

Distribution Sort with Randomized Cycling*

Jeffrey Scott Vitter[†] David A. Hutchinson[‡]

July 18, 2005

Abstract

Parallel independent disks can enhance the performance of external memory (EM) algorithms, but the programming task is often difficult. Each disk can service only one read or write request at a time; the challenge is to keep the disks as busy as possible. In this paper we develop a randomized allocation discipline for parallel independent disks, called *randomized cycling*. We show how it can be used as the basis for an efficient distribution sort algorithm, which we call *randomized cycling distribution sort* (RCD). We prove that the expected I/O complexity of RCD is optimal. The analysis uses a novel reduction to a scenario with significantly fewer probabilistic interdependencies. We demonstrate RCD's practicality by experimental simulations. Using the randomized cycling discipline, algorithms developed for the unrealistic multihead disk model can be simulated on the realistic parallel disk model for two important classes of algorithms: the class of *multi-pass* algorithms, which make a complete pass through their data before accessing any element a second time, and the algorithms based upon the well-known distribution and merge paradigms of EM computation.

*An earlier version of this paper appeared in *Proceedings of the 12th Annual SIAM/ACM Symposium on Discrete Algorithms*, Washington, DC, January 2001.

[†]Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-2066. Email: jsv@purdue.edu. Web: <http://www.science.purdue.edu/jsv/>. Support was provided in part by the Army Research Office through grant DAAD19-01-1-0725 and by the National Science Foundation through grants CCR-9877133, CCR-0082986, and IIS-0415097.

[‡]Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada K1S 5B6. Email: David.Hutchinson@pteran.ca. This work was done while the author was at Duke University, supported in part by the National Science Foundation through grant CCR-0082986.

1 Introduction

External memory (EM) algorithms are designed to be efficient when the problem data are too numerous to fit into the high-speed random access memory (RAM) of a computer and must reside on external devices such as disk drives [18]. In order to cope with the high cost of accessing data, 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 steps on the elements in the block while in the high-speed memory. The speedup can be considerable.

A second effective strategy for EM algorithms is the use of multiple parallel disks; whenever an input/output operation is performed, D blocks are transferred in parallel between memory and each of the D disks (one block per disk). An easy way to convert an EM algorithm designed for a single disk into an EM algorithm that utilizes parallel disks is the well-known technique of *disk striping*, in which the D blocks that are accessed at any given time reside at the same offset on each of the D respective disks. Disk striping can be shown to be equivalent to having a single disk with larger block-size BD , and its I/O performance for problems like sorting is suboptimal when D is large. An optimal EM algorithm for such problems thus 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 offset on its disk [18].

Designing algorithms for independent parallel disks has turned out to be ad hoc and relatively difficult [21, 14, 15, 5, 6, 7, 8, 3, 4, 18], and in practice the added overhead often makes the algorithms slower than those based upon disk striping. It is therefore highly desirable to develop efficient techniques for converting serial EM algorithms into EM algorithms that use parallel disks independently.

In this paper we develop a randomized distribution sort algorithm motivated by the simple randomized merge sort (SRM) algorithm of Barve et al. [5, 13] and Barve and Vitter [6], but with provably optimal performance for all parameter setting. We also show how the techniques can be generalized to provide optimal speedup for the class of *multipass EM algorithms* when run on parallel disks. Before we elaborate on these contributions, let us first review past work.

1.1 Previous Work

Sorting is a heavily used application and subroutine in external memory computing. The two main approaches to sorting are merge sort and distribution sort. *Merge sort* consists of two phases: the run formation phase and the merging phase. During run formation, the N input elements are input one memory-load at a time; each memory-load is sorted and written to the disks as a “run”. In the merge phase, the sorted runs are merged together $\Theta(M/B)$ at a time (where M is the internal memory size and B is the block size) in a round-robin manner until a single sorted run remains. In *distribution sort* the approach is to partition the data into S approximately equally sized subfiles (or “buckets”). In a *splitter selection phase*, we judiciously choose $S - 1 = \min\{(M/B)^{\Theta(1)}, 2N/M\}$ splitter elements from the data. In the subsequent *distribution phase* the input data are read sequentially and partitioned via the splitters into buckets. The blocks of each bucket are stored on the disks as they are formed. The splitter selection and distribution phases are repeated recursively until the buckets are small enough to be sorted in internal memory. The individual buckets are then concatenated together to form the desired output.

Barve et al. [5, 13] and Barve and Vitter [6] develop a sorting algorithm for parallel disks called *simple randomized merge sort* (SRM). SRM is noticeably faster than merge sort with disk striping, even when the number of disks is small. Randomization is used to choose a disk on which the first block of each run is located. Subsequent blocks in that run follow a simple round-robin order over the disks. The blocks input from the disks at each step of the merge phase tend to be distributed evenly among the disks, but for some values of the parameters an effect akin to the maximum bucket occupancy in statistics [20] results in a provably uneven distribution, and the I/O performance of SRM is suboptimal.

The two strategies of blocked access and parallel block transfer were proposed in the I/O model of Aggarwal and Vitter [1]. The model permits D blocks to be accessed at arbitrary locations in a single I/O operation, which is convenient for algorithm design, although unrealistic. In order to support D simultaneous I/O operations, the more realistic *parallel disk model* of Vitter and Shriver [21, 18] requires that each of the D blocks accessed in an I/O must reside on a separate disk. Recently, Sanders et al. [16] proposed an elegant and provably good randomized simulation technique for converting algorithms designed for the Aggarwal-Vitter model to the more realistic parallel disk model with only a constant factor slowdown. Their technique involves creating multiple copies of the disk blocks and randomly allocating

each block to one of the D disks. The resulting disk occupancies are not completely independent since the number of blocks is fixed, but they are negatively correlated, which encourages an even distribution. (A summary of the derivation is given in Section 3.)

A number of important EM algorithms have been proposed with support for blocked access to external memory but without support for parallel independent disks. These include distribution sweeping and a variety of geometric algorithms based upon distribution sweeping [10], trapezoidal decomposition, triangulation of a simple polygon, red-blue line intersection in GIS [3], and spatial join [2]. In many cases, the algorithms can be adapted to use parallel disks on an ad hoc basis by applying techniques from the previously developed parallel-disk sorting algorithms, such as those by Vitter and Shriver [21], Nodine and Vitter [14, 15], and Dehne et al. [7, 8], but in practice these techniques are often slower than simpler approaches based upon disk striping.

1.2 Our Contributions

In this paper we develop a simple, practical and provably optimal randomized algorithm for distribution sort with parallel disks. Our method is motivated by the simplicity and practicality of the SRM approach for merge sort. We therefore examine simple randomization schemes that promise to be practical and efficient. The key point in distribution sort is that the writing in the distribution phase can be done in a lazy manner. There is no need for all D blocks specified in one write operation to actually be written to disk before the blocks specified in the next write operation are written. As long as there is buffer space to temporarily cache the buckets waiting to be written to the disks, the writing can proceed optimally (up to a constant factor). There is thus no suboptimal effect akin to maximum bucket occupancy as there is with SRM, as mentioned in Section 1.1.

In Section 2 we define our notation and propose randomized variants of distribution sort that are simple and practical to implement, but their analysis is difficult because of extensive dependencies between the random variables in question. We also present our main result (proven later in Section 4), which shows that our *randomized cycling distribution sort* (RCD) variant has optimal expected I/O complexity. In Section 3 we discuss in detail the *fully randomized distribution sort* (FRD) variant, to which the analysis of Sanders et al. [16] can be applied. FRD is sometimes non-optimal in terms of I/O and it is somewhat complicated to implement, although it is optimal in terms of write operations. In Section 4 we analyze RCD, our

substantially simpler variant. We give a novel reduction to show that the number of write operations are bounded by those of FRD, and the number of read operations is trivially optimal. In Section 6 we discuss some interesting applications. Our analysis of RCD has subsequently been applied to optimal design and analysis of merge sort for parallel disks [12]. We can also use RCD to simulate a large class of algorithms for the Aggarwal-Vitter I/O model on the parallel disk model. In many cases, in contrast to more general simulation technique of Sanders et al. [16], we need only one copy of each data block. In other cases, we need multiple copies of blocks as before, but the randomization is always simpler to implement and can be done in a local manner, which is useful to exploit locality in algorithm design. Finally, we give concluding remarks and open problems.

2 Randomized Distribution Sort

We now consider the write component of the distribution phase of distribution sort. Each disk has an associated first-in first-out *disk queue*, and the D queues share a common buffer pool of internal memory, capable of holding W blocks collectively. The S output buckets in the distribution phase issue blocks of data to the disk queues, as shown in Figure 1. The algorithm variants FRD, SRD, RSD, and RCD, introduced below, differ primarily in their *allocation discipline*, which is the method by which their buckets allocate blocks to the disk queues. In each disk write cycle, up to D blocks, at most one per disk, are removed from the queues and physically written to the disks. Since the queue space is limited, we consider a collective block-arrival rate to the queues of $(1 - \epsilon)D$ per disk write cycle, for some $0 \leq \epsilon < 1$.

Fully Randomized Distribution Sort (FRD). In FRD each bucket selects a separate, randomly chosen disk for each block that it issues. While this variant is not considered for distribution sorting in [16], the analysis of [16] does apply to the writing phase for this variant. FRD is complicated to implement because of the bookkeeping necessary during the partitioning; each bucket must keep a list of its blocks on each disk so that they can be linked together. Another disadvantage of FRD is that the blocks being read during the reading phase (which correspond to a bucket in the previous pass) are not striped on the disks or perfectly evenly distributed. When N is relatively small or $M/BD = o(\log D)$, the algorithm is non-optimal, and we get a noticeably uneven distribution.

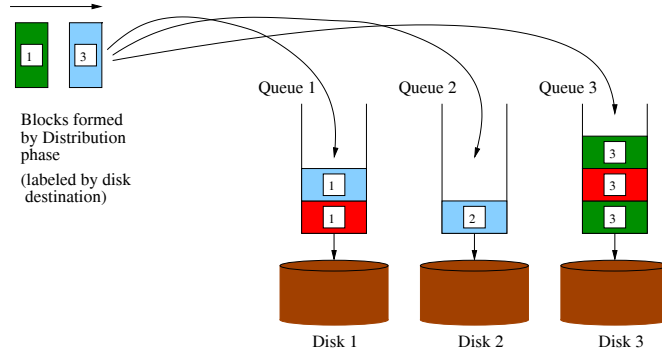


Figure 1: Distribution phase for the case of $D = 3$ disks. The blocks are formed during the bucket partitioning process and are assigned to disk queues according to the allocation discipline.

Simple Randomized Distribution Sort (SRD). Each bucket issues its blocks to consecutive positions in a simple, round-robin manner. The disk selected for the first block is called the *starting point* and is chosen uniformly randomly.

Randomized Striping Distribution Sort (RSD). Each bucket issues its blocks to consecutive positions in a simple, round-robin manner, but a new random starting point is chosen after every D blocks. This technique was suggested in [16] for distribution sorting, but no analysis or experiments were provided.

Randomized Cycling Distribution Sort (RCD). Each bucket chooses a random cycle order of the disk numbers (among the $D!$ possible permutations) and allocates its blocks to disks in a round-robin manner according to the cycle order.

The advantage of SRD, RSD, RCD is that they are easy to implement and the blocks from each bucket are striped together for reading in the next phase. The blocks actually written in a single I/O are therefore to different stripes, since the blocks for a given bucket go to a certain stripe. If a RAID system is used and parity information is maintained for error recovery, an extra parity block for each bucket can be maintained in internal memory. At any given time, the parity block is the parity for the blocks written so far in the current stripe for that bucket. When we write the $(D - 1)$ st block

of the stripe, we can then write out the parity block as the D th block of the stripe.

By contrast, in FRD writing is done by stripes. However the blocks in a bucket are not situated together in stripes; they appear at various locations on the disks, and as a result the extra bookkeeping is needed by FRD to link together the blocks in the bucket. If desired, the FRD method of striped writing but non-striped reading can also be supported by SRD, RSD, and RCD.

We define the following important random variables that govern the behavior of these methods. The main difficulty with the analysis of SRD, RSD, RCD (which is still open for SRD and RSD), is the extensive dependence between the random variables in question.

Definition 1 [Queuing Model] For purposes of our analysis we assume the following definitions and sequence of events during each time step t and for each queue $0 \leq i \leq D - 1$:

1. We define the queue length $Q_i^{(t)}$ to be the size of queue i at the beginning of time step t before any arrivals or consumption occur for that time step. We let $Q^{(t)} = \sum_{0 \leq i < D} Q_i^{(t)}$ denote the total of the queue sizes (or simply the total queue size) at time t .
2. The consumption process removes a block, if any exist, from queue i at time step t and writes it to the corresponding disk.
3. Some number $A_i^{(t)}$ of new blocks arrive for queue i at time step t (after the consumption for that time step). The total number of arrivals at time t among all the queues, namely, $A^{(t)} = \sum_{0 \leq i < D} A_i^{(t)}$, is $D(1 - \epsilon)$.

The above order of events, in which consumption is done before arrivals, is easier to analyze than the reverse, and results in a slightly more conservative analysis.

So far we haven't discussed what happens if the buffer pool of size W overflows. To handle that case, we insert the following event as event 2.5 to occur between events 2 and 3 above:

- 2.5. **while** $Q^{(t)} + A^{(t)} > W$ **do**
 Remove a block from queue i and write it to the corresponding disk
 enddo

Definition 2 We define $I^{(t)}(b)$ to be the number of blocks issued by bucket b in time step t . In particular, we have $\sum_b I^{(t)}(b) = A^{(t)} = D(1 - \epsilon)$.

Definition 3 For SRD, RCD, and RSD, we define the *cycle order* of a bucket to be the permutation $\langle i_0, i_1, \dots, i_{D-1} \rangle$ of queue indices that specify the round-robin order in which the bucket places blocks into the queues. In other words, the j th block of the bucket is issued to queue $i_{j \bmod D}$. For RCD, the cycle order of a bucket can be an arbitrary permutation of $\{0, 1, \dots, D-1\}$. For SRD, we have $i_{j+1} = i_j + 1 \bmod D$, where i_0 can be any value $0 \leq i_0 < D$.

Definition 4 The *configuration* of a bucket b specifies the schedule $I^{(1)}(b)$, $I^{(2)}(b)$, ... of blocks issued by the bucket and its cycle order $\langle i_0, i_1, \dots, i_{D-1} \rangle$. In other words, $Q^{(t)}$ and each $Q_i^{(t)}$ are deterministic functions of the configurations of the buckets.

There are two important issues for I/O efficiency in distribution sort: reading in the blocks using the parallel disks, and writing the blocks using parallel disks. Both must be done optimally. Our main result, which we prove in Section 4, is Theorem 1. It shows for RCD that the conditional consumption step will very seldom be needed, and the expected number of parallel write operations will be linear in the number of queuing events.

Theorem 1 *Consider the model of Definition 1 and the allocation discipline of RCD. Let $n^{(t)}$ be the number of parallel writes executed in time step t . Then for buffer pool size $W = (\ln 2 + \delta)D/\epsilon$, for some constant $\delta > 0$, we have $\mathbb{E}(n^{(t)}) = 1 + e^{-\Omega(D)}$.*

We conjecture that similar bounds hold for the write I/Os in SRD and RSD, as suggested by the experiments in Section 5. In RCD, SRD and RSD the blocks are striped on the disks, so their reading components are automatically optimal. We show in Section 3 that FRD satisfies the same write bound as does RCD in Theorem 1, but that the reading component is nonoptimal.

3 FRD: Almost Independent Scenario

The writing component of the distribution phase of FRD is optimal and satisfies the same bound given for RCD in Theorem 1. The analysis of the writing component of FRD is given by Sanders et al. [16]). In particular, with the appropriate setting of the buffer size, Sanders et al. show that the probability that the buffer overflows is exponentially small. We summarize the derivation below. We then demonstrate that the reading component of FRD is not optimal.

In FRD each bucket contributes only one block in total; that is, $\sum_t I^{(t)}(b) = 1$ for all buckets b . Therefore, the assignment of a bucket's single block to a queue is independent from the assignments of all other blocks. The only (very limited) dependence arises because there are a total of $D(1 - \epsilon)$ blocks issued collectively by the buckets in step t ; that is, $\sum_b I^{(t)}(b) = A^{(t)} = D(1 - \epsilon)$ for all time steps t . Let us define the notation $\widehat{Q}^{(t)}$ and $\widehat{Q}_i^{(t)}$ to denote $Q^{(t)}$ and $Q_i^{(t)}$ for the special case of FRD.

The size of queue i at time $t + 1$ can be expressed recursively by

$$\widehat{Q}_i^{(t+1)} = \widehat{Q}_i^{(t)} - 1 + [\widehat{Q}_i^{(t)} = 0] + A_i^{(t)}. \quad (1)$$

where $A_i^{(t)}$ is the number of blocks that arrive for queue i at time t . In the steady state $t = \infty$, we get the recurrence

$$\widehat{Q}_i^{(\infty)} = \widehat{Q}_i^{(\infty)} - 1 + [\widehat{Q}_i^{(\infty)} = 0] + A_i^{(\infty)}. \quad (2)$$

Let us define the probability generating functions

$$\widehat{Q}_i(z) = \sum_{t \geq 0} \text{Prob}\{\widehat{Q}_i^{(\infty)} = k\} z^k; \quad (3)$$

$$A_i(z) = \sum_{t \geq 1} \text{Prob}\{A_i^{(\infty)} = k\} z^k. \quad (4)$$

The recurrence relation (2) translates into the following equation on the generating functions:

$$\widehat{Q}_i(z) = \left(\frac{1}{z} \widehat{Q}_i(z) + \widehat{Q}_i(0) - \frac{1}{z} \widehat{Q}_i(0) \right) A_i(z), \quad (5)$$

whose solution is

$$\widehat{Q}_i(z) = \frac{\widehat{Q}_i(0) \left(1 - \frac{1}{z}\right)}{A_i(z)^{-1} - \frac{1}{z}}. \quad (6)$$

The random variable $A_i^{(\infty)}$ is the sum of $(1 - \epsilon)D$ binary random variables, each with probability $1/D$ of having value 1 and probability $1 - 1/D$ of having value 0. Therefore, we have

$$A_i(z) = \left(\frac{z}{D} + 1 - \frac{1}{D} \right)^{(1 - \epsilon)D}. \quad (7)$$

We can solve for $\widehat{Q}_i(0)$ in (6) by taking the limit of (6) as $z \rightarrow 1$. Since $\widehat{Q}_i(z)$ is a probability generating function, we have $\widehat{Q}_i(1) = 1$. By

¹We use the notation $[condition]$ to denote 1 if *condition* is true and 0 otherwise.

L'Hôpital's rule and by plugging in (7) for $\mathcal{A}_i(z)$, we get $\widehat{Q}_i(0) = \epsilon$. Substituting this into (6) and simplifying, we get the following closed form:

$$\begin{aligned}\widehat{Q}_i(z) &= \frac{(1 - \frac{1}{z})\epsilon}{\mathcal{A}_i(z)^{-1} - \frac{1}{z}} \\ &= \frac{(1-z)\epsilon}{1 - z\mathcal{A}_i(z)^{-1}} \\ &= \frac{(1-z)\epsilon}{1 - z/(\frac{z}{d} + 1 - \frac{1}{d})^{(1-\epsilon)D}}.\end{aligned}\quad (8)$$

Let W denote the total buffer pool size. We get a Chernoff-like tail bound on the probability of buffer overflow by starting with Markov's inequality applied to $e^{s\widehat{Q}^{(t)}}$:

$$\text{Prob}\{\widehat{Q}^{(t)} > W\} = \text{Prob}\{e^{s\widehat{Q}^{(t)}} > e^{sW}\} < e^{-sW} \mathbb{E}(e^{s\widehat{Q}^{(t)}}). \quad (9)$$

By definition of $\widehat{Q}^{(t)}$, the expected value term is

$$\mathbb{E}(e^{s\widehat{Q}^{(t)}}) = \mathbb{E}(e^{\sum_{0 \leq i < D} s\widehat{Q}_i^{(t)}}) = \mathbb{E}\left(\prod_{0 \leq i < D} e^{s\widehat{Q}_i^{(t)}}\right). \quad (10)$$

If the random variables $\langle \widehat{Q}_i^{(t)} \rangle_{0 \leq i < D}$ were independent, the expected value operator could be moved inside the product and thus the right-hand-side of (10) replaced by

$$\prod_{0 \leq i < D} \mathbb{E}(e^{s\widehat{Q}_i^{(t)}}) = (\mathbb{E}(e^{s\widehat{Q}_1^{(t)}}))^D. \quad (11)$$

The random variables $\langle \widehat{Q}_i^{(t)} \rangle_{0 \leq i < D}$ are not independent, but fortunately they are negatively associated², which allows the right-hand-side of (10) to be bounded by (11).

We are now ready to bound $\text{Prob}\{\widehat{Q}^{(t)} > W\}$ using (9). Choosing $s = \epsilon$ and using the fact that $\mathbb{E}(e^{s\widehat{Q}_i^{(t)}}) < \mathbb{E}(e^{s\widehat{Q}_1^{(\infty)}}) = \widehat{Q}_1(e^s)$, we get

$$\text{Prob}\{\widehat{Q}^{(t)} > W\} < e^{-\epsilon W} (\widehat{Q}_1(e^\epsilon))^D. \quad (12)$$

By applying L'Hôpital's rule to (8), we find that $\widehat{Q}_1(e^\epsilon) < 2$, and hence we get

$$\text{Prob}\{\widehat{Q}^{(t)} > W\} < e^{-(\epsilon W - D \ln 2)} = e^{-\delta D} \quad (13)$$

²Discrete random variables X_1, X_2, \dots, X_n are *negatively associated* [9] if, for any nondecreasing functions f and g and any disjoint subsets I and J of $[1, n]$, we have $\mathbb{E}(f(X_i, i \in I)g(X_j, j \in J)) \leq \mathbb{E}(f(X_i, i \in I))\mathbb{E}(g(X_j, j \in J))$. Intuitively, if X_i is large, then X_j tends to be small.

when $W = (\ln 2 + \delta)D/\epsilon$. In other words, a conditional consumption step is only executed with a probability that is exponentially small in δD . If such a rare event occurs, the number of conditional consumption steps needed to eliminate overflow can be conservatively bounded by $D(1 - \epsilon) + W$, since after that number of steps the queues would be empty. Therefore the total expected number $E(n^{(t)})$ of parallel write operations made by FRD at step t is bounded by

$$1 + (D(1 - \epsilon) + W) \text{Prob}\{\widehat{Q}^{(t)} > W\} < 1 + O(D) e^{-\delta D} = 1 + e^{-\Omega(D)}, \quad (14)$$

which is the bound used in Theorem 1 for RCD.

Although this shows that the writing component of FRD is optimal, the reading component of FRD can be nonoptimal by a factor of $\ln D / \ln \ln D$ when D is large because of unbalanced I/O operations [20]. Consider the following example: Let the block size be large, say 250 KB to amortize the seek latency over many data elements. Let the number of disks be $D = 400$, and let the memory size be 250 MB. Let the problem size be 400 MB (i.e., 1600 blocks) and let the number of buckets be 4, giving about 400 blocks per bucket. For each bucket, the expected maximum occupancy of its blocks on the disks is $\approx (\ln 400) / \ln \ln 400 \approx 3.3$, even though the average number of blocks per disk is only 1. The reading is thus about three times slower than if the input file were striped, which would be the case with SRD, RCD, and RSD. The amount of imbalance can be reduced somewhat by choosing smaller block sizes, but then the I/O costs would increase because of the increased number of random accesses to disk [19].

FRD requires that lists be maintained to link together the blocks in each bucket on each disk, and this bookkeeping complicates the implementation. For similar effort, a better approach would be to use Phase I of the algorithm of Vitter and Shriver [21], where data are written to the disks in stripes and the buckets tend to be more evenly distributed.

4 Analysis of the Total Queue Size $Q^{(t)}$ in RCD

In this section we give a proof of Theorem 1. Our objective is to derive a type of Chernoff bound on the total queue size $Q^{(t)}$ of RCD during its writing phase. This bound is the same one given in (9)–(11) for FRD, which has substantially more independence. After we derive the Chernoff bound, the rest of the proof of Theorem 1 proceeds as in (13) and (14) for FRD.

Our strategy for getting the desired bound on $E(e^{sQ^{(t)}})$ is to do a series of transformation steps on an instance of RCD, after which all buckets are

```

for  $r := 1$  to  $t$  do
  while there is a nonsingleton bucket  $b$ 
    that issues at least one block at time step  $r$ 
  do the following transformation step
    Remove one block from bucket  $b$  at time step  $r$ ;
    Create a new singleton bucket that issues one block at time step  $r$ 
      to a randomly chosen queue
    enddo
  enddo

```

Notation: For each individual transformation step, we let $Q^{(t)}$ represent the sum of queues at time t . $Q'^{(t)}$ will represent the sum of queues at time t after a block b has been removed but before a corresponding singleton bucket has been created. $Q''^{(t)}$ will represent the sum of queues at time t after bucket b has been transformed.

Figure 2: The bucket transformation steps.

singleton buckets (which corresponds to FRD). We reduce an instance of RCD to an instance of FRD via the series of bucket transformation steps shown in Figure 2.

Definition 5 A bucket b is a *singleton* bucket if it issues a total of one block over all time steps; that is, $\sum_t I^{(t)}(b) = 1$.

As mentioned ((10) and (11) in Section 3), Sanders et al. [16] have shown that the total queue size $\widehat{Q}^{(t)}$ for FRD (the situation in which all buckets are singletons) satisfies $\mathbb{E}(e^{s\widehat{Q}^{(t)}}) \leq \prod_{0 \leq i < D} \mathbb{E}(e^{s\widehat{Q}_i^{(t)}}) = (\mathbb{E}(e^{s\widehat{Q}_i^{(t)}}))^D$. The right-hand-side is the corresponding quantity for the case in which the queue sizes are completely independent. In the remainder of Section 4 we will show that each bucket transformation step causes $\mathbb{E}(e^{sQ^{(t)}})$ to increase or remain the same. Hence we will have

$$\mathbb{E}(e^{sQ^{(t)}}) \leq \mathbb{E}(e^{s\widehat{Q}^{(t)}}) \leq \prod_{0 \leq i < D} \mathbb{E}(e^{s\widehat{Q}_i^{(t)}}) = (\mathbb{E}(e^{s\widehat{Q}_i^{(t)}}))^D, \quad (15)$$

which establishes a Chernoff-type bound for $Q^{(t)}$ that is bounded by the Chernoff-type bound for FRD. Once (15) is established, the remainder of the proof of Theorem 1 follows from (13) and (14) applied to RCD.

4.1 Main Lemmas

The bucket transformations of Figure 2 iteratively convert an instance of RCD into an instance of FRD. Our objective is to show that (15) holds when the transformations of Figure 2 are applied. First we define $f(x) = e^{sx}$ so that the left-hand-side of (15) is $E(f(Q^{(t)}))$. Lemma 1 below is our main lemma. It shows that the bucket transformations of Figure 2 have the desired effect.

Lemma 1 *Each bucket transformation step described in Figure 2, in which one block at time step r is removed from the bucket and a new singleton bucket is created with one block at the same time step, causes the quantity $E(f(Q^{(t)})) = E(e^{sQ^{(t)}})$ to either increase or stay the same. In other words, for each transformation step in Figure 2, $E(f(Q''^{(t)})) - E(f(Q^{(t)})) \geq 0$.*

We will prove Lemma 1 in Section 4.2. Critical blocks and critical starting points are key concepts used to prove Lemma 1.

Definition 6 (Critical block) Consider an arbitrary bucket b which issues one or more block(s) at time step $r < t$, and consider any fixed configurations for the other buckets. Consider the following two scenarios:

1. One of the blocks issued at time step r from bucket b is placed into queue i . Let $Q^{(t)}$ be the total size of the queues at time step t .
2. Same as case 1, except that we remove the block that bucket b contributes to queue i at time r (without moving the times or placements of any of the other blocks). Let $Q'^{(t)}$ be the resulting total size of the queues at time step t .

We say that the block issued to queue i is a *critical block* for bucket b with respect to time step t if

$$Q'^{(t)} = Q^{(t)} - 1. \tag{16}$$

Note that it is always true that $Q^{(t)} - 1 \leq Q'^{(t)} \leq Q^{(t)}$. Criticality means that the first “ \leq ” is actually an equality.

Definition 7 (Critical starting point) Consider the case where the first block issued by bucket b (say, to queue i) is a critical block for bucket b with respect to time step t . We say that queue i is a *critical starting point* for bucket b with respect to time step t .

| | | | | | | | | | | | |
|---------------|----------|----------|----------|-----|----------|----------|----------|----------|----------|----------|-------------|
| time t' | r | $r + 1$ | $r + 2$ | ... | | | | | | $t - 1$ | t |
| $Q_i^{(t')}$ | ≥ 2 | ≥ 2 | ≥ 2 | ... | ≥ 2 | ≥ 2 | ≥ 2 | ≥ 2 | ≥ 2 | ≥ 2 | $Q_i^{(t)}$ |
| Item Arrivals | \odot | | | ... | \odot | \odot | | \odot | | | |

Figure 3: A queue with a critical starting point: Removing the arrival at time r will reduce $Q_i^{(t)}$ by one block; in fact, $Q_i^{(t')}$ gets reduced by 1 for each $r < t' \leq t$.

Lemmas 2–4 are useful for reasoning about the effect of block arrivals upon the sizes of queues. Please refer to Figure 3.

Lemma 2 *The following conditions are equivalent:*

1. *Initially, we have $Q_i^{(t')} \geq 1$, for all $r < t' < t$.*
2. *If we add a new block to queue i at time step r , then $Q_i^{(t')}$ increases by 1 at each time step t' , for $r < t' \leq t$.*

Proof: Consider the addition of a single new block to queue i at time step r , and assume that this new block increases $Q_i^{(t)}$. It must be true that a block is already consumed from queue i in every time step t' , where $r < t' < t$; otherwise, the new block would simply be consumed at some step where previously no block was consumed from queue i , thus not affecting $Q_i^{(t)}$. Recall that in our model $Q_i^{(t)}$ is the queue size at the *beginning* of time step t , and consumption occurs prior to arrival of blocks in each time step. Therefore, prior to the addition of the new block, we have $Q_i^{(t')} \geq 1$, for each time step t' , for $r < t' < t$.

Conversely, by a similar argument, if $Q_i^{(t')} \geq 1$, for all $r < t' < t$, then one block is consumed at each time step t' . Thus, adding a new block to queue i at time step r will propagate through all the time steps and increase $Q_i^{(t')}$ by 1 for each $r < t' \leq t$. \square

Lemma 3 *Consider an arbitrary bucket b with starting point i whose first block(s) appear at time step $r < t$. The following conditions are equivalent:*

1. *The starting point i is critical for bucket b with respect to time step t .*

2. Initially we have $Q_i^{(t')} \geq 2$, for all $r < t' < t$.
3. If we remove the block from queue i at time step r , then $Q_i^{(t')}$ decreases by 1 for each time step t' , for $r < t' \leq t$.

Proof: Conditions 1 and 3 are easily seen to be equivalent. Conditions 2 and 3 are equivalent to the two conditions in Lemma 2, but in reverse order of time; that is, they deal with removal of a block rather than addition of a block. All three conditions are therefore equivalent. \square

One way to think about critical starting points is by an analogy between the queue size and the water level in a lake. Suppose that each day the sun removes a gallon of water from a lake. Then, later in the evening, it may rain, in which case the lake gets replenished with one or more gallons of water. If the lake always has at least two gallons at the start of each day, then if we remove a gallon of water early one morning in April, the lake will contain one gallon less in September than it would normally have. If on the other hand, the lake has only one gallon at the start of June 15, then the sun will empty the lake. In this case, if we remove a gallon in April, there will be no change in September, since all the water in September must have been added after June 15. In other words, in order for a removal in April to have an effect in September, the water level of the lake must be sufficiently high each day in between and never go dry. The same is true for a queue with a critical starting point. Removing a block at time r will decrease the queue size by 1 at a later time t when there are initially two or more items in the queue at the beginning of each time step.

Using similar ideas, one can show that the closer to time step t that a block is inserted, the more likely it will affect the queue size at time step t .

Lemma 4 *If a block arrival at queue i is changed from time step r to instead be at time step r' , where $r < r' < t$, without moving the times or placements of any of the other blocks, then $Q_i^{(t)}$ either increases by 1 or stays the same.*

4.2 Proof of Lemma 1

To prove Lemma 1, let us consider the effect of a single bucket transformation step (see Figure 2) applied to bucket b at time step r . We assume that b does not issue any blocks before time r . We let the configurations of the other buckets be arbitrary and fixed. Let the cycle order for bucket b be

$\langle i_0, i_1, \dots, i_{D-1} \rangle$. Recall that the cycle order of a bucket is a permutation of the queue indices (see Definition 3). We consider for the moment a fixed starting point i_0 for the cycle order before the transformation, and after the transformation we assume a shifted cycle order $\langle i_1, \dots, i_{D-1}, i_0 \rangle$. The reason for the shifted cycle order is that, after the transformation, the blocks of the bucket are issued to the same queues as before the transformation, except for the single block that was removed.

We can express the total queue size $Q''^{(t)}$ after the transformation by

$$Q''^{(t)} = Q'^{(t)} + [\text{new bucket increases queue size}]$$

Recall that $Q^{(t)}$, $Q'^{(t)}$ and $Q''^{(t)}$ are defined in the description of Figure 2.

If bucket b 's starting point i_0 is critical (refer to Definition 6), then $Q'^{(t)} = Q^{(t)} - 1$. Suppose that c of the D possible starting points for bucket b are critical with respect to time step t . The new singleton bucket issues its single block to a randomly chosen queue, just like any other RCD or FRD bucket. The key point is that if the new singleton bucket issues its block to one of the c critical starting points, which happens with probability c/D , then by Lemmas 2 and 3, that queue size will be incremented and we will have $Q''^{(t)} = Q'^{(t)} + 1 = Q^{(t)}$. If the new bucket issues its block to one of the $D - c$ non-critical starting points, which happens with probability $1 - c/D$, then by Lemmas 2 and 3, we will have $Q''^{(t)} \geq Q'^{(t)} = Q^{(t)} - 1$.

With the notation $f(x) = e^{sx}$ from Section 4.1, the above relations give us the following lower bound on the conditional expectation, given that the starting point of bucket b is critical:

$$\begin{aligned} & \mathbb{E}(f(Q''^{(t)}) \mid \text{starting point is critical}) \\ & \geq \left(1 - \frac{c}{D}\right) \mathbb{E}(f(Q^{(t)} - 1) \mid \text{starting point is critical}) \\ & \quad + \frac{c}{D} \mathbb{E}(f(Q^{(t)}) \mid \text{starting point is critical}), \end{aligned} \quad (17)$$

where the expectation is over the starting point of the new bucket.

If instead b 's starting point i_0 is non-critical, then by similar reasoning either $Q''^{(t)} = Q^{(t)}$ or $Q''^{(t)} = Q^{(t)} + 1$, and we get the conditional expectation

$$\begin{aligned} & \mathbb{E}(f(Q''^{(t)}) \mid \text{starting point is non-critical}) \\ & \geq \left(1 - \frac{c}{D}\right) \mathbb{E}(f(Q^{(t)}) \mid \text{starting point is non-critical}) \\ & \quad + \frac{c}{D} \mathbb{E}(f(Q^{(t)} + 1) \mid \text{starting point is non-critical}). \end{aligned} \quad (18)$$

Using the identities $E(f(X+1)) = f(1)E(f(X))$ and $E(f(X-1)) = E(f(X))/f(1)$, we can rewrite (17) and (18) as

$$\begin{aligned} & E(f(Q^{(t)}) \mid \text{starting point is critical}) \\ & \geq \left(\left(1 - \frac{c}{D}\right) \frac{1}{f(1)} + \frac{c}{D} \right) \\ & \quad \times E(f(Q^{(t)}) \mid \text{starting point is critical}); \end{aligned} \quad (19)$$

$$\begin{aligned} & E(f(Q^{(t)}) \mid \text{starting point is non-critical}) \\ & \geq \left(\left(1 - \frac{c}{D}\right) + \frac{c}{D} f(1) \right) \\ & \quad \times E(f(Q^{(t)}) \mid \text{starting point is non-critical}). \end{aligned} \quad (20)$$

Since there are c critical starting points and $D - c$ non-critical starting points, we can remove the conditioning as follows: Before the transformation we have

$$\begin{aligned} & E(f(Q^{(t)})) \\ & = \frac{c}{D} E(f(Q^{(t)}) \mid \text{starting point is critical}) \\ & \quad + \left(1 - \frac{c}{D}\right) E(f(Q^{(t)}) \mid \text{starting point is non-critical}). \end{aligned} \quad (21)$$

After the transformation we get

$$\begin{aligned} & E(f(Q^{(t)})) \\ & \geq \frac{c}{D} \left(\left(1 - \frac{c}{D}\right) \frac{1}{f(1)} + \frac{c}{D} \right) \\ & \quad \times E(f(Q^{(t)}) \mid \text{starting point is critical}) \\ & \quad + \left(1 - \frac{c}{D}\right) \left(\left(1 - \frac{c}{D}\right) + \frac{c}{D} f(1) \right) \\ & \quad \times E(f(Q^{(t)}) \mid \text{starting point is non-critical}). \end{aligned} \quad (22)$$

Combining (21) and (22), we get

$$\begin{aligned} & E(f(Q^{(t)})) - E(f(Q^{(t)})) \\ & \geq -\frac{1}{f(1)} \left(\frac{c}{D} (f(1) - 1) - \frac{c^2}{D^2} (f(1) - 1) \right) \\ & \quad \times E(f(Q^{(t)}) \mid \text{starting point is critical}) \\ & \quad + \left(\frac{c}{D} (f(1) - 1) - \frac{c^2}{D^2} (f(1) - 1) \right) \\ & \quad \times E(f(Q^{(t)}) \mid \text{starting point is non-critical}). \end{aligned} \quad (23)$$

The following lemma shows that the left-hand-side of (23) is nonnegative, thereby finishing the proof of Lemma 1.

Lemma 5 $E(f(Q^{(t)} \mid \text{starting point is non-critical}) \geq \frac{1}{f(1)} E(f(Q^{(t)} \mid \text{starting point is critical}))$.

Proof: Consider the effect of modifying the cycle order of an arbitrary bucket b , keeping the cycle orders of the other buckets fixed. Consider a cycle order $\langle i_0, i_1, \dots, i_{D-1} \rangle$ for bucket b in which the starting point i_0 is critical with respect to t . Now exchange i_0 with one of bucket b 's $D - c$ non-critical starting points i_j to give the cycle order $\langle i_j, i_1, \dots, i_{j-1}, i_0, i_{j+1}, \dots, i_{D-1} \rangle$. We will show that any such exchange reduces the queue size $Q^{(t)}$ by at most 1.

Each cycle order with a critical starting point maps into $D - c$ cycle orders with non-critical starting points. Similarly, each cycle order with a non-critical starting point is mapped into by c cycle orders with critical starting points. The net effect is that the set of all such exchanges (mappings) induces a directed bipartite graph in which the two vertex sets are as follows:

1. *the $c(D - 1)!$ cycle orders of bucket b for which the starting point is critical, and*
2. *the $(D - c)(D - 1)!$ cycle orders of bucket b for which the starting point is non-critical.*

The outdegree of each vertex in set 1 is $D - c$, and the indegree of each vertex in set 2 is c . Therefore, a uniform distribution on vertex set 1, when a random exchange is applied, is mapped into a uniform distribution on vertex set 2.

Our approach to proving the lemma is to compare the expected value of $f(Q^{(t)}) = e^{sQ^{(t)}}$ conditioned on a critical starting point with the value of $f(Q^{(t)}) = e^{sQ^{(t)}}$ conditioned on a non-critical starting point. Since the mappings preserve a uniform distribution, it suffices to show that each exchange can decrease the queue size $Q^{(t)}$ at time step t by at most 1. If we succeed in doing so, we will have $E(f(Q^{(t)} \mid \text{starting point is non-critical})) \geq E(f(Q^{(t)} - 1 \mid \text{starting point is critical}))$. Using the fact that $f(x) = e^{sx}$ and thus $f(Q^{(t)} - 1) = \frac{1}{f(1)} f(Q^{(t)})$, we get the desired result $E(f(Q^{(t)} \mid \text{starting point is non-critical})) \geq \frac{1}{f(1)} E(f(Q^{(t)} \mid \text{starting point is critical}))$.

The rest of the proof consists of showing the claim that each exchange can decrease the queue size $Q^{(t)}$ at time step t by at most 1. In each exchange, the only queues whose sizes are affected are i_0 and i_j .

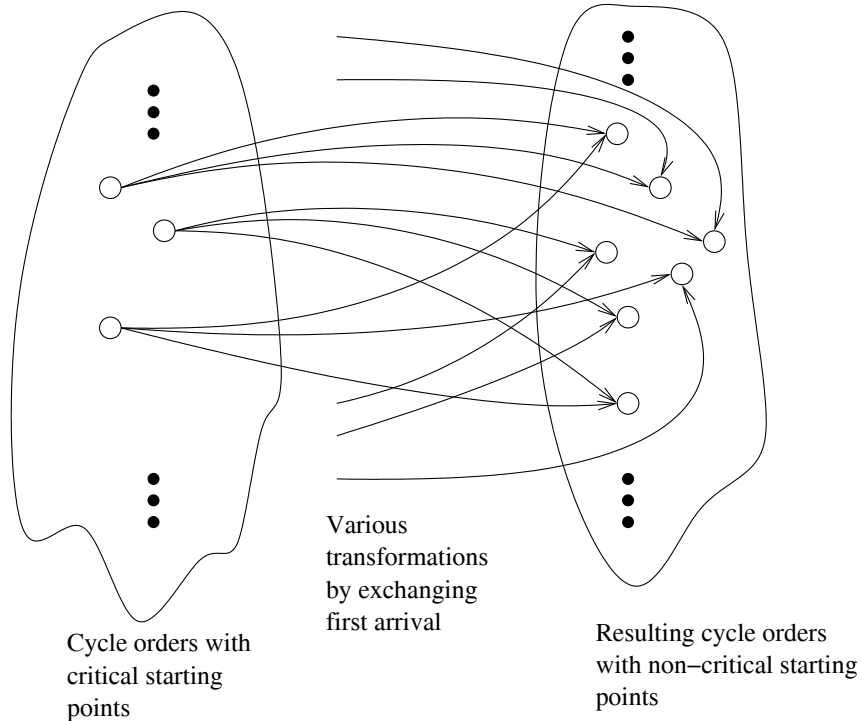


Figure 4: The mapping from critical to non-critical starting points for bucket b induces a directed bipartite graph. The figure illustrates a graph for $c = 2$ and $D - c = 3$.

Suppose first that the number of arrivals ℓ at queue i_0 before time step t does not change under the new cycle ordering. We can decompose our proposed swap in the queue order of b 's cycle as follows: Consider the ℓ' th time through the cycle order, for $\ell' = 0, 1, \dots, \ell$. In the original cycle order, let the time steps at which a block is contributed to queue i_0 be denoted $r_{\ell'}$, and let the time steps at which a block is contributed to queue i_j be denoted as $s_{\ell'}$. For each $\ell' = 0, 1, \dots, \ell$, we do the following two steps:

1. Remove the block that bucket b contributes at time $r_{\ell'}$ to queue i_0 , without moving the times or placements of any of the other blocks.
2. Insert a block into queue i_0 at later time $s_{\ell'}$.

By the definition of a critical starting point, the effect of Step 1 will be to decrease the final queue size $Q_{i_0}^{(t)}$ at time step t by 1. By Lemma 3,

for each $r_{\ell'} < t' < t$, we have $Q^{(t')}_{i_0} \geq 2$ before the removal, and thus $Q^{(t')}_{i_0} \geq 1$ after the removal. By Lemma 2, the effect of Step 2 will be to make $Q^{(t')}_{i_0} \geq 2$, for each $s_{\ell'} < t' < t$, and increase the final queue size $Q_{i_0}^{(t)}$ by 1 back to its original value. Therefore, each iteration of Steps 1 and 2 does not change $Q_{i_0}^{(t)}$.

We can do a similar analysis of the effect of the cycle reordering on the size of $Q_{i_j}^{(t)}$. As before we can decompose the reordering into ℓ iterations of a two-step procedure. For each $\ell' = 0, 1, \dots, \ell$, using the prior definitions of $r_{\ell'}$ and $s_{\ell'}$, we do the following two steps:

1. Remove the block that bucket b contributes at time $s_{\ell'}$ to queue i_j , without moving the times or placements of any of the other blocks.
2. Insert a block into queue i_j at the earlier time $r_{\ell'}$.

By Lemma 4, the effect of each iteration of Steps 1 and 2 will be to make $Q_{i_j}^{(t)}$ decrease by 1 or stay the same. Suppose that the ℓ' th iteration of Steps 1 and 2 causes $Q_{i_j}^{(t)}$ to decrease by 1. We will now show that all subsequent iterations of Steps 1 and 2 do not further decrease $Q_{i_j}^{(t)}$. By Lemma 3, before the ℓ' th iteration, we have $Q_{i_j}^{(t')} \geq 2$ for time steps $s_{\ell'} < t' < t$, and afterwards we have $Q_{i_j}^{(t')} \geq 1$. Now let's consider any subsequent iteration of Steps 1 and 2, say, the ℓ'' th iteration. Step 1 may cause $Q_{i_j}^{(t')}$ to decrease by 1 for $s_{\ell''} < t' \leq t''$, for some $t'' > t'$; that is, before Step 1, it must be the case that $Q_{i_j}^{(t')} \geq 2$, for $s_{\ell''} < t' < t''$, and after Step 1 we have $Q_{i_j}^{(t')} \geq 1$, for $s_{\ell''} < t' < t''$. Thus, immediately before Step 2, we have $Q_{i_j}^{(t')} \geq 1$, for $s_{\ell''} < t' < t''$. By Lemma 2, Step 2 will cause $Q_{i_j}^{(t')}$ to increase for each $r_{\ell''} < t' \leq t''$, and thus the values of $Q_{i_j}^{(t')}$ for $r_{\ell''} < t' < t$ will be restored to how they were before the ℓ'' th iteration. Therefore, the net effect of all ℓ iterations of Steps 1 and 2 is that $Q_{i_j}^{(t)}$ may decrease by 1 or stay the same. Since $Q_{i_0}^{(t)}$ does not change, the sum $Q_{i_0}^{(t)} + Q_{i_j}^{(t)}$ either decreases by 1 or stays the same.

By a similar analysis, we can show that if the number of arrivals at queue i_0 before time step t decreases by 1 under the new cycle ordering, then $Q_{i_0}^{(t)}$ decreases by 1 and $Q_{i_j}^{(t)}$ either increases by 1 or remains the same. As before, $Q_{i_0}^{(t)} + Q_{i_j}^{(t)}$ never decreases by more than 1 decreases by 1 or remains the same. \square

Substituting the bound of Lemma 5 into (23) and simplifying, we find that (23) is nonnegative, which completes the proof of Lemma 1. As noted earlier, the rest of the proof of Theorem 1 follows from (13) and (14) applied to RCD.

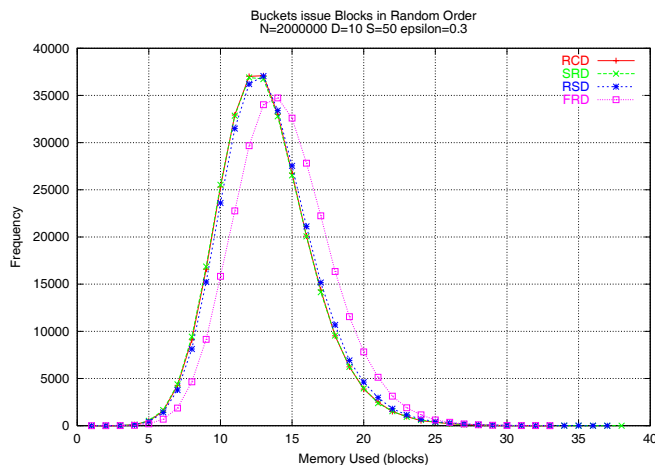


Figure 5: (Randomly permuted input file.) The distribution of memory usage is shown for each variant when $\epsilon = 0.3$. FRD is slightly worse (i.e., tends to need more memory) than the other variants, which have similar performance.

5 Experimental Results

In this section we describe the results of simulation experiments designed to investigate the behavior of the different allocation disciplines outlined in Section 2. As the theoretical analysis provides useful guidance primarily when D is large, we were especially interested in the performance for smaller, practical numbers of disks. We conducted experiments with two types of input files: (1) randomly permuted, and (2) balanced, in that the buckets issue their blocks in turn (i.e., bucket $b_{(i+1) \bmod S}$ issues a block immediately after bucket b_i). We also considered allowing a bucket to issue multiple blocks in its turn, but this becomes “embarrassingly easy” for the cyclic variants we favor in this paper.

We constructed a simulation program in C++ that repeatedly executed the following general procedure:

1. Create a “block” and assign it to one of the S buckets.
2. Assign the block to a queue, depending on the bucket and the queue management discipline in place.

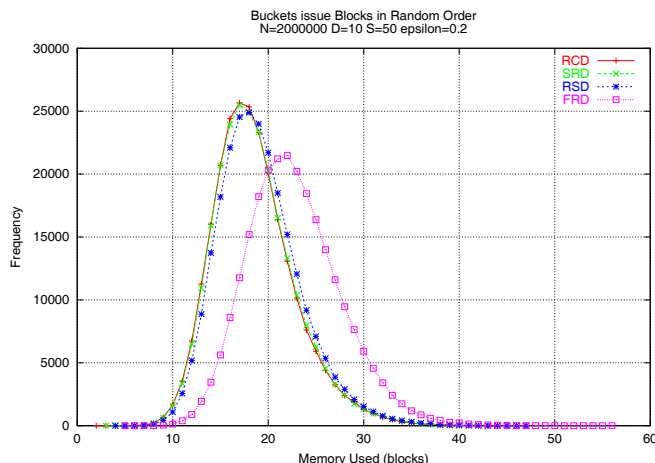


Figure 6: (Randomly permuted input file.) The distribution of memory usage is shown for each variant when $\epsilon = 0.2$. FRD is slightly worse (i.e., tends to need more memory) than the other variants. All the variants tend to use more memory than when $\epsilon = 0.3$ (Figure 5).

3. If $D(1 - \epsilon)$ blocks have been queued since the last write step, do the following:
 - a. Remove the first block, if any, from each queue and “consume it.”
 - b. Measure the total number of blocks in the queues and increment a corresponding frequency histogram counter.

This basic procedure is repeated for each block that is generated. The collected data was graphed using `gnuplot`.

Figures 5–7 show the memory usage frequency distributions for FRD, SRD, RSD, and RCD for case (1), with the ϵ values 0.3, 0.2, and 0.1, 10 queues, 50 buckets, and 2×10^6 total blocks. Also shown (Figures 8–9) are the curves for SRD, RSD and RCD when $\epsilon = 0.01$ and $\epsilon = 0.001$, respectively. We wait for a period of time, 1000 write cycles in this case, for the system to reach a steady state before beginning the measurements. No conditional consumption steps were performed. The curves for SRD and RCD are nearly identical for ϵ values 0.01 to 0.3. RCD is noticeably better than RSD and FRD. FRD’s memory usage is worse in all cases than those

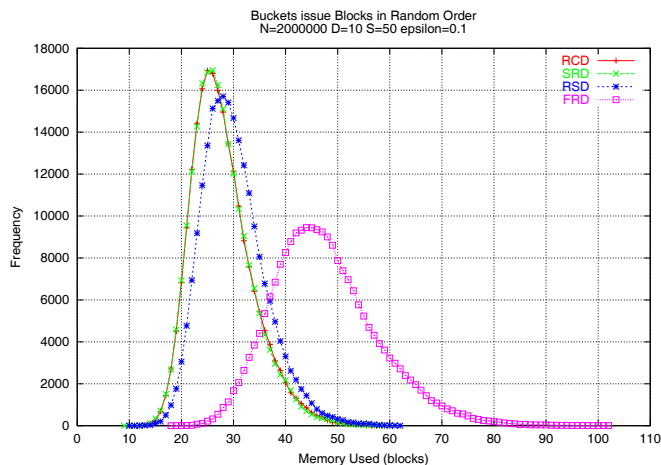


Figure 7: (Randomly permuted input file.) The distribution of memory usage is shown for each variant when $\epsilon = 0.1$. FRD is slightly worse (i.e., tends to need more memory) than the other variants. All the variants tend to use more memory than when ϵ is larger (Figures 5–6).

of SRD and RCD, and it could not be shown for $\epsilon = 0.01$ since its memory consumption went off the scale of the graphs. The graphs indicate that the mean and variance of all of the variants increase with decreasing ϵ , but for FRD more so than for SRD or RCD.

Figures 10–14 show the memory usage frequency distributions for case (2). RCD performs better than the other variants.

6 Applications

While FRD is an elegant method to analyze, it is not ideal for use in sorting because of the large amount of randomness required. In addition, as pointed out in Section 3, it is theoretically nonoptimal for sorting because buckets can be distributed in an unbalanced way across the disks. RCD is provably optimal and seems to be the most efficient striping discipline in practice. Our analysis of RCD has subsequently been applied, using the notion of duality, to optimal design and analysis of merge sort for parallel disks [12].

The FRD-based parallel disk simulation technique of Sanders et al. [16]

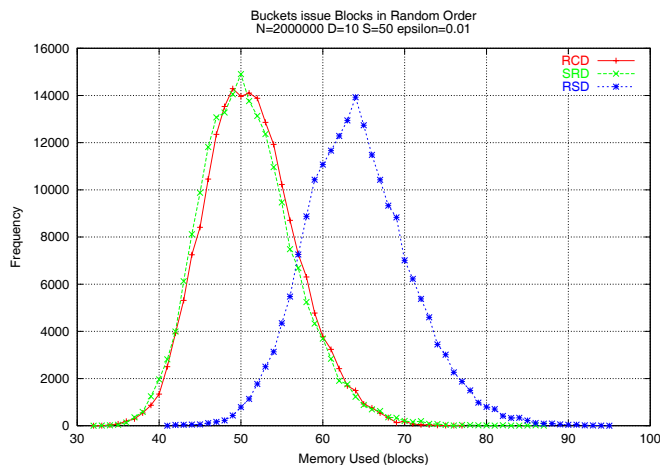


Figure 8: (Randomly permuted input file.) The distribution of memory usage is shown for each variant except FRD when $\epsilon = 0.01$. The consumption of FRD was much larger for this case and goes off the graph to the right. All of the variants tend to use more memory than when ϵ is larger (Figures 5–7).

has some practical constraints. Each block must be duplicated and each copy randomly relocated. Moreover, in order for the analysis to be valid, before each write of a block, all the copies of the block must be re-mapped using a directory structure, and the old copies must be deallocated on disk. This rather severe restriction is made in order to guarantee that any two writes are to independent disks.

A more practical simulation technique was proposed in [16] using the notion of randomized striping (which is the allocation discipline of RSD). We conjecture that a similar alternative, based instead upon a modification of RCD, will also work. Neither method has been analyzed theoretically for the general case of simulating an arbitrary multiheaded disk algorithm by using instead a collection of D separate disks. We conjecture that they do allow optimal simulation.

The RCD technique can be generalized to simulate an important class of multiheaded disk algorithms. This class includes all *multipass algorithms*, by which we mean that the algorithms read and write the data in passes; all of the data elements are read and written once before being read and

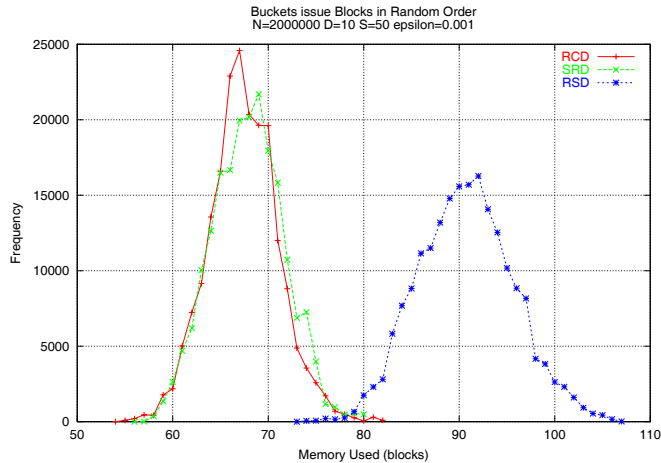


Figure 9: (Randomly permuted input file.) The distribution of memory usage is shown for each variant except FRD when $\epsilon = 0.001$. The consumption of FRD was much larger for this case and goes off the graph to the right. All of the variants tend to use more memory than when ϵ is larger (Figures 5–8).

written a second time, and so on. Duplication is done as before, but the duplicate blocks do not need to be individually remapped to a random disk. Instead, the ordering of the D blocks in each stripe is randomly scrambled (thus allowing the algorithm to take advantage of locality optimizations on the disks for extra speed). The analysis is an extension of the analysis of Section 4. The notion of “bucket” is replaced by the notion of “track”.

Theorem 2 *Multipass algorithms for the multiheaded disk I/O model can be emulated on independent disks with only a constant factor slowdown in terms of I/O cost.*

The multipass property is present in a large number of EM algorithms, including those based upon the data stream model of computation [11].

An even simpler approach with no duplication of blocks is possible for the important subclass of the class of multipass algorithms that are based upon the stream paradigms of distribution and distribution sweeping [17, 18] and, by duality, merging [12, 18]. For these algorithms, the RCD method works almost exactly as described for distribution sort, and the same analysis

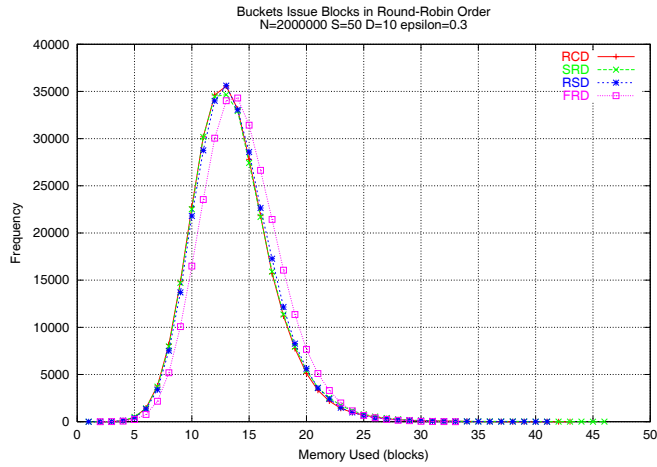


Figure 10: (Round-robin bucket order.) The distribution of memory usage is shown for each variant when $\epsilon = 0.3$.

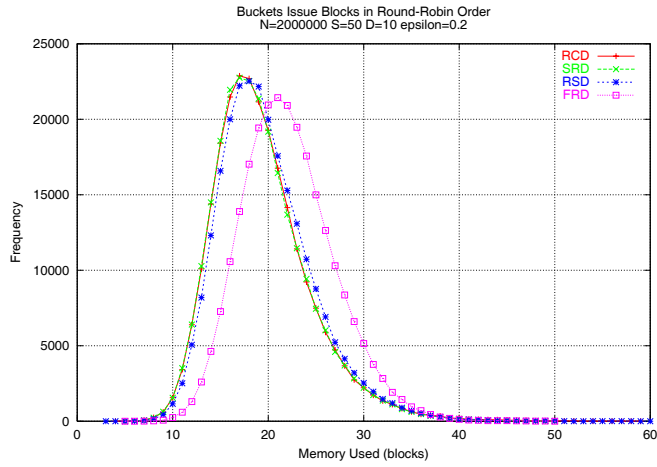


Figure 11: (Round-robin bucket order.) The distribution of memory usage is shown for each variant when $\epsilon = 0.2$. All of the variants tend to use more memory than when ϵ is larger (Figure 10).

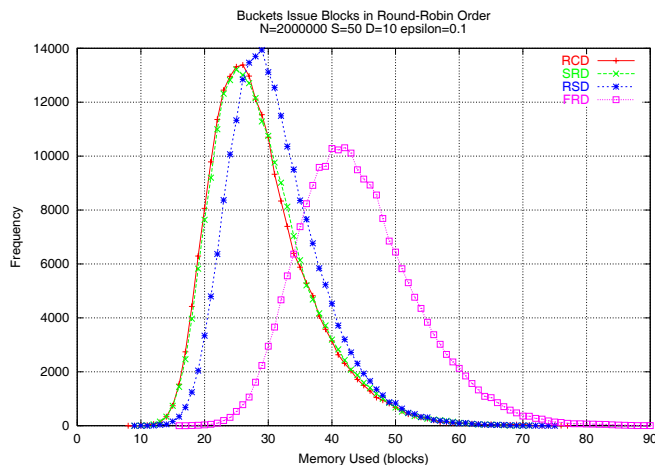


Figure 12: (Round-robin bucket order.) The distribution of memory usage is shown for each variant when $\epsilon = 0.1$. All of the variants tend to use more memory than when ϵ is larger (Figures 10–11).

applies. No duplicate copies of blocks are needed. Relevant algorithms include orthogonal segment line segment intersection, all nearest neighbors of a point set and a variety of other geometric algorithms [10], trapezoidal decomposition, triangulation of a simple polygon, red-blue line intersection in GIS [3], and spatial join [2].

7 Conclusions

In this paper we showed that randomized cycling distribution sort RCD is theoretically optimal for sorting with parallel disks, and it is practical for implementation. We conclude by mentioning some open problems.

We observed that the distribution sort variants SRD and RSD performed similarly to RCD in our experiments. We conjecture that they have similar behavior in general, but proof of their I/O complexity is open.

There is an interesting correspondence between hashing with linear probing [13] and the total queue size of SRD. For the case in which there are $S = D(1 - \epsilon)$ buckets and each of the S buckets issues one block per time step, which seems to be a “hard” instance of SRD, the expected queue size in the limit is precisely the average number of probes for all S possible success-

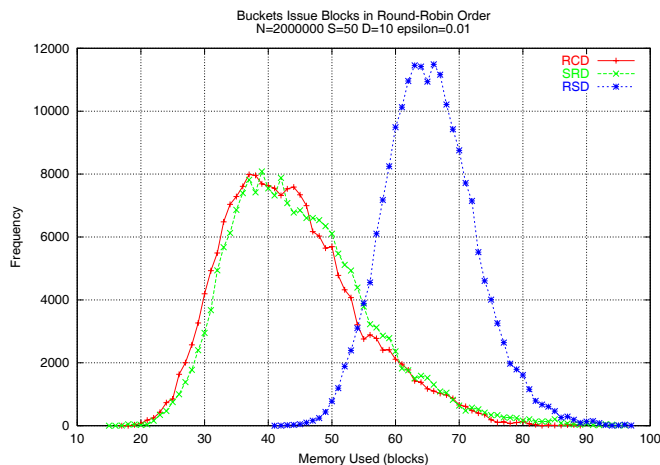


Figure 13: (Round-robin bucket order.) The distribution of memory usage is shown for each variant except FRD when $\epsilon = 0.01$. The consumption of FRD was much larger for this case and goes off the graph to the right. All of the variants tend to use more memory than when ϵ is larger. All of the variants tend to use more memory than when ϵ is larger (Figures 10–12).

ful searches in hashing with linear probing, where the hash table size is D and the number of inserted elements is S . Asymptotically, this quantity is $(S/2)(1 + 1/(1 - S/D)) = \frac{1}{2}D(1 - \epsilon)(1 + 1/\epsilon)$. This correspondence suggests that the I/O performance of SRD may also be optimal up to a constant factor for any fixed $\epsilon > 0$.

In our analysis of RCD the configuration of a single bucket was modified while the configurations of the other buckets were fixed. An interesting question is whether a more general approach, in which the configurations of the other buckets are allowed to vary over all the possible configurations, would work for the analysis of SRD or RSD. We conjecture that the answer is yes.

We also conjecture that a striped variant of RCD admits an optimal general simulation of a multiheaded disk algorithm on the parallel disk model.

Acknowledgments. The authors would like to thank Rakesh Barve and Peter Sanders for interesting discussions about the SRD and RSD algo-

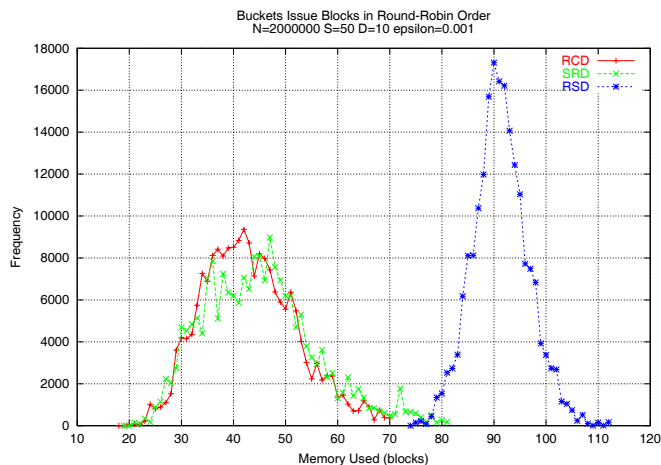


Figure 14: (Round-robin bucket order.) The distribution of memory usage is shown for each variant except FRD when $\epsilon = 0.001$. The consumption of FRD was much larger for this case and goes off the graph to the right. All of the variants tend to use more memory than when ϵ is larger. All of the variants tend to use more memory than when ϵ is larger (Figures 10–13).

rithms.

8 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] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the International Conference on Very Large Databases*, volume 24, pages 570–581, New York, August 1998.
- [3] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, to appear.

- [4] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.
- [5] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized merge-sort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.
- [6] R. D. Barve and J. S. Vitter. A simple and efficient parallel disk merge-sort. *Theory of Computing Systems*, 35(2):189–215, March/April 2002.
- [7] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 9, pages 106–115, June 1997.
- [8] F. Dehne, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proceedings of the International Parallel Processing Symposium*, volume 13, pages 14–20, April 1999.
- [9] D. Dubhasi and D. Ranjan. Balls and bins: A study in negative dependence. *Random Structures & Algorithms*, 13:99–124, 1998.
- [10] 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.
- [11] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report 1998–011, Digital Equipment Corporation Systems Research Center, Palo Alto, 1998.
- [12] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with parallel disks. submitted to journal. An earlier version appeared in *Proceedings of the European Symposium on Algorithms*, Springer-Verlag, Lecture Notes in Computer Science, 2161, August 2001.
- [13] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1998.
- [14] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, volume 5, 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 *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, volume 11, pages 849–858, San Francisco, January 2000.
- [17] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, volume 7, pages 74–77, San Antonio, TX, October 1995. IEEE Computer Society Press.
- [18] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.
- [19] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.
- [20] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 9, pages 431–524. North-Holland, 1990.
- [21] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.