Slabpose Columnsort: A New Oblivious Algorithm for Out-of-Core Sorting on Distributed-Memory Clusters

Geeta Chaudhry
Thomas H. Cormen

Dartmouth College
Department of Computer Science
{geetac,thc}@cs.dartmouth.edu

Abstract

Our goal is to develop a robust out-of-core sorting program for a distributed-memory cluster. The literature contains two dominant paradigms for out-of-core sorting algorithms: merging-based and partitioning-based. We explore a third paradigm, that of oblivious algorithms. Unlike the two dominant paradigms, oblivious algorithms do not depend on the input keys and therefore lead to predetermined I/O and communication patterns in an out-of-core setting. We have developed several out-of-core sorting programs using this paradigm. Our baseline implementation, 3-pass columnsort, was based on Leighton's columnsort algorithm. Though efficient in terms of I/O and communication, 3-pass columnsort has a restriction on the maximum problem size. As our first effort toward relaxing this restriction, we developed two implementations: subblock columnsort and M-columnsort. Both of these implementations incur substantial performance costs: subblock columnsort performs additional disk I/O, and M-columnsort needs substantial amounts of extra communication and computation. In this paper, we present slabpose columnsort, a new oblivious algorithm that we have designed explicitly for the out-of-core setting. Slabpose columnsort relaxes the problem-size restriction at no extra I/O or communication cost. Experimental evidence on a Beowulf cluster shows that unlike subblock columnsort and M-columnsort, slabpose columnsort runs almost as fast as 3-pass columnsort. To the best of our knowledge, our implementations are the first out-of-core multiprocessor sorting algorithms that make no assumptions about the keys and produce output that is perfectly load balanced and in the striped order assumed by the Parallel Disk Model.

Keywords: columnsort, out-of-core, parallel sorting, distributed-memory cluster, oblivious algorithms.

1 Introduction

Sorting is universally acknowledged to be an important and fundamental problem in computing. Sorting very large datasets appears as a key subroutine in many applications. Geographical information systems, seismic modeling, and web-search engines are a few example applications that store, sort, and search through enormous amounts of data. For some applications, the amount of data exceeds the capacity of main memory, and the data then typically reside on one or more disks. As a result, the cost of transferring data between memory and disks (relative to the cost of computation) can be a major performance bottleneck. For large-scale applications, therefore, it is essential to design and engineer algorithms that minimize disk-I/O times. These algorithms are called *out-of-core* algorithms [37].

Goal

Our goal is to design a robust out-of-core sorting algorithm for a distributed-memory cluster and to develop an implementation of the algorithm using off-the-shelf software. Because distributed-memory clusters [3] offer a good price-performance ratio, they have become a popular platform for high-performance computing. Two ways to sort out-of-core data dominate the literature: merging-based algorithms and partitioning-based algorithms. In both of these approaches, the I/O and communication patterns are highly dependent on the input keys. We explore a third way of out-of-core sorting: oblivious algorithms. An *oblivious sorting algorithm* is a comparison-based sorting algorithm in which the sequence of comparisons is predetermined [25, 26]. In other words, the result of a comparison has no effect on the indices of elements compared later on. When adapted to an out-of-core setting, the I/O and communication patterns of such an algorithm should be independent of the input keys.

An algorithm whose I/O and communication patterns depend on the input keys makes the task of overlapping I/O, communication, and computation difficult. Such an algorithm has the additional challenge of beating "bad" sequences of input keys. When the I/O and communication patterns depend on the input, it is hard to predict at coding time the best way to overlap I/O, communication, and computation. Such overlapping is crucial to the performance of any out-of-core application on a distributed-memory cluster. For the best performance, it is not enough to minimize the costs of these three individual operations. Instead, we should overlap the three operations since they use three different resources: disks, network, and processors.

The amount of overlap actually achieved in an out-of-core sorting implementation depends on two factors: the opportunity of overlap in the underlying algorithm and the software used to achieve this overlap. For a given algorithm, it is really an implementation issue whether the maximum overlap can actually be achieved by the implementation, but choosing an algorithm that has ample opportunity for overlap is more of an algorithm design issue.

Related work

Here are the two predominant paradigms in the literature for out-of-core sorting, along with the primary hindrances in adapting each to a distributed-memory cluster:

- *Merging-based* algorithms first create sorted runs of the input and then merge these sorted runs. As we shall discuss in Section 2, the procedure of merging the sorted runs is difficult to parallelize, and there is a fair amount of unpredictability in the disk I/O patterns even on a single processor. We know of no robust implementation of a merging-based algorithm for a distributed-memory cluster.

- *Partitioning-based* algorithms are based on partitioning the input into approximately equal-sized buckets and then sorting each bucket. A distribution sort is a classical example. As we shall see in Section 2, the standard partitioning schemes do not have predictable I/O and communication patterns in the out-of-core scenario on a cluster of machines.

Oblivious algorithms: the third way

In marked contrast to the two dominant paradigms of merging and partitioning, both of which have unpredictable I/O and communication patterns, we offer the third paradigm of oblivious algorithms. With this paradigm, we have achieved our goal: out-of-core sorting on a distributed-memory cluster using off-the-shelf software. An oblivious algorithm in an out-of-core setting leads to predetermined I/O and communication patterns, allowing excellent opportunity for both planning and achieving good overlap among I/O, communication, and computation. We have begun to explore this third paradigm by designing and implementing several algorithms that sort out-of-core data on clusters. The following list summarizes our background work on out-of-core sorting:

- As a first step, we have designed and implemented several out-of-core sorting programs [10, 12] for distributed-memory clusters. Each program is based on Leighton's columnsort [27], a sorting algorithm consisting of eight steps. Columnsort is oblivious, and it also has other benefits that make it amenable to an out-of-core adaptation. We shall see these benefits later in this paper.

  Although our first implementations are robust, they suffer from a restriction on the maximum problem size that they can sort. Columnsort sorts N records[1] arranged as an $r \times s$ matrix, where $N = rs$, s divides r, and $r \geq 2s^2$. We refer to the last restriction on the height of the matrix ($r \geq 2s^2$) as the *height restriction*. In our first implementations, we set r to be M/P, where M is the overall memory, in records, on the entire cluster, and P is the number of processors in the cluster. Therefore, M/P is the number of records that a single processor can hold in its memory. We refer to this setting of r, the height of the matrix, to M/P as the *height interpretation*. Together, the height interpretation and the height restriction lead to the following *problem-size restriction*, which limits the maximum number of records, N, that we can sort on a cluster of P processors:

$$N \leq \sqrt{(M/P)^3/2}\,. \tag{1}$$

- We designed and implemented *subblock columnsort* [13], a new oblivious algorithm that relaxes the height restriction by a factor of $\sqrt{s}/2$, to $r \geq 4s^{3/2}$, by adding two steps to columnsort. With a height interpretation of $r = M/P$, we get a problem-size restriction of [2]

$$N \leq (M/P)^{5/3}/4^{2/3}\,. \tag{2}$$

  The two additional steps of subblock columnsort add an extra pass to the three passes of *3-pass columnsort*, our best previous implementation. Each *pass* consists of reading each record once from disk, working on it in memory, and writing it back to disk.

[1] Each record contains a *key* according to which the records are to be sorted.

[2] This improvement in problem size can be quite substantial in an out-of-core setting. For most current systems ($M/P \geq 2^{12}$ records), this change will enable us to more than double the largest problem size.

- *M-columnsort* changes the height interpretation from $r = M/P$ to $r = M$, thus leading to the improved problem-size restriction

$$N \leq \sqrt{M^3/2}\,. \tag{3}$$

  Although M-columnsort adds no extra passes, it does incur substantial amounts of communication and additional computation, when compared to 3-pass columnsort.

The work presented in this paper is a result of our efforts toward answering the following question: can we relax the problem-size restriction without any extra I/O or communication costs? The following list summarizes the contributions of this paper:

- We present *slabpose columnsort*, a new oblivious algorithm designed explicitly for the out-of-core setting of a cluster. Just like subblock columnsort and M-columnsort, slabpose columnsort relaxes the problem-size restriction. Unlike subblock columnsort and M-columnsort, however, slabpose columnsort does so at no extra I/O or communication cost. Slabpose columnsort relaxes the height restriction to $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$, for a parameter k such that $k < s$. In our implementation, we set $k = P$. With a height interpretation of $r = M/P$, we get a problem-size restriction of

$$N \leq \begin{cases} (M/P)^{3/2} \cdot \dfrac{\sqrt{P}}{2} & \text{if } s/P \geq P, \\ \dfrac{(M/P)^2}{4P} & \text{if } s/P < P\,. \end{cases}$$

  As discussed later, in most cases, this restriction is an improvement over the original problem-size restriction.

- All of our implementations use only standard, off-the-shelf software, such as MPI [36] and MPI-2 [19] for communication and I/O.

- There are no assumptions required about the keys. In fact, the I/O and communication patterns are oblivious to the keys in each of our programs. To the best of our knowledge, our implementations are the first ones to sort out-of-core data on a distributed-memory cluster without making any assumptions about the input keys.

- We parallelize all disk I/O operations across all the disks in the cluster. That is, we have engineered our implementation to make sure that all reads and writes are evenly distributed across the nodes in the cluster. Furthermore, the reads and writes on a given node are evenly distributed across all the disks owned by that node.

- The output appears in the standard striped ordering used by the Parallel Disk Model (PDM) [40]. PDM ordering balances the load for any consecutive set of records across processors and disks as evenly as possible. A further advantage to producing sorted output in PDM ordering is that our algorithm can be used as a subroutine in other PDM algorithms. (See [37] for a compendium of PDM algorithms.) To the best of our knowledge, our programs are the first multiprocessor sorting algorithms whose output is in striped PDM order.

Experimental results on a Beowulf cluster with fast processors and a Myrinet interconnection network show that slabpose columnsort indeed runs almost as fast as 3-pass columnsort. Both subblock columnsort

and M-columnsort run much slower than 3-pass columnsort, due to the extra I/O and communication costs, respectively.

The remainder of this paper is organized as follows. Section 2 discusses related work, focusing on the two dominant paradigms in the literature. Section 3 presents the columnsort algorithm, our adaption of columnsort to an out-of-core setting, and a summary of subblock columnsort and M-columnsort. Section 4 describes slabpose columnsort. Section 5 presents experimental results. Finally, Section 6 offers some concluding remarks and discusses future work. We give the proof of correctness of slabpose columnsort in Appendix A.

2 Related work

As one would expect for a fundamental problem such as sorting, the research community has investigated it extensively. Two thorough surveys catalog out-of-core algorithms [4, 38]. This section touches on previous work in developing out-of-core sorting algorithms under both the merging-based and the partitioning-based paradigms. For each paradigm, we first outline the major design issues and theoretical results, and then we discuss the algorithms that actually have been implemented.

Merging-based algorithms

There are two main challenges in designing a merging-based out-of-core sorting algorithm for a distributed-memory cluster. The primary challenge is to ensure good data locality when the sorted runs are merged. Even on a uniprocessor system, disk reads and writes depend on the input keys. The second challenge is to parallelize the merging with data in a distributed memory; we know of no efficient algorithm that does so. It is not surprising, therefore, that the known implementations [8, 15, 33, 35] of merging-based algorithms which produce output in PDM order run only on a single processor.

Theoretically, the challenge of assigning blocks to disks has been met in various ways. Greed sort by Nodine and Vitter [31], (l, m)-merge sort by Rajasekaran [34], Sharesort by Aggarwal and Plaxton [1], and Simple Randomized Mergesort by Vitter and Barve [7] are a few examples of merging-based out-of-core sorting algorithms. More recently, there has been some work on efficient prefetching strategies with applications to sorting [23, 24, 39].

Partitioning-based algorithms

When designing a partitioning-based sorting algorithm, the challenge is to ensure that the amount of data in each partition is approximately equal. An uneven partitioning requires load balancing, which in turn requires additional communication and possibly additional I/O. It turns out that it is difficult to find the exact partitioning elements (leading to a perfectly load-balanced partition) in an I/O-efficient way [38]. Even when the partitions are approximately equal in size, the I/O and communication patterns are highly dependent on the keys. Out-of-core data are read one *memoryload* (i.e., M records, with M/P records per processor) at a time. In any such read step, when each processor reads M/P records, it is entirely possible that many more than M/P records in memory belong to the partition mapped to a given processor, leading to highly unbalanced I/O and communication.

The literature describes several partitioning-based algorithms. Some of them are deterministic [2, 6, 22, 30], and others are randomized [16, 20, 21, 40]. In any case, an out-of-core sorting algorithm using

these methods must have a provision for recovering from a bad partition. To date, we do not know of any out-of-core sorting implementation for clusters that uses any of these methods.

One might think that a randomized partitioning-based algorithm such as the one in [16] is ideal for sorting out-of-core data on distributed-memory clusters. Below, we discuss the steps of a typical partitioning-based algorithm, along with the main issues in adapting each step to an out-of-core setting of distributed-memory clusters:

- **Find the partitioning elements**. In most cases, each processor selects a random sample of its data, and the partitioning elements are decided by sorting the set of the samples of all the processors [16, 21]. The sorting of this set is most often serialized and takes place on one processor, which then broadcasts the $P - 1$ partitioning elements $s_1, s_2, \ldots, s_{P-1}$ to all processors. This step, therefore, consists of two rounds of communication (sending samples to one processor and the broadcast of the partitioning elements) and requires less than a full pass, since no records are written to the disks and perhaps not even all records need to be read.

- **Distribute the data**. All N records are distributed among the P processors as follows. Letting $s_0 = -\infty$ and $s_P = \infty$, then for $i = 0, 1, \ldots, P - 1$, processor i gets all elements with keys k such that $s_i < k \leq s_{i+1}$. This step proceeds on each processor in several stages, where each stage has two steps. First, each processor reads in a memoryload of data, divides it into P groups, one per processor, and then sends each group to the appropriate processor. Next, each processor writes out the data that it received. None of the stages are guaranteed to be load balanced. In other words, almost all the records in a stage can belong to one single processor, leading to unbalanced communication and unbalanced I/O, the latter being more costly. Note that this imbalance within individual stages can exist even if the partitioning elements divide the entire data into approximately equal-sized partitions.

- **Locally sort each partition**. After the distribution step, there is no guarantee that the partitions are equal in size. Since any load balancing requires each partition to be sorted, the total time for this step is dictated by the time to sort the largest partition. Each processor then locally sorts its partition using an out-of-core sorting method of choice, which takes at least two passes due to the data being out-of-core. Depending on the sorting method in use, however, one may combine the step of distributing the data and the first pass of sorting each partition into a single pass. The succeeding out-of-core local sorting passes remain.

- **Produce output that is sorted, load-balanced, and striped across all the disks**. Even if all the partitions are equal in size, the output so far is sorted in processor-major order. Since our goal is to produce output that is striped across all the disks, load balanced and with the striping unit being any desired size, an additional pass is needed.

We claim that any partitioning-based algorithm that produces striped output on a distributed-memory cluster needs to make at least three passes over the data: one for distribution, at least one for local sorting, and one for load balancing and striping. Our programs, though restricted in the maximum problem size, require at most four passes over the data; except for subblock columnsort, all our implementations make three passes over the data. Our programs have predictable worst-case performance, however, and they make no assumptions about the input keys. When engineering an out-of-core sorting algorithm for a distributed-memory cluster, it is helpful to know in advance the exact pattern of the I/O and communication steps; this knowledge allows the programmer to efficiently plan the overlap of I/O, communication, and computation. Curiously, our upper bound of four passes is better than the $\omega(1)$-pass lower bound for the multiheaded,

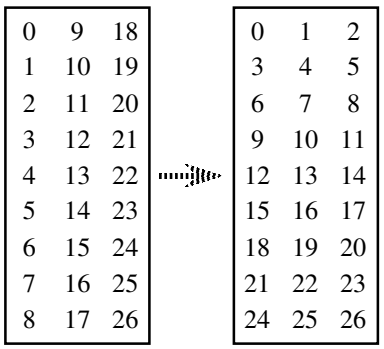

Figure 1: The transpose-and-reshape operation of columnsort's step 2, shown for $r = 9$ and $s = 3$.

single-disk model proven by Aggarwal and Vitter [2], which is at least as strong a model as the PDM. Our upper bound is achieved at the cost of the problem-size restriction.

Existing implementations

Jim Gray maintains the sorting benchmark web page [18], which keeps track of the fastest current sorting programs. Some of these sorting programs are almost in-core ($N \leq 2M$). Others target either a shared-memory system or a uniprocessor system. A few other implementations fit into the broad category of partitioning-based algorithms [14, 32], but they do not target the out-of-core setting of a cluster.

We know of only two implementations [5, 17] that target the same setting as ours: sorting out-of-core data on a distributed-memory cluster. Both are partitioning-based, and they make simplifying assumptions, thereby avoiding the main challenge of a partitioning-based sort: obtaining perfectly load-balanced partitions. Neither of the implementations produces output in striped PDM order. One implementation, NOW-Sort [5], assumes that keys are uniformly distributed, and the other [17] assumes apriori knowledge of partitioning elements that divide the entire data equally among the P processors.

3 Background

In this section, we review the columnsort algorithm. Next, we describe our out-of-core adaptation, followed by a brief outline of our three preliminary implementations. We then summarize subblock columnsort and M-columnsort. We conclude by recalling the implications of the problem-size restriction.

The basic columnsort algorithm

Columnsort sorts N records arranged as an $r \times s$ matrix, where $N = rs$, s divides r, and $r \geq 2s^2$. When columnsort completes, the matrix is sorted in column-major order. Columnsort proceeds in eight steps. Steps 1, 3, 5, and 7 are all the same: sort each column individually. Each of steps 2, 4, 6, and 8 permutes the matrix entries as follows:

- *Step 2: Transpose and reshape:* As shown in Figure 1, transpose the $r \times s$ matrix into an $s \times r$ matrix. Then "reshape" it back into an $r \times s$ matrix.

- *Step 4: Reshape and transpose:* This permutation is the inverse of that of step 2.
- *Step 6: Shift down by $r/2$:* Shift each column down by $r/2$ positions, wrapping the bottom half of each column into the top half of the next column. The top half of the leftmost column is filled with $-\infty$ keys and a new rightmost column is created, with its bottom half filled with ∞ keys.
- *Step 8: Shift up by $r/2$:* This permutation is the inverse of that of step 6.

Out-of-core columnsort

In our adaptation of columnsort to an out-of-core setting on a distributed-memory cluster, we assume that the cluster has P processors and D disks, where $D \geq P$. A processor *owns* the D/P disks that it accesses.[3] The data are placed so that each column is stored in contiguous locations on the disks owned by a single processor. Specifically, processor j *owns* columns j, $j + P$, $j + 2P$, and so on. We use buffers that hold exactly r records. Each processor has several such buffers. For convenience, we assume that all configuration parameters as well as the matrix dimensions r and s are powers of 2. (Thus, P must divide D.)

Here, we outline the basic structure of each pass; for key implementation features and performance results, see [10, 12]. Each pass in our first implementation performs two consecutive steps of columnsort. That is, pass 1 performs steps 1 and 2, pass 2 performs steps 3 and 4, pass 3 performs steps 5 and 6, and pass 4 performs steps 7 and 8. Each pass is decomposed into s/P *rounds*. Each round processes the next set of P consecutive columns, one column per processor, through a pipeline of five phases. This pipeline runs on each processor. In each round on each processor, an r-record buffer travels through the following five phases:

Read phase: Each processor reads a column of r records from the disks that it owns into the buffer associated with the given round.

Sort phase: Each processor locally sorts, in memory, the r records it has just read. In the first pass, we call the system `qsort` call. In all the other passes, we merge sorted runs.

Communicate phase: Each record is destined for a specific column, depending on which even-numbered columnsort step this pass is performing. In order to get each record to the processor that owns this destination column, processors exchange records.

Permute phase: Having received records from other processors, each processor rearranges them into the correct order for writing.

Write phase: Each processor writes a set of r records onto the disks that it owns.

Because we implemented the phases asynchronously, at any one time each phase could be working on a buffer from a different round.

[3] When $D \geq P$, each processor accesses exactly D/P disks over the entire course of the algorithm. When $D < P$, we require that there be P/D processors per node and that they share the node's disk; in this case, each processor accesses a distinct portion of the disk. We treat this distinct portion as a separate "virtual disk," allowing us to assume that $D \geq P$.

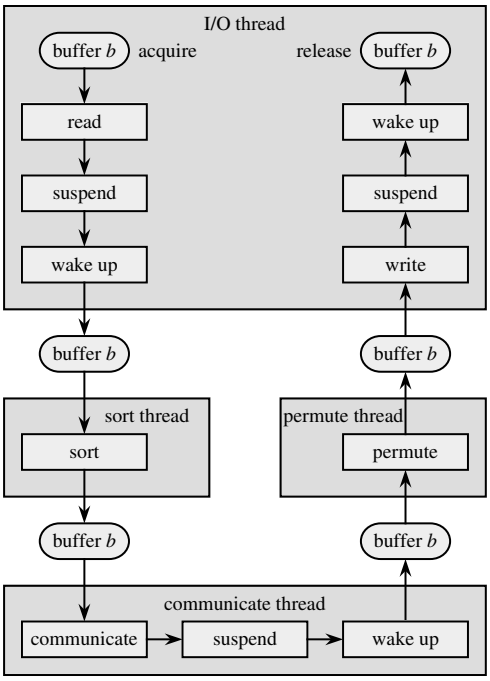

Figure 2: The history of a buffer b as it progresses within a given round of a given pass of threaded 4-pass columnsort. The I/O thread acquires the buffer from the global pool and then reads into it from disk. The I/O thread suspends during the read, and when it wakes up, it signals the sort thread. The sort thread sorts buffer b and signals the communicate thread. The communicate thread suspends during interprocessor communication, and when it wakes up, it signals the permute thread. The permute thread then permutes buffer b and signals the I/O thread. The I/O thread writes the buffer to disk, suspending during the write. When the I/O thread wakes up, it releases buffer b back to the global pool.

Preliminary implementations

Non-threaded columnsort, our first implementation [12], used asynchronous I/O and asynchronous communication calls to overlap I/O, communication, and computation. This implementation had performance results that, by certain measures, made it competitive with the NOW-Sort program [5].

Threaded 4-pass columnsort, our second implementation [10], was an engineering effort that used threads in order to provide greater flexibility in overlapping I/O, communication, and computation. Threads allow scheduling to be more dynamic, and they permit greater flexibility in memory usage. Experimental results showed that this improvement reduced the running time to about half of that of the non-threaded implementation. In this implementation, there were four threads per processor. As Figure 2 shows, the sort, communicate, and permute phases each had their own threads, and the read and write phases shared an I/O thread. The read and write phases appeared in a common thread because they would have serialized at the disk anyway. The threads operated on buffers, each capable of holding exactly r records, and which were drawn from a global pool. The threads communicated with each other via a standard semaphore mechanism.

Threaded 3-pass columnsort, which we will simply call 3-pass columnsort throughout this paper, reduced the number of passes from four down to three by combining the last two passes into a single pass. This improvement has both algorithmic and engineering aspects. Subblock columnsort and M-columnsort

use this implementation as the starting point.

Problem-size restriction

We recall the problem-size restriction (1) from Section 1. All of the previous implementations are subject to this problem-size restriction, since they use the original columnsort algorithm, inheriting its height restriction, and they use the height interpretation of $r = M/P$. In other words, substituting $r = M/P$ and $s = N/r = NP/M$ in the height restriction $r \geq 2s^2$ gives restriction (1): $N \leq \sqrt{(M/P)^3/2}$. There are two main implications of this restriction. The first implication is the obvious one: the maximum problem size has an upper bound. The second implication is one of scalability. As we can see in restriction (1), the problem size N depends on M/P, the memory per processor, rather than on the total memory M of the system. In other words, if we double the number of processors, without changing the memory per processor, the maximum problem size does not increase.

3.1 Subblock columnsort

Subblock columnsort relaxes the height restriction by a factor of $\sqrt{s}/2$, to $r \geq 4s^{3/2}$, by adding two steps to columnsort. With a height interpretation of $r = M/P$, we get problem-size restriction (2): $N \leq (M/P)^{5/3}/4^{2/3}$. Subblock columnsort requires s to be a perfect square.

Since subblock columnsort differs from columnsort only in the two additional steps, our implementation of subblock columnsort started with the 3-pass columnsort program and integrated these two steps as one extra pass. The overall thread structure of subblock columnsort is the same as that of the 3-pass columnsort program.

3.2 M-columnsort

In order to achieve the problem-size restriction (3), $N \leq \sqrt{M^3/2}$, we consider each column to be M elements, so that $r = M$. Recall that with this more relaxed restriction, the maximum problem size now scales with the memory in the entire system, so that adding more processors with the same amount of memory per processor increases the maximum problem size. In fact, this increase is superlinear in the total memory size.

As with subblock columnsort, our implementation of M-columnsort is a modification of 3-pass columnsort. Instead of adding a pass, however, we increase the complexity of the sort phase of each pass. When $r = M/P$, the sort phase is just a local sort on each processor. In M-columnsort, since $r = M$, the sort phase becomes a multiprocessor sort with distributed memory. One benefit of performing a multiprocessor sort is that we can eliminate the communicate phase in the first two passes.

4 Slabpose columnsort

This section presents our new oblivious algorithm: slabpose columnsort. Just like subblock columnsort and M-columnsort, slabpose columnsort relaxes the problem-size restriction. Unlike subblock columnsort and M-columnsort, however, slabpose columnsort does so at no extra I/O or communication cost. We first describe the algorithm; Appendix A presents the proof of its correctness. Next, we explain how a 3-pass implementation of this 10-step algorithm is possible. We conclude with a discussion on the new problem-size restriction and some implementation notes.

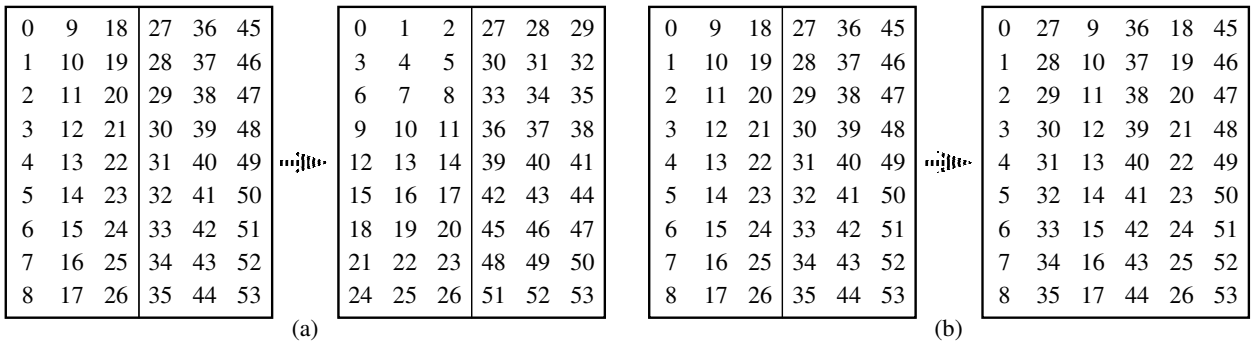

Figure 3: k-slabpose and k-shuffle operations, shown for $k = 3$ on a 9×6 matrix. **(a)** A k-slabpose operation. **(b)** A k-shuffle operation.

4.1 The algorithm

Slabpose columnsort is based on partitioning the matrix into several vertical "slabs," where we define a *k-slab* as a set of k consecutive columns, with the leftmost column in the slab at an index that is a multiple of k. (The inspiration for using slabs comes from the work of Marberg and Gafni [28] for sorting on a square matrix.) When forming slabs, we assume that k is a divisor of s (and hence, by the divisibility restriction, a divisor of r). For a given value of k, there are s/k k-slabs; since k is a divisor of s, so is s/k.

This algorithm requires two new operations, both of which are oblivious to the data being sorted:

- A *k-slabpose*, shown in Figure 3(a), is a transpose and reshape operation, but within each k-slab. Just as step 2 of the original columnsort algorithm transposes the matrix and reshapes it back into an $r \times s$ arrangement, a k-slabpose transposes within each k-slab and reshapes it back into an $r \times k$ configuration.

- A *k-shuffle*, shown in Figure 3(b), is a permutation of the s columns of the matrix in which we first take in order all columns whose indices are congruent to 0 modulo k, then take in order all columns whose indices are congruent to 1 modulo k, then 2 modulo k, and so on. More precisely, to determine which index column j maps to, let $j = lk + m$, where $0 \le l < s/k$ and $0 \le m < k$; then column j maps to index $ms/k + l$. Since $l = \lfloor j/k \rfloor$ and $m = j \bmod k$, column j maps to index $\mathcal{T}_j$, where

$$\mathcal{T}_j = (j \bmod k)s/k + \lfloor j/k \rfloor \,. \tag{4}$$

With these two new operations, we present the algorithm, which we call *slabpose columnsort*. Start by choosing any divisor k of s. Then perform the following 11 steps:

- Step 1 sorts each column.
- Step 2 performs a k-slabpose.
- Step 3 sorts each column.
- Step 4 performs a k-shuffle.
- Step 5 performs an (s/k)-slabpose.

- Steps 6–11 are the same as steps 3–8 of the original columnsort algorithm.

Note that because steps 4 and 5 are consecutive and both perform fixed permutations, they can be replaced by a single step that performs the composition of the k-shuffle and (s/k)-slabpose permutations. The resulting algorithm would have 10 steps rather than 11. To ease understanding, however, we shall focus on the 11-step formulation in the remainder of this section.

When $k = 1$, slabpose columnsort reduces to the original columnsort algorithm. Steps 2 and 4 become identity permutations, and so step 3 becomes redundant. Step 5 becomes a transpose-and-reshape operation over the whole matrix, just like step 2 of the original columnsort algorithm.

We refer the reader to Appendix A for the proof of correctness of slabpose columnsort.

4.2 A 3-pass implementation

In this section, we describe a 3-pass implementation of one particular case of slabpose columnsort. We set the parameter k of slabpose columnsort to be equal to the number of processors in the out-of-core setting, i.e., $k = P$. Hereafter, we assume that $k = P$ in any implementation of slabpose columnsort.

The out-of-core setting is inherited from our earlier implementations [10, 11, 12, 13]. The three passes of the out-of-core implementation are structured so that the first pass performs steps 1–5 of slabpose columnsort and passes 2 and 3 together perform steps 6–11 of slabpose columnsort.

First, we elaborate how to complete steps 1–5 of slabpose columnsort in one single pass over the data. We then explain why passes 2 and 3 of slabpose columnsort are the same as passes 2 and 3, respectively, of 3-pass columnsort.

4.2.1 Pass 1 performs steps 1–5

In all our implementations, we have been assuming that column j maps to processor $\mathcal{P}_{j \bmod P}$. We shall refer to this mapping as the *P-cyclic mapping*. We will show that if we stick to the P-cyclic mapping, step 4 of slabpose columnsort has several rather severe performance drawbacks. Then, we will explain how we can overcome all of these drawbacks by carefully changing the column-to-processor mapping for a few chosen steps.

Step 4 with the *P*-cyclic mapping

In each communication phase of all the passes of all our implementations so far, each processor receives M/P records. First, we will show how it is possible that some processors receive many more than M/P records, if we assume the P-cyclic mapping in step 4 of slabpose columnsort. We will then point out some performance implications of this fact.

To show that a processor can receive more than M/P records, consider step 4 (the P-shuffle) of any round, say round x. According to the P-cyclic mapping, in round x, processor $\mathcal{P}_i$ has the column numbered $j = xP + i$. Let $\mathcal{T}_j$ be the target column of column j, according to the P-shuffle. By formula (4), with $k = P$, $j \bmod P = i$, and $\lfloor j/P \rfloor = x$, we have $\mathcal{T}_j = is/P + x$. According to the P-cyclic mapping, column $\mathcal{T}_j$ maps to the processor numbered $\mathcal{T}_j \bmod P$. Now, consider the case in which P divides s/P,[4] implying that $\mathcal{T}_j \bmod P = x \bmod P$. Therefore, if P divides s/P, all P columns in the memory of the system map to one single processor, namely $\mathcal{P}_{x \bmod P}$, implying that $\mathcal{P}_{x \bmod P}$ will receive M records. This situation leads to the following performance drawbacks:

[4]This assumption is true in all our experiments.

- All M records go to processor $\mathcal{P}_{x \bmod P}$, leading to highly unbalanced communication.
- Since each processor can hold only M/P records in its memory, processor $\mathcal{P}_{x \bmod P}$ somehow needs to manage the M records that it receives.
- Processor $\mathcal{P}_{x \bmod P}$ must write these M records to its disks. Because all the other processors receive no records, they must wait for processor $\mathcal{P}_{x \bmod P}$ to finish its write operation, which leads to highly unbalanced I/O as well.
- The first pass must end with step 4, since the M records must be written out to disks. Step 5, therefore, will need another pass over the data.

Steps 1–5 in our implementation

Now, we explain how, by carefully changing the column-to-processor mapping in some steps, we not only overcome the performance drawbacks mentioned above, but we do so while requiring step 4 to perform no communication whatsoever.

We first define a new way to map columns to processors: the (s/P)-*blocked mapping*. According to the (s/P)-blocked mapping, the first s/P columns map to $\mathcal{P}_0$, the next s/P columns map to $\mathcal{P}_1$, and so on. That is, columns with indices $\{0, 1, 2, \ldots, (s/P - 1)\}$ map to processor $\mathcal{P}_0$, columns with indices $\{(s/P), (s/P) + 1, \ldots, 2s/P - 1\}$ map to processor $\mathcal{P}_1$, and so on. In general, column j maps to processor $\mathcal{P}_{\lfloor j/(s/P) \rfloor}$. We observe that according to the (s/P)-blocked mapping, processor $\mathcal{P}_i$ owns the ith (s/P)-slab.

Similar to most passes in our implementations, pass 1 of slabpose columnsort proceeds in s/P rounds. In any given round x of this pass, where $0 \leq x < s/P$, a column goes through seven phases. The mapping of columns to processors differs among some phases. When we say that a phase assumes a given column-to-processor mapping $\mathcal{M}$, we mean that at the end of the phase, each column should reside with the processor that owns this column according to the mapping $\mathcal{M}$. Below, we describe the seven phases, specifying the column-to-processor mapping assumed in each individual phase:

- *Read phase* (P-cyclic mapping). Each processor reads in a column. In round x, processor $\mathcal{P}_i$ reads the column numbered $j = xP + i$. After the read phase, the xth P-slab is in the internal memory of the system.
- *First sort phase* (P-cyclic mapping). This phase corresponds to step 1 of slabpose columnsort. Each processor sorts the $r = M/P$ records in its local memory.
- *Communicate phase* (P-cyclic mapping). This phase corresponds to step 2, a P-slabpose, of slabpose columnsort. An all-to-all communication performs the P-slabpose on the xth P-slab. At the end of this phase, each processor has P sorted runs, where each run has r/P records.
- *Second sort phase* (P-cyclic mapping). This phase corresponds to step 3 of slabpose columnsort. Each processor sorts the r records in its local memory. Since the r records are composed of P sorted runs, this step is a recursive merge sort with just $\lg P$ levels of recursion.
- *Shuffle phase* ((s/P)-blocked mapping). This phase corresponds to step 4, a P-shuffle, of slabpose columnsort. Surprisingly, this phase is just a no-op, i.e., each processor does nothing.

Let us see why each processor does nothing in the shuffle phase. By formula (4), with $k = P$, $j \bmod P = i$, and $\lfloor j/P \rfloor = x$, column j maps to column $\mathcal{T}_j = is/P + x$. We know that column j maps to processor $\mathcal{P}_i$ according to the P-cyclic mapping, the mapping prior to the shuffle phase. For the shuffle phase to be a no-op, it is enough to show that column $\mathcal{T}_j$ maps to processor $\mathcal{P}_i$ as well, albeit according to the (s/P)-blocked mapping, the mapping that the shuffle phase assumes. By the definition of the (s/P)-blocked mapping, column $\mathcal{T}_j$ maps to the processor numbered $\lfloor \mathcal{T}_j/(s/P) \rfloor = i$. Therefore, according to the (s/P)-blocked mapping, column $\mathcal{T}_j$ maps to processor $\mathcal{P}_i$, thus proving that the shuffle phase is a no-op.

- *Permute phase* ((s/P)-blocked mapping). To execute step 5 of slabpose columnsort, each processor performs an (s/P)-slabpose on one slab. According to the (s/P)-blocked mapping, processor $\mathcal{P}_i$ owns columns with indices $\{i(s/P), i(s/P)+1, \ldots, (i+1)(s/P)-1\}$, i.e., $\mathcal{P}_i$ owns the ith (s/P)-slab. Therefore, each processor owns one of the P (s/P)-slabs, implying that the (s/P)-slabpose requires no communication.

- *Write phase* ((s/P)-blocked mapping). Each processor writes out the r records in its local memory to its disks.

4.2.2 Passes 2 and 3 perform steps 6–11

Here, we describe how the last two passes of subblock columnsort are the same as the last two passes of 3-pass columnsort.

Pass 2 performs steps 6 and 7

In pass 2 of slabpose columnsort, we assume the P-cyclic mapping, just as in pass 2 of 3-pass columnsort. Note that pass 2 of slabpose columnsort and pass 2 of 3-pass columnsort execute the same steps: sort each column and then do a reshape-and-transpose operation on the entire matrix. The mapping prior to step 2 of slabpose columnsort is different from the mapping prior to step 2 of 3-pass columnsort. In slabpose columnsort, since the column-to-processor mapping at the end of pass 1 is (s/P)-blocked, the mapping prior to step 2 is (s/P)-blocked as well.

We claim that pass 2 of slabpose columnsort is identical to pass 2 of 3-pass columnsort, despite the difference in the column-to-processor mappings prior to step 2. In other words, slabpose columnsort can revert back to the P-cyclic mapping from pass 2 onward. Each processor can execute pass 2 as if the mapping has always been P-cyclic.

To prove our claim, we argue the following about slabpose columnsort: step 6 is independent of any column-to-processor mapping, and step 7 is independent of the mapping prior to pass 2. Step 6 just sorts each individual column and therefore is independent of the column-to-processor mapping. In other words, each processor can locally sort the column it has in its local memory without violating the correctness of the algorithm.

At the beginning of step 7, it does not matter which column is stored on which processor. Figure 4 illustrates the mechanics of step 7. Let us examine any single column, say column j. Step 7 distributes the r elements in column j among all s columns of the matrix, with r/s elements each. An element in row i of column j maps to column $\lfloor i/r \rfloor$; this target column number depends only on the row number i and not on the column number j. Because step 8, the next step, sorts each column, the placement of an element within a column during step 7 is immaterial. Therefore, step 7 is independent of the mapping prior to pass 2.

$$\begin{bmatrix} 1 & 7 & 13 \\ 2 & 8 & 14 \\ 3 & 9 & 15 \\ 4 & 10 & 16 \\ 5 & 11 & 17 \\ 6 & 12 & 18 \end{bmatrix} \xrightarrow{\text{reshape-and-transpose}} \begin{bmatrix} 1 & 3 & 5 \\ 7 & 9 & 11 \\ 13 & 15 & 17 \\ 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

Figure 4: The operation of the reshape-and-transpose step of columnsort. For simplicity, we choose this small 6×3 matrix to illustrate the step.

Pass 3 performs steps 8–11

Pass 3 of slabpose columnsort is identical to pass 3 of 3-pass columnsort. Given that the column-to-processor mapping of pass 3 of both algorithms is the same (P-cyclic), and given that steps 8–11 of slabpose columnsort are the same as steps 5–8 of 3-pass columnsort, this observation does not come as a surprise.

4.3 Problem-size bound of the 3-pass implementation

As shown in Appendix A, as long as k is chosen as a divisor of s, s is a divisor of r, and $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$, slabpose columnsort sorts correctly. In our case, since $k = P$, the restriction on the height of the column translates to $r \geq (2s^2/P)(\lceil P^2/s \rceil + 1)$.

We consider two cases:

- Case 1: $P^2 \leq s$, implying that $\lceil P^2/s \rceil + 1 = 2$, which in turn implies that the restriction $r \geq 4s^2/P$ suffices. Letting $r = M/P$ and $s = N/r$, we get the problem-size restriction

$$N \leq (M/P)^{3/2} \frac{\sqrt{P}}{2} . \tag{5}$$

  Compared to the problem-size restriction (1) of 3-pass columnsort, this problem-size restriction is relaxed by a factor of $\sqrt{P}$. Note that slabpose columnsort does the same amount of I/O and communication as 3-pass columnsort. Therefore, this improvement in problem-size restriction comes at almost no performance cost.

- Case 2: $P^2 > s$, implying that $\lceil P^2/s \rceil \geq 1$ and $\lceil P^2/s \rceil + 1 \leq 2P^2/s$. Here, the restriction $r \geq 4sP$ suffices, since

$$\begin{aligned} (2s^2/P)(\lceil P^2/s \rceil + 1) &\leq (2s^2/P)(2P^2/s) \\ &= 4sP . \end{aligned}$$

  Letting $r = M/P$ and $s = N/r$, we get the problem-size restriction

$$N \leq \frac{(M/P)^2}{4P} . \tag{6}$$

  Although the exponent of M/P in this restriction is much better than that of restriction (5), the maximum problem size decreases as we increase the number of processors. As long as

$(M/P)^{1/3}/4^{1/3} \geq P$, however, this limit will be better than that of restriction (5). Therefore, we have a crossover point in terms of the number of processors; increasing the number of processors beyond this point makes the problem-size restriction worse than that of restriction (5). This crossover point, however, improves with increases in the amount of memory per processor. For example, if $M/P = 2^{21}$, the crossover point is 64 processors, whereas if $M/P = 2^{24}$, the crossover point is 128 processors.

4.4 Implementation notes

Slabpose columnsort differs from 3-pass columnsort only in the first pass. Furthermore, this difference is restricted to the permute phase. In the first pass of 3-pass columnsort, the permute phase rearranges records in the correct order of writing. In the first pass of slabpose columnsort, however, since the shuffle phase is a no-op, the second sort phase can be combined into the permute phase, adding to the complexity of the permute phase.

5 Experimental results

This section presents the results of our experiments on *Jefferson*, a Beowulf cluster that belongs to the Computer Science Department at Dartmouth. We start with a brief description of Jefferson. After outlining our experimental setup, we analyze the performance of our four implementations.

5.1 Jefferson

Jefferson is a Beowulf cluster of 32 dual 2.8-GHz Intel Xeon nodes. Each node has 4 GB of RAM and an Ultra-320 36-GB hard drive. A high-speed Myrinet connects the network. At the time of our experiments, each node ran Redhat Linux 8.0. We use the C `stdio` interface for disk I/O, the `pthreads` package of Linux, and standard synchronous MPI calls within threads. We use the MPI/Pro package for MPI calls.

5.2 Experimental setup

Our experimental runs were for combinations of the following:

Algorithm: We ran 3-pass columnsort, subblock columnsort, M-columnsort, and slabpose columnsort. For a baseline, we also ran just the I/O portions of three and four passes of columnsort. Recall that subblock columnsort makes 4 passes over the data, and slabpose columnsort, M-columnsort, and 3-pass columnsort each make 3 passes over the data.

Buffer size: For all our implementations except for subblock columnsort, we use 64-MB buffers. For subblock columnsort, since s has to be a perfect square, some problem sizes were impossible using 64-MB buffers; we used 128-MB buffers in such cases. Note that these buffer sizes, being in bytes, are not in terms of number of records, and so they should not be construed as equaling M/P. The record size in all experiments is 64 bytes.

Number of processors and volume of data: We ran various combinations with 2, 4, 8, and 16 processors and with an amount of data varying from 8 GB up to 128 GB. We did not run any experiments with less than 4 GB of data per processor because file-caching effects masked the out-of-core nature of

the problem. We were unable to perform any runs with more than 8 GB of data per processor due to disk-space limitation.

5.3 Relative performance of the four threaded implementations

In this section, we review various aspects of the observed performance of our four implementations. We first examine how I/O bounded each implementation was. Then, we analyze the running times of our three algorithms that relax the problem-size restriction: slabpose columnsort, subblock columnsort, and M-columnsort.

Due to the problem-size restriction (1), 3-pass columnsort could not handle more than 32 GB of data. The problem-size restrictions of the other three algorithms allowed all the problem sizes that we could sort, given the disk-space limitation.

I/O boundedness

We ran our experiments in two sets. In the first set, the per-processor problem size was 4 GB; in the second set, it was 8 GB. For a given algorithm and buffer size, we found that the amount of data per processor was by far the most important factor in determining run time. Given the large amount of disk I/O that each of the algorithms has to perform, this characteristic did not come as a surprise. M-columnsort, however, was an exception to this rule. In M-columnsort, the substantial amounts of communication and computation exceeded the I/O times by far.

Figures 5 and 6 show the ratios of total running times and the baseline I/O times for the 4-GB case and the 8-GB case, respectively. For the 4-GB case, the maximum value for this ratio was 1.78, and the average value was 1.16. For the 8-GB case, the maximum value was 1.5, and the average was 1.07. The maximum values, in both cases, are due to M-columnsort, which is clearly not I/O bound. Each data point is an average of three runs.

Excluding M-columnsort, the maximum values of these ratios are 1.33 and 1.20 for the 4-GB and the 8-GB cases, respectively; the average values are 1.09 and 1.02. These average values show that, but for M-columnsort, all our implementations are fairly I/O-bound. Furthermore, the average as well as the maximum values are lower for the 8-GB case, showing that the 8-GB case is more representative of an out-of-core scenario.

The observed running times

Figure 7 shows the running times of our experiments with 4 GB of data per processor; Figure 8 covers the experiments with 8 GB of data per processor. Each plotted point in the figure represents the average of three runs of an algorithm. Variations in running times were relatively small (within 5%). The horizontal axis is organized by the total amount of data, in GB, sorted across all processors.

For the problem sizes of 16 GB and 32 GB in Figure 8, subblock columnsort is almost below the baseline 4-pass I/O time. It might seem like an anomaly that an implementation takes less time than the baseline I/O time. This difference in timings, however, is not much of an anomaly since the presented timings are averages of multiple runs.

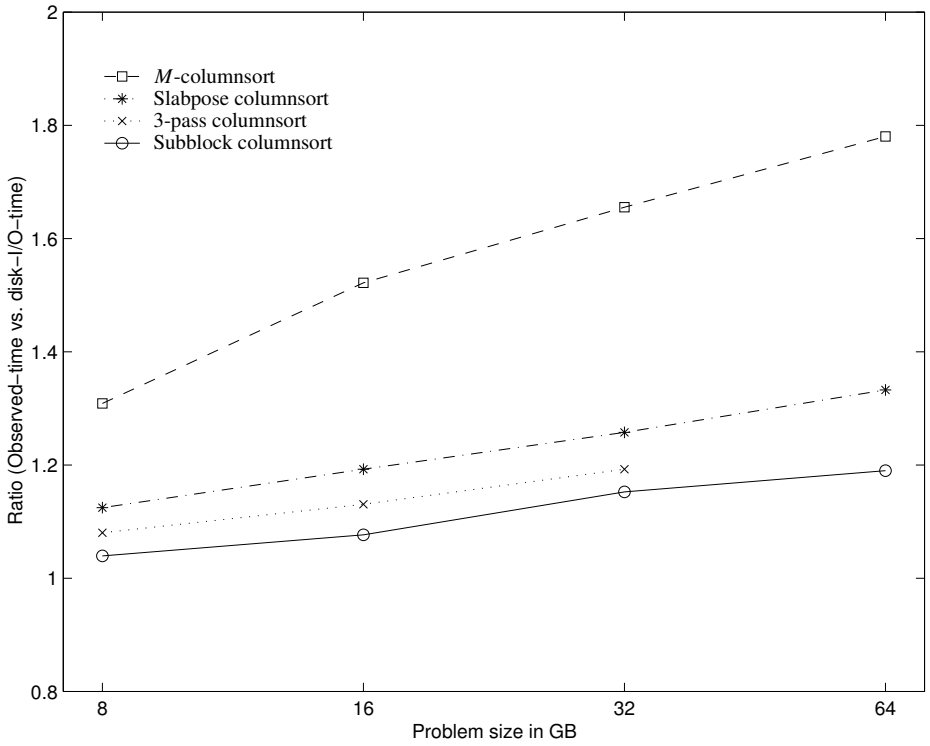

Figure 5: Ratio of observed running time vs. the baseline I/O time for the four threaded implementations with 4 GB of data per processor. The ratio (observed-time/I/O-time) is on the vertical axis. For 3-pass columnsort, only problem sizes up to 32 GB were possible, due to the problem-size restriction.

Slabpose columnsort

Slabpose columnsort performs similarly to 3-pass columnsort. The performance of slabpose columnsort degrades as the problem size increases. The good news is that this degradation is less pronounced in the 8-GB case than in the 4-GB case; we can expect that for per-processor problem sizes that are prominently out-of-core, slabpose columnsort would scale well. Since slabpose columnsort relaxes the problem-size restriction at almost no extra cost (it has one extra sort phase), we expect slabpose columnsort to be the algorithm of choice for problem sizes that 3-pass columnsort cannot handle. Our experiments show that, for problem sizes above 32 GB, slabpose columnsort is the clear winner.

Subblock columnsort

For both the 4-GB and 8-GB cases, subblock columnsort runs in approximately 125% of the time taken by 3-pass columnsort. Also, the performance of subblock columnsort scales well with the problem size. Hence, subblock columnsort scales slightly better than slabpose columnsort. We believe that there are two reasons for this difference in scalability. First, for most cases ($s \geq P^2$), the one additional pass does no communication. Second, the first pass of slabpose columnsort has two sort phases, compared to the single sort phase of the first pass of subblock columnsort. The scenario in which we might consider using subblock columnsort instead of slabpose columnsort would be a system in which I/O is much cheaper than

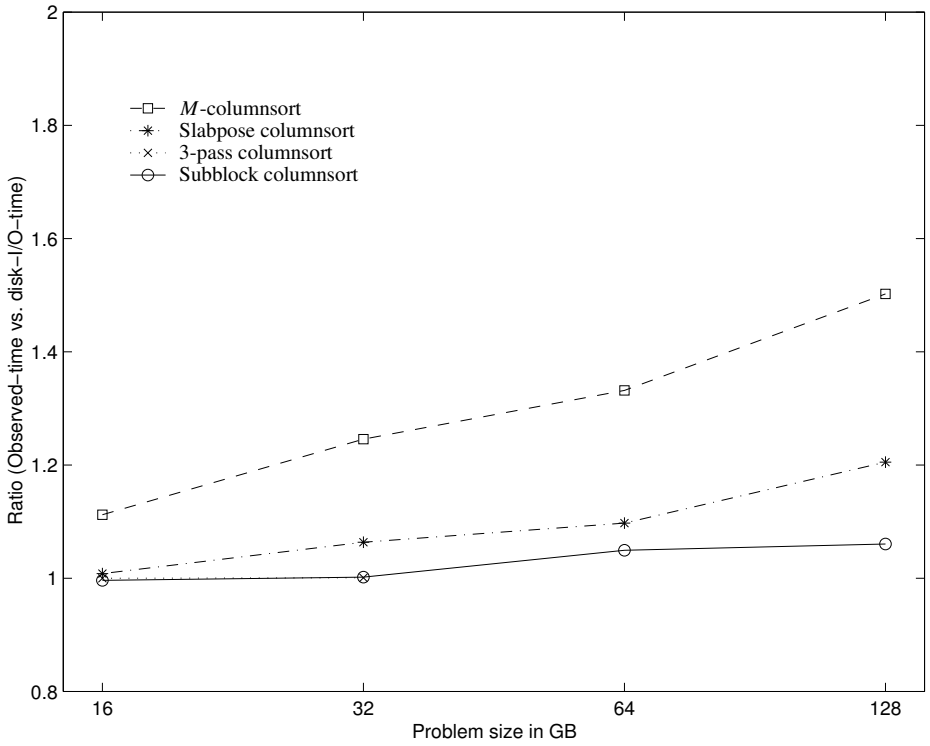

Figure 6: Ratio of observed running time vs. the baseline I/O time for the four threaded implementations with 8 GB of data per processor. The ratio (observed-time/I/O-time) is on the vertical axis. For 3-pass columnsort, only problem sizes up to 32 GB were possible, due to the problem-size restriction.

communication and computation.

M-columnsort

For all problem sizes, and all across our experiments, M-columnsort is the slowest. Each pass of M-columnsort has several phases, compared to at most seven of 3-pass columnsort.[5] Each pass, however, has only two I/O phases: the read phase and the write phase. Jefferson is not nearly as I/O bound to hide so many communication and computation times behind disk I/O. The increased number of phases in M-columnsort is due to the column being M records tall, thus introducing a parallel in-core sort step in each round. For this parallel sorting, we use an in-core parallel version of columnsort. In future, we would like to explore a different sorting algorithm for this purpose, in particular, one that does not have as many computation and communication phases.

6 Conclusion and future work

We have begun to explore the third paradigm, that of oblivious algorithms, for out-of-core sorting on distributed-memory clusters. As a first step, we developed three programs [10, 12] that sort out-of-core

[5]Passes 1 and 2 have a total of 11 phases, whereas pass 3 has a total of 20 phases.

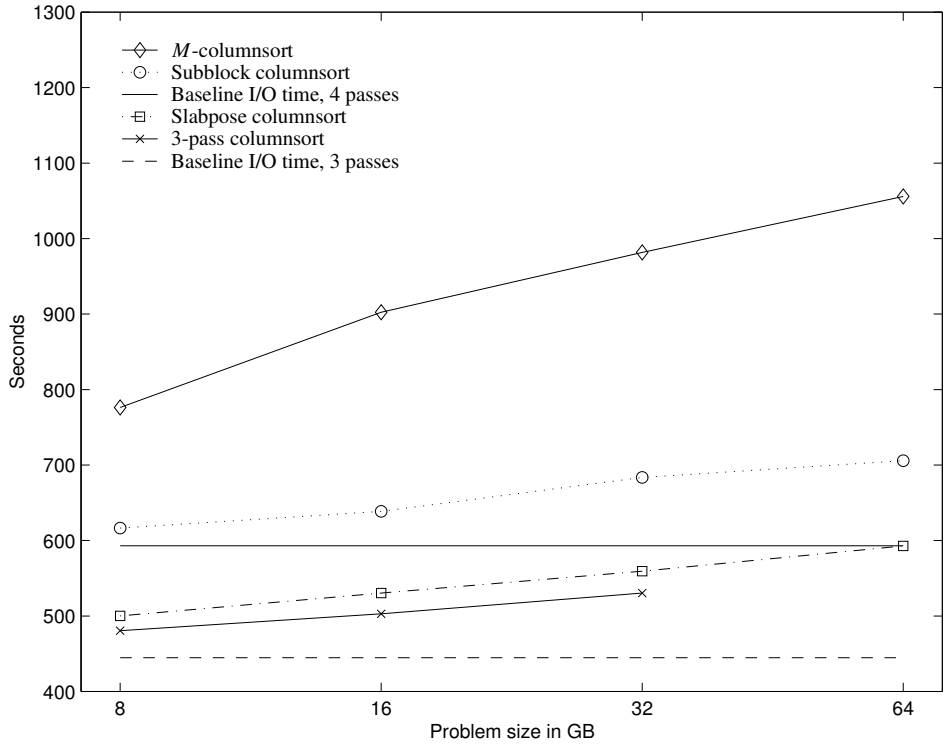

Figure 7: Actual completion times, with 4 GB as the per-processor problem size. The times are on the vertical axis, in seconds. For each of subblock columnsort, slabpose columnsort, and M-columnsort, the figure shows completion times for four problem sizes. For 3-pass columnsort, however, only problem sizes up to 32 GB were possible, due to the problem-size restriction. The two horizontal lines represent the baseline I/O times for three and four passes.

data on distributed-memory clusters. Each program is based on Leighton's columnsort [27]. Although these algorithms are robust, they suffer from the restriction $N \leq \sqrt{(M/P)^3/2}$ on the maximum problem size N that they can sort. To relax this restriction, we developed subblock columnsort and M-columnsort. Subblock columnsort is an algorithmic extension to columnsort; it relaxes the restriction at the cost of an additional pass. M-columnsort, more of an engineering effort, relaxes the restriction at the cost of substantial amounts of extra communication and computation.

In this paper, we present a new oblivious algorithm, slabpose columnsort, that we have designed explicitly for the out-of-core setting. Unlike subblock columnsort and M-columnsort, slabpose columnsort relaxes the problem-size restriction at no extra I/O or communication costs. Slabpose columnsort has ten steps, compared to columnsort's eight. We show how to adapt slabpose columnsort to the out-of-core setting, resulting in a 3-pass algorithm which does the same exact amounts of I/O and communication as 3-pass columnsort.

In all our implementations, there are no assumptions required about the keys. In fact, our algorithm's I/O and communication patterns are oblivious to the keys. We have engineered our implementations to make sure that the I/O operations are parallelized and the output appears in the standard striped ordering used by the Parallel Disk Model [40]. To the best of our knowledge, our implementations are the first out-of-core multiprocessor sorting algorithms that produce output in the order assumed by the Parallel Disk Model and make no assumptions about the keys.

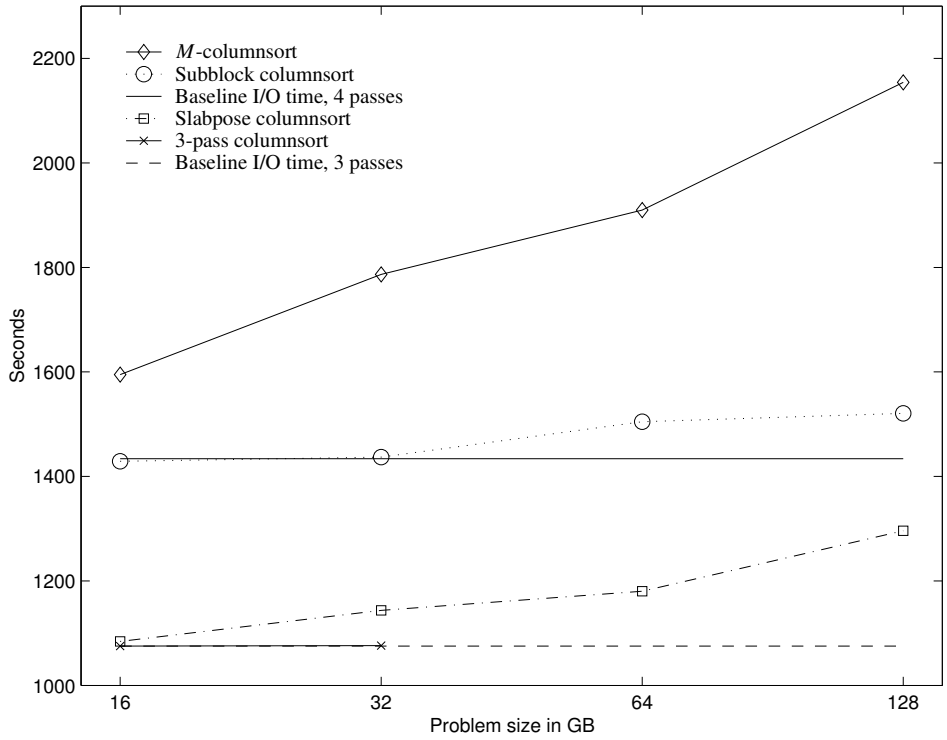

Figure 8: Actual completion times, with 8 GB as the per-processor problem size. The times are on the vertical axis, in seconds. For each of subblock columnsort, slabpose columnsort, and M-columnsort, the figure shows completion times for four problem sizes. For 3-pass columnsort, however, only problem sizes up to 32 GB were possible, due to the problem-size restriction. The two horizontal lines represent the baseline I/O times for three and four passes.

There are several directions for our future work; we describe two of them here. We are exploring whether our oblivious sorting algorithms run faster than out-of-core partitioning-based algorithms on clusters. To this end, we have had to design and implement our own partitioning-based algorithm. Our first cut at such an algorithm takes advantage of simplifying assumptions about the input keys that are unlikely to hold in practice. Even with these simplifying assumptions, our oblivious sorting algorithm runs faster in many cases [9]. We are in the midst of developing a more robust partitioning-based algorithm, which makes no assumptions about the keys. We expect that this more robust algorithm will take even longer, and might be slower than our oblivious method in all cases. Another direction for future work is to design an oblivious algorithm that can be engineered into a 2-pass implementation.

Acknowledgments

MPI/Pro is a commercial product from MPI Software Technology, Inc. We thank Tony Skjellum, David Leimbach, Rossen Dimitrov, Weiyi Chen, and Tracey Miller of MPI Software Technology, Inc., for assistance with installing and using MPI/Pro.

References

[1] Alok Aggarwal and C. Greg Plaxton. Optimal parallel sorting in multi-level storage. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 659–668, January 1994.

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

[3] Thomas E. Anderson, David E. Culler, David E. Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[4] Lars Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, volume 1340 of *Lecture Notes in Computer Science*, pages 213–254. Springer-Verlag, 1997.

[5] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *SIGMOD '97*, pages 243–254, May 1997.

[6] David A. Bader, David R. Helman, and Joseph Jájá. Practical parallel algorithms for personalized communication and integer sorting. *Journal of Experimental Algorithmics*, 1(3):1–42, 1996.

[7] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, June 1997.

[8] Rakesh D. Barve and Jeffrey Scott Vitter. A simple and efficient parallel disk mergesort. *Theory of Computing Systems*, 35(2):189–215, April 2002.

[9] Geeta Chaudhry and Thomas H. Cormen. Oblivious vs. distribution-based sorting: An experimental evaluation. To appear in ESA 2005.

[10] Geeta Chaudhry and Thomas H. Cormen. Getting more from out-of-core columnsort. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX 02)*, pages 143–154, January 2002.

[11] Geeta Chaudhry, Thomas H. Cormen, and Elizabeth A. Hamon. Parallel out-of-core sorting: The third way. *Cluster Computing*. To appear.

[12] Geeta Chaudhry, Thomas H. Cormen, and Leonard F. Wisniewski. Columnsort lives! An efficient out-of-core sorting program. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 2001.

[13] Geeta Chaudhry, Elizabeth A. Hamon, and Thomas H. Cormen. Relaxing the problem-size bound for out-of-core columnsort. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, June 2003. SPAA Revue paper.

[14] Thomas H. Cormen and Melissa Hirschl. Early experiences in evaluating the Parallel Disk Model with the ViC* implementation. *Parallel Computing*, 23(4–5):571–600, June 1997.

[15] Roman Dementiev and Peter Sanders. Asynchronous parallel disk sorting. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 138–148, June 2003.

[16] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 280–291. IEEE Computer Society Press, 1991.

[17] Goetz Graefe. Parallel external sorting in Volcano. Technical Report CU-CS-459-90, University of Colorado at Boulder, Department of Computer Science, March 1990.

[18] Jim Gray. http://research.microsoft.com/barc/sortbenchmark/. Sort Benchmark Home Page.

[19] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI—The Complete Reference, Volume 2, The MPI Extensions*. The MIT Press, 1998.

[20] David R. Helman, David A. Bader, and Joseph Jájá. A randomized parallel sorting algorithm with an experimental study. *Journal of Parallel and Distributed Computing*, 52(1):1–23, July 1998.

[21] David R. Helman and Joseph Jájá. Sorting on clusters of SMPs. Technical Report CS-TR-3833, University of Maryland, College Park, Department of Computer Science, 1997.

[22] David R. Helman, Joseph Jájá, and David A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *Journal of Experimental Algorithmics*, 3(4):1–24, 1998.

[23] David A. Hutchinson, Peter Sanders, and Jeffrey Scott Vitter. Duality between prefetching and queued writing with applications to external sorting. In *European Symposium on Algorithms*, volume 2161 of *Lecture Notes in Computer Science*, pages 62–73. Springer-Verlag, August 2001.

[24] David A. Hutchinson, Peter Sanders, and Jeffrey Scott Vitter. The power of duality for prefetching and sorting with parallel disks. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 334–335, July 2001.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley, 1998.

[26] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[27] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[28] John M. Marberg and Eli Gafni. Sorting in constant number of row and column phases on a mesh. *Algorithmica*, 3:561–572, 1988.

[29] Mark H. Nodine and Jeffrey Scott Vitter. Large-scale sorting in parallel memories. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 29–39, July 1991.

[30] Mark H. Nodine and Jeffrey Scott Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, June 1993.

[31] Mark H. Nodine and Jeffrey Scott Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, July 1995.

[32] Markus Pawlowski and Rudolf Bayer. Parallel sorting of large data volumes on distributed memory multiprocessors. In Arndt Bode and Mario Dal Cin, editors, *Parallel Computer Architectures: Theory, Hardware, Software, Applications*, pages 246–264. Springer-Verlag, 1993.

[33] Matthew D. Pearson. Fast out-of-core sorting on parallel disk systems. Technical Report PCS-TR99-351, Dartmouth College Department of Computer Science, June 1999.

[34] Sanguthevar Rajasekaran. A framework for simple sorting algorithms on parallel disk systems. *Theory of Computing Systems*, 34(2):101–114, 2001.

[35] Sanguthevar Rajasekaran and Xiaoming Jin. A practical realization of parallel disks. In *International Workshop on Parallel Processing*, pages 337–344, August 2000.

[36] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference, Volume 1, The MPI Core*. The MIT Press, 1998.

[37] Jeffrey Scott Vitter. Efficient memory access in large-scale computation. In *Proceedings of the 1991 Symposium on Theoretical Aspects of Computer Science (STACS '91)*, pages 26–41, Berlin, 1991. Springer-Verlag. Published as *Lecture Notes in Computer Science* volume 480.

[38] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[39] Jeffrey Scott Vitter and David A. Hutchinson. Distribution sort with randomized cycling. In *12th Annual SIAM/ACM Symposium on Discrete Algorithms*, pages 77–86, January 2001.

[40] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.

A Proof of correctness of slabpose columnsort

To prove the correctness of slabpose columnsort, we use the *0-1 Principle* [26, pp. 141–142]:

> If an oblivious algorithm sorts all input sets consisting solely of 0s and 1s, then it sorts all input sets with arbitrary values.

We first need to establish that slabpose columnsort is oblivious. Clearly, the non-sort steps perform oblivious permutations. The sorting method used in sort steps might not be oblivious, however. As pointed out by Leighton [26, p. 147], "No matter how the columns are sorted, the end result will look the same." Thus, we can imagine that an oblivious sorting method was used for the sort steps, knowing that we can substitute any sorting method of our choosing.

When given a 0-1 input, we say that an area of the matrix is *clean* if it consists either of all 0s or all 1s. An area that might have both 0s and 1s is *dirty*. We shall show that steps 1–7 of the 11-step slabpose columnsort algorithm reduce the size of the dirty area to at most half a column and that steps 8–11 complete the sorting, assuming that the dirty area is at most half a column in size. As we read 0-1 values in a prescribed order within the matrix, a *0 → 1 transition* occurs when a 0 is followed immediately by a 1; we define a *1 → 0 transition* analogously.

We begin with the following lemma, which shows that slabpose columnsort sorts correctly if the size of the dirty area, after step 7, is at most half a column.

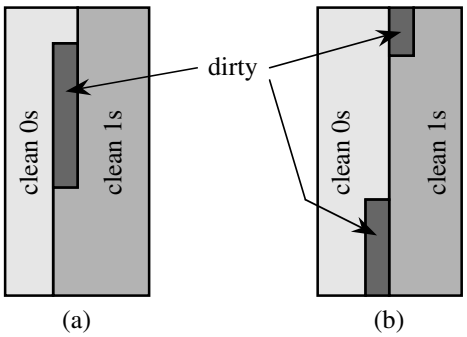

Figure 9: If after step 7 the dirty area is at most half a column in size, then it either **(a)** fits in a single column or **(b)** crosses from one column to the next.

Lemma 1 *Assuming a 0-1 input, if the dirty area is at most half a column in size after step 7, then steps 8–11 produce a fully sorted output.*

Proof: As Figure 9 shows, because the size of the dirty area is at most half a column, it either fits in a single column or crosses from one column into the next. If the dirty area fits in a single column, then the sorting of step 8 cleans it, and steps 9–11 do not corrupt the sorted 0-1 output. If the dirty area is in two columns after step 7, then it is in the bottom half of one column and the top half of the next. Step 8 does not change this property, step 9 ensures that the dirty area resides in one column, step 10 cleans the dirty area, and step 11 moves all values back to where they belong. ■

We shall assume a 0-1 input and show that after step 7, the dirty area is at most half a column in size. The following lemma bounds the number of dirty rows in each k-slab after step 3 of slabpose columnsort.

Lemma 2 *Assuming a 0-1 input, after step 3 of slabpose columnsort, each k-slab consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most k dirty rows between them.*

Proof: As Figure 10(a) shows, after step 1, reading from top to bottom, each column consists of 0s followed by 1s. As we read a given column from top to bottom after step 1, there is at most one transition, and it is a $0 \rightarrow 1$ transition.

Step 2 turns each column into exactly r/k rows within a k-slab, as shown in Figure 10(b). Because k divides r, the quotient r/k must be an integer. If we read any k-slab in row-major order, each $1 \rightarrow 0$ transition occurs at the end of one row within the k-slab and the beginning of another. Therefore, read in row-major order, each set of consecutive r/k rows within a k-slab has at most the $0 \rightarrow 1$ transition from the corresponding column after step 1. Within each set of consecutive r/k rows of a k-slab, the only row that may be dirty is the row containing the $0 \rightarrow 1$ transition. Since there are k such sets of rows within a k-slab, each k-slab now has at most k dirty rows altogether.

Figure 10(c) shows the effect of step 3: moving the clean rows of 0s to the top rows of the k-slab and the clean rows of 1s to the bottom rows of the k-slab. The dirty rows, of which there are at most k, end up between the clean 0s and 1s. ■

Figure 11(a) shows the matrix after step 3. The dirty rows in one k-slab bear no relation to the dirty rows in any other k-slab, and so overall it is possible that up to s rows in the matrix are still dirty. As the following lemma shows, steps 4 and 5 reduce the number of dirty rows in the entire matrix.

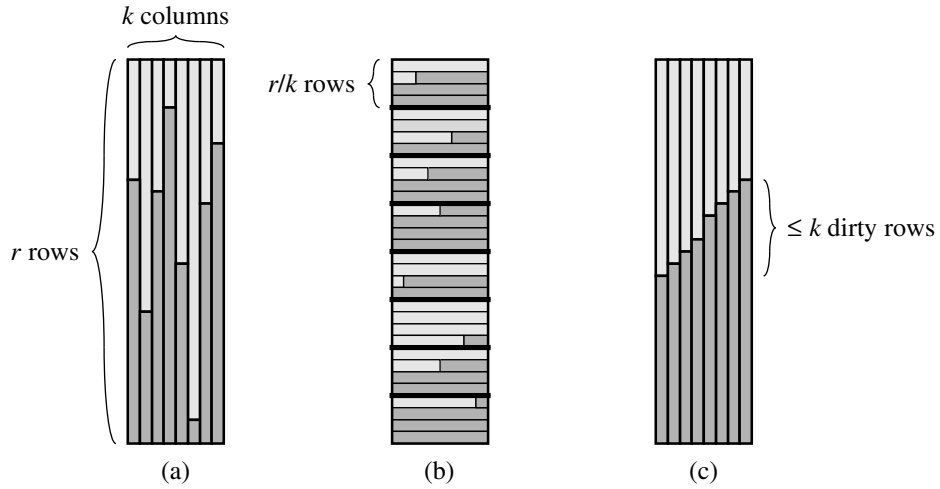

Figure 10: The effect of steps 1–3 of slabpose columnsort on a single k-slab, assuming a 0-1 input. 0s are lightly shaded and 1s are more darkly shaded. **(a)** After step 1, each column consists of 0s followed by 1s. **(b)** After step 2, each set of consecutive r/k rows has at most one $0 \rightarrow 1$ transition. There are at most k $0 \rightarrow 1$ transitions altogether. Heavy lines separate the rows formed from each column. **(c)** After step 3, the clean rows of 0s are at the top, the clean rows of 1s are at the bottom, and there are at most k dirty rows between them.

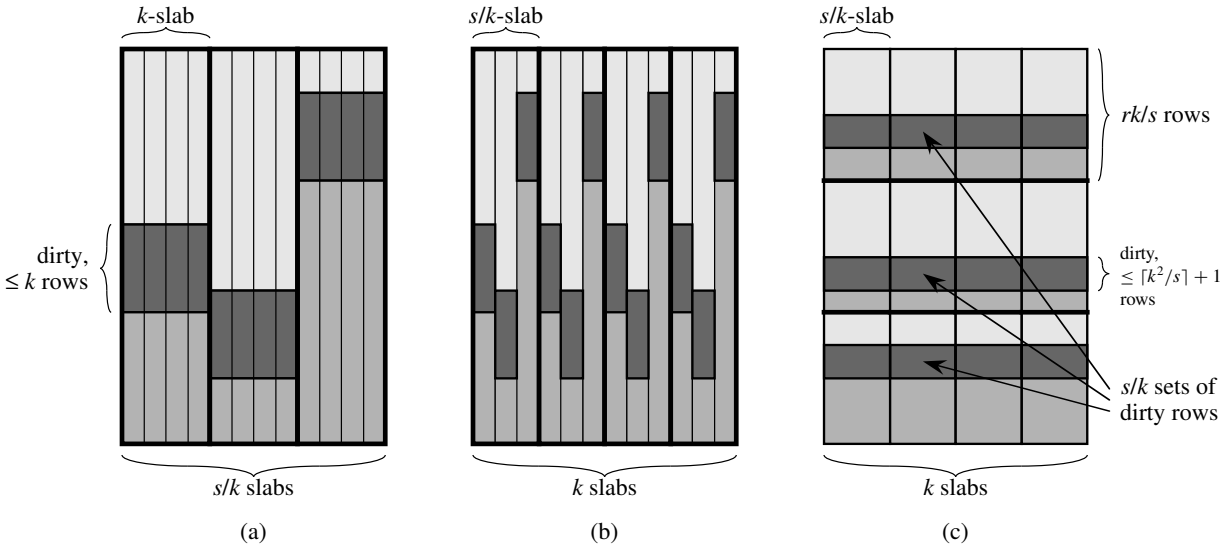

Figure 11: The matrix after steps 3, 4, and 5 of slabpose columnsort. **(a)** After step 3, there are s/k k-slabs, and each k-slab has a dirty area at most k rows high. The dirty areas do not necessarily align among k-slabs. **(b)** After step 4, there are k (s/k)-slabs. If we look at the jth column of each slab, the dirty area is confined to the same set of rows. **(c)** After step 5, within each set of rk/s consecutive rows, the dirty rows are confined to the same set of $\lceil k^2/s \rceil + 1$ rows.

Lemma 3 *Assuming a 0-1 input, after step 5 of slabpose columnsort, at most* $(s/k)(\lceil k^2/s \rceil + 1)$ *rows of the matrix are dirty.*

Proof: As Figure 11(b) shows, the k-shuffle of step 4 has the effect of creating k (s/k)-slabs. For $j = 0, 1, \ldots, k-1$, the dirty part of the jth column is confined to the same set of k rows in each of the (s/k)-slabs. Since s/k is a divisor of r, the (s/k)-slabpose operation of step 5 permutes each column within an (s/k)-slab into a set of $r/(s/k) = rk/s$ consecutive rows within the same (s/k)-slab. Figure 11(c) illustrates the result. Because the jth column of each (s/k)-slab forms the jth set of rk/s consecutive rows, and the dirty part of the jth column is confined to the same set of rows in each of the (s/k)-slabs, we see that within each set of rk/s consecutive rows, the dirty rows are confined to the same set of rows.

Next we determine how many rows within each set of rk/s consecutive rows are dirty. Because the dirty rows align among (s/k)-slabs, we need examine just a single (s/k)-slab. By Lemma 2, prior to step 4, any given column of a k-slab has a dirty area at most k rows high. Because step 4 only permutes columns, we see that prior to step 5, any given column of an s/k-slab has a dirty area at most k rows high. When these k values are moved into rows of width s/k, they fall into at most $\lceil k/(s/k) \rceil + 1 = \lceil k^2/s \rceil + 1$ rows.

Since there are s/k sets of rk/s consecutive rows, the total number of dirty rows after step 5 is at most $(s/k)(\lceil k^2/s \rceil + 1)$. ∎

Theorem 4 *As long as k is chosen as a divisor of s, s is a divisor of r, and $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$, slabpose columnsort sorts correctly.*

Proof: If we assume a 0-1 input, then by Lemma 3, there are at most $(s/k)(\lceil k^2/s \rceil + 1)$ dirty rows after step 5. After step 6 (which sorts each column), there are clean rows of 0s at the top of the matrix, clean rows of 1s at the bottom, and at most $(s/k)(\lceil k^2/s \rceil + 1)$ dirty rows between them. Since the dirty area is at most s columns wide, it is confined to an area of the matrix of size $(s^2/k)(\lceil k^2/s \rceil + 1)$. After step 7, which is a full reshape-and-transpose, when we read the matrix in column-major order, the dirty area is confined to at most $(s^2/k)(\lceil k^2/s \rceil + 1)$ consecutive entries. By Lemma 1, as long as this dirty area is at most half a column in size, the final four steps produce a sorted 0-1 output. This bound on the dirty area's size—$(s^2/k)(\lceil k^2/s \rceil + 1) \leq r/2$—is equivalent to the condition $r \geq (2s^2/k)(\lceil k^2/s \rceil + 1)$ in the theorem statement. Noting that slabpose columnsort is oblivious and applying the 0-1 Principle completes the proof. ∎