CS–1996–15

Simple Randomized Mergesort on Parallel Disks[1]

Rakesh Barve Edward F. Grove
Jeffrey Scott Vitter

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

October 1996

Simple Randomized Mergesort on Parallel Disks[1]

Rakesh D. Barve [2]
Dept. of Computer Science
Duke University
Durham, NC 27708–0129
rbarve@cs.duke.edu

Edward F. Grove[3]
Dept. of Computer Science
Duke University
Durham, NC 27708–0129
efg@cs.duke.edu

Jeffrey Scott Vitter[4]
Dept. of Computer Science
Duke University
Durham, NC 27708–0129
jsv@cs.duke.edu

[1]An earlier version of the paper appeared in [BGV96].

[2]Support was provided in part by an IBM Graduate Fellowship.

[3]Support was provided in part by the U.S. Army Research Office under grant DAAH04–93–G–0076 while the author was at Duke University.

[4]Support was provided in part by the National Science Foundation under grant CCR–9522047 and by the U.S. Army Research Office under grant DAAH04–96–1–0013.

Abstract

We consider the problem of sorting a file of N records on the D-disk model of parallel I/O in which there are two sources of parallelism. Records are transferred to and from disk concurrently in blocks of B contiguous records. In each I/O operation, up to one block can be transferred to or from each of the D disks in parallel. We propose a simple, efficient, randomized mergesort algorithm called SRM that uses a forecast-and-flush approach to overcome the inherent difficulties of simple merging on parallel disks. SRM exhibits a limited use of randomization and also has a useful deterministic version. Generalizing the technique of forecasting, our algorithm is able to read in, at any time, the "right" block from any disk, and using the technique of flushing, our algorithm evicts, without any I/O overhead, just the "right" blocks from memory to make space for new ones to be read in. The disk layout of SRM is such that it enjoys perfect write parallelism, avoiding fundamental inefficiencies of previous mergesort algorithms. By analysis of generalized maximum occupancy problems we are able to derive an analytical upper bound on SRM's expected overhead valid for arbitrary inputs.

The upper bound derived on expected I/O performance of SRM indicates that SRM is provably better than disk-striped mergesort (DSM) for realistic parameter values D, M, and B. Average-case simulations show further improvement on the analytical upper bound. Unlike previously proposed optimal sorting algorithms, SRM outperforms DSM even when the number D of parallel disks is small.

Key Words : I/O, External Memory, Disk, Parallel Disks, Sorting, Mergesort, Merging, Forecasting, Maximum Occupancy, Disk Striping.

1 Introduction

The classical problem of sorting and related processing is reported to consume roughly 20 percent of computing resources in large-scale installations [Knu73, LV85]. In the light of the rapidly increasing gap between processor speeds and disk memory access times, popularly referred to as the *I/O bottleneck*, the specific problem of *external memory sorting* assumes particular importance. In external sorting, the records to be sorted are simply too many to fit in internal memory and so have to be stored on disk, thus necessitating I/O as a fundamental, frequently used operation during sorting.

One way to alleviate the effects of the I/O bottleneck is to use parallel disk systems [HGK+94, PGK88, Uni89, GS84, Mag87]. Aggarwal and Vitter [AV88], generalizing initial work done by Floyd [Flo72] and Hong and Kung [HK81], laid the foundation for I/O algorithms by studying the I/O complexity of sorting and related problems. The model they studied [AV88] considers an internal memory of size M and I/O reads or writes that each result in a transfer of D blocks, where each block is comprised of B contiguous records, from or to disks. Subsequently, Vitter and Shriver [VS94] considered a realistic *D-disk* two-level memory model in which secondary memory is partitioned into D physically distinct and independent disk drives or read-write heads that can simultaneously transmit a block of data, with the requirement that $M \geq 2DB$.

In the D-disk two-level memory hierarchy [VS94], there are two sources of parallelism. First, as in traditional I/O systems, records are transferred concurrently in blocks of B contiguous records. Secondly, in a single I/O operation, each of the D disks can simultaneously transfer one (but only one) block of B records, so that each I/O operation can potentially transfer DB records in parallel.

To be more precise, D is the number of blocks that can be transmitted in parallel at the speed at which data comes off or goes onto the magnetic disk media. The parameter D may thus be smaller than the actual number of disks if the channel bandwidth from the disks is not sufficient for each disk to transmit or receive a block simultaneously. It then might be useful to consider two disk parameters D and D', where D is the channel bandwidth in terms of the number of blocks coming off or going onto disk that can be transferred simultaneously, and D' is the number of disks sharing the bandwidth. As long as at least D of the D' disks have a block to transmit, the I/O channel can remain busy. Hybrid models with multiple channels and several disks per channel are also possible. In this paper we adopt the simpler and more restrictive model in which $D = D'$ and develop algorithms that are near-optimal even for this more restrictive model. A more detailed discussion of disk models and characteristics appears in [RW94].

The problem of external sorting in the D-disk model has been extensively studied. Algorithms have been developed that use an asymptotically optimal $\Theta(\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)})$ number of I/O operations[1], as well as doing an optimal amount of work in internal memory. The previously developed sorting algorithms for the D-disk model have larger than desired constant factors, and some are complicated to implement. As a result, the simple technique of *mergesort with disk striping* (DSM), which can be asymptotically sub-optimal in terms of number of I/Os by a multiplicative factor of $\ln(M/B)$, is commonly used in practice for

[1]Throughout this paper we use the standard asymptotic notation $f(n) = O(g(n))$ to mean that there are constants C, $n_0 > 0$ such that $|f(n)| \leq C|g(n)|$ for all $n \geq n_0$. We say that $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$. We write $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. We say that $f(n) = o(g(n))$ if $f(n)/g(n) \to 0$ as $n \to \infty$. The above asymptotic notations can be applied in a generalized manner as well. For example, we write $f(n) = g(n)+O(h(n))$ if $f(n)-g(n) = O(h(n))$, and we say that $f(n) \leq g(n)+O(h(n))$ if there is some $j(n)$ such that $f(n) \leq j(n)$ and $j(n) = g(n) + O(h(n))$.

external sorting on parallel disks[NV90, NV93]. In DSM, the disks are coordinated so that in each parallel read and write, the locations of the blocks accessed on each disk are the same, which has the logical effect of sorting with $D' = 1$ disk and block size $B' = DB$. In each merge pass, $R = \Theta(M/DB)$ runs are merged together at a time, except possibly in the last pass. The number of passes required is thus $\ln(N/M)/\ln(M/DB)$, with an additional pass for initial run formation. The resulting I/O complexity of DSM is $\Theta(\frac{N}{DB}(1 + \frac{\log(N/M)}{\log(M/DB)}))$. The advantage of DSM is its simplicity, which results in very small constant factors, thus making it attractive for efficient implementation on existing parallel disk systems. But the disadvantage of DSM is that it becomes inefficient as the number of disks gets larger.

Thus, on the one hand we have external sorting algorithms that are optimal (up to a constant factor) but are not efficient in practice unless the number of disks is very large, and on the other hand we have a simple disk striping technique that works well with a small number of disks.

In this paper, we propose a *Simple Randomized Mergesort* (SRM) that is practical and provably efficient for a wide range of values of D. We show that SRM is asymptotically optimal within very small constant factors when $M = \Omega(DB \log D)$, which is generally satisfied in practice. Moreover, even for ranges of D, M, and B when SRM is sub-optimal, it still remains efficient and faster than DSM, as borne out by our theoretical analysis and empirical simulations.

In the next section we describe SRM and some previous approaches to external mergesort, and we list our main theoretical result Theorem 1, which gives an analytical upper bound on SRM's expected number of I/Os in the worst case. Sections 3–8 focus on SRM and its analysis. Section 3 discusses the basic idea of SRM and the way that records are distributed on the disks, which is important to attain efficient read and write parallelism. Section 4 discusses the forecasting data structure needed for parallel reads. Section 5 describes the merge process in detail. We analyze the merge process in Sections 6, 7, and 7.1 and prove Theorem 1. In Section 8, we briefly discuss the potential benefits of a deterministic version of our algorithm.

In Sections 9 and 10, we demonstrate the practical merit of SRM by comparison with DSM. We base our comparisons on both our analytical worst-case expected bounds as well as the more optimistic empirical simulations of average-case performance, for a wide variety of values for the parameters M, D, and B.

2 Main Results

2.1 Background

The first step in mergesorting a file of N records is to sort the records internally, one half memoryload at a time so as to overlap computation with I/O, to get $2N/M$ sorted runs each of length $M/2$. Techniques like replacement selection [Knu73] can produce roughly N/M runs each of length about M. Then, in a series of passes over the file, R sorted runs of records are repeatedly merged together until the file is sorted. At each step of the merge, the record with the smallest key value among the R leading records of the R runs being merged is appended to the output run. The merge order parameter R is preferably chosen as close to $M/2B$ as possible so as to reduce the number of passes and still allow double buffering.

In order to carry out a merge, the leading portions of all the runs need to be in main memory. As soon as any run's leading portion in memory gets depleted by the merge process, an I/O read operation is required. Since a large number of runs (close to $M/2B$

runs) are merged together at a time, the amount of memory available for buffering leading portions of runs is very small on average. It is of paramount importance that only those blocks that will be required in the near future with respect to the merge process are fetched into main memory by the parallel reads. However, this memory management is difficult to achieve since the order in which the data in various disk blocks will participate in the merge process is input dependent and unknown. Vitter and Shriver [VS94] give further intuition regarding the difficulty of mergesorting on parallel disks. In Greed Sort [NV90], the trick of "approximate merging" is used to circumvent this difficulty. Aggarwal and Plaxton [AP94] use the Sharesort technique that does repeated merging with accompanying overhead.

Recently, Pai et al [PSV94] considered the average-case performance of a simple merging scheme for $R = D$ sorted runs, one run on each disk. They use an approximate model of average-case inputs and require that the internal memory be sufficiently large. They also require that each run reside entirely on a single disk; in order to get full output bandwidth, the output run must be striped across disks. A mergesort based on their merge scheme thus requires an extra transposition pass between merge passes so that striped output runs of the previous merge pass can be realigned onto individual disks.

2.2 Overview of SRM

We present an efficient, practical, randomized algorithm for external sorting called *Simple Randomized Mergesort* (SRM). Our algorithm uses a generalization of the forecasting technique [Knu73] in order to carry out parallel prefetching of appropriate disk blocks during merging. Randomization is used only while choosing, for each input run, the disk on which to place the first block of the run; the remaining blocks of that run are cyclically striped across the disks. The use of randomization is thus very restricted; the merging itself is deterministic.

If the internal memory available is of size M, with block size B and with D parallel disks, SRM merges R runs together at a time in each merge pass, where R is the largest integer satisfying $M/B \geq 2R + 4D + RD/B$. We note that M/B is the number of internal memory blocks available. Hence the merge order (the number of runs SRM merges together at a time) is determined by the amount of internal memory at its disposal. As a function of M, B, and D, the merge order R is given by $(M/B - 4D)/(2 + D/B)$. In the the realistic case that $D = O(B)$, SRM merges an optimal number (namely, $R = \Theta(M/B)$) of runs together at a time. For simplicity in the exposition, we assume that $D = O(B)$. (We can use the partial striping technique of [VS94] to enforce the assumption if needed.)

There are two aspects to our analysis of the algorithm—one theoretical and one practical. The first (and main) part of our analysis is bounding the expected number of I/Os of SRM, where the expectation is with respect to the randomization internal to SRM and the bound on the expectation holds for any input. SRM uses an elegant internal memory management scheme to solve the fundamental prefetching difficulties of simple merging on parallel disks alluded to earlier in Section 2.1. Below we briefly sketch this technique:

At any time while merging together R runs, let us consider how SRM reads in the next R blocks on disk required by the merge process. Observing that the next R blocks on disk will not, in general, be uniformly distributed among the D disks, we denote as d^* the maximum number of blocks among these R blocks that reside on a single disk. The blocks of the R input runs are striped across the disks. In order to maximize parallelism, in each I/O read operation, SRM fetches into memory one block from each disk whenever there is space to accomodate D blocks in main memory. In any I/O read operation, the block that is read in from each disk is the one that contains the smallest key among all the input

blocks on that disk that are not yet in memory. This is achieved by using the forecasting technique. In the event that there is only enough free memory to read in F additional disk blocks, where $F < D$, SRM *flushes out* $D - F$ previously read blocks from memory back to disk. Among the blocks already in memory, the $D - F$ blocks chosen for flushing are precisely the $D - F$ blocks that will participate in the merge farthest in the future. The internal memory buffer SRM that uses is just large enough to ensure that no block that is among the R blocks required next by the merge is ever flushed out. Moreover, there is no actual I/O that accompanies the flushing operation; SRM merely pretends that those blocks had never been read into memory. The forecasting and flushing are enough to ensure that bringing precisely those R blocks that are next required by the merge from disk into memory will take no more than d^* I/O read operations.

This memory management scheme enables us to obtain a handle on SRM's performance. We are able to relate SRM's performance to a combinatorial problem we call the *dependent maximum occupancy* problem, which is a generalization of the classical maximum occupancy problem [KSC78, VF90].

Our main theorem below gives expressions for the I/O performance of SRM for three patterns of growth rate between the number M/B of blocks in internal memory and the number D of disks in the parallel disk system. The different sizes of internal memory considered correspond to certain values of the merge order R as defined by the formula $R = (M/B - 4D)/(2 + D/B)$ given above. The general formulation in terms of occupancy statistics is given later.

Theorem 1 *Under the assumption that $D = O(B)$, the merge order $R = \Theta(M/B)$ of SRM is optimal. SRM mergesort uses $\frac{N}{DB}(1 + \frac{\ln(N/M)}{\ln R})$ I/O write operations[2], which is optimal. The expected number $Reads_{SRM}$ of I/O read operations done by SRM can be bounded from above as follows:*

1. *If $R = kD$ for constant k, as R, $D \to \infty$, we have*

$$Reads_{SRM} \leq \frac{N}{DB} + \frac{N}{DB}\frac{\ln(N/M)}{\ln kD}\frac{\ln D}{k \ln\ln D}\left(1 + \frac{\ln\ln\ln D}{\ln\ln D} + \frac{1 + \ln k}{\ln\ln D} + O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right).$$

2. *If $R = rD\ln D$ for constant r, as R, $D \to \infty$, we have*

$$Reads_{SRM} \leq \frac{N}{DB} + c\frac{N}{DB}\frac{\ln(N/M)}{\ln(rD\ln D)} + O(1)$$

which is optimal within a constant factor, namely c. The magnitude of c depends on r.

3. *If $R = rD\ln D$ where $r = \Omega(1)$, we have*

$$Reads_{SRM} \leq \frac{N}{DB} + \frac{N}{DB}\frac{\ln(N/M)}{\ln(rD\ln D)}\left(1 + \sqrt{\frac{2}{r}} + \frac{\ln r}{\sqrt{2r}\ln D} + O\left(\frac{1}{r} + \frac{\log r}{r\log D} + \frac{1}{D\sqrt{r}}\right)\right)$$

which is asymptotically optimal; that is, the factor of proportionality multiplying the $\frac{N}{DB}\frac{\ln(N/M)}{\ln(rD\ln D)}$ term is 1.

[2]Strictly speaking, the expressions for the exact number of "passes" and hence the number of I/O operations required to sort N records using a mergesort involves applying the ceiling ($\lceil\ \rceil$) function to certain logarithmic terms. Employing these exact expressions would mean dealing with a complicated dependence on N, without having any significant impact on our results for large N. Thus in this exposition we choose to overlook this issue and not apply the ceiling function, thus obtaining simplified expressions for I/O complexity.

The mergesort algorithm by Pai et al [PSV94] uses significantly more I/Os and internal memory. In their scheme, the internal memory size needs to be $\Omega(D^2B)$ to attain an efficient merging algorithm.

The other aspect of our analysis deals with the practical merit of our SRM mergesort algorithm on existing parallel disk systems. In Sections 9 and 10, we consider implementations of disk striping (DSM) and SRM mergesort algorithms for the interesting case $R = kD$, in which the number of blocks in internal memory scales linearly with the number of disks D being used. We demonstrate that SRM's good I/O performance is not merely asymptotic, but extends to practical situations as well. Like DSM, SRM also overlaps I/O operations and internal computation, which is important in practice.

3 SRM Data Layout

Our primary goal in this paper is to take advantage of mergesort's simplicity and potential for high efficiency in practice when used with parallel disk systems. We use striped input runs in our merging procedure so that the output runs of one merge pass written with full write parallelism can participate as input runs in the next pass without any reordering or transposition. In this scheme, if the 0th block of a run r is on disk d_r, then the ith block resides on disk $(i + d_r) \bmod D$. Striping alone is not enough to ensure good performance. If many runs are being merged and the disk d_r for each run r is chosen deterministically, the merging algorithm can have abysmal worst case performance. Throughout the entire duration of the merge, the R leading blocks of the R runs may always lie on the same disk, thus causing I/O throughput to be only a factor of $1/D$ of optimal.

A natural alternative that we pursue in this paper is to consider a randomized approach to mergesort. We randomly assign the starting disk d_r for each input run r. Then, on average, it is not necessary to read many blocks from the same disk at one time. The analysis involves an interesting reduction to a maximum bucket occupancy problem.

In order to facilitate analysis, *for each run r we choose the disk d_r on which r's initial block will reside independently and identically with uniform probability.* The location of every subsequent block of run r is determined deterministically by cycling through the disks. We will measure our algorithm's performance by bounding the *expected* number of I/Os required to merge R runs for any *arbitrary* (that is, *worst-case*) set of input runs; the expectation is with respect to the randomization in the placement of the first block of the runs. The actual key values of the records that constitute the runs can be arbitrary and their relative order does not affect the bounds we derive.

If we consider the problem of deterministic mergesorting with random inputs, as opposed to randomized mergesorting, we believe that the same analysis technique can yield similar bounds (but for the average-case) for a version of our algorithm in which the starting disk d_r for run r is deterministically chosen to be uniformly staggered, so that $d_r = 0$ for $r = 0, 1, \ldots, R/D - 1$, $d_r = 1$ for $r = R/D, R/D + 1, \ldots, 2R/D - 1$ and so on. Moreover, as we indicate in Section 8, by taking into consideration the lengths of the runs being merged in different stages of the mergesort, it may be possible to improve the bound on overall average I/O performance. In this paper, however, we concentrate on our SRM method, since a very limited amount of explicit randomization in SRM (used to determine the random starting disks d_r) enables us to get around the average-case assumption of the deterministic version, thus facilitating the elegant analysis presented here.

4 Forecasting Format and Data Structure

As observed earlier, each of the R runs that participate in a merge is output onto disk either during the run formation stage or during some earlier merge. This enables us to begin each run on a uniformly random disk. We can also format each block of every run so as to implant some *future* information as explained below. This is a generalization of the *forecasting* technique of [Knu73]. We denote the ith block of a run r as $b_{r,i}$ and the *smallest (first) key* in block $b_{r,i}$ as $k_{r,i}$. Blocks of a run r are formatted as follows:

- Implanted in the initial block $b_{r,0}$ of run r are the key values $k_{r,j}$ for $0 \leq j \leq D-1$. (Typically in practice, the D key values will indeed fit in a small portion of one block. Even if this is not so, since $D = O(B)$, this information will fit in at most $O(1)$ blocks.)
- Implanted in the ith block $b_{r,i}$, where $i > 0$, of run r is the single key value $k_{r,(i+D)}$.

The extra information for forecasting in each block is only one key value, not one record. So the extra space taken up by the forecasting format of runs is negligible. For simplicity, we assume that all key values are distinct.

Definition 1 At any time t during the merge, consider the unprocessed portions of the R runs. Among these records, the record with the smallest key is called the *next* record of the merge at time t. A block belonging to any run is said to begin *participating* in the merge process at the time t when its first record becomes the next record of the merge. A block ceases to participate in the merge as soon as all of its records get depleted by the merge. At any point of time, consider all the blocks not in memory that reside on disk i. The *smallest block of run j on disk i* is the block that will be the earliest participating block among all the blocks of run j on disk i. The *smallest block on disk i* is the the earliest participating block on disk i at that time. At any time, the *leading* block of a run is the block (possibly partly consumed) that contains the smallest key value in that run at that time. Note that a block can become a leading block much before beginning to participate in the merge.

On an I/O read, the block read in by SRM from disk i is always the smallest block on disk i at that time. To be able to do this, SRM maintains a forecasting data structure.

Definition 2 The *forecasting data structure* (FDS) consists of D arrays $\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_{D-1}$, one corresponding to each disk. At any point of time, $\mathcal{H}_i[j]$ stores $K_{i,j}$, where $K_{i,j}$ is the smallest key value in the smallest block of run j on disk i.

At any time, in order for SRM to read in the smallest block from disk i, it merely needs to read in the smallest block of run j on disk i, where j is the run with the smallest key in $\mathcal{H}_i$.

5 The SRM Merging Procedure

In this section, we discuss how SRM merges R runs striped across D disks, with the randomized layout described in Section 3.

The SRM merging process can be specified as a set of two concurrent logical control flows corresponding to *internal merge processing* and *I/O scheduling*, respectively. By internal merge processing, we refer to the computation that merges the leading portions of runs after these leading portions have been brought into internal memory. Internal merge processing

has to wait when some run's leading portion in memory gets depleted or when the write buffers get full and need to be written to disk. By *I/O scheduling* we refer to the algorithm that decides which blocks to bring in from the disks. I/O scheduling decisions are based on information from the FDS data structure and the amount of unoccupied internal memory. Implementing internal merge processing and the I/O schedule as concurrent modules makes the overlapping of CPU computation and I/O operations an easier task.

Internal merge processing can be implemented using a variety of techniques; we refer the reader to [Knu73]. In this section, we focus on the I/O scheduling algorithm. We first discuss issues pertaining to management of internal memory and maintaining the FDS data structure during the merge. We then go on to introduce some terminology and notation that helps us describe I/O operations and the I/O scheduling algorithm. The terminology we develop here will be used even in subsequent sections when we analyze the algorithm.

5.1 Internal Memory Management

Management of internal memory blocks plays a key role in the interaction between the I/O scheduling algorithm and internal merge processing. In SRM, internal memory management of merge data is done at block level granularity, so internal fragmentation within internal memory blocks is not an issue. By an internal block, we mean a set of contiguous internal memory locations with fixed boundaries, large enough to hold B records. Note that the internal structures pertaining to internal memory management need to be updated whenever internal merge processing is in certain critical states and after every I/O read operation. In this subsection, we describe some of the more important aspects of SRM's internal memory management.

The number M/B of blocks in internal memory, expressed in terms of R, B and D, is $2R + 4D + RD/B$. Of these, SRM maintains a dynamically changing partition $\{M_L, M_R, M_D, M_W\}$ of $2R + 4D$ physical blocks of internal memory.

Definition 3 The set M_L of R internal blocks is maintained such that at any time, if the leading block of any run is in internal memory, its internal block is in M_L. The set M_D contains D internal blocks maintained such that each I/O read operation of SRM can read in D blocks from disks into the internal memory blocks of M_D. The remaining $R + D$ internal memory blocks comprise the set M_R. The sets are maintained such that as soon as there are D unoccupied internal blocks, a parallel read can be initiated into M_D. In addition, SRM uses a set M_W of $2D$ internal memory blocks as an output buffer.

The D internal memory blocks of M_D are used specifically to initiate I/O read operations at the earliest possible time, potentially bringing in blocks even before they begin participating. The output buffer M_W is large enough to ensure that the output run can be written out with full parallelism in the forecasting format described earlier.

The forecasting data structure FDS and related auxiliary data structures occupy not more than about RD/B blocks.

5.2 Maintaining the dynamic partition of internal memory

As internal merging proceeds and the parallel I/O operations keep transferring blocks between the parallel disk system and internal memory, internal memory blocks need to be exchanged among the sets M_L, M_R, M_D, and M_W in order to ensure that the assertions in Definition 3 are met. Below, we describe three types of exchanges of internal memory blocks in the management of M_L, M_R, and M_D:

1. Consider any state of internal merge processing when the last record in the leading block of a run r gets consumed by the merge. If run r's new leading block occupies an internal block of M_R, then M_R and M_L mutually exchange the internal blocks corresponding to the new and old leading blocks so that M_R gets an unoccupied internal block and M_L gets an occupied one.

2. Consider any I/O read operation that has just completed. I/O read operations always read in blocks into the D internal memory blocks of set M_D. If the read operation brings into M_D a block that is the leading block of some run r, then M_L and M_D exchange internal blocks corresponding to the old and new leading blocks, so that M_D gets an unoccupied internal block and M_L gets an occupied one.

3. Whenever M_R has at least one unoccupied internal memory block and M_D has at least one occupied internal memory block, M_R and M_D exchange internal memory blocks so that M_D obtains an unoccupied block and M_R gets an occupied one.

In addition, records are added to blocks of M_W as the internal merging proceeds.

5.3 Maintaining the Forecasting Data Structure

Whenever a block belonging to run j is read into memory from disk i, the forecasting data from that block is used to update the entry $\mathcal{H}_i[j]$ of FDS.

We will see in Definition 6 that SRM may at times flush a block belonging to run j to the disk i it originated from. This will not require any I/O but merely require that the entry $\mathcal{H}_i[j]$ of FDS be updated with the smallest key of the particular block being flushed. If more than one block from a particular run j are flushed together, all to the same disk i, then the entry $\mathcal{H}_i[j]$ is updated with the smallest key value among all the blocks from run j being flushed to disk i.

5.4 Terminology and Notation

Definition 4 Suppose we order a set A of blocks in ascending order by the blocks' smallest key values. We define $Rank_A(b)$, for $b \in A$, to be the rank of block b in this order. The first (smallest-valued) block has rank 1 and the last (largest-valued) block has rank $|A|$. For any time t during the merge, we define the following terms:

- $\mathcal{F}_t$ denotes the set of full blocks in internal memory such that no block b in $\mathcal{F}_t$ is the leading block of any run at time t.

- $\mathcal{S}_t$ denotes the set of D blocks, each of which is the smallest block on one of the D disks.

- $Fset_t(l) = \{b \mid \ b \in \mathcal{F}_t,\ Rank_{\mathcal{F}_t}(b) \geq |\mathcal{F}_t| - l + 1\}$ denotes the set of the l highest ranked blocks of $\mathcal{F}_t$. By definition, no block of $Fset_t(l)$ is a leading block.

- $OutRank_t = \min_{b \in \mathcal{S}_t}\{Rank_{\mathcal{F}_t \cup \mathcal{S}_t}(b)\}$ denotes the rank of the smallest-ranked block of $\mathcal{S}_t$ in the set $\mathcal{F}_t \cup \mathcal{S}_t$.

Definition 5 The operation $ParRead_t$ is executed at a time t only when D unoccupied internal blocks are available in M_D. Using FDS, it reads in the set $\mathcal{S}_t$ of D blocks into M_D, exchanging internal blocks of M_D with M_L as in point 2 of Section 5.1 if necessary. FDS is updated as described in Section 5.3 using the implanted information in blocks of $\mathcal{S}_t$.

Definition 6 The operation $Flush_t(j)$ virtually flushes out the j blocks of the set $Fset_t(j)$ occupying internal blocks in M_R at time t. In SRM, we have $j \leq D$ whenever $Flush_t(j)$ is invoked. The flushing of blocks is virtual in the sense that the blocks *do not actually have to be written out to disk*, so there is no I/O involved. FDS is updated as described in Section 5.3 to reflect the fact that the flushed out blocks will have to be read back from their original disks when needed.

An important feature of SRM exploited in Section 6 is that leading blocks of runs are never flushed out.

5.5 I/O Scheduling Algorithm

In this subsection, we specify the parallel I/O schedule of SRM. As far as writes go, Each write is executed with full write parallelism as soon as the next output stripe of D formatted blocks are ready in the output buffer M_W. With respect to reads, the basic idea is to read in D blocks of the set $\mathcal{S}_t$ whenever possible; flushing a few of the highest ranked non-leading blocks in $\mathcal{F}_t$, if necessary.

1. At the beginning, SRM reads the first block from each of the R runs into the R internal blocks of M_L using parallel reads.
2. Until the merge is completed, whenever the I/O system is free at a time t when D unoccupied blocks are available in M_D, SRM does the following:
   (a) If there are D unoccupied internal blocks in the set M_R at time t, a $ParRead_t$ operation is initiated.
   (b) Else if the number of occupied internal blocks in M_R at time t is $R + extra$, where $1 \leq extra \leq D$, and if $OutRank_t > extra$, a $ParRead_t$ operation is initiated.
   (c) Else if the number of occupied internal blocks in M_R at time t is $R + extra$ where $1 \leq extra \leq D$, and if $OutRank_t \leq extra$, a $Flush_t(extra - OutRank_t + 1)$ operation is invoked, followed by initiation of a $ParRead_t$ operation.

In Step 2c, the flushed out blocks are the ones that will not be used in the "near future," as we will see in the next section. Lemma 1 below ensures that SRM merging runs to completion, needing only the specified amount of memory.

Lemma 1 *Consider any $ParRead_t$ operation invoked by SRM at time t. No block not in internal memory when $ParRead_t$ has completed begins participating in the merge until D unoccupied physical blocks are available in the set M_D of internal memory blocks. This ensures that there will be enough space for the next read operation to begin before any outside block begins participating.*

Proof: Consider step 2a of the algorithm: D unoccupied internal blocks are available in the set M_R before $ParRead_t$ gets initiated. In this case, the requirement of having D unoccupied blocks in internal memory for the next parallel read is trivially met by exchange operations as in point 3 of Section 5.1. Consider step 2b of the algorithm in which only $D - extra$ unoccupied blocks are available in M_R but $OutRank_t > extra$ just before $ParRead_t$ gets initiated. In this case, by definition of $OutRank_t$, $extra$ blocks that belong to the set M_R just before $ParRead_t$ gets initiated begin participation before any block not yet in memory after completion of $ParRead_t$. These $extra$ blocks become *leading* blocks of their respective runs before participating. Thus by exchange operations as in point 1 of Section 5.1, M_R

gets *extra* unoccupied internal memory blocks from M_L. This means that by means of exchange operations as in point 3 of Section 5.1, M_D has $D - extra + extra = D$ unoccupied blocks before participation of any block that is still on disk after $ParRead_t$ is completed. Similarly, in case of step 2c of the algorithm, $OutRank_t - 1$ internal blocks of M_R become unoccupied owing to participation while $extra - OutRank_t + 1$ blocks of M_R are flushed. These unoccupied blocks, along with the $D - extra$ unoccupied blocks already present in M_R ensure that the set M_D does obtain D unoccupied blocks before any outside block begins participating in the merge. This completes the proof of the lemma. □

Lemma 1 shows that a *ParRead* operation can be initiated before any block brought in by that operation begins participation. Thus there is genuine prefetching ability, which is useful in overlapping I/O operations with internal processing.

6 Using Phases to Count *ParRead* Operations

For the purpose of analysis, we break the process of merging R runs comprising a total of N' records into a sequence of *phases*, and we upper bound the expected number of read operations required during a phase. The overall bound on the merge process can then be computed using the linearity of expectations.

Consider the set of all blocks except blocks in the initial set. Each such block can participate in the merge only after its preceding blocks have ceased to participate, so that if the ith block of a run is currently participating, at least $(i-1)B$ records from that run have already been output.

Definition 7 Consider the set $\mathcal{R}_0$ of all blocks of all runs excluding the initial block of each run. The *participation index* of any block in $\mathcal{R}_0$ is i if it is the ith block, in chronological order, to begin participating in the merge, where $1 \leq i \leq \frac{N'-RB}{B}$. We denote by set $\mathcal{P}_j$, where $1 \leq j \leq \frac{N'-RB}{RB}$, the subset of $\mathcal{R}_0$ consisting of blocks whose participating indices are in the range $[(j-1)R+1, jR]$.

Using parallel reads, SRM tries to fill the R internal blocks of M_R using FDS. Every incoming block provides new implanted information. When a block with small participation index is on disk while memory is full, we bring it in, by flushing blocks that are *surely not* among the next R blocks to begin participating in the merge, as Lemma 2 shows.

Lemma 2 *Consider a flush $Flush_t$ at time t. The $R + OutRank_t - 1$ smallest-ranked blocks of $\mathcal{F}_t$ (in M_R) do not get flushed out: they remain in M_R after the flush completes.*

Proof: Since $Flush_t$ is invoked at time t, it means that the number of blocks in M_R is $R + extra$ ($1 \leq extra \leq D$) and $0 < OutRank_t \leq extra$. By definition of the $Flush_t$ operation, $extra - OutRank_t + 1$ blocks corresponding to the $extra - OutRank_t + 1$ highest ranked blocks in the set of all blocks occupying internal blocks of M_R are flushed out. This means that $R + OutRank_t - 1$ of the lowest ranking blocks remain in internal memory even after the $Flush_t$ operation. Hence the lemma is proved. □

The merge proceeds in a sequence of phases, each contributing R blocks to the output run.

Definition 8 The first phase begins at the time p_0 of the last $ParRead_{p_0}$ read operation invoked in Step 1 of SRM. The jth phase ends at the time p_j of the read $ParRead_{p_j}$ such that any block $b \in \mathcal{R}_0$ with participation index i_b, where $i_b \leq jR$, has been read into

memory (at least once) from disk by some $ParRead_t$ such that $t \leq p_j$. The $ParRead_t$ read operations such that $p_{j-1} < t \leq p_j$ are the *reads forming the jth phase*, and the set $\mathcal{P}_j$ defined above is the *set of blocks forming the jth phase.*

The next step is to show that when the jth phase is over, blocks with participation indices smaller than jR have already been "taken care of."

Lemma 3 *After the read $ParRead_{p_j}$ at time p_j is complete, the following invariants hold:*

1. *No block still on disk has a participating index less than $jR + 1$.*
2. *No block b with participating index $i_b \leq (j+1)R$ can be flushed out by any $Flush_t$, where $t > p_j$.*

Proof: We prove the claims by induction. For the base case $j = 1$, observe that by time p_1, all blocks having participation indices at most R are read in at least once. By Lemma 2, none of these blocks could be flushed out at any time $t \leq p_1$ because they would be among the R smallest ranked blocks of M_R, thus proving the claim. Now consider the second claim for $j = 1$. Suppose there is a flush $Flush_t$ at any time $t > p_1$. By the previous claim, if X is the number of blocks in M_R that have participating indices at most R, then $OutRank_t > X$. By Lemma 2, the $R + OutRank_t - 1$ smallest ranked blocks at time t remain in M_R after the flush. Of these, at most $X \leq OutRank_t - 1$ blocks have ranks smaller than R. Thus if there is any block b with rank at most $2R$ at time t, then b must be among the $R + OutRank_t - 1$ smallest ranked blocks. Hence the base case is proved for both claims.

Now suppose both claims are true for some $j - 1$. Then by inductive hypothesis for Claim 1, when the read $ParRead_{p_{j-1}}$ at p_{j-1} completes, the smallest participating index of any block still on disk is $(j-1)R + 1$. By the inductive hypothesis for Claim 2 none of the blocks with participating indices at most jR can get flushed at any time $t > p_{j-1}$. By definition, all blocks with indices at most jR are read in by p_j. Since none of them can get flushed after p_{j-1}, the first claim is true for j. The second claim for j follows by an argument identical to the argument for its base case. □

We are led immediately to the following lemma that gives the number of blocks in the output run of the merge at the end of the first j phases.

Lemma 4 *At least jRB records from blocks of the set $\mathcal{R}_0$ are already in the merged output of SRM* before *any block still on disk after time p_j begins participating in the merge. Thus all reads $ParRead_t$ such that $t \leq p_j$ and the reads I_0 required in the step 1 of SRM can be charged to these jRB records in the output of SRM.*

We wish to obtain a handle on the number of read operations $ParRead_t$ where $p_{j-1} < t \leq p_j$ (the reads associated with the jth phase). We will show that this number is the largest number of blocks in $\mathcal{P}_j$ that need to be read in from any particular disk.

Definition 9 Consider the blocks of the set $\mathcal{P}_{j+1}$ just after $ParRead_{p_j}$ is completed. Such a block b is said to be on *level* 0 if it is already in internal memory and level h if it is the hth smallest block on its disk at that time. Let L_{j+1} denote the highest level of any block of $\mathcal{P}_{j+1}$. In the case of the first phase, let L_1 be the highest level of any block of $\mathcal{P}_1$ after Step 1 of the algorithm in Section 5.5.

We prove the following characterization of the number of reads associated with the $(j+1)$st phase.

Lemma 5 *The L_{j+1}th read of SRM* after *time p_j is done at time p_{j+1}.*

Proof: The proof follows from Lemma 3. No block with participation index at most jR is still on disk after the read at p_j completes. Moreover no block with participation index at most $(j+1)R$ can be flushed out after p_j. Clearly then, since any $ParRead_t$ operation reads in the smallest block from every disk at time t, we have the nice property that *all blocks of $\mathcal{P}_{j+1}$ that are at the hth level will get read by the hth read after p_j*, never to get flushed out. This proves the lemma. □

We can now relate progress in the merge to the number of read operations involved in phases by means of the following lemma that follows from Lemmas 5 and 4.

Lemma 6 *The first $I_0 + \sum_{1 \le i \le j} L_i$ reads of SRM can be charged to at least jRB records in the output of the merge, where I_0 is the number of reads SRM incurs in Step 1 of the algorithm in Section 5.5. The total number of I/O read operations required to complete the merge is given by the above sum with $j = \frac{N'/B-R}{R}$.*

In the next section we give a bound on the expectation of L_i with respect to the randomization involved in the choice of disks for the initial blocks of the runs. Thereafter, Lemma 6 can be used to obtain the overall bound on the number of reads involved in SRM.

7 Probabilistic Analysis

Consider any arbitrary R runs input to SRM. The initial block of each run is placed on a disk chosen with a uniform probability of $1/D$ independently of other runs. Since runs are cycled across disks, the position of the initial block of the run determines the position of the other blocks of the run. As a result, for any j, the jth block of a run is on any one of the D disks, each with probability $1/D$, depending only on the disk containing that run's initial block.

Lemma 7 *Consider the ith phase and the related set $\mathcal{P}_i$ of blocks such that $1 \le i \le \frac{N'-RB}{RB}$. Let n_j denote the number of blocks from the jth run in $\mathcal{P}_i$, so that $\sum_{0 \le j \le R-1} n_j = R$. Let C_j denote the ordered list $\langle b_{j,0}, b_{j,1}, \ldots, b_{j,n_j-1} \rangle$ of the n_j blocks of run j in $\mathcal{P}_i$, where the order is the chronological order of participation. Let D_j denote the disk from which the block $b_{j,0}$ originates, for each run j, $0 \le j \le R-1$. Then the disks D_j are independently distributed, each with uniform probability over the set $\{0, 1, \ldots, D-1\}$ of D disks.*

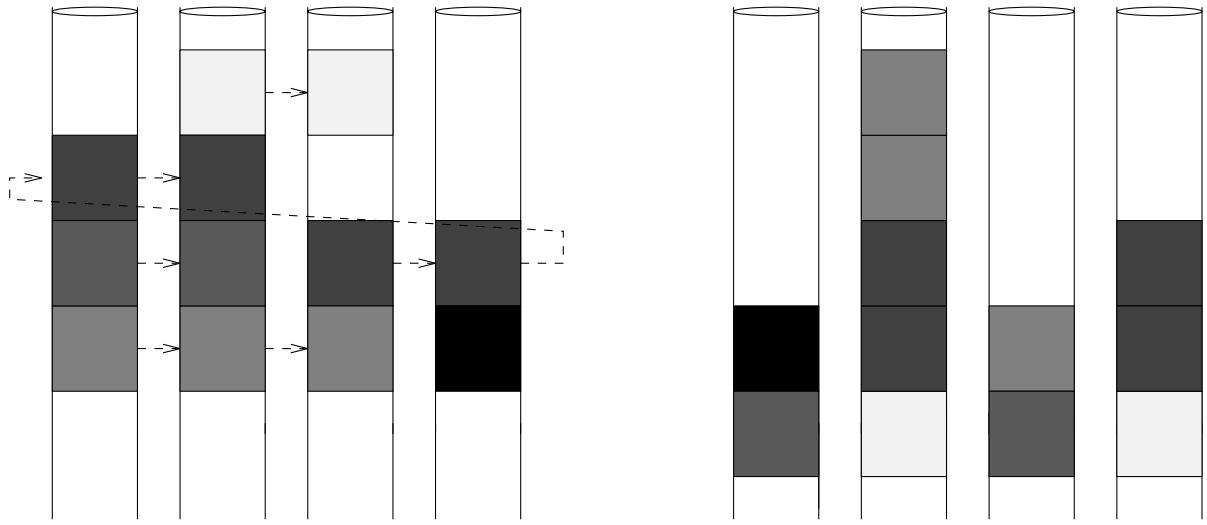
Definition 10 We call each C_j a *chain* of length n_j of contiguous blocks of run j. The disk corresponding to any given block of C_j is determined by the disk of the lead block $b_{j,0}$ of C_j.

We recall that the number of reads L_i in the ith phase is the maximum level of any block on any disk at time p_{i-1}.

Definition 11 Consider the set $\mathcal{P}_i$. Let L'_i be the maximum level of any block of $\mathcal{P}_i$ on disk, considering all of $\mathcal{P}_i$'s blocks on their respective original disks.

In the following subsections, we will be able to bound $E(L'_i)$. The following lemma shows that L'_i is an overestimate of L_i, so we get a bound on L_i too.

Lemma 8 *With L_i and L'_i defined as above, we have $L_i \le L'_i$ and $E(L_i) \le E(L'_i)$.*



(a) Dependent Occupancy (b) Classical Occupancy

Figure 1: (a) Dependent occupancy instance with $N_b = 12$, $C = 5$, $D = 4$. Arrows indicate the cyclical order of blocks in the same chain. The maximum occupancy is 4, as realized in the second bin. (b) Classical occupancy instance with $N_b = 12$, $D = 4$. Blocks fall independently of each other. The maximumm occupancy is 5, as realized in the second bin.

7.1 The Dependent Occupancy Problem

In this section, we define the *dependent occupancy* problem, which along with the well-studied *classical occupancy* problem (a special case of the former), is directly related to the I/O performance of SRM.

> In the **classical occupancy** problem of parameters $\{N_b, D\}$, N_b balls are thrown into D bins independently each with uniform probability of $1/D$ of falling into any bin. We denote the asymptotically tight expression for the expectation of the maximum number of balls in any bin [VF90, KSC78] by $\mathcal{C}(N_b, D)$.
>
> In the **dependent occupancy** problem, we consider D bins but instead of balls, we consider C chains of balls, such that the total number of balls summed over the C chains, is N_b. A chain of length ℓ consists of ℓ balls linked together in a chain. A chain of length ℓ is said to fall or get thrown into a bin s, where $0 \leq s \leq D-1$, if the leading ball of the chain falls into bin s and the remaining $\ell - 1$ balls of that chain are deposited cyclically into bins; that is the i-th ball (with $0 \leq i \leq \ell - 1$) of that chain falls in bin $(s + i) \bmod D$. Since the sum of the number of balls from all the chains is exactly N_b, the maximum length of a chain is N_b. Denoting as n_j, the number of chains of length j, we have $\sum_{1 \leq j \leq N_b} n_j = C$ and $\sum_{1 \leq j \leq N_b} j n_j = N_b$. We are interested in the expectation of the maximum occupancy of any bin when each of the C chains gets independently and identically randomly thrown, such that the probability of each chain falling into any of the D bins is $1/D$.

The classical occupancy problem is the special case of the dependency occupancy problem when $C = N_b$ and $n_1 = N_b$ with $n_j = 0$ for $1 < j \leq N_b$. Figure 1 above shows one instance of both types of problems for $N_b = 12$, $D = 4$, and with $C = 5$ for the depen-

dent occupancy problem. The figure uses square blocks instead of balls and uses arrows to connect together blocks from the same chain.

7.2 Asymptotic Expansions of the Maximum Occupancy

Our goal here is to obtain a bound on the expected maximum occupancy in dependent occupancy problems. As mentioned earlier, the leading terms in our upper bounds for this quantity are the same as those in the well-known bounds for $\mathcal{C}(N_b, D)$ in the classical occupancy problem [KSC78]. We conjecture that the expected maximum classical occupancy $\mathcal{C}(N_b, D)$ is an *upper bound* for the maximum dependent occupancies.

Consider the following intuition: in dependent occupancy, a chain of length ℓ results in a *maximum occupancy* of $\lceil \ell/D \rceil$ because of the constraint that balls of the chain are cyclically distributed among the D bins. The cyclic distribution tends to reduce the variance of the individual occupancies, thus lowering the maximum occupancy. In classical occupancy, the *average occupancy* of any bin when ℓ balls are independently and uniformly randomly thrown is ℓ/D but more than ℓ/D of them can easily fall into the same bin.

For instance, in Figure 1, it can be seen that pairs of blocks from the same chain that are forced to occupy different bins in the dependent occupancy problem may occupy the same bin in the classical version of the occupancy problem. Intuitively, independence increases the expected maximum occupancy.

In [BGV96], our approach was to use the well known bounds for the classical maximum occupancy to bound dependent maximum occupancies; however, the proof of the bound we gave was erroneous. We do believe, however, that the bound is correct, as we conjecture above. In this paper, we instead derive a direct bound on the expectation of the maximum dependent occupancy via interesting analytical and asymptotic techniques, and as a result our proof is independent of the classical occupancy bounds of [KSC78].

As a first step toward obtaining an upper bound on the maximum dependent occupancy, we first state and prove a simple lemma that shows that it will suffice for us to focus on dependent problems having no chains of length greater than D:

Lemma 9 *Consider any dependent occupancy problem $\mathcal{X}$ involving C chains, D bins and a total of N_b balls such that one or more chains in $\mathcal{X}$ are of length greater than D. Let X_{max} denote the maximum occupancy random variable of $\mathcal{X}$. Then there is a dependent occupancy problem $\mathcal{X}^*$ involving a total of N_b balls and D bins such that no chain involved in $\mathcal{X}^*$ has length greater than D and $E[X_{max}] = E[X^*_{max}]$, where X^*_{max} is the maximum occupancy random variable for $\mathcal{X}^*$.*

Proof: Consider the dependent occupancy problem $\mathcal{X}^1$ obtained by replacing a chain $\mathcal{C}_1$ of $\mathcal{X}$ of length $aD + b$, where $a \geq 1$ and $0 \leq b < D$, with, instead, a chains of length D and one chain of length b. Let X^1_{max} denote the maximum occupancy random variable of $\mathcal{X}^1$. We claim that the occupancy distribution of N_b balls among the D bins in the problem $\mathcal{X}$ is exactly the same as that of the problem $\mathcal{X}^1$.

To see this, first consider the occupancy distribution of balls of chain $\mathcal{C}_1$ among the bins when it is randomly thrown into the bins. Let us suppose that $1 \leq b < D$. When the chain $\mathcal{C}_1$ gets thrown, b contiguously placed bins numbered, say $s, (s+1) \bmod D, \ldots, (s+b-1) \bmod D$ with $0 \leq s \leq D-1$, each get $a+1$ balls. The other $D-b$ bins each get exactly a balls. There are D distinct sets $S_0, S_1, \ldots, S_{D-1}$, each set consisting of b contiguously placed bins. Let us denote by S'_i the set of bins other than those in S_i. Then for $0 \leq i \leq D-1$, the probability that set S_i's b bins each receive $a+1$ balls of $\mathcal{C}_1$ whereas bins in the set S'_i each receive a balls is exactly $1/D$.

Now let us consider the occupancy distribution of balls when a chains each of length D and one chain of length b get randomly thrown into the bins. It is not hard to see that the occupancy distribution of balls that results from throwing in these $a+1$ chains independently and uniformly randomly among the bins is precisely the occupancy distribution of balls that results from throwing chain $\mathcal{C}_1$, described in the previous paragraph. Even in the case when $b = 0$, the occupancy distribution of balls that results from throwing in chain $\mathcal{C}_1$ is the same as that obtained from throwing in a chains of length D each.

Moreover, the two occupancy distributions that result from the other balls are obviously identical since the same $C-1$ chains are involved in both the distributions. By independence and the claim in the previous paragraphs, the occupancy distribution of balls in problem $\mathcal{X}$ is the same as the one in problem $\mathcal{X}^1$.

Since we can replace one chain ($\mathcal{C}_1$) of problem $\mathcal{X}$ with a multiple number of chains of length at most D to obtain a problem $\mathcal{X}^1$ with the same occupancy distribution of balls, we can repeat this process on $\mathcal{X}^1$ to replace one more chain of length greater than D to obtain another problem $\mathcal{X}^2$, and so on, retaining the same occupancy distribution of balls as in the orginal problem $\mathcal{X}$. We continue this process until we have a dependency problem with no chain of length greater than D and denote this problem as $\mathcal{X}^*$. Since the occupancy distributions of $\mathcal{X}$ and $\mathcal{X}^*$ are identical, so are their expected maximum occupancies. This proves the lemma. □

We now use the above lemma in proving an upper bound on the expected maximum occupancy of dependent occupancy problems involving a total of N_b balls and D bins.

Theorem 2 *Consider any dependent occupancy problem $\mathcal{X}'$ that involves C' chains, D bins and a total of N_b balls from the C' chains. Let X'_{max} denote the maximum occupancy random variable for $\mathcal{X}'$. Let $N_b = kD$, for $k > 0$. Then*

1. *If k is a constant as $D \to \infty$,*

$$E[X'_{max}] \leq \frac{\ln D}{\ln \ln D}\left(1 + \frac{\ln \ln \ln D}{\ln \ln D} + \frac{1 + \ln k}{\ln \ln D} + O\left(\frac{(\log \log \log D)^2}{(\log \log D)^2}\right)\right).$$

2. *In the case where $k = r \ln D$ and $r = \Omega(1)$, we have*

$$E[X'_{max}] \leq \left(1 + \sqrt{\frac{2}{r}} + \frac{\ln r}{\sqrt{2r} \ln D} + O\left(\frac{1}{r} + \frac{\log r}{r \log D} + \frac{1}{D\sqrt{r}}\right)\right)\frac{N_b}{D}.$$

Note that the right hand side is of the form $\frac{N_b}{D}(1 + o(1))$ when $r \to \infty$. In the case when $r = \Theta(1)$, we have $E[X'_{max}] \leq cN_b/D$ for some constant $c > 0$.

Since the classical occupancy problem is a special case of the dependent occupancy problem, an immediate implication is the following corollary.

Corollary 1 *Consider a classical occupancy problem in which N_b balls are independently, uniformly randomly thrown into D bins. The expected maximum occupancy $\mathcal{C}(N_b, D)$ of this problem is bounded from above by the same upper bounds that we proved in Theorem 2 for the expectation $E[X'_{max}]$ of the maximum occupancy of any dependent occupancy problem involving a total of N_b balls and D bins.*

The upper bounds we prove for the expectation of maximum occupancy in the dependent occupancy problem are precisely the *asymptotically tight bounds* for the classical maximum occupancy problem derived in [KSC78]. In fact, our proof for the upper bound

on dependent expected maximum occupancy constitutes an alternate approach to obtaining the same asymptotically leading terms as those in [KSC78] for the classical expected maximum occupancy. (The proof in [KSC78] also serves as a lower bound on classical expected maximum occupancy but our techniques can be modified to do the same.)

Proof of Theorem 2: The given dependency problem $\mathcal{X}'$ might involve some chains that are longer than D in length. Lemma 9 ensures us that there always exists a dependency problem $\mathcal{X}$ such that $E[X_{max}] = E[X'_{max}]$ where X_{max} is the maximum occupancy random variable of $\mathcal{X}$, and $\mathcal{X}$ involves no chain of length greater than D. We will thus focus our attention on this problem $\mathcal{X}$.

Let C denote the number of chains in $\mathcal{X}$, and for $1 \leq j \leq N_b$, let n_j denote the number of chains of length j. Thus we have

$$\sum_{1 \leq j \leq D} n_j = C, \tag{1}$$

and

$$\sum_{1 \leq j \leq D} j n_j = N_b. \tag{2}$$

Let T denote any positive integer. One way of computing $E[X_{max}]$ is

$$\begin{aligned} E[X_{max}] &= \sum_{m \geq 0} \Pr\{X_{max} > m\} && (3) \\ &\leq T + \sum_{m \geq T} \Pr\{X_{max} > m\} && (4) \\ &\leq T + \sum_{m \geq T} D \cdot \Pr\{X > m\}. && (5) \end{aligned}$$

where X is the random variable corresponding to the *occupancy of one particular bin*, say bin b_0, the bin numbered 0. We will derive an apppropiate bound on the quantity $\sum_{m \geq T} \Pr\{X > m\}$ and then apply it in inequality (5) above to bound $E[X_{max}]$.

If all the chains involved are of length 1, computing $\Pr\{X > m\}$ is relatively straightforward. The fact that there can be up to $C = N_b$ chains whose lengths may vary from 1 through D introduces dependencies that complicate the task of obtaining a bound for $\Pr\{X > m\}$. To obtain a bound on $\Pr\{X > m\}$, we will use a generating function approach.

Definition 12 Consider a random variable W that takes integral values. The *probability generating function* (PGF) of W, denoted $G_W(z)$, is defined to be the function

$$G_W(z) = \sum_t \Pr\{W = t\} z^t.$$

Thus the coefficient of z^t in $G_W(z)$ is $\Pr\{W = t\}$.

In the simple case when dependency problem $\mathcal{X}$ has only one chain, and the chain has length ℓ, the PGF $G_X(z)$ is

$$G_X(z) = \left(1 - \frac{1}{\ell}\right) z^0 + \frac{1}{\ell} z^1 = 1 - \frac{1}{\ell} + \frac{1}{\ell} z.$$

In the general case when dependency problem $\mathcal{X}$ has C independent chains, the PGF is the product of the PGFs corresponding to each individual chain, which gives us

$$G_X(z) = \prod_{1 \le j \le D} \left(1 - \frac{j}{D} + \frac{jz}{D}\right)^{n_j}. \tag{6}$$

For $0 \le m \le \sum_{1 \le j \le D} n_j = C$, the coefficient $\Pr\{X = m\}$ of z^m in $G_X(z)$ is also the *residue* at $z = 0$ of the complex analytic function $G_X(z)/z^{m+1}$ [GK81]. Moreover, $z = 0$ is the only pole of $G_X(z)/z^{m+1}$ enclosed in a circle of radius P centered at $z = 0$ in the complex plane, where P is any positive number. Therefore, the residue theorem states that

$$\Pr\{X = m\} = \frac{1}{2\pi i} \oint_{|z|=P} \frac{G_X(z)}{z^{m+1}} dz, \tag{7}$$

where $i = \sqrt{-1}$. It follows that

$$\begin{aligned} \Pr\{X = m\} &\le \frac{1}{2\pi} \oint_{|z|=P} \left| \frac{G_X(z)}{z^{m+1}} \right| dz & (8) \\ &\le \frac{1}{2\pi} (2\pi P) \frac{G_X(P)}{P^{m+1}}. & (9) \end{aligned}$$

Step (9) follows because the coefficients of the PGF $G_X(z)$ are non-negative real numbers and thus $|G_X(z)|$ is maximized on the circle of radius P when z is positive, namely, when $z = P$. By (6) and (9), we get

$$\begin{aligned} \Pr\{X = m\} &\le \frac{G_X(P)}{P^m} & (10) \\ &= \frac{1}{P^m} \prod_{1 \le j \le D} \left(1 + \frac{(P-1)j}{D}\right)^{n_j} & (11) \\ &\le \frac{1}{P^m} \prod_{1 \le j \le D} \left(1 + \frac{P-1}{D}\right)^{jn_j} & (12) \\ &= \frac{1}{P^m} \left(1 + \frac{P-1}{D}\right)^{N_b}. & (13) \end{aligned}$$

Step (12) follows from the inequality $(1 + (P-1)j/D) \le (1 + (P-1)/D)^j$ when $j \ge 0$ and $P \ge 1$. Step (13) follows from (2). Note that if the probability generating function for the maximum occupancy random variable of classical occupancy with N_b balls and D bins is denoted $G_Y(z)$, then $G_Y(P)$ is precisely the expression $(1 + (P-1)/D)^{N_b}$.

We need to select an appropriate value of P in the bound (13) for $\Pr\{X = m\}$. The idea is to optimize the value of P so that the right hand side of (5) is minimal. In what follows, we will denote P as $1 + \alpha$, where α is a positive quantity to be determined later. It is understood that P (and thus α) may be a function of D.

Substituting $P = 1 + \alpha$ into (13), we have

$$\Pr\{X = m\} \le \frac{1}{(1+\alpha)^m} \left(1 + \frac{\alpha}{D}\right)^{N_b}. \tag{14}$$

Using (14) to bound $\Pr\{X > m\}$, we have

$$\Pr\{X > m\} = \sum_{t > m} \Pr\{X = t\} \tag{15}$$

$$\leq \left(1+\frac{\alpha}{D}\right)^{N_b} \sum_{t>m} (1+\alpha)^{-t} \tag{16}$$

$$= \left(1+\frac{\alpha}{D}\right)^{N_b} \frac{1}{(1+\alpha)^m} \sum_{t>0} (1+\alpha)^{-t} \tag{17}$$

$$= \left(1+\frac{\alpha}{D}\right)^{N_b} \frac{1}{\alpha(1+\alpha)^m}. \tag{18}$$

We know that $E[X_{max}] \geq N_b/D$ trivially. In order to conveniently manipulate the right hand side of (5), which will be a function of T and α, we parametrize T as $T = 1 + \rho N_b/D$, where $\rho > 1$ is chosen so that $\rho N_b/D$ is an integer. Plugging in this parametrization and simplifying (5) using (18), we have

$$E[X_{max}] \leq \left(\frac{\rho N_b}{D}+1\right) + \sum_{m>\rho N_b/D} D \cdot \Pr\{X > m\} \tag{19}$$

$$\leq \left(\frac{\rho N_b}{D}+1\right) + D \sum_{m>\rho N_b/D} \left(1+\frac{\alpha}{D}\right)^{N_b} \frac{1}{\alpha(1+\alpha)^m} \tag{20}$$

$$\leq \left(\frac{\rho N_b}{D}+1\right) + D \left(1+\frac{\alpha}{D}\right)^{N_b} \frac{1}{\alpha(1+\alpha)^{\rho N_b/D}} \sum_{m>0} \frac{1}{(1+\alpha)^m} \tag{21}$$

$$= \left(\frac{\rho N_b}{D}+1\right) + D \left(1+\frac{\alpha}{D}\right)^{N_b} \frac{1}{\alpha^2(1+\alpha)^{\rho N_b/D}}. \tag{22}$$

If the value of ρ is large enough, the second term of (22) will be negligible. Now we derive the smallest value ρ can take that still ensures that the second term in (22) is at most 1:

$$D \left(1+\frac{\alpha}{D}\right)^{N_b} \frac{1}{\alpha^2(1+\alpha)^{\rho N_b/D}} \leq 1.$$

Taking natural logarithms on both sides, we have

$$\ln D + N_b \ln\left(1+\frac{\alpha}{D}\right) - \rho\frac{N_b}{D}\ln(1+\alpha) - 2\ln\alpha \leq 0.$$

Rearranging terms, we get

$$\rho\frac{N_b}{D}\ln(1+\alpha) \geq N_b \ln\left(1+\frac{\alpha}{D}\right) + \ln D - 2\ln\alpha \tag{23}$$

$$\rho \geq \frac{D\ln\left(1+\frac{\alpha}{D}\right)}{\ln(1+\alpha)} + \frac{D\ln D}{N_b\ln(1+\alpha)} - \frac{2D\ln\alpha}{N_b\ln(1+\alpha)}. \tag{24}$$

We take ρ^* to be the smallest real number satisfying (24) such that $\rho^* N_b/D$ is integral. Replacing ρ by ρ^* in (22) and exploiting the fact that the second term in (22) is at most 1, we get

$$E[X_{max}] \leq \left(\frac{\rho^* N_b}{D}+1\right) + 1 \tag{25}$$

$$\leq \frac{\rho^* N_b}{D} + 2. \tag{26}$$

We are now ready to consider the different cases regarding the relationship between N_b and D. The basic idea is to choose different values of α for the different cases so as to optimize the bounds obtained using (26). Throughout this derivation we will make use of the bound $\ln(1+x) = x - x^2/2 + O(x^3)$ for bounded positive x.

Case 1: $N_b = kD$ for constant $k > 0$. In this case, we choose

$$\alpha = \frac{1/r}{\ln(1/r)} = \frac{(\ln D)/k}{\ln\ln D - \ln k}.$$

Substituting and simplifying, the first term in expression (24) can be written as

$$\frac{\ln D}{k(\ln\ln D)^2}\left(1 + O\left(\frac{\log\log\log D}{\log\log D}\right)\right),$$

and the second term in expression (24) can be written as

$$\frac{\ln D}{k\ln\ln D}\left(1 + \frac{\ln\ln\ln D}{\ln\ln D} + \frac{\ln k}{\ln\ln D} + O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right).$$

The third term of expression (24) is $O(1)$. Hence the bound on ρ^* can be written as

$$\rho^* = \frac{\ln D}{k\ln\ln D}\left(1 + \frac{\ln\ln\ln D}{\ln\ln D} + \frac{1+\ln k}{\ln\ln D} + O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right). \tag{27}$$

Using inequality (27) in inequality (26) and simplifying, we get

$$E[X_{max}] \le \frac{\ln D}{\ln\ln D}\left(1 + \frac{\ln\ln\ln D}{\ln\ln D} + \frac{1+\ln k}{\ln\ln D} + O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right). \tag{28}$$

This completes the proof of Case 1 of Theorem 2.

Case 2: $N_b = rD\ln D$ with $r = \Omega(1)$. In this case, we choose $\alpha = \sqrt{2/r}$. Since $\alpha/D = \sqrt{2/r}/D \to 0$ asymptotically as $D \to \infty$, it follows that $D\ln(1+\alpha/D) = \sqrt{2/r}(1 + O(\sqrt{2/r}/D))$.

Let us first consider the more interesting subcase when $r \to \infty, \alpha \to 0$. In this the first term on the right hand side of inequality (24) equals

$$\frac{D\ln\left(1+\frac{\alpha}{D}\right)}{\ln(1+\alpha)} = \frac{\sqrt{2/r}\left(1 + O\left(\frac{\sqrt{2/r}}{D}\right)\right)}{\sqrt{2/r} - 1/r + O(1/r\sqrt{r})} = 1 + \sqrt{\frac{1}{2r}} + O\left(\frac{1}{r} + \frac{1}{D\sqrt{r}}\right). \tag{29}$$

The second term on the right hand side of (24) equals

$$\frac{D\ln D}{N_b\ln(1+\alpha)} = \frac{1/r}{\sqrt{2/r} - 1/r + O(1/r\sqrt{r})} = \frac{1}{\sqrt{2r}} + O\left(\frac{1}{r}\right). \tag{30}$$

The third term of (24) is

$$\begin{aligned}
\frac{-2D\ln\alpha}{N_b\ln(1+\alpha)} &= \frac{-2\ln\alpha}{(r\ln D)\alpha(1+O(\alpha))} && (31)\\
&= \frac{\ln r - \ln 2}{r(\ln D)\sqrt{2/r}(1+O(1/\sqrt{r}))} && (32)\\
&< \frac{\ln r}{\sqrt{2r}(\ln D)(1+O(1/\sqrt{r}))} && (33)\\
&= \frac{\ln r}{\sqrt{2r}\ln D} + O\left(\frac{\log r}{r\log D}\right). && (34)
\end{aligned}$$

By (29),(30), and (34), we can choose ρ^* to be

$$\rho^* = 1 + \sqrt{\frac{2}{r}} + \frac{\ln r}{\sqrt{2r}\ln D} + O\left(\frac{1}{r} + \frac{\log r}{r\log D} + \frac{1}{D\sqrt{r}}\right). \tag{35}$$

For the bound on the expected maximum occupancy, applying (35) in inequality (26) and simplification yields

$$\begin{aligned} E[X_{max}] &\leq \left(1+\sqrt{\frac{2}{r}}+\frac{\ln r}{\sqrt{2r}\ln D}+O\left(\frac{1}{r}+\frac{\log r}{r\log D}+\frac{1}{D\sqrt{r}}\right)\right)\frac{N_b}{D}+2 \\ &\leq \left(1+\sqrt{\frac{2}{r}}+\frac{\ln r}{\sqrt{2r}\ln D}+O\left(\frac{1}{r}+\frac{\log r}{r\log D}+\frac{1}{D\sqrt{r}}\right)\right)\frac{N_b}{D} \end{aligned}$$

In the subcase when $r = k = \Theta(1)$ for some $k > 0$, it is easily seen that the right hand side of (24) depends on r but it is $O(1)$, and thus $\rho^* = O(1)$. Using $\rho^* \leq c'$, where c' is a positive constant, in inequality (26), we have

$$E[X_{max}] \leq c'\frac{N_b}{D}+2. \tag{36}$$

This completes the proof of Case 2 of Theorem 2.

□

7.3 Proof of Theorem 1

An intuitive way to look at the analysis of SRM is that the number of I/O read operations corresponding to a phase is the expected maximum occupancy of a dependent occupancy problem involving $N_b = R$ balls and D bins. Thus to compute the number of reads for the entire mergesort, we have to multiply the expected maximum occupancy by the total number of phases in the mergesort, which is the product of the number $\ln(N/M)/\ln R$ of passes over the file other than the initial run formation pass and the number N/RB of phases in a merge pass.

Using Lemmas 6 and 8, with the random variables I_0 and L'_i defined as in those lemmas, the random variable that bounds the number of I/O read operations during a merge of N' records during SRM is $I_0 + \sum_{1\leq i\leq J} L'_i$, where $J = \frac{N'/B-R}{R}$. The expected number of read operations in a merge is thus

$$E[I_0] + \sum_{1\leq i\leq J} E[L'_i]. \tag{37}$$

The random variable I_0 is the maximum occupancy of a classical occupancy problem involving R balls and D bins and each L'_i is the maximum occupancy of a dependent occupancy problem involving R balls and D bins.

Let us consider Case 1 of Theorem 1, in which we have $R = kD$ for constant $k > 0$. In terms of occupancies, this corresponds to Case 1 of Theorem 2, in which a total of $N_b = R = kD$ balls are thrown into D bins. From the bounds for Case 1 of Theorem 2 and Corollary 1, the expectations $E[I_0]$ and $E[L'_i]$ are all bounded by the right hand side of (28). Substituting (28) into (37) and simplifying, we find that the number of reads in a merge of N' records during SRM is given by

$$\begin{aligned} &(J+1)\left(\frac{\ln D}{\ln\ln D}\left(1+\frac{\ln\ln\ln D}{\ln\ln D}+\frac{1+\ln k}{\ln\ln D}+O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right)\right) \\ &= \frac{N'}{kDB}\cdot\frac{\ln D}{\ln\ln D}\left(1+\frac{\ln\ln\ln D}{\ln\ln D}+\frac{1+\ln k}{\ln\ln D}+O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right). \end{aligned} \tag{38}$$

In each pass over the file, the sum of the sizes of all the runs that are merged is $\sum N' = N$. Therefore by (38), the expected number of reads done by SRM in each pass on the file is

$$\frac{N}{kDB}\cdot\frac{\ln D}{\ln\ln D}\left(1+\frac{\ln\ln\ln D}{\ln\ln D}+\frac{1+\ln k}{\ln\ln D}+O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right). \tag{39}$$

There are $(\ln(N/M))/(\ln(kD))$ merge passes other than the initial run formation pass, which costs N/DB read operations. Thus, the expected number of read operations required to sort N using SRM in Case 1 of Theorem 1 is

$$\frac{N}{DB} + \frac{\ln(N/M)}{\ln(kD)} \frac{N}{DB} \frac{\ln D}{k \ln\ln D} \left(1 + \frac{\ln\ln\ln D}{\ln\ln D} + \frac{1+\ln k}{\ln\ln D} + O\left(\frac{(\log\log\log D)^2}{(\log\log D)^2}\right)\right).$$

Similarly, we can apply occupancy bounds from other cases of Theorem 2 and Corollary 1 to bound the expectations $E[I_0]$ and $E[L'_i]$, where $1 \le i \le J$, in Cases 2 and 3 of Theorem 1, and we get the desired expressions shown in the statement of Theorem 1. This completes the proof of Theorem 1.

8 A Deterministic Variant

As mentioned earlier in Section 3, when the input data records of the runs are randomly and uniformly distributed, we expect a version of our algorithm that uses a deterministic staggered distribution of starting blocks of runs on disks to give an average performance comparable to the bounds we showed in Theorem 1. Intuitively, this effect occurs because with long, random input runs, the R leading blocks of the R input runs at any time during the merge tend to be spread no worse than randomly among the D disks.

When the R input runs are staggered on the D disks, the first run has its first block on disk 0 with subsequent blocks cyclically stored on the disks, the next run has ts first block stored on disk 1, and so on. If the runs are "short enough," we can expect the runs to maintain their stagger on the disks *throughout* the duration of the merge on the average, ensuring very efficient I/O. On the other hand, as observed in the previous paragraph, even when runs are "long," we can expect I/O to be reasonably efficient. Hence, if an analysis of having runs staggered deterministically during initial merge stages (when runs are "short") is combined with an analysis that exploits the randomness in maximum occupancy situations during later merge stages (when runs are "long"), we might be able to obtain an improvement in overall I/O performance of the mergesort. We might thus be able to prove theoretical optimality (within a constant factor) for an increased range of values of M, B and D for our mergesort, compared with the range in Theorem 1.

9 Comparisons between SRM and DSM in Practice

In DSM, the popularly-used merging technique with disk striping, there are $O(M/DB)$ runs merged in each merge step with perfect read parallelism. The price that DSM pays for not using the disks independently is that it can merge only $\Theta(M/DB)$ rather than $\Theta(M/B)$ runs at a time. However, DSM is often more efficient in practice than the previously developed asymptotically optimal sorting algorithms, since the latter algorithms have larger overheads [VV96].

When the amount M of internal memory is small or the number D of disks is large, our SRM method is clearly superior to DSM. For example, if $M = \Theta(DB)$, DSM is suboptimal by a factor of $\Theta(\log D)$ whereas SRM is suboptimal by a factor of $\Theta(\log D/\log\log D)$. The improvement of SRM over DSM shows up in practice even when M is substantially larger or the number of disks D is small, which is when DSM has better I/O performance than previously developed external sorting algorithms.

In this section, we compare the performance of the DSM and SRM mergesorts on existing parallel disk systems. In Subsection 9.2 we make a comparison based on *an estimate of the*

upper bound on the expected worst-case number of I/O operations of an SRM mergesort with the exact number of I/O operations for the standard DSM mergesort with disk striping, using parameters of presently existing or feasible parallel disk systems. In Subsection 9.3 we make a similar comparison based on actual simulations of SRM to estimate its average-case I/O performance on random inputs. Both the comparisons show that SRM's I/O performance is always better than DSM's. Moreover, simulations indicate that SRM's actual I/O performance is much better than that implied by the estimate of the analytical bound in Subsection 9.2.

9.1 Expressions for the number of I/O operations

In this subsection and the following ones, we assume that SRM is able to merge $R = kD$ runs at a time, for interesting values of k and D. We consider DSM using the same amount of memory and estimate the relative performances of SRM and DSM.

We will first consider SRM, which merges $R = kD$ runs at a time. The amount of internal memory needed to support the merging is $M = (2kD + 4D)B + kD^2$. The total number of writes SRM requires is $\frac{N}{DB}(1 + \frac{\ln(N/M)}{\ln(kD)})$ since it has perfect write parallelism. In terms of reads, SRM requires N/DB reads corresponding to the initial run formation pass. It subsequently requires $\frac{\ln(N/M)}{\ln(kD)}$ passes to perform the merging, each incurring $v\frac{N}{DB}$ reads, where $v = v(k, D)$ is the *overhead factor* that represents a multiplicative overhead over and above the minimum number $\frac{N}{DB}$ of reads for a single pass. The total number of I/O operations SRM takes to sort N records is thus

$$\frac{N}{DB}\left(2 + \frac{\ln(N/M)}{\ln(kD)}(1 + v)\right) = \frac{N}{DB}\left(2 + C_{SRM} \ln \frac{N}{M}\right),$$

where

$$C_{SRM} = \frac{1 + v}{\ln(kD)}. \tag{40}$$

Now let us compute in a similar way the number of I/O operations needed by DSM to sort N records using the same amount of memory as SRM above. We assume that DSM uses $2D$ blocks per run for I/O read buffers and $2D$ blocks for I/O write buffers. DSM merges $(M/B - 2D)/2D = k + 1 + kD/2B$ runs at a time. It can be verified that the number of I/O operations required by DSM to sort N records is

$$\frac{N}{DB}\left(2 + 2\frac{\ln(N/M)}{\ln(k + 1 + kD/2B)}\right) = \frac{N}{DB}\left(2 + C_{DSM} \ln \frac{N}{M}\right),$$

where

$$C_{DSM} = \frac{2}{\ln(k + 1 + kD/2B)}. \tag{41}$$

9.2 Comparison based on expected worst-case performance of SRM

Our goal in this subsection is to compare SRM and DSM based on the *expected worst-case performance* of SRM. Our comparison uses k and D values corresponding to realistic systems. To make the comparison possible, we need to obtain an estimate of the overhead factor $v(k, D)$ defined in the previous subsection corresponding to these values of k and D.

Note that for a pair of given finite values of k and D, the estimate of the expected worst-case value of v using Theorem 1 is lax because of contributions from lower-order terms. As explained in Section 7.3, the expected number of reads done by SRM is bounded by an expression involving expected maximum occupancies of the classical and dependent

	$D = 5$	$D = 10$	$D = 50$	$D = 100$	$D = 1000$
$k = 5$	1.6	1.7	2.2	2.3	2.7
$k = 10$	1.4	1.5	1.8	1.9	2.2
$k = 20$	1.3	1.4	1.5	1.6	1.8
$k = 50$	1.2	1.2	1.3	1.4	1.5
$k = 100$	1.11	1.16	1.22	1.26	1.3
$k = 1000$	1.04	1.05	1.08	1.08	1.1

Table 1: The overhead $v(k, D)$, computed by estimating $\mathcal{C}(kD, D)/k$ using computer simulations.

cases. We get more useful comparisons between SRM and DSM by replacing the maximum dependent occupancy by the maximum classical occupancy in the bound for the expected number of reads required by SRM. This approach can be justified on two counts. First, the upper bound in Theorem 2 that we proved for the expected maximum dependent occupancy is the same as the expression for the expected maximum of the classical occupancy problem. Moreover, we conjecture that expected classical maximum occupancy is never less than the expected maximum occupancy of dependent occupancy problems involving the same number of balls and bins, as explained in Section 7.2.

For given values of k and D, we simulate throwing kD balls into D bins repeatedly to estimate $\mathcal{C}(kD, D)$. Thus the expected worst-case overhead $v(k, D)$ is estimated by repeated ball-throwing experiments to estimate $\frac{\mathcal{C}(kD,D)}{k}$. Table 1 shows the values of v based on such experiments.

For every k, D pair considered, we use (40) to estimate an expected worst-case value for C_{SRM} using the abovementioned estimates for v. In Table 2 we present the C_{SRM}/C_{DSM} ratio for several values of k and D. The ratio C_{SRM}/C_{DSM} represents the relative I/O advantage of SRM over DSM, neglecting the $2N/DB$ I/Os that both methods require during the initial run formation. We used block size $B = 1000$ records for all k, D pairs. (The choice of B is not significant so long as it is reasonable.) In our representation, $2k$ is roughly the number of internal memory blocks available per disk.

Table 2 shows that the SRM uses signficantly fewer I/Os than does DSM. For example, for $D = 50$, $k = 100$, which translates into $M = 10.5$ million records of internal memory, SRM uses 0.60 times as many I/Os as does the slower DSM, not counting the initial run formation pass that they share in common. When D is small, as k increases relative to D, the C_{SRM}/C_{DSM} ratio gradually increases toward 1, which indicates a lessening advantage of SRM over DSM when there are few disks and a huge amount of internal memory. As we show in the next subsection, since we overestimate the number of reads required by SRM in our analysis, SRM actually exhibits even better performance than indicated in Tables 1 and 2.

9.3 Using simulations to count SRM's I/O operations

In the previous subsection we showed by using classical maximum occupancy to estimate the overhead term v that SRM performs well compared to DSM on arbitrary inputs. In this section we use *simulations of the algorithm itself* to estimate v on *average-case inputs* and make a similar comparison.

Our experiments consist of simulating SRM while merging $R = kD$ sorted runs, and each of length L, for a large range of k and D values. The runs input to the merge were generated such that *each set of input runs was equally likely*. SRM's actions while merging

	$D = 5$	$D = 10$	$D = 50$	$D = 100$	$D = 1000$
$k = 5$	0.71	0.62	0.51	0.48	0.46
$k = 10$	0.72	0.66	0.54	0.50	0.48
$k = 20$	0.75	0.68	0.56	0.53	0.49
$k = 50$	0.77	0.71	0.59	0.55	0.50
$k = 100$	0.78	0.72	0.61	0.57	0.51
$k = 1000$	0.83	0.77	0.67	0.63	0.56

Table 2: The performance ratio C_{SRM}/C_{DSM} for memory size $M = (2k + 4)DB + kD^2$, with block size $B = 1000$. (Both M and B are expressed in units of records.) The overhead factor v in C_{SRM} is based on computer simulations of $\mathcal{C}(kd, D)/k$.

	$D = 5$	$D = 10$	$D = 50$
$k = 5$	1.0	1.0	1.2
$k = 10$	1.00	1.0	1.1
$k = 50$	1.00	1.00	1.00

Table 3: The overhead factor $v(k, D)$ for memory size $M = (2k + 4)DB + kD^2$ obtained from simulations.

depend only on the relative order of the input keys. Thus, there is an obvious one-to-one correspondence between the set of all possible input runs to the merge and the set of partitions of the set $I = \{1, 2, \ldots, LkD\}$, each partition splitting I into kD disjoint subsets of size L. We generate average-case inputs to the merge by generating partitions of the set I, with each partition being equally likely.

Our simulations indicate that SRM's I/O overhead is noticeable only when k is small compared with the number of disks D. We ran our simulations for several different sets of parameters. Not only did we vary k and D, but for each k, D pair we also varied B and L (where L is as defined above). We present here an illustrative sample of typical simulation outcomes, with our main focus on the parameters k and D. Table 3 shows the multiplicative overhead v corresponding to simulations for interesting k, D pairs. It can be seen that the estimates of v based on average-case inputs are smaller than the corresponding estimates of the expectation of worst-case values of v in Table 1. (For values of k larger than the ones shown here, the simulations gave values v that were practically equal to 1.) The simulations show in the average-case that SRM has little or no overhead when k is reasonably large and that the number of I/O read operations required to carry out the merge is practically N'/DB, where $N' = LkD$ is the number of records in the merged output. In our simulations, N' was always 1000 times bigger than kDB. Longer simulations tend to be time consuming.

The average-case simulations indicate when k is reasonably large that there is little or no flushing of blocks by SRM. Intuitively, the rate of consumption (due to merging) of the blocks in internal memory is such that there is almost always space to read in the next "smallest" D blocks without the need to flush.

In order to make a similar comparison as the previous subsection, but based on results of simulations of the algorithm itself, we computed the term C'_{SRM} analogously to C_{SRM} using (40), by using the average-case values for v obtained from simulations of the algorithm.

Table 4 below shows the C'_{SRM}/C_{DSM} ratios so obtained, for various k, D pairs. We note that the entries in Table 4 are smaller than the corresponding entries in Table 2, indicating that SRM's performance is indeed better than that implied by the more pessimistic upper

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

Table 4: The performance ratio C'_{SRM}/C_{DSM} for memory size $M = (2k + 4)DB + kD^2$ where C'_{SRM} is computed using the overhead value $v(k, D)$ obtained from simulations.

bound. Moreover, SRM's low overhead in the average-case indicates that for all practical purposes it is an optimal external sorting algorithm.

10 Realistic Values for Parameters *k*, *D*, and *B*

We have shown in the previous section for a wide range of k, D, and B values that SRM performs very efficiently. Since not every k, D, B triplet corresponds to that of a realistic machine, we try in this section to obtain a crude approximation of the relative magnitudes of k, D, and B in typical computers. We argue that k is generally much larger than D in realistic machines, which implies that SRM is the method of choice on such machines, still noticeably faster than DSM.

In our terminology, where the internal memory size is $M = (2kD+4D)B+kD^2$ and the merge order is $R = kD$, the expression $2kD$ is roughly equal to M/B, the number of blocks that fit in main memory, under the realistic assumption that $D = O(B)$. Let us consider a fast, uniprocessor workstation attached to, say, $D = 5$ independent disks for parallel I/O. We are likely to find internal memories of the order of 100 megabytes on such machines and disk block or track sizes of the order of 10–50 kilobytes. This would mean that k may be on the order of 200–1000 when $D = 5$ on such workstations. If the same amount of main memory is used with instead 10 disks and 50-kilobyte disk blocks, k would be on the order of 100. On the other hand when the number of disks is relatively high as in large-scale multiprocessor computing systems, we would still expect k to be large, even after factoring in the increased block sizes that such machines have. This is primarily because large-scale computing systems tend to have huge internal memories. For instance, in a system with 100 parallel disks and 100-kilobyte disk blocks, one would expect on the order of at least 5–10 gigabytes or more of aggregate internal memory. This would correspond to k being on the order of 500–1000 or more. The relative magnitudes of k, D, and B cited here are meant to represent likely scenarios; it is conceivable that there are systems with different relationships among the values of k, D, and B.

For most values of of k, D, and B, the previous section shows that SRM is extremely efficient. Looking back at the occupancy analysis, the reason for SRM's efficiency is that throwing a large number (say, kD) of balls uniformly and independently into a small number of bins (say, D) does result in a more or less balanced distribution. Even DSM will perform well when D is small and k is large, since it will merge k runs at a time, which is not much worse than merging the optimal kD runs at a time. However, SRM's extremely low overhead still gives it an advantage over DSM.

11 Conclusions

In this paper, we presented a simple, efficient mergesort algorithm for parallel disks that makes a limited use of randomization. The analysis of the I/O performance involved a

reduction to certain maximum occupancy problems. We demonstrated the practical merit of the algorithm by showing that it incurs fewer I/O operations than the commonly used disk-striped mergesort, even on realistic parallel disk systems with a small number of disks. We did so analytically by estimating the maximum bucket occupancy values for several k, D pairs and empirically by using simulations to count the number of I/O operations needed by SRM in the average case. We argued that SRM may be considered an optimal external sorting algorithm in practice. The technique of staggering runs might yield further gains in practice if combined with our general approach.

References

[AP94] Alok Aggarwal and C. Greg Plaxton. Optimal parallel sorting in multi-level storage. *Proc. Fifth Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 659–668, 1994.

[AV88] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[BGV96] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Simple randomized mergesort on parallel disks. *Proc. of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 109–118, 1996.

[Flo72] R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 105–109. Plenum, 1972.

[GK81] Daniel H. Greene and Donald E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhäuser, Boston., 1981.

[GS84] D. Gifford and A. Spector. The TWA reservation system. *Comm. ACM*, 27(7):650–665, July 1984.

[HGK+94] L. Hellerstein, G. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2–3), 1994.

[HK81] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. *Proc. 13th Annual ACM Symp. on Theory of Computation*, pages 326–333, may 1981.

[Knu73] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[KSC78] V. F. Kolchin, B. A. Sevastyanov, and V. P. Chistyakov. *Random Allocations*. Winston & Sons, Washington, 1978.

[LV85] E. E. Lindstrom and J. S. Vitter. The design and analysis of bucketsort for bubble memory secondary storage. *IEEE Transactions on Computers*, C-34:218–233, March 1985.

[Mag87] N. B. Maginnis. Store more, spend less: Mid-range options abound. *Computerworld*, pages 71–82, November 1987.

[NV90] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, 1990.

[NV93] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. *Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). *Proc. 1988 ACM-SIGMOD Conf. on Management of Data*, pages 109–116, 1988.

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

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, pages 17–28, March 1994.

[Uni89] University of California at Berkeley. *Massive Information Storage, Management, and Use (NSF Institutional Infrastructure Proposal)*, January 1989. Technical Report No. UCB/CSD 89/493.

[VF90] J. S. Vitter and Ph. Flajolet. Average-case analysis of algorithms and data structures. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter 9, pages 431–524. North-Holland, 1990.

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

[VV96] D. E. Vengroff and J. S. Vitter. I/O-efficient computation: The tpie approach. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, September 1996.