# Distribution Sort with Randomized Cycling

*Jeffrey Scott Vitter*[*] *David A. Hutchinson*[*]

## Abstract

Parallel independent disks can enhance the performance of external memory (EM) algorithms, but the programming task is often difficult. In this paper we develop randomized variants of distribution sort for use with parallel independent disks. We propose a simple variant called *randomized cycling distribution sort* (RCD) and prove that it has optimal expected I/O complexity. The analysis uses a novel reduction to a model with significantly fewer probabilistic interdependencies. Experimental evidence is provided to support its practicality. Other simple variants are also examined experimentally and appear to offer similar advantages to RCD. Based upon ideas in RCD we propose general techniques that transparently simulate algorithms developed for the unrealistic multihead disk model so that they can be run on the realistic parallel disk model. The simulation is optimal for two important classes of algorithms: the class of *multipass* 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 paradigm of EM computation.

## 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 [17]. 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 [17].

Designing algorithms for independent parallel disks has turned out to be ad hoc and relatively difficult [19, 13, 14, 5, 6, 7, 8, 3, 4, 17], 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, 12] and Barve and Vitter [6], but with provably optimal performance for all parameter settings. 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, which are stored on the disks. 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.

[*]Department of Computer Science, Duke University, Durham, NC 27708–0129 {*jsv,hutchins*}*@cs.duke.edu*.

Barve et al. [5, 12] 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 [18] 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 [19, 17] requires that each of the $D$ blocks accessed in an I/O must reside on a separate disk. Recently, Sanders et al. [15] 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 previous parallel-disk sorting algorithms, such as those by Vitter and Shriver [19], Nodine and Vitter [13, 14], 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, and apply the analysis of Sanders et al. [15] to its writing component. 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 5 we conjecture, based upon experimental evidence and an interesting relation to hashing with linear probing, that another simple variant, called *simple randomized distribution sort* (SRD) (akin to SRM), is also optimal.

In Section 6 we report on simulation experiments that confirm the theoretical analysis of RCD and demonstrate the practicality of the methods. We are separately pursuing an implementation as part of a parallel disk environment and plan to report our findings in a subsequent paper.

In Section 7 we discuss several interesting applications. We develop a general technique for simulating 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. [15], 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. 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. 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.

**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 [15] 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 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 $D(1-\epsilon)$.

In practice, we would normally wait until new block arrivals appear before attempting to consume from the queues, but the above order of events, in which consumption is attempted before arrivals, is easier to analyze and results in a slightly more conservative analysis.

So far we haven't discussed what happens if the buffer pool of size $W$ overflows. In that case, we insert the following event number 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 $\langle i_0, i_1, \ldots, i_{D-1}\rangle$ 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 $\{0, 1, \ldots, D-1\}$. For SRD, we have $i_{j+1} = i_j + 1 \bmod D$, where $i_0$ can be any value $0 \le i_0 < D$.

**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 $\langle i_0, i_1, \ldots, i_{D-1}\rangle$. 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 $\mathrm{E}(n^{(t)}) = 1 + e^{-\Omega(D)}$.*

We conjecture that similar bounds hold for the write I/Os in SRD and RSD, as suggested by the experiments in Section 6. In RCD, SRD, and RSD, the blocks of each bucket 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 essentially given by Sanders et al. [15]); we summarize it below. We then demonstrate that the reading component of FRD is not theoretically optimal because of possible global imbalance.

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, $\sum_b I^{(t)}(b) = D(1-\epsilon)$ for all time steps. Let the notation $\widehat{Q}^{(t)}$ and $\widehat{Q}_i^{(t)}$ 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[1]

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

The probability generating function $\widehat{\mathcal{Q}}_i^{(t)}(z) = \sum_{t\ge 1} \mathrm{Prob}\{\widehat{Q}_i^{(t)} = k\} z^k$ has the following closed form in the steady state $t = \infty$:

$$\widehat{\mathcal{Q}}_i(z) = \frac{(1-z)\epsilon}{1 - z/(\frac{z}{d} + 1 - \frac{1}{d})^{(1-\epsilon)D}}. \tag{2}$$

With the appropriate setting of the buffer size $W = (\ln 2 + \delta)D/\epsilon$, the probability that the buffer overflows is exponentially small; the Chernoff-like tail bound of buffer overflow is derived by starting with Markov's inequality applied to $e^{s\widehat{Q}^{(t)}}$:

$$\begin{aligned} \mathrm{Prob}\{\widehat{Q}^{(t)} > W\} &= \mathrm{Prob}\{e^{s\widehat{Q}^{(t)}} > e^{sW}\} \\ &< e^{-sW}\mathrm{E}(e^{s\widehat{Q}^{(t)}}). \end{aligned} \tag{3}$$

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

$$\mathrm{E}(e^{s\widehat{Q}^{(t)}}) = \mathrm{E}(e^{\sum_{0\le i<D} s\widehat{Q}_i^{(t)}}) = \mathrm{E}\left(\prod_{0\le i<D} e^{s\widehat{Q}_i^{(t)}}\right). \tag{4}$$

If the random variables $\langle \widehat{Q}_i^{(t)}\rangle_{0\le i<D}$ were independent, the expected value operator could be moved inside the product and thus the right-hand-side of (4) replaced by

$$\prod_{0\le i<D} \mathrm{E}(e^{s\widehat{Q}_i^{(t)}}) = \left(\mathrm{E}(e^{s\widehat{Q}_1^{(t)}})\right)^D. \tag{5}$$

The random variables $\langle \widehat{Q}_i^{(t)}\rangle_{0\le i<D}$ are not independent, but fortunately they are negatively associated[2], which allows the right-hand-side of (4) to be bounded by (5).

Finally, we use the fact that $\mathrm{E}(e^{s\widehat{Q}_i^{(t)}}) < \mathrm{E}(e^{s\widehat{Q}_i^{(\infty)}}) = \widehat{\mathcal{Q}}_i(e^s)$ (which by l'Hôpital's rule can be bounded by 2 when $s = \epsilon$), and the following bound emerges:

$$\mathrm{Prob}\{\widehat{Q}^{(t)} > W\} < e^{-(\epsilon W - \ln 2)D} = e^{-\delta D}. \tag{6}$$

In other words, a conditional consumption step is only executed with exponentially small probability. 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

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

[2] A sequence $\langle X_1, \ldots, X_n\rangle$ of discrete RVs are *negatively associated* [9] if for any nondecreasing function $f$ and for any disjoint subsets $I$ and $J$ of $[1, n]$, $\mathrm{E}\big(f(X_i,\ i \in I)g(X_j,\ j \in J)\big) \le \mathrm{E}\big(f(X_i,\ i \in I)\big)\,\mathrm{E}\big(g(X_j,\ j \in J)\big)$. Intuitively, if $X_i$ is large, then $X_j$ tends to be small.

the total expected number $\mathrm{E}(n^{(t)})$ of parallel write operations made by FRD at step $t$ is bounded by

$$\begin{aligned} & 1 + \big(D(1-\epsilon) + W\big)\,\mathrm{Prob}\{\widehat{Q}^{(t)} > W\} \\ & \quad < \quad 1 + O(D)\,e^{-\delta D} = 1 + e^{-\Omega(D)}, \end{aligned} \tag{7}$$

which is the bound used in Theorem 1 for RCD.

The reading component of FRD can be nonoptimal by a $\ln D / \ln\ln D$ factor when $D$ is large because of unbalanced I/O operations [18]. 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 is 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 [17].

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 [19], where data are written to the disks in stripes and the buckets tend to be more evenly distributed.

## 4 Analysis of 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 of RCD. This bound is the same one given in (3)–(5) for the substantially more independent case FRD. After we derive the Chernoff bound, the rest of the proof of Theorem 1 proceeds as in (6) and (7) for FRD.

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

**Definition 5** A bucket $b$ is a *singleton* bucket if it issues a total of one block over all time steps; that is, $\sum_{t'} I^{(t')}(b) = 1$.

We reduce an instance of RCD to an instance of FRD via the following series of transformations:

**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 $r$
  **enddo**
**enddo**

As mentioned in (4) and (5), the total queue size $\widehat{Q}^{(t)}$ for FRD (the situation in which all buckets are singletons) satisfies $\mathrm{E}(e^{s\widehat{Q}^{(t)}}) \le \prod_{0\le i<D} \mathrm{E}(e^{s\widehat{Q}_i^{(t)}}) = \big(\mathrm{E}(e^{s\widehat{Q}_i^{(t)}})\big)^D$. The right-hand-side is the corresponding quantity for the case in which the queue sizes are completely independent. In this section we will show for each transformation step that $\mathrm{E}(e^{sQ^{(t)}})$ never decreases as a result of the transformation. Hence we will have

$$\mathrm{E}(e^{sQ^{(t)}}) \le \prod_{0\le i<D} \mathrm{E}(e^{s\widehat{Q}_i^{(t)}}) = \big(\mathrm{E}(e^{s\widehat{Q}_i^{(t)}})\big)^D, \tag{8}$$

which establishes a Chernoff-type bound for $Q^{(t)}$ that is bounded by the Chernoff-type bound for FRD. The remainder of the proof of Theorem 1 follows from (6) and (7) applied to RCD.

### 4.1 Main Lemmas

Let us define $f(x) = e^{sx}$, so that the term we are interested in (namely, the left-hand-side of (8)) is $\mathrm{E}\big(f(Q^{(t)})\big)$. Our main lemma below, which we prove in the next section, shows that the transformations have the desired effect.

**Lemma 1** *Each bucket transformation step as described above, in which one block at time step $r$ is removed from the bucket and a new bucket is created with one block at the same time step, causes the quantity $\mathrm{E}\big(f(Q^{(t)})\big)$ to increase or stay the same.*

A key concept in proving Lemma 1 is the notion of critical starting points.

**Definition 6** Consider an arbitrary bucket $b$ whose first block(s) appear at time step $r < t$, and consider any fixed configurations for the other buckets. Consider the following two scenarios:

1. Queue $i$ is the starting point for bucket $b$ (i.e., the first block 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 any of the other blocks). Let $Q'^{(t)}$ be the resulting total size of the queues at time step $t$.

| $t'$ | $r$ | $r+1$ | $\ldots$ | | | $t-1$ | $t$ |
|---|---|---|---|---|---|---|---|
| $Q_i^{(t')}$ | $\geq 2$ | $\geq 2$ | $\ldots$ | $\geq 2$ | $\geq 2$ | $\geq 2$ | $Q_i^{(t)}$ |
| Item Arrivals | $\bullet$ | | $\ldots$ | $\bullet$ | $\bullet$ | | |

Figure 1: A Critical Queue. The size of queue $i$ is shown at each time step $t' \in [r, t]$. An arrival at queue $i$ during a time step is indicated by $\bullet$. Since the size of the $i$th queue is at least 2 for $r \leq t' < t$, it will remain at least 1 even without the arrival at time step $r$, and a block will continue to be consumed at each time step. Hence, the final queue size will contain one fewer block than before.

We say that queue $i$ is a *critical starting point* for bucket $b$ with respect to time step $t$ if

$$Q'^{(t)} = Q^{(t)} - 1 \tag{9}$$

(or equivalently if $Q'^{(t)}_i = Q^{(t)}_i - 1$, since the other queues are not affected).

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. The following lemmas are important for reasoning about the effect of block arrivals upon the sizes of queues. See Figure 1. The proofs (by induction) are omitted for brevity.

**Lemma 2** *The following conditions are equivalent:*

1. *Consider an arbitrary bucket $b$ with starting point $i$ whose first block(s) appear at time step $r < t$. The starting point $i$ is critical for bucket $b$ with respect to time step $t$.*
2. *$Q_i^{(t')} \geq 2$, for all $r < t' < t$.*

**Lemma 3** *The following conditions are equivalent:*

1. *$Q_i^{(t')} \geq 1$, 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 $r < t' \leq t$.*

For critical buckets the following lemma shows that the value of $Q_i^{(t)}$ is maximized when the starting point of the bucket is queue $i$.

**Lemma 4** *If an block in a queue is moved from time step $r$ to time step $r'$, where $r < r' < t$, then the size of the queue increases by 1 or stays the same.*

### 4.2 Proof of Lemma 1

To prove Lemma 1, let us consider the effect of a transformation step applied to bucket $b$ at time step $r$. We assume that $b$ does not issue any blocks before time $r$. We use $Q^{(t)}$ to represent the sum of queues at time $t$, and we use $Q''^{(t)}$ to represent the sum of queues at time $t$ after bucket $b$ has been transformed. We let the configurations of the other buckets be arbitrary and fixed. Let the permutation cycling order for bucket $b$ be $\langle i_0, i_1, \ldots, i_{D-1}\rangle$. 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 $\langle i_1, \ldots, i_{D-1}, i_0\rangle$. 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 block that was removed.

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

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

where $Q'^{(t)}$ is defined as in Definition 6 of criticality. If bucket $b$'s starting point $i_0$ is critical, 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 $t$. If the new bucket issues its block to one of the $c$ critical starting points, which happens with probability $c/D$, then by Lemmas 2 and 3 the total queue size will be incremented and $Q''^{(t)} = Q'^{(t)} + 1 = Q^{(t)}$. 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 $Q''^{(t)} \geq Q'^{(t)} = Q^{(t)} - 1$.

In the following we will use the notation CR and NCR to represent the events "the starting point of bucket $b$ is critical" and "the starting point of bucket $b$ is non-critical", respectively. The above relations give us the following lower bound on the conditional expectation, given that the starting point of bucket $b$ is critical:

$$\begin{aligned}\mathrm{E}\big(f(Q''^{(t)}) \,\big|\, \mathrm{CR}\big) \;\geq\; &\Big(1 - \frac{c}{D}\Big)\,\mathrm{E}\big(f(Q^{(t)} - 1) \,\big|\, \mathrm{CR}\big) \\ &+ \frac{c}{D}\,\mathrm{E}\big(f(Q^{(t)}) \,\big|\, \mathrm{CR}\big),\end{aligned} \tag{10}$$

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

$$\begin{aligned}&\mathrm{E}\big(f(Q''^{(t)}) \,\big|\, \mathrm{NCR}\big) \\ &\quad\geq \Big(1 - \frac{c}{D}\Big)\,\mathrm{E}\big(f(Q^{(t)}) \,\big|\, \mathrm{NCR}\big) \\ &\qquad + \frac{c}{D}\,\mathrm{E}\big(f(Q^{(t)} + 1) \,\big|\, \mathrm{NCR}\big).\end{aligned} \tag{11}$$

Using the identities $\mathrm{E}\big(f(X+1)\big) = f(1)\,\mathrm{E}\big(f(X)\big)$ and $\mathrm{E}\big(f(X-1)\big) = \mathrm{E}\big(f(X)\big)/f(1)$, we can rewrite (10)

and (11) as

$$\begin{aligned}
&\mathrm{E}\big(\, f(Q''^{(t)}) \;\big|\; \mathrm{CR}\,\big) \\
&\qquad \geq \quad \left(\left(1-\frac{c}{D}\right)\frac{1}{f(1)}+\frac{c}{D}\right)\mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{CR}\,\big); \\
&\mathrm{E}\big(\, f(Q''^{(t)}) \;\big|\; \mathrm{NCR}\,\big) \\
&\qquad \geq \quad \left(\left(1-\frac{c}{D}\right)+\frac{c}{D}\,f(1)\right)\mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{NCR}\,\big).
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{E}\big(f(Q^{(t)})\big) \quad &= \quad \frac{c}{D}\,\mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{CR}\,\big) + \left(1-\frac{c}{D}\right) \\
&\qquad \times \mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{NCR}\,\big). \qquad (12)
\end{aligned}$$

After the transformation we get

$$\begin{aligned}
\mathrm{E}\big(f(Q''^{(t)})\big) \quad &\geq \quad \frac{c}{D}\left(\left(1-\frac{c}{D}\right)\frac{1}{f(1)}+\frac{c}{D}\right) \\
&\qquad \times \mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{CR}\,\big) \\
&\qquad + \left(1-\frac{c}{D}\right)\left(\left(1-\frac{c}{D}\right)+\frac{c}{D}\,f(1)\right) \\
&\qquad \times \mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{NCR}\,\big). \qquad (13)
\end{aligned}$$

Combining (12) and (13) we get

$$\begin{aligned}
&\mathrm{E}\big(f(Q''^{(t)})\big) - \mathrm{E}\big(f(Q^{(t)})\big) \\
&\qquad \geq \quad \left(\frac{c}{D}\left(\frac{1}{f(1)}-1\right)+\frac{c^2}{D^2}\left(1-\frac{1}{f(1)}\right)\right) \\
&\qquad\qquad \times \mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{CR}\,\big) \\
&\qquad\qquad + \left(1-\frac{c}{D}\right)\frac{c}{D}\big(f(1)-1\big) \\
&\qquad\qquad \times \mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathrm{NCR}\,\big). \qquad (14)
\end{aligned}$$

The following important lemma is needed to show that (14) is nonnegative.

**Lemma 5** *We have*

$$\mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathit{NCR}\,\big) \geq \frac{1}{f(1)}\,\mathrm{E}\big(\, f(Q^{(t)}) \;\big|\; \mathit{CR}\,\big).$$

*Proof Sketch*: We prove the lemma by showing a correspondence between the cycle orders of bucket $b$ for which the starting point is critical and the cycle orders of $b$ for which the starting point is non-critical. Consider the cycle order sequence $\langle i_0, i_1, \ldots, i_{D-1}\rangle$ where $i_0$ is critical and $i_j$ is non-critical, for some $0 < j < D$. If we swap the positions of $i_0$ and $i_j$ to get the cycle order $\langle i_j, i_1, \ldots, i_{j-1}, i_0, i_{j+1}, \ldots\ i_{D-1}\rangle$, then the only queues whose sizes are affected are $i_0$ and $i_j$.

Suppose that the number of arrivals at queue $i_0$ before time step $t$ under the new cycle ordering is the same as before, then by Lemmas 2–4 and further analysis it follows that $Q_{i_0}^{(t)}$ remains the same and that $Q_{i_j}^{(t)}$ either decreases by 1 or remains the same. If instead the number of arrivals at queue $i_0$ before time step $t$ decreases by 1 under the new cycle ordering, then by Lemmas 2–4 and further analysis it follows that $Q_{i_0}^{(t)}$ decreases by 1 and that $Q_{i_j}^{(t)}$ either increases by 1 or remains the same. In either case, $Q_{i_0}^{(t)} + Q_{i_j}^{(t)}$ never decreases by more than 1, and the lemma follows immediately. □

Substituting the bound of Lemma 5 into (14) shows that (14) is nonnegative, which completes the proof of Lemma 1. As noted earlier, the rest of the proof of Theorem 1 follows from (6) and (7) applied to RCD.

In our analysis above, 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. Evidence in favor of SRD is given in the next section.

## 5 SRD: Linear Probing Analogy

There is an interesting correspondence between hashing with linear probing [12] 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 successful 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 $(S/2)\big(1+1/(1-S/D)\big) = \frac{1}{2}D(1-\epsilon)(1+1/\epsilon)$. For lack of space, we do not elaborate on the correspondence, but it suggests that the I/O performance of SRD may also be optimal up to a constant factor for any fixed $\epsilon > 0$.

## 6 Experimental Results

In this section we describe experiments designed to explore the memory usage of FRD, SRD, RSD and RCD. As the theoretical analysis provides guidance primarily for large $D$, we were especially interested in the performance for smaller, practical numbers of disks. We simulated a stream of blocks arriving at the $D$ disk queues; each block was labeled with a bucket index. Labeling was done via: (1) random assignment of buckets to blocks, or (2) balanced assignment (bucket $b_{(i+1) \bmod S}$ issues a block immediately after bucket $b_i$). A write cycle was simulated every $D$ block arrivals. After an initial startup period (1000 write cycles in this case) to permit the system to reach a steady state, the number of queued blocks was recorded following each write cycle.

Figures 2–4 show the memory usage frequency distributions for FRD, SRD, RSD, and RCD for case (1),

with the $\epsilon$ values 0.1 and 0.3, 10 queues, 50 buckets, and $2 \times 10^6$ total blocks. Also shown are the curves for SRD and RCD when $\epsilon = 0$. No conditional consumption steps were performed. The curves for SRD and RCD completely overlap for $\epsilon$ values 0.3 and 0.1, and they differ only when $\epsilon$ becomes extremely small or zero, with RCD slightly better than SRD. RCD is noticeably better than RSD and FRD. FRD's memory usage is worse in all cases than those of SRD and RCD, and it could not be shown for $\epsilon = 0$ since its memory consumption increased without apparent bound. The graphs indicate that the mean and variance of all of the variants increase with decreasing $\epsilon$, but FRD more so than SRD or RCD. Figures 5–7 show the memory usage frequency distributions for case (2). These seem to be harder cases for SRD but RCD continues to perform better than the other variants.

## 7 Applications

The elegant simulation technique of Sanders et al. [15] can simulate an arbitrary multiheaded disk algorithm by using instead a collection of $D$ separate disks, but it is somewhat cumbersome for practical use. Each block must be duplicated and each copy randomly relocated. In order for the analysis to be valid, before each write of a block, all the copies of the block must be re-mapped and the old copies deallocated on disk. This rather severe assumption is made in order to guarantee that any two writes are to independent disks.

A more practical simulation technique was proposed in [15] using the notion of randomized striping (which is the allocation discipline of RSD). It has not been analyzed theoretically for the general case, and we conjecture that it and related techniques do allow optimal general 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 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". Details are deleted for brevity; they will be included in the full paper.

**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, distribution and distribution sweeping paradigms [16, 17]. For these algorithms, the RCD method works almost exactly as described for distribution sort, and the same analysis 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].

## 8 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. A detailed implementation is being pursued as part of a parallel disk environment we are developing.

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.

**Acknowledgments.** We thank Rakesh Barve and Peter Sanders for interesting discussions about the SRD and RSD algorithms. This research was supported in part by by Army Research Office MURI grant DAAH04–96–1–0013 and National Science Foundation grants CCR–9877133 and EIA–9870734. The second author was supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and National Science Foundation grant CCR–0082986.

## 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 24th International Conference on Very Large Databases*, 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. Special issue on cartography and geographic information systems. An earlier version appeared in *Proceedings of the Third European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, 295–310, Springer-Verlag, September 1995.

[4] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. of the IEEE*

*Symposium on Foundations of Computer Science*, pages 560–569, Burlington, VT, October 1996.

[5] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, 1997.

[6] 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.

[7] 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.

[8] 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.

[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 *IEEE Foundations of Computer Science*, pages 714–723, Palo Alto, CA, November 1993.

[11] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical report, Digital Equipment Corp. Systems Research Centre, Palo Alto, CA, 1998.

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

[13] 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.

[14] 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.

[15] P. Sanders, S. Egner, and J. Korst. Fast concurrent access to parallel disks. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 849–858, San Francisco, January 2000.

[16] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, pages 74–77, San Antonio, TX, October 1995. IEEE Computer Society Press.

[17] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. American Mathematical Society Press, Providence, RI, 1999. An expanded version is available via the author's web page `http://www.cs.duke.edu/~jsv/`.

[18] 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.

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

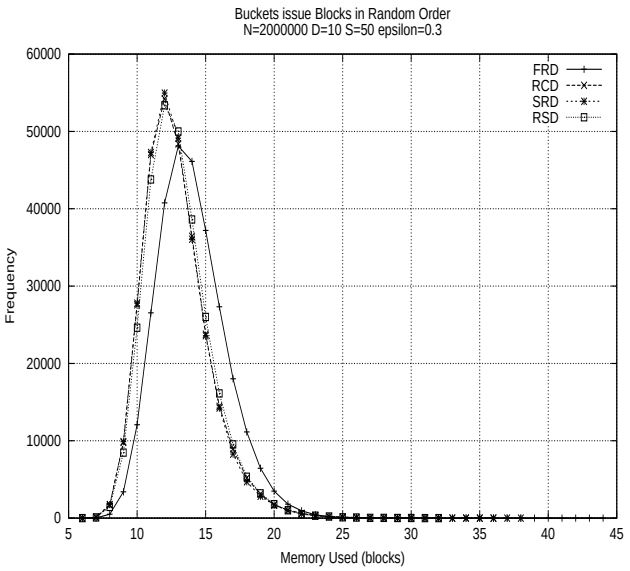


Figure 2: (Random bucket order) The distribution of memory usage is shown for each variant when $\epsilon = 0.3$. FRD tends to need more memory than the others.

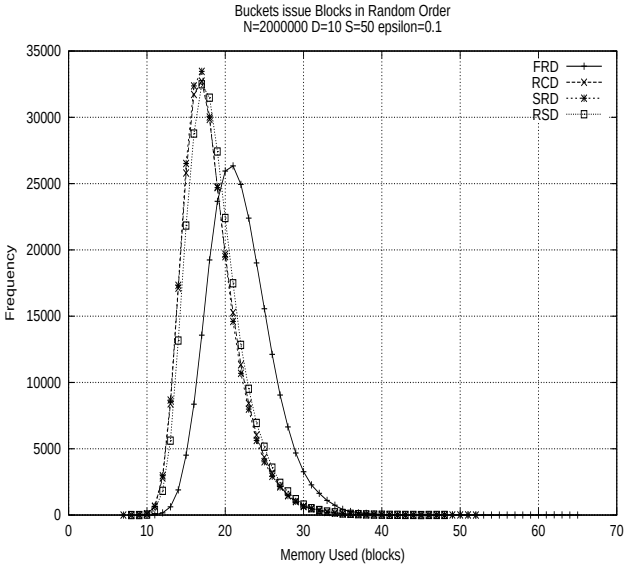


Figure 3: (Random bucket order) The distribution of memory usage is shown for each variant when $\epsilon = 0.1$. FRD tends to need the most memory. All of the variants tend to use more memory than when $\epsilon = 0.3$.

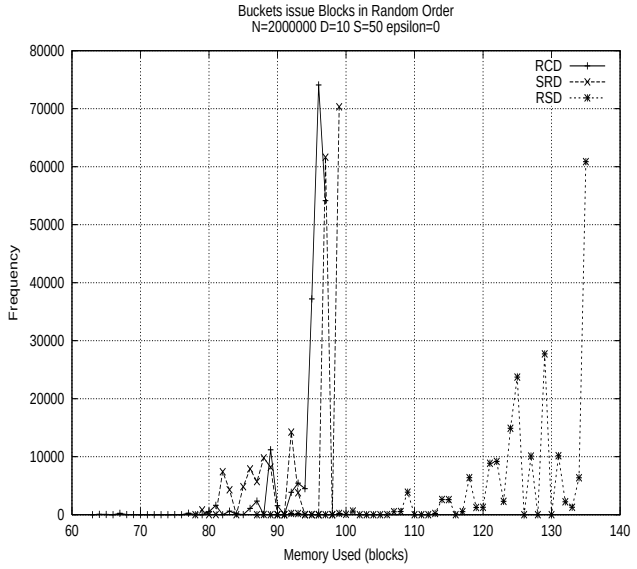


Figure 4: (Random bucket order) The distribution of memory usage is shown for each variant except FRD when $\epsilon = 0$. The consumption of FRD was much larger and goes off the graph to the right. All of the variants tend to use more memory than when $\epsilon = 0.1$.

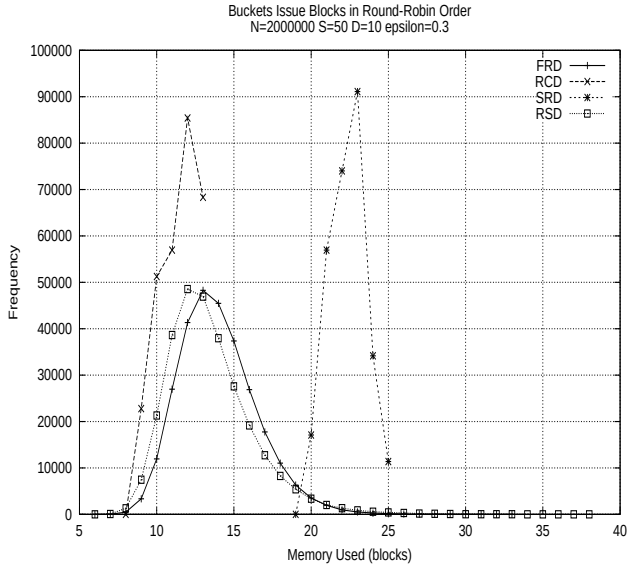


Figure 5: (Round-robin bucket order) The distribution of memory usage is shown for each variant when $\epsilon = 0.3$. This seems to be a more difficult case for SRD than when buckets issue blocks in random order (Figure 2).

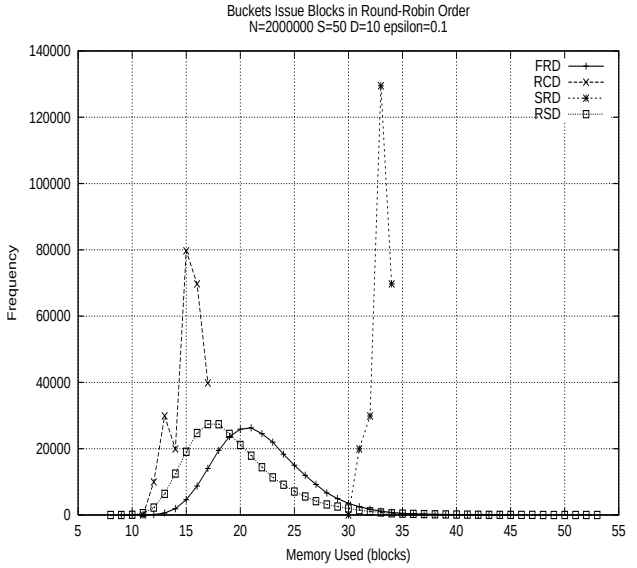


Figure 6: (Round-robin bucket order) The distribution of memory usage is shown for each variant when $\epsilon = 0.1$. Again, this seems to be a more difficult case for SRD than when buckets issue blocks in random order (Figure 3).

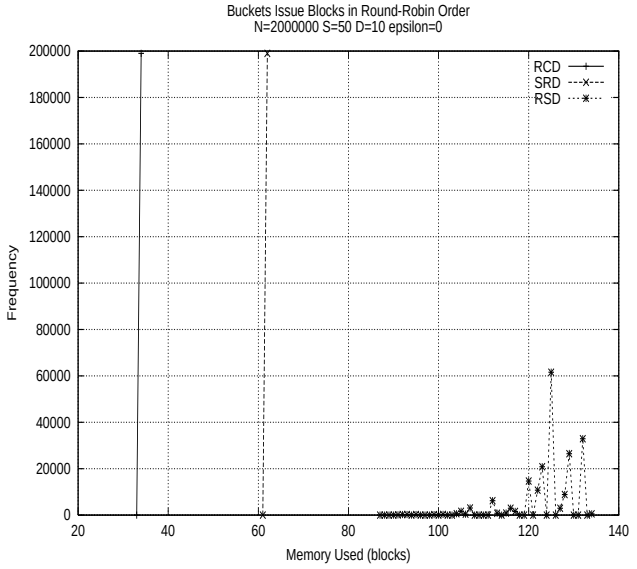


Figure 7: (Round-robin bucket order) The distribution of memory usage is shown for each variant except FRD when $\epsilon = 0$. The consumption of FRD was much larger and goes off the graph to the right. All of the variants tend to use more memory than when $\epsilon = 0.1$. RCD and SRD used a fixed amount of memory after steady state was reached.