6-1-1999

Fast Out-of-Core Sorting on Parallel Disk Systems

Matthew D. Pearson
Dartmouth College

Dartmouth College Computer Science Technical Report PCS-TR99-351

Fast Out-of-Core Sorting on Parallel Disk Systems

Matthew D. Pearson
mdp@alum.dartmouth.org

Dartmouth College Department of Computer Science
Hanover, NH 03755

Abstract

This paper discusses our implementation of Rajasekaran's (l,m)-mergesort algorithm (LMM) for sorting on parallel disks. LMM is asymptotically optimal for large problems and has the additional advantage of a low constant in its I/O complexity. Our implementation is written in C using the ViC* I/O API for parallel disk systems.

We compare the performance of LMM to that of the C library function `qsort` on a DEC Alpha server. `qsort` makes a good benchmark because it is fast and performs comparatively well under demand paging. Since `qsort` fails when the swap disk fills up, we can only compare these algorithms on a limited range of inputs. Still, on most out-of-core problems, our implementation of LMM runs between 1.5 and 1.9 times faster than `qsort`, with the gap widening with increasing problem size.

1. Introduction

Researchers in many fields often wish to solve problems that are too large to fit into main memory, but traditional in-core methods are generally unable to handle very large sets of data. Performance suffers due to excessive demand paging; worse, many in-core implementations simply crash when the data is larger than available swap space.

Improving the performance of out-of-core sorting is an important goal. Sorting is a fundamental problem and is a key component in many algorithms. While recursive in-core methods such as `qsort` fare reasonably well under demand paging (as they eventually reduce the problem size to one that fits in-core), they still are limited by available swap space. Sorting algorithms designed for out-of-core situations, because they minimize I/O operations, can run considerably faster than in-core algorithms pushed past memory limits. Furthermore, when such algorithms are implemented to run on parallel disk systems, they can handle much larger problems. This paper examines the implementation and performance of one such algorithm, Rajasekaran's (l,m)-mergesort (LMM) [3].

The remainder of this paper is organized as follows. In Section 2, we describe Vitter and Shriver's Parallel Disk Model (PDM) [4], under which LMM is asymptotically optimal for large problems. Section 3 presents a theoretical outline of LMM. Section 4 is an in-depth discussion of our implementation of the algorithm where we focus on practical matters of implementation. Test results from a DEC workstation are presented in Section 5. Section 6 presents some suggestions on altering LMM to work faster and more efficiently. We conclude the paper in Section 7.

2. The Parallel Disk Model

The Parallel Disk Model, or PDM, is an abstraction under which many out-of-core algorithms such as LMM are designed. Imagine a problem with N records on a computer with D disks and a random-access memory capable of holding M records. The records are distributed evenly across the disks $\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{D-1}$, such that each disk contains N / D records. The records on each disk are arranged in blocks of B records each,[1] and blocks are striped across the disks. Each stripe consists of D blocks, or BD records. We assume for simplicity that N, M, D, and B are all exact powers of 2. If N is 64, D is 8, and B is 2, the records are laid out as in Figure 1. A record with index x is located on stripe $\lfloor x / BD \rfloor$ of disk $\mathcal{D}_{\lfloor (x \bmod B) / D \rfloor}$.

	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	$\mathcal{D}_5$	$\mathcal{D}_6$	$\mathcal{D}_7$
stripe 0	0 1	2 3	4 5	6 7	8 9	10 11	12 13	14 15
stripe 1	16 17	18 19	20 21	22 23	24 25	26 27	28 29	30 31
stripe 2	32 33	34 35	36 37	38 39	40 41	42 43	44 45	46 47
stripe 3	48 49	50 51	52 53	54 55	56 57	58 59	60 61	62 63

Figure 1. PDM layout of records for $N = 64$, $D = 8$, $B = 2$. Each rectangle delimits one block. Numbers indicate the index of each record.

A *parallel I/O operation* can move up to D blocks between the disks and main memory, with no more than one block transferred per device. The most general form of I/O is *independent I/O*, where the blocks in a given I/O operation can be in any stripe on their respective disks. Another common form of I/O is called *striped I/O*, where the blocks accessed must be in the same stripe on each device.

We make two assumptions concerning memory. Since a parallel I/O operation can move up to BD records, we assume that main memory is large enough to hold this data: $M \geq BD$. We also make the assumption that $N > M$ to ensure that the problem is truly out-of-core. PDM convention uses lower-case letters to indicate the

1. These 'logical blocks' might consist of one or several sectors of a physical device, or even several physical devices in the case of a RAID.

base-2 logarithm of the corresponding capital letter; in this paper, we use b as shorthand for $\log_2(B)$ and set $d = \log_2(D)$.

Note that the PDM ignores the variations in disk-access times caused by latency and head travel. While at first this may seem a serious omission, recall that the RAM model glosses over performance-affecting factors such as secondary cache, yet still provides a reasonable framework for predicting an algorithm's efficiency.

In the RAM model, an algorithm's performance is measured by the number of primitive operations needed to solve a problem of size N. The PDM was designed to analyze out-of-core problems, however, and uses a different metric. Because I/O is the limiting factor in out-of-core problems, under the PDM an algorithm is assessed by the number of parallel I/O operations it requires.

3. How LMM works

LMM is essentially an extension of merge-sort, but instead of merging 2 sequences at a time, we merge l sequences at a time. Suppose we have l sorted sequences $U_1, U_2, \ldots, U_l$ of equal length: for $1 \le i \le l$, $U_i = u_i^1, u_i^2, \ldots, u_i^r$, where r is some arbitrary constant. First, break each sequence into m parts by unshuffling; in this way, U_i becomes $U_i^1, U_i^2, \ldots, U_i^m$. The specific sequence U_i^j, where $1 \le j \le m$, consists of the elements $u_i^j, u_i^{j+m}, u_i^{j+2m}, \ldots$. After all l sequences are unshuffled, there are lm subsequences, each of which is sorted. It's like dealing a deck of cards out to a group of friends — if the deck at first is sorted, then each friend's hand will be sorted.

Next, for $1 \le j \le m$, recursively merge $U_1^j, U_2^j, \ldots, U_l^j$ to create the sorted sequence $X_j = x_j^1, x_j^2, \ldots$. Each sequence X has a very similar distribution of numbers, which results in some interesting properties. Consider the nth element of X_i. It is greater than or equal than the nth element in sequence X_j where $1 \le j \le i$. This property, coupled with x_i^n being greater than or equal to any x_i^k, $1 \le k \le n$, means that the data as laid out in Figure 2 is sorted both horizontally and vertically.

$$
\begin{array}{cccccc}
x_1^1 & x_1^2 & x_1^3 & x_1^4 & x_1^5 & \ldots \\
x_2^1 & x_2^2 & x_2^3 & x_2^4 & x_2^5 & \ldots \\
x_3^1 & x_3^2 & x_3^3 & x_3^4 & x_3^5 & \ldots \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
x_m^1 & x_m^2 & x_m^3 & x_m^4 & x_m^5 & \ldots
\end{array}
$$

Figure 2. Table of sequences $X_1, \ldots, X_m$. When laid out in this way, rows and columns are in increasing order left-to-right and top-to-bottom. Each row is a sequence X_i; each consecutive group of m columns forms a sequence Z_i.

The similar distribution of elements in the m sequences makes merging them quite a simple task. We shuffle the elements of the X_i's together to form the sequence $Z = x_1{}^1, x_2{}^1, \ldots, x_m{}^1, x_1{}^2, x_2{}^2, \ldots, x_m{}^2, \ldots$. Because the X_i's have such similar distribution of elements, Z will be "mostly sorted." That is, all of its elements will be very close to where they are supposed to be. In fact, it can be shown that every element in Z will be no more than lm spaces away from its proper place. There are many ways to perform the cleanup operation. One way is to partition Z into segments of length lm each, called Z_1, Z_2, Z_3, ... Sort each of the Z_i's, then merge Z_1 with Z_2, Z_3 with Z_4, etc. Finally, merge Z_2 with Z_3, Z_4 with Z_5, and so on. We discuss a more efficient way to clean up in our discussion of implementation in Section 4.

When we discuss LMM in the remainder of the paper, it will be helpful to refer to steps of the algorithm by number. Here, then, is a recipe for sorting:

To merge l sorted runs of equal length,

1. *Unshuffle.* Unshuffle each run U_i, for $1 \le i \le l$, into m parts, and call them U_i^j for $1 \le j \le m$.
2. *Recursive merge.* For each i, recursively merge U_i^1, U_i^2, ..., U_i^m, and call the result X_i. If all m subsequences fit in memory, use a base-case algorithm.
3. *Shuffle-merge.* Shuffle the elements of the X sequences together again.
4. *Clean up.* Compare each element with its neighbors and shift its position by up to lm places if necessary.

4. Implementation

Our implementation of LMM, `slmm` (for *S*imple *LMM*), is written in C and uses the ViC* API for I/O [1]. The ViC* API is a portable interface that implements the PDM abstraction for parallel I/O operations. It supports many architectures and file systems; our implementation is built atop UFS, the traditional UNIX File System.

Our goal was an implementation of LMM that was fast and had efficient I/O patterns. The top-down approach of LMM described in Section 3 recurses into many subproblems which can spread out over the disks and require large amounts of head travel. To avoid recursion on the call stack and to provide more efficient accesses to the disks, we work bottom-up, starting with small sorted runs and merging them together repeatedly. Data is laid out on the disks in stripe-major order, like the data pictured in Figure 1. We set $l = m$ for simplicity; this way, merge and unshuffle operations mirror one another. To take maximum advantage of disk parallelism, we set $m = D$ whenever possible. (Occasionally we will need to merge fewer than D sorted runs, in which cases we simply use a smaller value of m.) `slmm` bottoms out with a base problem size of $F = BD^q$, where q is the greatest integer

such that $BD^q \leq M$; that is, $q = \lfloor(\log_2(M) - b) / d\rfloor$. Since there are BD records per stripe, our base case consists of D^{q-1} stripes' worth of data.

Since the operation of slmm is complex, we will supplement the description of our implementation with a running example. Imagine a computer with $D = 4$ disks and $B = 4$ records per block. To keep the description simple, we set $M = BD$, so that $q = 1$ and $F = 8$. Suppose we want to sort the following $N = 128$ integers:

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	1	8	8	7	7	1	5	7	3	7	1	2	4	6	7	7
stripe 1	7	4	9	7	4	9	6	9	8	2	9	6	4	8	9	6
stripe 2	9	1	5	9	9	5	9	0	9	9	1	0	9	5	0	9
stripe 3	2	1	1	2	3	5	2	1	1	0	3	2	1	3	0	1
stripe 4	8	2	1	7	2	1	3	4	4	3	2	1	1	9	3	5
stripe 5	7	5	5	7	5	6	5	7	6	5	7	5	5	6	5	5
stripe 6	8	5	6	3	8	4	6	8	6	5	4	9	5	6	5	5
stripe 7	2	0	1	0	0	2	2	0	0	1	2	2	2	0	2	2

Figure 3. 128 integers, arbitrarily arranged.

On the first pass over the data, we read F records at a time and sort them in core. This procedure is easy to do using striped I/O and any in-core sorting algorithm; slmm uses qsort for in-core sorting. The initial sort pass yields N/F sorted sequences. In our example, we end up with 8 of them. Since m is limited by the number of disks, we can merge only 4 at a time. Thus, we break the problem into 2 subproblems, each of which contains 4 sorted sequences. We call these groups of D sequences "run-sets"; each iteration of slmm takes s run-sets each with D sequences as input, and produces s sequences D times longer as output. Sometimes we need to merge fewer than D sequences; in that case we have one run-set containing m sequences. What we'll need to do in our example is run slmm twice, once with $s = 2$ to form two sequences of length 64 from eight of length 16, and once more with $s = 1$ to form one sequence of length 128 from those two 64-element sequences.

Speaking more generally, after the first sorting pass we end up with $\lceil N/FD\rceil$ run-sets containing D sorted sequences each; each one of these run-sets is a list of sorted sequences $U_1, \ldots, U_D$, following our notation of Section 3. When $FD \geq N$, we have just one run-set containing $m \leq D$ sequences. Since N is a power of 2, the number of run-sets, and sequences within each run-set, must always be a power of 2. Below, we show our example after the initial sorting pass. The first run-set con-

sists of sequences occupying stripes 0 – 3, and the second contains the sequences on stripes 4 – 7. The first element of each sequence is written in boldface.

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	**1**	1	1	2	3	4	6	6	7	7	7	7	7	7	8	8
stripe 1	**2**	4	4	4	6	6	6	7	7	8	8	9	9	9	9	9
stripe 2	**0**	0	0	1	1	5	5	5	9	9	9	9	9	9	9	9
stripe 3	**0**	0	1	1	1	1	1	1	2	2	2	2	3	3	3	5
stripe 4	**1**	1	1	1	2	2	2	3	3	3	4	4	5	7	8	9
stripe 5	**5**	5	5	5	5	5	5	5	5	6	6	6	7	7	7	7
stripe 6	**3**	4	4	5	5	5	5	5	6	6	6	6	8	8	8	9
stripe 7	**0**	0	0	0	0	0	1	1	2	2	2	2	2	2	2	2

Figure 4. Sequence of numbers after the initial in-core sort pass.

Unshuffling

The idea behind the unshuffle pass is to break up each sequence into m parts, so that later when we merge them back together we have a nice, even distribution of numbers. In the case of our example, we want to unshuffle the elements of each sequence U_i so that 4 of them are part of each sequence X_i after merging. U_1 provides the first 4 elements of each sequence to be merged in the base case, U_2 the next 4, and U_m the last 4 of each. U_1 contains the elements with indices 0 – 15, which should be permuted to indices 0 – 3, 16 – 19, 32 – 35, 48 – 51 (these numbers are the first 4 indices of each X_i). From the unshuffling discussion in Section 3 we know the permute should be done in this way: 0→0, 1→16, 2→32, 3→48, 4→1, 5→17, 6→33, 7→49, …. A simple and fast way to unshuffle them uses the binary representation of the source and destination indices. Look at the last two bits of the source indices: 0, 4, 8, and 12 all end in 00; 1, 5, 9, and 13 all end in 01. The other even indices end in 10, and the remaining odds end in 11. If you look at the *first* two bits of the destination indices[2], 0 – 3 start with 00, 16 – 19 begin with 01, 32 – 35 start with 10, and 48 – 51 start with 11. Thus we can map source to destination indices by promoting the 2 least significant bits to the most significant bits, and shifting the rest right by 2.

In the general case, we can implement the unshuffle operation by a $\log_2(m)$ right barrel shift of each element's index. Since our parameters are all powers of 2, it is

2. In the case of our example, we express the indices using 6 bits because each run-set has 64 elements and $\log_2(64) = 6$.

easy to break the bit-wise representation of the index into sections and shift them around. As each sorted subsequence is $F = BD^q$ records long, the least significant $b + qd$ bits of each element's index represent its position within the sequence. The next $\log_2(m)$ bits represent the sequence's position relative to the others in its run-set, and the bits higher than that mark its position within the problem as a whole. Since we are breaking the sequence into m parts, but wish to keep it within its run-set, we only alter the lower $\log_2(m) + b + qd$ bits. Our index permutation is as follows, where bit 0 is the least significant index bit.

For each bit x from 0 to the most significant bit, move x to position $f(x)$, where

$$f(x) = \begin{cases} x + b + d(q + c) & \text{if } 0 \le x < \log_2(m) \\ x - \log_2(m) & \text{if } \log_2(m) \le x < \log_2(m) + b + d(q + c) \\ x & \text{if } \log_2(m) + b + d(q + c) \le x \end{cases}$$

(the value c is explained later when we discuss recursion; here, $c = 0$.)

Our running example does not illustrate very well how the unshuffle step works, since the values are so small and simple. In a more complicated situation, say where $b = 4$, $d = 3$, $q = 2$, $m = 4$, a 16-bit address with bits `a` – `p` is permuted as follows:

`abcd efgh ijkl mnop` $\Rightarrow$ `abcd opef ghij klmn`

The unshuffle pass, then, could be done by reading in one sequence at a time, permuting its elements' indices, and writing the data out. Since a sequence can be of arbitrary length this isn't a good way to unshuffle, because in later iterations main memory may not be large enough to contain a full sequence. Our implementation passes over the data one stripe at a time, reading in BD records of a sequence, unshuffling them in-core, and then writing them out. Note that `slmm` may read from one section of the data and write to a different location altogether. The bottom b bits represent an element's position within a data block, the next d bits determine which disk it is on, and the upper bits are the stripe number. In the above example, before the permute, the element in question was located on block [`abcdefghi`] of disk [`jkl`]; afterwards it moves to disk [`hij`], block [`abcdopefg`]. Whenever `e` is 0 and `o` is 1, this element permutes to a higher location on the disks. Thus, if we blindly wrote out data as we permuted it, we would overwrite blocks we have not yet read. By writing to a different file, we avoid overwriting data we need later. Using this scratch file, however, doubles the disk space needed for sorting.

Figure 5 shows how our example looks after the unshuffle pass. We again mark the first element of each sequence in bold type; we use subscripts to represent the elements' indices before they were unshuffled.

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	$\mathbf{1}_0$	3_4	7_8	7_{12}	2_{16}	6_{20}	7_{24}	9_{28}	0_{32}	1_{36}	9_{40}	9_{44}	0_{48}	1_{52}	2_{56}	3_{60}
stripe 1	0_{49}	1_{53}	2_{57}	3_{61}	$\mathbf{1}_1$	4_5	7_9	7_{13}	4_{17}	6_{21}	8_{25}	9_{29}	0_{33}	5_{37}	9_{41}	9_{45}
stripe 2	0_{34}	5_{38}	9_{42}	9_{46}	1_{50}	1_{54}	2_{58}	3_{62}	$\mathbf{1}_2$	6_6	7_{10}	8_{14}	4_{18}	6_{22}	8_{26}	9_{30}
stripe 3	4_{19}	7_{23}	9_{27}	9_{31}	1_{35}	5_{39}	9_{43}	9_{47}	1_{51}	1_{55}	2_{59}	5_{63}	$\mathbf{2}_3$	6_7	7_{11}	8_{15}
stripe 4	$\mathbf{1}_{64}$	2_{68}	3_{72}	5_{76}	5_{80}	5_{84}	5_{88}	7_{92}	3_{96}	5_{100}	6_{104}	8_{108}	0_{112}	0_{116}	2_{120}	2_{124}
stripe 5	0_{113}	0_{117}	2_{121}	2_{125}	$\mathbf{1}_{65}$	2_{69}	3_{73}	7_{77}	5_{81}	5_{85}	6_{89}	7_{93}	4_{97}	5_{101}	6_{105}	8_{109}
stripe 6	4_{98}	5_{102}	6_{106}	8_{110}	0_{114}	1_{118}	2_{122}	2_{126}	$\mathbf{1}_{66}$	2_{70}	4_{74}	8_{78}	5_{82}	5_{86}	6_{90}	7_{94}
stripe 7	5_{83}	5_{87}	6_{91}	7_{95}	5_{99}	5_{103}	6_{107}	9_{111}	0_{115}	1_{119}	2_{123}	2_{127}	$\mathbf{1}_{67}$	3_{71}	4_{75}	9_{79}

Figure 5. After the unshuffle pass. Boldface numbers mark the beginning of each sequence; subscripted numbers mark elements' original indices. The contents of each block are sorted left-to-right, but runs across adjacent blocks may not be.

After the unshuffle pass the sequences start on different disks. This staggering is not part of the unshuffle function described above, but is fairly straightforward to implement. We want to have each sequence start on a different disk so that the merge step can access the disks more efficiently. When a sequence hits the end of $\mathcal{D}_{D-1}$, it wraps around to the same block of $\mathcal{D}_0$. That is, indices are wrapped modulo F. We'll return to this point after we discuss the base merge pass.

Merging in the Base Case

Step 2 is the recursive step, but since this is the first iteration of the algorithm we use a base-case operation here. We want to create m sorted runs in each run-set that are properly prepared for the shuffle-merge step. This task can be done in-core, since the end result of the operation is a run of length F and $F \leq M$. The unshuffle step distributed the elements nicely in our example so that the elements that make up a specific X_i are all in one stripe. For larger values of q, the elements to be merged in the base case are all located in D^{q-1} contiguous stripes. The easiest way to merge the base case is to read in F stripes at a time and sort the data in core, then write it out, keeping track of which disk the sequence started on. In our example where B is small this isn't a bad way to do it. But for real-life applications where $B = 8$ kilobytes or so, and where $q > 1$, this is awfully inefficient.

Our implementation reads in F records, does an m-way merge in-core, then writes out the data. This can be done in-place with regard to the file system because the unshuffle step wrote out data to its proper sequence (the base merge simply rearranges that data). This locality holds for $q > 1$ as well. In Figure 6 we show the

example data after the base merge. We now have m sorted sequences in each run-set, and those sequences have an even distribution of numbers, a property we will use to our advantage in the next step.

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	**0**	0	1	1	1	2	2	3	3	6	7	7	7	9	9	9
stripe 1	8	9	9	9	**0**	0	1	1	2	3	4	4	6	6	7	7
stripe 2	6	6	7	8	8	9	9	9	**0**	1	1	1	2	3	4	5
stripe 3	2	4	5	5	6	7	7	8	9	9	9	9	**1**	1	1	2
stripe 4	**0**	0	1	2	2	2	3	3	5	5	5	5	5	6	7	8
stripe 5	6	7	7	8	**0**	0	1	2	2	2	3	4	5	5	5	6
stripe 6	5	5	5	6	6	7	8	8	**0**	1	1	2	2	2	4	4
stripe 7	2	3	4	5	5	5	5	6	6	7	9	9	**0**	1	1	2

Figure 6. After the base-case merge, the elements are distributed evenly as discussed in Section 3. Notice how the vertical ordering is skewed diagonally by the need to offset each sequence in preparation for the shuffle-merge step.

Shuffle-Merging and Cleanup

For step 3, we want to take the first BD / m elements of each sequence in the run set and merge them together, and write them out as Z_1. The next BD / m elements from each sequence are then read in, merged, and written out as Z_2, and so on. Since $1 < m \leq D$, we know $BD > BD / m \geq B$, which means we read data in blocks, not stripes. Reading data from different areas of the disks in units smaller than a stripe calls for independent I/O. In the case where $m = D$, such as in our example, the merge reads one block at a time from each sequence. In other cases where $m < D$, we read D / m blocks from each sequence, so that in any case we read BD blocks into memory at a time. These many independent I/O accesses to different sequences is why we stagger the runs across the disks, because if they were not distributed in this fashion, we would have to read D blocks from one disk while the others sat unused. Staggering lets us read one block each from the D disks, which speeds up the merge considerably.

Our implementation does not write out Z_i until Z_{i+1} has been read in and merged. When we start the merge step on a run-set, we read in the first BD / m elements from each sequence and merge them, and then read in the next BD / m elements. We then merge these two sorted lists and write out the first BD elements to disk. The latter BD elements are held over to be merged with the results of the next independent I/O call. We continue to read in records until we have the last $2BD$

records of the run-set merged in memory. The final step is to flush all records to disk to make room for the beginning of the next run-set.

We keep a buffer of BD elements in-core as a way of performing Step 4, the cleanup step. If we assume that $B \geq D$, then $BD \geq D^2$. Since $l = m$ and $m \leq D$, we have $D^2 \geq lm$. We know that lm is the maximum displacement an element can have in the reshuffled list. So if $BD \geq lm$,[3] why do we keep $2BD$ records in memory? The first element of our merged stripe could be lm spaces off-kilter, and might actually belong in the previous stripe. So, in order to ensure that an element is cleaned up to its proper place, we maintain BD records in memory from the previous merge. When we merge this "holdover" stripe with the current stripe, we clean up elements that may need to be moved across a stripe boundary.

This step, too, requires a scratch file. If we view the disk system as a table, the merge step reads in diagonally, and writes out horizontally. Figure 7 shows the read/write operations of Step 3 on 4 disks with 4 sequences each containing 4 blocks. The number contained in each block indicates the time it is read from or written to. Note how our first two I/O operations are reads, in order to get a sorted run of $2BD$ elements in memory. At $t = 3$, we write out the first BD elements. At $t = 4$, we read in BD elements, at $t = 5$ we write some out, and so on until the end of the merge operation where we write two stripes of data out. Notice how the write at $t = 3$ obliterates data we need to read at $t = 4$. Since the data is in a scratch file already, to prevent overwriting we simply write it back to the original file.

$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$
1	2	4	6
6	1	2	4
4	6	1	2
2	4	6	1

$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$
3	3	3	3
5	5	5	5
7	7	7	7
8	8	8	8

Figure 7. Table representation of read/write operations during the shuffle-merge step. Numbers indicate the time a block is accessed, either via an independent read on the left, or a striped write on the right.

After finishing steps 3 and 4, our example contains two sorted runs of length 64, as shown in Figure 8 on the next page.

3. Generally by a long shot. On our DEC machine, $B = 2^{13}$. When $m = D$, BD is roughly a thousand times larger than lm.

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	**0**	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
stripe 1	2	2	2	2	2	2	3	3	3	3	4	4	4	4	5	5
stripe 2	5	6	6	6	6	6	6	7	7	7	7	7	7	7	7	8
stripe 3	8	8	8	9	9	9	9	9	9	9	9	9	9	9	9	9
stripe 4	**0**	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2
stripe 5	2	2	2	2	2	2	2	3	3	3	3	4	4	4	4	5
stripe 6	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6
stripe 7	6	6	6	6	6	7	7	7	7	7	8	8	8	8	9	9

Figure 8. After the first iteration of `slmm`, we have 2 sorted runs *m* times longer than before. The offset of a merged run-set is the offset of its first sequence – for both of these run-sets, that offset was 0.

Larger Problem Sizes and Recursion

Each iteration of `slmm` creates sorted runs m times longer than those with which it started. If $N > FD$ we need to perform at least one additional iteration after the first one. We unshuffle the longer runs into run-sets whose sequences are as long as the sequences on which the previous iteration worked, and then recurse. That is, for a second-level iteration, we want to unshuffle the sequences into runs of length greater than or equal to F. This sets up the data so we can repeat the first iteration to even out the distributions by unshuffling and merging the sequences together. Finally, we merge them back together again to create runs of length up to FD^2. In our example we unshuffle the elements of the two sorted sequences, setting m to be 2 instead of 4.

Since these sequences are D times longer than before, our permutation should move the rightmost bits d bits further to the left than in the previous iteration. In the index-permutation function $f(x)$ from page 7, we set c to 1. Adding 1 to q increases the range of the barrel shift by d bits. Later unshuffles, since they work on still longer sequences, use correspondingly higher values of c.[4] In Figure 9 on the next page we show the results of this second-level unshuffle operation.

4. The value c is the depth of recursion minus 1.

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	$\mathbf{0}_0$	0_2	1_4	1_6	1_8	1_{10}	1_{12}	1_{14}	2_{16}	2_{18}	2_{20}	3_{22}	3_{24}	4_{26}	4_{28}	5_{30}
stripe 1	$\mathit{5}_{32}$	6_{34}	6_{36}	6_{38}	7_{40}	7_{42}	7_{44}	7_{46}	8_{48}	8_{50}	9_{52}	9_{54}	9_{56}	9_{58}	9_{60}	9_{62}
stripe 2	$\mathbf{0}_{64}$	0_{66}	0_{68}	1_{70}	1_{72}	1_{74}	2_{76}	2_{78}	2_{80}	2_{82}	2_{84}	2_{86}	3_{88}	3_{90}	4_{92}	4_{94}
stripe 3	$\mathit{5}_{96}$	5_{98}	5_{100}	5_{102}	5_{104}	5_{106}	5_{108}	6_{110}	6_{112}	6_{114}	6_{116}	7_{118}	7_{120}	8_{122}	8_{124}	9_{126}
stripe 4	6_{113}	6_{115}	6_{117}	7_{119}	7_{121}	8_{123}	8_{125}	9_{127}	$\mathbf{0}_1$	0_3	1_5	1_7	1_9	1_{11}	1_{13}	1_{15}
stripe 5	2_{17}	2_{19}	2_{21}	3_{23}	3_{25}	4_{27}	4_{29}	5_{31}	$\mathit{6}_{33}$	6_{35}	6_{37}	7_{39}	7_{41}	7_{43}	7_{45}	8_{47}
stripe 6	8_{49}	9_{51}	9_{53}	9_{55}	9_{57}	9_{59}	9_{61}	9_{63}	$\mathbf{0}_{65}$	0_{67}	0_{69}	1_{71}	1_{73}	1_{75}	2_{77}	2_{79}
stripe 7	2_{81}	2_{83}	2_{85}	3_{87}	3_{89}	4_{91}	4_{93}	5_{95}	$\mathit{5}_{97}$	5_{99}	5_{101}	5_{103}	5_{105}	5_{107}	5_{109}	6_{111}

Figure 9. Results of unshuffling the larger run-set containing 2 sequences of length 64. Elements 5_{32}, 5_{96}, 6_{33}, and 5_{97} are italicized to indicate the beginnings of "phantom runs."

Notice that the resultant runs are half as long as the originals, and that the bottom run-set is offset by 2 relative to the top one. The reason for this larger offset is that the top and bottom run-set will be merged recursively to form one sequence each, and we want these two sequences X_1 and X_2 to be laid out properly to prepare for the last merge step. Since we will read 2 blocks from each sequence on each independent I/O read and want to access each disk equally, we stagger the runs here by two disks. Notice also how the bottom sequences wrap modulo FD rather than modulo F as in the previous iteration. The third sequence starts on stripe 4 and ends on 6, while the fourth starts on stripe 6 but has $2B$ elements that are written on the beginning of stripe 4. The reason we wrap around in this fashion is to keep the sequence as contiguous as possible. Suppose we offset over individual stripes, so that a sequence filled each stripe before moving to the next. Then every stripe's worth of data would have a discontinuity in its ordering. Reading data with gaps such as these is more complicated than reading in continuous stripes that wrap around to the beginning because it requires more independent I/O calls. A little added complexity on write operations makes reading much easier, so we wrap on the sequence level rather than on the stripe level.

Even though each run-set contains 2 sorted sequences, we treat it as if there were 4 for the next iteration. A sorted sequence can be viewed as two sequences that do not overlap. In our example, we break each sequence half, and show the leaders of these new "phantom sequences" in italics. Treating the run-sets in this way greatly simplifies the algorithm because we can reuse our last set of parameters. Another, more important, reason for increasing m in this step is that if there were several lev-

els of recursion, the algorithm reaches the base case far more quickly using larger values of m, so setting m high for one step yields good savings for later steps.

More generally, whenever we need to merge the results of the first iteration of `slmm`, we use that first iteration recursively as a base step. The results of our first `slmm` iteration are of length FD. A level-2 iteration would consist of the following: First we perform one m-unshuffle operation to create run-sets whose sequences are of length FD/m. Since these unshuffled runs are at least as long as our initial runs that consisted of F elements, we can use a level-1 `slmm` iteration to recursively merge them together. Per the above discussion, we perform one unshuffle pass, an in-core merge pass, and a shuffle-merge pass to create nicely distributed runs of length FD. We perform a shuffle-merge pass again to result in sorted runs of length FDm. Figure 10 shows the state of our example problem after the level-1 `slmm` has finished merging the two run-sets into sequences X_1 and X_2 of length 64. Notice how X_2 starts on stripe 4 and wraps around back to it with its final $2B$ elements.

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	**0**	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2
stripe 1	2	2	2	2	2	2	2	3	3	3	3	4	4	4	4	5
stripe 2	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	7
stripe 3	7	7	7	7	7	8	8	8	8	9	9	9	9	9	9	9
stripe 4	8	9	9	9	9	9	9	9	**0**	0	0	0	0	1	1	1
stripe 5	1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3
stripe 6	3	3	4	4	4	4	5	5	5	5	5	5	5	5	5	5
stripe 7	6	6	6	6	6	6	6	7	7	7	7	7	7	8	8	8

Figure 10. Sample case after the level-1 merge, ready for the final level-2 merge pass. Note how evenly distributed the elements of these two sequences are. They have the same number of each value, except the first sequence has one more 2, and the second has one more 5.

The level-2 shuffle-merge step for our example is very similar to the level-1 shuffle-merge, except that it reads two blocks at a time from each sequence. We first read block 0 of D_0 and D_1, and block 4 of D_2 and D_3. Next we read block 0 of D_2 and D_3, and block 5 of D_0 and D_1. Our final independent I/O call reads block 3 of D_2 and D_3, and block 4 of D_0 and D_1. Figure 11 shows the results of this merge, a single sorted list of 128 elements.

	$\mathcal{D}_0$				$\mathcal{D}_1$				$\mathcal{D}_2$				$\mathcal{D}_3$			
stripe 0	**0**	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
stripe 1	1	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
stripe 2	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3	3
stripe 3	3	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5
stripe 4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
stripe 5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7
stripe 6	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8	8
stripe 7	8	8	9	9	9	9	9	9	9	9	9	9	9	9	9	9

Figure 11. After the final merge pass, the sequence is sorted.

Inductively, to merge together D sorted runs of length FD^k, $k > 0$, we perform k unshuffles, followed by a base merge pass, followed by k shuffle-merge passes. When we have $C < D$ runs of length L, $FD^{k-1}C \leq L \leq FD^k$, we sort in the same fashion, except that the first unshuffle and last merge use $m = C$ rather than $m = D$.

Thus, in order to sort FD^k elements (a "problem of depth k"), we need to perform $1 + 3 + 5 + \ldots + (2k + 1) = (k + 1)^2$ passes. The first pass forms sorted runs of length F, and each successive group of passes forms sorted runs D times longer than before. Again, in cases where the problem size lies between FD^c and FD^{c+1}, the first unshuffle and last shuffle-merge of the last iteration use a value of m less than D. Even though each unshuffle or shuffle-merge operation copies the data, the total number of these copies is always even, so that when `slmm` finishes the data is sorted in the original file.

Our example hinted that offsetting can get complicated when we are performing multiple shuffle-merges. The result of one pass must be offset properly for the next, and the result of that pass must be positioned correctly for the next, and so on, to get the best performance. As we have seen, if we are performing two shuffle-merge passes, we want the original run-sets to be offset, as well as their members. That way, when the first pass is done, each resultant sequence is offset properly.

In the general case, shuffle-merging a problem of depth k, we have k "levels of offset". When we start merging from a base run of length F, we must track its position within k nested run-sets. An easy way to track this is to imagine a k-digit number in base m associated with each base run of length F. The least significant bit represents the base run's position in its run-set. The next bit represents that run-set's position relative to its parent run-set, and so on. We label the original sequences by counting, starting at 0. The absolute offset of any sequence, that is, the disk it starts on, is simply the sum of its label's digits, modulo D.

5. Test Results

We now present and compare the running times of slmm and qsort. We ran test runs on adams, a DEC 2100 server with two 175-MHz Alpha processors. adams has 320 megabytes of memory, uses eight 2-gigabyte disks for storage, and runs Digital UNIX 4.0E. The disks are distributed among three SCSI chains connected to a DEC RAID controller. We allotted 64 megabytes of main memory to the slmm and qsort executables.[5] Both were coded in C and compiled using cc with options -fast, -arch host, -tune host, and -inline speed.

The following tables show the run-times we measured on inputs of randomly ordered[6] sequences of 4-byte ints. First we show qsort's performance, in Table 1. Table 2 gives run times for slmm using synchronous I/O, and Table 3 shows slmm's run time using asynchronous I/O. Since the shuffle-factor m is bound by the number of disks, we show slmm's performance for $D = 8$ and $D = 4$.

Table 1. Test runs of qsort. We show the run time in seconds, as well as the normalized time (in μs) which is the run time divided by $N\log_2 N$.

	qsort	
Problem size	runtime (s)	normalized (μs)
$N = 2^{22}$	34.14	0.417
$N = 2^{23}$	73.00	0.378
$N = 2^{24}$	301.57	0.749
$N = 2^{25}$	694.65	0.828
$N = 2^{26}$	1686.40	0.967
$N = 2^{27}$	5439.42	1.501
$N = 2^{28}$	—	—
$N = 2^{29}$	—	—

Not surprisingly, qsort is the fastest way to sort smaller problems of $N \leq 2^{23}$ integers; it generally runs 20 – 30 percent faster than slmm. While slmm's I/O times for these problems are small due to file caching, it has significantly more computational overhead than qsort, which accounts for its poorer performance for in-core problems. For $N \geq 2^{24}$, qsort pages, making its performance drop sharply. Its normalized times for $N = 2^{27}$ are nearly 4 times greater than for $N = 2^{23}$. Because even more recursive steps would be required to reduce a problem to an in-core size, we expect that for $N \geq 2^{28}$ qsort's performance would be even worse. However, qsort

5. Of course, qsort uses far more swap space than slmm on larger problems.
6. We used the standard C library function rand for this purpose.

cannot handle these problem sizes because it completely fills the scratch disk at $N = 2^{27}$ and causes a segmentation violation at $N = 2^{28}$.

Table 2. w the run time in seconds and the normalized time in μs. We also give *k*, the problem depth, and the percentage of time spent on I/O.

	slmm / synchronous I/O / 8 disks				**slmm / synchronous I/O / 4 disks**			
Problem size	*k*	runtime (s)	normalized (μs)	I/O %	*k*	runtime (s)	normalized (μs)	I/O %
$N = 2^{22}$	*	52.09	0.564	8.6	*	48.37	0.524	8.8
$N = 2^{23}$	1	100.61	0.521	8.4	*	100.30	0.520	9.5
$N = 2^{24}$	1	202.76	0.504	8.5	1	214.75	0.533	18.1
$N = 2^{25}$	1	444.01	0.529	14.0	1	477.40	0.569	24.7
$N = 2^{26}$	2	1872.43	1.073	41.7	2	2011.45	1.152	49.6
$N = 2^{27}$	2	3663.03	1.011	38.8	2	4071.05	1.123	49.2
$N = 2^{28}$	2	7423.74	0.988	38.4	3	13295.01	1.769	56.1
$N = 2^{29}$	3	23550.42	1.513	44.8	3	27385.43	1.759	56.6

Table 3. Test runs of asynchronous slmm. Data are as in Table 2, except the I/O column now gives time spent waiting for asynchronous calls to complete.

	slmm / asynchronous I/O / 8 disks				**slmm / asynchronous I/O / 4 disks**			
Problem size	*k*	runtime (s)	normalized (μs)	I/O %	*k*	runtime (s)	normalized (μs)	I/O %
$N = 2^{22}$	*	51.11	0.554	1.1	1	46.77	0.507	3.7
$N = 2^{23}$	1	99.16	0.514	2.2	1	95.20	0.493	1.7
$N = 2^{24}$	1	198.87	0.494	1.6	2	288.93	0.718	7.8
$N = 2^{25}$	1	414.59	0.494	1.9	2	689.97	0.823	22.9
$N = 2^{26}$	2	1400.16	0.802	15.8	3	2572.50	1.474	40.9
$N = 2^{27}$	2	2814.41	0.777	13.8	3	5219.11	1.440	41.2
$N = 2^{28}$	2	5659.27	0.753	11.8	4	15451.34	2.056	45.0
$N = 2^{29}$	3	17518.78	1.125	17.1	4	**	**	**

* Indicates *k* should be 0. Since slmm with $k = 0$ is essentially be qsort, we reduce *M* for these problems to raise *k* to 1.

** Due to hardware trouble on adams we did not complete this test. We expect sorting $N = 2^{29}$ integers under these conditions to take approximately 32,000 seconds.

slmm, on the other hand, does not suffer the same sharp performance drop when the problem size exceeds main memory. Normalized times do rise, however, when *k* increases. Recall that for a given problem size, *k* is the smallest integer such that

$N \leq FD^k$, and the number of passes required to solve a given problem is $(k + 1)^2$. Thus, every time N increases by a factor of D, k increases by one and the number of passes rises.[7] For example, normalized times for asynchronous 8-disk slmm increase by about 50–60 percent from $N = 2^{25}$ to 2^{26}, and again from $N = 2^{28}$ to 2^{29}. These sizes mark boundaries where the data requires more passes to be sorted. At $N = 2^{26}$, k increases from 1 to 2 for 8-disk slmm, which means the number of passes required to sort more than doubles from 4 to 9. At $N = 2^{29}$, the number of passes nearly doubles again, to 16.

For $D = 4$, the number of passes needed to solve a problem rises far more quickly. Let's look at the run times with asynchronous I/O for $N = 2^{23}$ and 2^{28} on 4 and 8 disks. On the smaller problem, both have $k = 1$ and the 4-disk run is slightly faster than the 8-disk run. For $N = 2^{28}$, however, $k = 2$ for 8-disk operation, indicating 9 passes are necessary to sort. Using only 4 disks, $k = 4$, so slmm needs 25 passes. 4-disk slmm requires nearly 3 times more passes over the disks to sort the numbers, which explains why it takes nearly 3 times longer to do so.

Looking over the I/O percentages for synchronous slmm we see it can spend a large amount of time waiting for I/O to complete, especially for larger problems. This is especially noticeable for $D = 4$ where slmm spends over half its time in I/O sorting $N = 2^{28}$ integers. By using asynchronous calls (reading ahead and writing behind) we can generally cut the time we wait for I/O by a factor of more than 2 for $D = 8$. Over 4 disks the improvement is not that great: I/O times drop by less than 25 percent. More interesting, switching from synchronous to asynchronous I/O actually slows slmm down from $N = 2^{24}$ on up. These issues point out some interesting tradeoffs involved in deciding what kind of I/O to use, and we will discuss them briefly.

When slmm runs on 4 disks, the bandwidth of the disk system is a real bottleneck. The processor can manipulate the data, whether by unshuffling or shuffle-merging, faster than the disks can supply it. Switching from synchronous to asynchronous I/O simply changes the read and write calls that slmm waits for. If a computation is heavily I/O bound, then overlapping I/O and computation increases the speed by at most the computation time, which is a relatively small percentage of the total time. Thus, prefetching by one operation does not provide as significant a speedup as it does on 8-disk slmm. Reading further ahead, say, 2 or 3 steps, might alleviate the I/O bottleneck further. But given the other performance limitations slmm has on 4 disks, it's probably a better idea to go find a machine with more disks instead.

The fact that asynchronous I/O can still spend a lot of time waiting explains why asynchronous operation isn't much faster than synchronous for 4 disks, but it cer-

7. In Section 4, we defined $F = BD^q \leq M$ for the largest possible q. For asynchronous slmm on adams, $B = 2^{13}$ and $M = 2^{22}$, so when $D = 8$, $q = 3$ and $F = BD^q = 2^{22}$.

tainly does not explain why using asynchronous I/O actually slows slmm down. The reason for this slowdown hinges on memory use. Using asynchronous I/O reduces M by a factor of 4: we need to allocate a prefetch and write-behind buffer each as big as our main buffer, and since M must be a power of 2 we must round it down. Our tests ran slmm in a memory partition of 64 megabytes (16 million records). Synchronous I/O use sets $M = 2^{24}$ ints, but asynchronous operation must use a smaller value, $M = 2^{22}$. This difference affects the base case size F. With synchronous I/O, since $B = 2^{13}$ and $D = 4$, we have $F = BD^q \leq 2^{24}$, $D^q \leq 2^{11}$ and $q = 5$. Switching to asynchronous I/O, we have $D^q \leq 2^9$ and $q = 4$.[8] Thus the base case size F for synchronous I/O is 4 times that for asynchronous when $D = 4$. For synchronous I/O and 4 disks, the value of k is 1 less than its asynchronous counterpart for $N = 2^{24}$ on up because we can use a base case algorithm on larger sequences. This great reduction in the number of passes more than makes up for the slight savings gained from using asynchronous I/O.

Note, however, that even in the slowest case, asynchronous I/O with $D = 4$, at $N = 2^{27}$ slmm is still faster than qsort. Under better conditions, asynchronous I/O with $D = 8$, it is nearly twice as fast. We expect slmm to really shine on larger disk systems with $D \geq 16$. By keeping k lower for larger problem sizes, slmm on larger disk systems should be able to sort even more quickly.

6. Potential Modifications

In deciding how to implement LMM we made a few changes to the algorithm as Rajasekaran presented it in [3]. For example, we always set l and m to be the same, and we work bottom-up, rather than top-down. Both of these changes simplify the algorithm and make it faster in practice. Now that we have a working implementation of LMM and have examined its test data, a few other potential improvements come to light.

slmm, like Rajasekaran's presentation of LMM, uses a logical block as the basic unit of data. The reads in the shuffle-merge step are independent I/O calls that read a different block from each disk. The unshuffle step repeatedly reads in a stripe of data, permute its indices, and writes it out to D different blocks. Looking back, using a logical block as the basic element was probably not the most efficient way to implement LMM.

When merging or unshuffling long sequences of data, reading in units as small as one block can result in a lot of head travel for the amount of data moved. When k is 3 or more, during the last iteration's first unshuffle and last merge I/O wait times can

8. When $D = 8$, F is the same for synchronous and asynchronous I/O. $F = 2^{22}$ and $q = 3$ in either case.

exceed 70 percent of that step's run time. The reason for this is the disk heads are traveling long distances from where they are reading to where they are writing. Reading a block, traveling a great distance, then writing it out, only to go back again is not the most effective way to perform I/O. Significantly faster would be working in larger units, say stripe size or more. If unshuffles read in m stripes at a time, they could write out a stripe rather than a block to the beginning of each sequence. Similarly the shuffle-merge operation could read a stripe rather than a block from each sequence per merge and hence write out a larger portion. This would cause the size of the buffers used in these steps to increase by a factor of D to BD^2, but since the base case works in units of BD^q this should not change the memory requirements for most real-world applications (on our test machine q ranged from 3 to 5). A good general case solution would be to use D^{q-1} blocks as the basic unit of data rather than just one block.

Another advantage of changing the basic unit of data from a block to a stripe or more is that if we never read or write in units smaller than a stripe, we do not need to stagger the runs. For each read step in the merge pass we would read a stripe from each run, which is D blocks each from D disks; offsetting the beginning of each run by any amount would not change this distribution at all. Reducing the different ways data can be offset to removes complexity from the code and makes it easier to maintain. Further, if all runs start on $\mathcal{D}_0$, striped I/O can be used exclusively. Since striped I/O is generally a bit faster than independent I/O, this should provide some speed improvement.

We believe that changing LMM's focus from blocks to stripes would greatly simplify its implementation and provide significant speed gains.

7. Conclusion

We have implemented LMM and analyzed its real-world performance. It runs substantially faster than `qsort` under demand paging and can handle much larger problems. As Rajasekaran predicted, LMM performs much more strongly when the number of disks is large.

Although it runs on a multiprocessor DEC 2100, `slmm` only uses one processor for sorting and merging. With a faster disk system, computation time, rather than I/O time, can become a bottleneck. We are investigating ways to adapt LMM to run on a shared-memory SMP. It is our belief that a multiprocessor implementation of LMM could run very quickly, since unshuffle and merge tasks are essentially independent across run-sets and could thus be handled by different processors.

Acknowledgments

Many thanks to Thomas Cormen for introducing me to the problem of out-of-core sorting, and for his help and support during all phases of this project. His revisions and advice have made this paper far stronger than it would have been otherwise. James Clippinger maintains the ViC* API and answered countless questions regarding programming for parallel disks. He also helped squash a pesky bug in `slmm`. Wayne Cripps keeps the server adams running, and on the side, answered many systems questions. Daniel Epstein provided many thoughtful comments on an earlier draft of this paper. The purchase of the DEC 2100 server named adams was made possible by a grant from Digital Equipment Corporation.

References

[1] T. H. Cormen, M. Hirschl, Early experiences in evaluating the parallel disk model with the ViC* implementation, Parallel Computing 23 (1997) 571–600.

[2] T. H. Cormen, D. M. Nicol, Performing out-of-core FFTs on parallel disk systems, Parellel Computing 24 (1998) 5–20.

[3] S. Rajasekaran, A Framework For Simple Sorting Algorithms on Parallel Disk Systems, 10th Annual ACM Symposium on Parallel Algorithms and Architectures (1998) 88–97.

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