Columnsort Lives!
An Efficient Out-of-Core Sorting Program

Geeta Chaudhry *
Dartmouth College
Department of Computer Science
geetac@cs.dartmouth.edu

Thomas H. Cormen †
Dartmouth College
Department of Computer Science
thc@cs.dartmouth.edu

Leonard F. Wisniewski
Sun Microsystems
lenbo@east.sun.com

Abstract

We present the design and implementation of a parallel out-of-core sorting algorithm, which is based on Leighton's columnsort algorithm. We show how to relax some of the steps of the original columnsort algorithm to permit a faster out-of-core implementation. Our algorithm requires only 4 passes over the data, and a 3-pass implementation is possible. Although there is a limit on the number of records that can be sorted—as a function of the memory used per processor—this upper limit need not be a severe restriction, and it increases superlinearly with the per-processor memory. To the best of our knowledge, our implementation is the first out-of-core multiprocessor sorting algorithm whose output is in the order assumed by the Parallel Disk Model. We define several measures of sorting efficiency and demonstrate that our implementation's sorting efficiency is competitive with that of NOW-Sort, a sorting algorithm developed to sort large amounts of data quickly on a cluster of workstations.

1. Introduction

Sorting is universally acknowledged to be an important and fundamental problem in computing. Sorting very large data sets is a key subroutine in many applications. For some applications, the amount of data exceeds the capacity of main memory (we call these "out-of-core" problems), and the data then typically reside on one or more disks. This paper explores the implementation and performance of an out-of-core algorithm for sorting data on multiple processors and with multiple disks.

The specific algorithm is based on Leighton's columnsort algorithm [6]. The implementation for an out-of-core, parallel-I/O setting is novel.

Our performance goal is to sort large volumes of data while consuming as few resources as possible. We define the *sorting efficiency* to be the ratio of the volume of data sorted, in gigabytes, to resources used during the sorting. We use four different resource measures: processor time, disk time, processor time plus disk time, and used-memory time. Thus, if V is the volume of data, there are P processors and D disks, G gigabytes of memory are actually used across all processors, and sorting takes T minutes, then we will be measuring sorting efficiencies given by the formulas V/PT, V/DT, $V/((P + D)T)$, and V/GT.

Arpaci-Dusseau et al. [2] used an alternative measure of large-scale sorting: the volume of data that can be sorted in one minute. This measure does not account for resource usage. Nevertheless, their achievement of sorting 6 gigabytes in just under one minute in 1997 is impressive. Their NOW-Sort algorithm did so on a network of 64 workstations with 128 disks and a Myrinet network. On a network of just 8 workstations with 32 disks, NOW-Sort sorted 1.4 gigabytes in one minute. Our out-of-core columnsort measurements are on 4- and 8-processor systems. Table 1 shows that in many instances, the sorting efficiencies of our algorithm are close to those of NOW-Sort, and in some cases, they are even higher than NOW-Sort's.

At first glance, it may appear inherently unfair to compare sorting efficiencies of two algorithms that were implemented and measured four years apart; one would expect the more recent implementation to be faster due to hardware improvements. Further examination reveals that each implementation has certain characteristics that give it a head start over the other. Comparing the sorting efficiencies of the two algorithms on various hardware configurations yields some insight into how these differences in characteristics can translate into differences in performance. The characteristics fall into three classes: restrictions due to the underlying algo-

*Supported in part by a donation from Sun Microsystems.

†Contact author. Postal address: Dartmouth College Department of Computer Science, 6211 Sudikoff Laboratory, Hanover, NH 03755, USA. Supported in part by a donation from Sun Microsystems. This reseach was supported in part by NSF Grant EIA-98-02068.

Algorithm	Columnsort	Columnsort	Columnsort	NOW-Sort	NOW-Sort	NOW-Sort	NOW-Sort
System	Sun™	Sun	Origin 2000	Sun	Sun	Sun	Sun
P	4	8	4	8	16	32	64
D	4	4	4	32	32	64	128
G (GB)	0.15625	0.3125	0.15625	0.72656	0.65625	1.3125	2.625
V (GB)	1	2	1	1.4	1.6	3.0	6.0
T (minutes)	3.875	6.1	5.2	1	1	1	1
V/PT	0.06451	0.04098	0.04808	0.17500	0.1	0.09375	0.09375
V/DT	0.06451	0.08197	0.04808	0.04375	0.05	0.04688	0.04688
$V/((P+D)T)$	0.03226	0.02732	0.02404	0.035	0.03333	0.03125	0.03125
V/GT	1.65154	1.04918	1.23077	1.92688	2.43810	2.28571	2.28571
				Ratio of Columnsort on Sun with $P = 4$ to NOW-Sort			
V/PT				36.86%	64.51%	68.81%	68.81%
V/DT				147.46%	129.03%	137.63%	137.63%
$V/((P+D)T)$				92.16%	96.77%	103.22%	103.22%
V/GT				85.71%	67.74%	72.25%	72.25%

Table 1: Parameters, results, and sorting efficiencies for our out-of-core columnsort algorithm on three systems and for NOW-Sort on four systems. The top section shows number of processors (P), number of disks (D), gigabytes of memory used (G), gigabytes sorted (V), and time to sort (T) for each algorithm and system. The middle section shows various sorting efficiencies. The bottom section shows ratios of the sorting efficiency of columnsort on a 4-processor Sun cluster (the most sorting-efficient of the columnsort results) to sorting efficiencies of the four NOW-Sort results.

	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
$\mathcal{D}_0$	0	16	32	48	64	80	96	112
	1	17	33	49	65	81	97	113
$\mathcal{D}_1$	2	18	34	50	66	82	98	114
	3	19	35	51	67	83	99	115
$\mathcal{D}_2$	4	20	36	52	68	84	100	116
	5	21	37	53	69	85	101	117
$\mathcal{D}_3$	6	22	38	54	70	86	102	118
	7	23	39	55	71	87	103	119
$\mathcal{D}_0$	8	24	40	56	72	88	104	120
	9	25	41	57	73	89	105	121
$\mathcal{D}_1$	10	26	42	58	74	90	106	122
	11	27	43	59	75	91	107	123
$\mathcal{D}_2$	12	28	44	60	76	92	108	124
	13	29	45	61	77	93	109	125
$\mathcal{D}_3$	14	30	46	62	78	94	110	126
	15	31	47	63	79	95	111	127

Figure 1: The striped layout produced by out-of-core columnsort. The data are sorted in column-major order. Each box represents one striping unit, and numbers represent record indices. Each stripe runs vertically within a column. This example has 128 records, 4 disks, striping unit 2, and the columnsort matrix has $r = 16$ rows, and $s = 8$ columns. Each column C_j consists of 2 consecutive stripes.

rithm, hardware and software differences, and experimental differences. The differences due to the underlying algorithm are the following:

- Our out-of-core columnsort requires all parameters to be integer powers of 2; NOW-Sort does not.

- NOW-Sort assumes that the sort keys are uniformly distributed; our algorithm makes no assumptions about the keys. In fact, our algorithm's I/O and communication patterns are oblivious to the keys. That is, the exact sequence of I/O and communication operations depend only on the problem size and the system parameters, not on the key values.

- NOW-Sort does not place the sorted data into a striped order across the disks, nor does it assure the best possible load balance.[1] Our algorithm produces sorted records in the standard striped ordering used by the Parallel Disk Model (PDM), as Figure 1 shows. 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.

The differences in hardware and software are the following:

- The processors on which we have measured our algorithm are faster (296 MHz) than the 1997 processors used by NOW-Sort (170 MHz), and the individual disks we used are faster than those used by NOW-Sort.[2]

  The NOW-Sort cluster, however, has a 160 megabyte/second Myrinet network, compared to the 155 megabit/second ATM OC-3 network used in our measurements.

- The NOW-Sort implementation measures the relative speeds of all disks in the system and uses this information to optimize I/O performance by altering the striping unit on each disk. Moreover, NOW-Sort performs I/O via `mmap` and `madvise` calls in order to prevent double buffering by the operating system. Furthermore,

[1] Arpaci-Dusseau et al. write, "At the end, the data is sorted across the disks of the processors, with the lowest-valued keys on processor 0 and the highest-valued keys on processor $P-1$. Note that the number of records per node will only be approximately equal, and currently depends on the distribution of key values."

[2] We are unable to obtain exact specifications for the NOW-Sort disks.

NOW-Sort performs interprocessor communication by active messages.

Our algorithm, on the other hand, uses only MPI and MPI-2 calls [5, 11] for communication and I/O. All striping units are the same on all disks.

There is one experimental difference. Due to the power-of-2 requirement, the measurements for our algorithm are on 64-byte records. The NOW-Sort measurements are on the Datamation-standard 100-byte records. Therefore, for a given volume of data, our measurements are for over 50% more records.

The following list summarizes the contributions of this paper:

1. We present an out-of-core sorting algorithm, based on a 16-year-old algorithm, columnsort, which was designed for a rectangular mesh of processors. Each processor in the original columnsort corresponds to one record in our algorithm, and each column in the original columnsort corresponds to one *memoryload*—the amount of data that can be held in one processor's memory—in our algorithm.
2. To the best of our knowledge, our implementation is the first multiprocessor sorting algorithm whose output is in PDM order. We know of three previous uniprocessor implementations for the PDM, one for out-of-core radix sort [4], one by Pearson [8] for Rajasekaran's LMM-sort method [9] (of which columnsort is a special case), and one by Barve and Vitter [3] for randomized merge sort. Adapting these algorithms for multiprocessors is not straightforward.
3. We have relaxed some of the steps of the original columnsort algorithm to permit a faster out-of-core implementation.
4. Our out-of-core columnsort requires only 4 passes over the data, and we shall also discuss a future implementation that works in only 3 passes. Curiously, our upper bound is better than the $\omega(1)$-pass lower bound for the multiheaded, single-disk model proven by Aggarwal and Vitter [1], which is at least as strong a model as the PDM. Our upper bound is achieved at a cost of an upper limit on the number of records that can be sorted, expressed as a function of the memoryload size. In practice, however, this upper limit need not be a severe restriction. Although our algorithm is not designed to be a PDM algorithm, it can in fact be viewed as a 4-pass PDM algorithm. Thus, our algorithm beats the lower bound because of its record limit rather than because it was designed outside the PDM.
5. Our implementation's sorting efficiency is competitive with that of NOW-Sort.

The remainder of this paper is organized as follows. Section 2 reviews Leighton's original columnsort algorithm, which we adapt in Section 3 for an out-of-core setting. Section 4 details the performance of our implementation, as summarized in Table 1. Finally, Section 5 offers some final thoughts on our columnsort implementation relative to NOW-Sort, and it outlines future work.

2. The basic columnsort algorithm

In this section, we review Leighton's columnsort algorithm from [6]. Along the way, we make some observations that will improve the out-of-core implementation.

Columnsort sorts N numbers, which are treated as an $r \times s$ matrix, where $N = rs$, s is a divisor of r, and $r \geq 2(s-1)^2$. When columnsort completes, the matrix is sorted in column-major order. That is, each column is sorted, and the keys in each column are no larger than the keys in columns to the right.

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.

Step 2: Transpose and reshape
As Figure 2(a) shows, we first transpose the $r \times s$ matrix into an $s \times r$ matrix. Then we "reshape" it back into an $r \times s$ matrix by taking each row of r entries and rewriting it as an $r/s \times s$ submatrix. In Figure 2(a), for example, the column with $r = 6$ entries $a\ b\ c\ d\ e\ f$ is transposed into a 6-entry row with entries $a\ b\ c\ d\ e\ f$ and then reshaped into the 2×3 submatrix $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$.

Step 4: Reshape and transpose
As Figure 2(a) shows, we first reshape each set of r/s rows into a single r-element row and then transpose the matrix. In other words, step 4 is the inverse of the permutation performed in step 2.

Step 6: Shift down by $r/2$
As Figure 2(b) shows, we shift each column down by $r/2$ positions, wrapping around into the next column as necessary. This shift operation vacates the first $r/2$ entries of the leftmost column, which are filled with keys of $-\infty$, and it creates a new column on the right, the bottommost $r/2$ entries of which are filled with keys of ∞. Looked at another way, we shift the top half of each column into the bottom half of that column, and we shift the bottom half of each column into the top half of the next column.

Step 8: Shift up by $r/2$
As Figure 2(b) shows, we shift each column up by $r/2$ positions, wrapping around into the previous column as necessary. In other words, we perform the inverse permutation of step 6.

We omit the proof of correctness and refer the reader to [6].

Simplifying observations

Three observations will improve our out-of-core implementation later on:

$$\begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix} \quad \begin{array}{c} \xrightarrow{\text{Step 2}} \\ \xleftarrow{\text{Step 4}} \end{array} \quad \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

(a)

$$\begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix} \quad \begin{array}{c} \xrightarrow{\text{Step 6}} \\ \xleftarrow{\text{Step 8}} \end{array} \quad \begin{bmatrix} -\infty & d & j & p \\ -\infty & e & k & q \\ -\infty & f & l & r \\ a & g & m & \infty \\ b & h & n & \infty \\ c & i & o & \infty \end{bmatrix}$$

(b)

Figure 2: The operations of even-numbered steps of columnsort. This figure is taken from [6]. For simplicity, this small 6×3 matrix is chosen to illustrate the steps, even though its dimensions fail to obey the columnsort restrictions on r and s. **(a)** The operations of steps 2 and 4. **(b)** The operations of steps 6 and 8.

Placement: Immediately following each of steps 2, 4, and 6, we sort each column. Therefore, we only need to permute each element into the correct column in steps 2, 4, and 6. Its placement within the column (i.e., its exact row) is immaterial.

Run: The sorts in steps 3, 5, and 7 can be implemented as multiway merge operations. That is, the entries to be sorted are arranged as several sorted runs, which need only be merged. In Leighton's basic algorithm, sorting each column in step 3 is a merge of s runs of length r/s each, and sorting each column in step 7 is a merge of 2 runs of length $r/2$ each. Our implementation takes advantage in steps 3 and 7 of the formation of these runs in the previous steps, and it also uses the placement observation in step 4 to form s sorted runs of length r/s each so that step 5 is also a multiway merge.

Pairing: We can combine steps 6–8 by pairing adjacent columns. We sort the bottom $r/2$ entries of each column along with the top $r/2$ entries of the next column, placing the sorted r entries into the same positions. The top $r/2$ entries of the leftmost column were already sorted by step 5 and can therefore be left alone, and similarly for the bottom $r/2$ entries of the rightmost column.

These observations are known within the mesh-sorting community. Rajasekaran [9] uses the run and pairing observations at various points in his LMM-sort algorithm.

3. Out-of-core columnsort

In this section, we describe how we adapt Leighton's original columnsort algorithm for an out-of-core setting on a multiprocessor with a parallel disk system.[3] After describing our machine model, we present the common structure of each pass of the out-of-core algorithm, followed by a discussion of how each pass is implemented within the common framework.

Machine model

We assume that our machine has P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ and D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. The processors may be on separate nodes of a cluster, they may be processors within a shared-memory node, or they may be a combination of the above. For example, one configuration that we used had a cluster of 4 nodes, with each node having 2 processors, for a total of 8 processors; in this configuration, each node had 1 disk, which was shared by the 2 processors on that node. More generally, we assume that each processor can access at least one disk directly.

When $D = P$, each processor accesses exactly one disk 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. In fact, in our implementation, we treat this distinct portion as a separate "virtual disk," allowing us to assume that $D \geq P$. When $D > P$, each processor has exclusive access to D/P disks, and it stripes its data across these disks in *local stripes* of BD/P records each. We shall say that a processor *owns* the D/P disks that it accesses.

The *striping unit*, also known in the PDM as the *block size*, is the maximum number of consecutive records that are stored on a single disk. A *record* is a contiguous sequence of bytes containing a numeric *key* embedded at a fixed offset within the record. For good performance, the striping unit should be at least one physical block on the disk. In practice, the striping unit is at least the system's physical page size. We parameterize the striping unit by the PDM parameter B. Any disk access transfers an entire block of records between the disk and the memory of some processor.

The memory of the entire system holds M records. Memory is partitioned among the P processors so that each processor can hold M/P records. In practice, a processor that holds M/P records has a physical memory that is larger than M/P records' worth. That is because physical memory holds more than just records—software and the runtime stack, for example. Moreover, our algorithm requires multiple data buffers for merging (which is not in-place), communication, and overlapping I/O with other operations.

For convenience, our current implementation assumes that all parameters are powers of 2.

[3]The Greed Sort algorithm by Nodine and Vitter [7] for the PDM uses columnsort as a finishing-up step. Nodine and Vitter show that columnsort makes a constant number of passes for sufficiently small problem sizes. Their paper also notes a property that is similar to our pairing observation.

Given the machine parameters, we set the columnsort matrix dimensions r and s as follows. We require a column's worth of data to fit within one processor's memory. Thus, we set $r = M/P$. Since $N = rs$, we set $s = N/r = NP/M$.

These choices of r and s induce an upper limit on the number of records N in the file to be sorted. Recall the columnsort requirement that $r \geq 2(s-1)^2$, which we simplify by requiring $r \geq 2s^2$. Using our parameter settings, we have the restriction that $M/P \geq 2(NP/M)^2$, which is equivalent to

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

Although the restriction (1) may appear to be overly constraining on the largest problem size that will be correctly sorted, in practice we can still sort some fairly large files. For example, suppose we are sorting 32-byte records on a machine with 32 megabytes of memory per processor available for holding data (after reducing the available memory for software, stack, and buffers). Then $M/P = 2^{20}$, since the memory size M is in terms of 32-byte records. By the restriction (1), the largest file we can sort has 2^{29} records, or 2^{34} bytes. In other words, even with a moderate memory size, we can sort up to 16 gigabytes of data. Moreover, by restriction (1), the maximum problem size increases with $(M/P)^{3/2}$, i.e., it increases superlinearly with the amount of memory per processor.

Common structure among passes

Our columnsort implementation proceeds in four passes over the data. Each pass reads records from one part of each disk and writes records to a different part of each disk. We alternate the portions read and written from pass to pass so that, apart from the input and output portions, we need just one other portion, whose size is that of the data.

Each pass performs two consecutive steps of columnsort. That is, pass 1 performs columnsort 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. (We shall discuss in Section 5 an optimization that uses the pairing observation to combine steps 5–8 into a single pass.)

Figure 3 shows how the data are organized on the disks at the start of each pass. 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.

Each pass is decomposed into s/P *rounds*. Each round processes the next set of P consecutive columns, one column per processor, in four phases:

1. Each processor reads a column of r records from the disks that it owns.

2. Each processor locally sorts, in memory, the r records it has just read.

3. 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.

4. Having received records from other processors, each processor writes a set of records onto the disks that it owns. These records are not necessarily all written consecutively onto the disks, though they are written as a small number of sorted runs.

C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
0	16	32	48	64	80	96	112
1	17	33	49	65	81	97	113
2	18	34	50	66	82	98	114
3	19	35	51	67	83	99	115
4	20	36	52	68	84	100	116
5	21	37	53	69	85	101	117
6	22	38	54	70	86	102	118
7	23	39	55	71	87	103	119
8	24	40	56	72	88	104	120
9	25	41	57	73	89	105	121
10	26	42	58	74	90	106	122
11	27	43	59	75	91	107	123
12	28	44	60	76	92	108	124
13	29	45	61	77	93	109	125
14	30	46	62	78	94	110	126
15	31	47	63	79	95	111	127
$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_0$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$

Figure 3: The layout used internally within out-of-core columnsort. Each column is stored contiguously on the set of disks owned by a single processor. Column C_i is stored on the disks owned by processor $\mathcal{P}_{i \bmod P}$. In this example, $P = D = 4$ and $s = 8$.

We shall see shortly how each phase differs in each pass.

An early implementation of ours performed rounds strictly in order. That is, processor j performed all four phases on column j, then it performed all four phases on column $j+P$, then all four phases on column $j+2P$, and so on. This implementation did not overlap local sorting, communication, or I/O. We improved the implementation by adding asynchronous I/O and communication to allow overlap. At any particular time, processor j might be communicating records belonging to column $j+kP$, locally sorting records in column $j+(k+1)P$, reading column $j+(k+2)P$, and writing column $j+(k-1)P$. However, our current implementation does not overlap reading with either local sorting or writing. A future implementation will use threads to allow the system to decide how to overlap phases for full flexibility.

How each pass works

The exact operation of three of the four phases within a round differs from pass to pass. The phase that remains the same is reading. That is, each processor in each round, regardless of the pass, reads a column of r records by simply reading r/B consecutive blocks from the disks that it owns. This style of reading yields the best possible I/O performance.

Now we examine how the four passes perform each of the other three phases.

Pass 1.
This pass reads in each column, locally sorts it—assuming nothing about the sortedness of the data within the column—and then writes it out according to the transpose-and-reshape permutation.

Local sorting: Because we make no assumptions about the sortedness of each column or about the key distribution, we locally sort using the system `qsort` call. (On average, `qsort` was the fastest of the various sorting methods that we tried.) Assuming that `qsort` performs $\Theta(r \lg r)$ comparisons on average to sort each column, pass 1's in-core local sorting complexity is $\Theta(sr \lg r) = \Theta(N \lg r)$ altogether, or $\Theta(N/P \lg r)$ per processor.

Communication: After locally sorting, each processor has a sorted column of r records. As in step 2 of columnsort, r/s records out of the r are destined for each of the s columns. Each processor locally permutes its r records to form s sorted runs of r/s records each. Every processor owns s/P columns, and so every processor sends s/P sorted runs to each of the other processors. Each processor makes $P-1$ calls to the MPI functions `MPI_Isend` and `MPI_Irecv` to perform this communication asynchronously. (Processors don't need to send to themselves.) Each processor, therefore, receives $P-1$ messages, each message consisting of s/P sorted runs. As each processor receives these runs, it assembles them into a buffer such that the runs that are destined for the same column within this pass are contiguous.

Writing: Every processor now has P sorted runs for each of the s/P columns that it owns. Each processor performs s/P writes of the P runs, i.e., Pr/s records. In the case in which $D > P$ and $Pr/s < BD/P$—so that the amount of data being written to each column is less than the size of a local stripe—the writes are not distributed evenly across the D/P disks owned by the processor. For example, Figure 4 shows a situation with $s = 8$, $P = 4$, $D = 16$ (so that $D/P = 4$), and $r = 4B$. Here, $D > P$ and $Pr/s = 2B < 4B = BD/P$. The figure shows the write patterns in the first round for processor $\mathcal{P}_0$. This processor owns columns 0 and 4. It needs to write $2B$ records, or 2 blocks, to each of these columns. As Figure 4(a) shows, if both columns start on disk $\mathcal{D}_0$, we write to only half the disks. Figure 4(b) demonstrates how we use the placement observation to write to different locations within the columns in order to distribute the load evenly among the disks.

Pass 2.
This pass reads in each column, locally sorts it—knowing that every column consists of s sorted runs of r/s records each—and then writes it out according to the reshape-and-transpose permutation.

Local sorting: We use merge sort, but because we have s sorted runs to start, we need to recurse only $\lg s$ levels rather than $\lg r$. It takes $\Theta(r \lg s)$ time to locally sort each column, so that the total local sorting time is $\Theta(rs \lg s) = \Theta(N \lg s)$, or $\Theta(N/P \lg s)$ per processor.

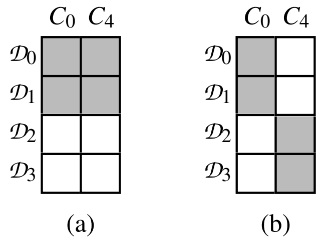

Figure 4: Write patterns in the first round of pass 1 for processor $\mathcal{P}_0$ when the amount of data being written to each column is less than the size of a local stripe. Shaded blocks are the ones written to if we **(a)** do not use and **(b)** do use the placement observation to distribute the load evenly.

Communication: Communication in pass 2 is exactly the same as in pass 1, except that immediately after local sorting, the records destined for each column are already formed into a sorted run of r/s records. Therefore, the local permutation prior to sending is skipped. As in pass 1, however, records are locally permuted after they have been received.

Writing: Pass 2 writes in exactly the same way as pass 1. It can do so even though the runs received within a given round are not destined for consecutive locations within a column according to the reshape-and-transpose permutation. These runs arrived in contiguous locations of a communication buffer, and we can write them together because, by the placement observation, it does not matter where in the column we write them. Moreover, writing these runs as we receive them makes the input to the next pass consist of sorted runs.

Pass 3.
Like pass 2, this pass reads in each column and locally sorts it, knowing that every column consists of s sorted runs of r/s records each. It then writes out the records according to the shift down by $r/2$ permutation.

Local sorting: We use merge sort, just as in pass 2.

Communication: The last $r/2$ records in each column are going to go to the next column. Therefore, in each round, processor $\mathcal{P}_j$ sends its last $r/2$ records to processor $\mathcal{P}_{(j+1) \bmod P}$. In each round, each processor makes just one call to `MPI_Isend` and `MPI_Irecv` to perform this communication asynchronously.

Writing: In each round, each processor writes a column of r records by simply writing r/B consecutive blocks to the disks that it owns. Just like how we read, this style of writing yields the best possible I/O performance.

Pass 4.
This last pass reads in each column and locally sorts it—knowing that every column consists of 2 sorted runs of $r/2$

records each—and then writes it out according to the shift up by $r/2$ permutation, but in PDM order.

Local sorting: We merge the 2 sorted runs, which takes only $\Theta(r)$ time per processor in each round. Over all rounds, local sorting takes $\Theta(N)$ time, or $\Theta(N/P)$ time per processor.

Communication: In each round, each processor has r records that need to go to r consecutive locations in PDM order. That is, they are striped across all D disks in the system. In the PDM order, each stripe has BD records, and so the r records form r/BD consecutive stripes. In other words, the r records will go to r/B blocks in all, or r/BD blocks per disk. Since every processor owns D/P disks, $(r/BD)(D/P) = r/BP$ blocks go to each processor. Every processor sends r/P records to each other processor by making asynchronous MPI calls similar to those of passes 1 and 2. Each processor, therefore, receives $P-1$ messages, each message consisting of r/BP blocks. As each processor receives these blocks, it assembles them into a buffer such that the blocks that are destined for the same disk are contiguous.

Writing: Each processor has D/P disks, and the records destined for each disk are already contiguous. A set of D/P asynchronous write calls finishes the job.

4. Performance results

In this section, we present experimental results for our algorithm running on Sun Microsystems and Silicon Graphics systems. For each system, we break down the time spent in I/O, local sorting, communication, and local permuting in each pass and in all four passes together. Seeing where the algorithm spends its time gives us a good indication of what parts of the implementation should be tuned in the future.

Our runs are on two clusters of Sun™ SMP nodes and on one Silicon Graphics Origin 2000. Although the columnsort code itself is exactly the same for each of these systems, it is linked with different MPI implementations.

On each system, we sorted 256 megabytes per processor. The times reported herein are the average of five runs on each system. Table 1 summarizes the results. The total times appearing in Table 1 are slightly lower than the total times given in this section due to the overhead introduced by calls to finer-grained timers.

The first system is a cluster of 4 Sun Enterprise™ 450 SMPs. We used only one processor on each node. Each processor is an UltraSPARC™-II, running at 296 MHz and with 128 MB of RAM, of which we used 40 MB for holding data. The nodes are connected by an ATM OC-3 network, with a peak speed of 155 megabits per second. This system has one disk per node. Two of these disks are Seagate ST39103LC spinning at 10,016 RPM and with an average latency of 2.99 msec, and the other two are IBM DNES309170 spinning at 7200 RPM and with an average latency of 4.17 msec. All disks are on Ultra2 SCSI buses. The MPI and MPI-2 implementations are part of Sun HPC ClusterTools™.

Figure 5 shows the time spent waiting for disk reads, disk writes, communication, local sorting, and local permuting in all four passes and in total for a sort of 1 gigabyte. Because operations are heavily overlapped, measurements are for the time spent waiting for operations to complete. For example, because local sorting and communication overlap extensively, the local sorting time is also communication time. The local sorting time shown is the actual time spent sorting locally, and the actual communication time includes both the local sorting time and the communication time shown as well as some I/O time.

Examination of Figure 5 reveals that columnsort is clearly not I/O bound. Although local sorting takes a significant percentage of the total time, because local sorting is overlapped with communcation, we see that the algorithm is communication-bound. We believe that a faster network, such as the Myrinet used in NOW-Sort, would yield significantly faster times overall.

The second system is also a cluster of 4 Sun Enterprise 450 SMPs. Here, we used two processors on each node. All other specifications are the same as the 4-processor Sun cluster, including the 4 disks. The two processors on each node share a single disk.

Figure 6 shows the wait times for the 8-processor Sun cluster for a sort of 2 gigabytes. The total time is approximately 57% higher than the 1-gigabyte sort on 4 processors. One should bear in mind that the network and disk bandwidths are the same on this system as on the 4-processor system, and so it is not surprising that the time increased. Because only processing power is greater than on the 4-processor system, the percentage of time spent in local sorting is less. Percentage of time spent waiting for I/O is roughly the same for the two systems, and the communication wait time is significantly higher on the 8-processor system.

The final system is a Silicon Graphics Origin 2000. This system has 8 processors, of which we used 4. Each processor is an IP27 running at 180 MHz. The entire system has 4 GB of RAM, of which we used 160 MB in all for holding data. The Origin 2000 has distributed shared memory. We used 4 disks on a common I/O bus. Each disk is a Seagate ST34501W spinning at 10,033 RPM and with an average latency of 2.99 msec. The MPI and MPI-2 implementations are MPICH from Argonne National Laboratories (http://www.mcs.anl.gov/mpi/mpich/). Although Silicon Graphics offers a native implementation of MPI for the Origin 2000, we could not use it because, to the best of our knowledge, it does not contain I/O functions.

From Figure 7, wait times for a 1-gigabyte sort, we see a rather different breakdown than we saw for the Sun systems. Communication is significantly faster because there is no external network, but I/O is much slower due to the shared I/O bus. Sorting is slower on the Origin 2000 as well, due to the slower processors.

In all three figures, we see the benefit of the run observation. Passes 2 and 3 use $\lg s$ levels of merging, and pass 4 uses just one level of merging. From Figures 5–7, we see that passes 2 and 3 spend much less time locally sorting than pass 1,

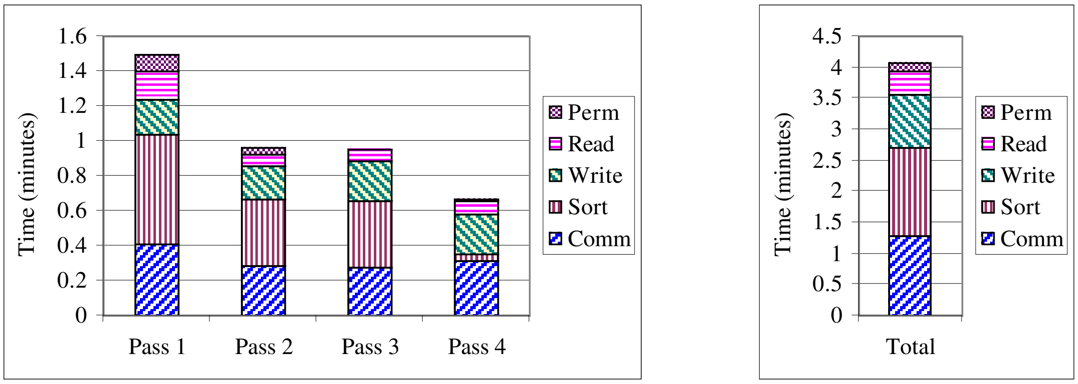

Figure 5: Wait times for disk reads, disk writes, communication, local sorting, and local permuting in all four passes and in total on the 4-processor Sun cluster for a 1-gigabyte sort.

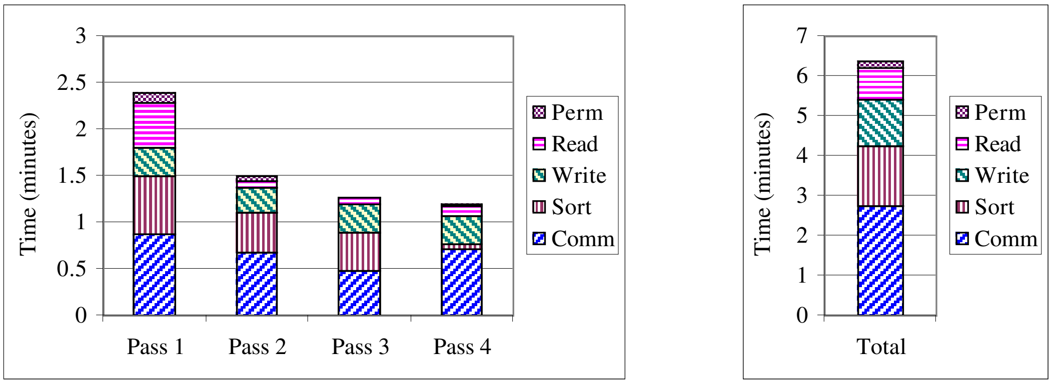

Figure 6: Wait times on the 8-processor Sun cluster for a 2-gigabyte sort.

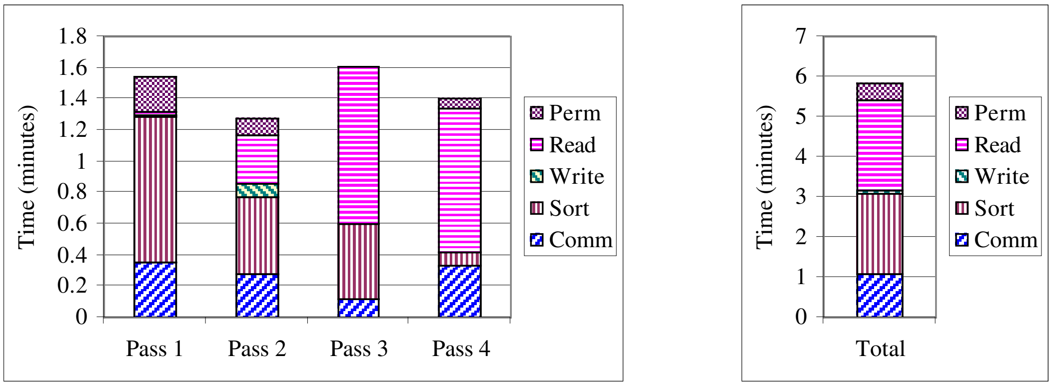

Figure 7: Wait times on the 4-processor Origin 2000 for a 1-gigabyte sort.

and pass 4 spends very little time locally sorting.

5. Conclusion

As Table 1 shows, by one measure, our out-of-core columnsort implementation is more sorting-efficient than NOW-Sort, but in other measures, NOW-Sort is more sorting-efficient. Our columnsort is more efficient in V/DT (gigabytes sorted per disk-minute), NOW-Sort is more efficient in V/PT (gigabytes sorted per processor-minute) and V/GT (gigabytes sorted per used-memory-minute). The measure $V/((P+D)T)$ combines elements of one measure in which columnsort is more sorting-efficient and one in which NOW-Sort is more sorting-efficient, and so the two algorithms turn out to be very close in this measure.

Each of the two implementations exploits certain advantages over the other. NOW-Sort has the following advantages:

System: A faster network, active messages, and I/O via `mmap` and `madvise`, whereas our columnsort implementation uses only MPI and MPI-2 calls for communcation and I/O.

Assumptions: Processor-major output ordering and uniformly distributed keys.

Experimental: Larger record size, hence fewer records per gigabyte.

Our columnsort implementation has the following advantages:

System: Faster processors and faster disks.

Assumptions: All parameters are powers of 2, and the problem size is restricted by the amount of memory per processor.

Experimental: None.

One might wonder why we go through the effort of implementing a columnsort-based approach when a distribution-sorting approach like NOW-Sort's makes fewer passes over the data. To perform out-of-core distribution sorting in the absence of uniformly distributed keys, there are typically at least three passes: one to distribute the data, one to sort on each processor (possibly out-of-core), and one for load balancing. In fact, an out-of-core distribution sort usually requires even more passes: sorting on each processor may require multiple passes over the data owned by that processor, and we need to read some subset of keys prior to distribution in order to select a set of "splitters." Although our columnsort-based algorithm makes four passes over the data, we shall describe below how to reduce the number of passes to only three. Consequently, in terms of number of passes over the data, a columnsort-based algorithm can be as good as a distribution sort.

Furthermore, when the output must be striped in PDM order, a distribution-sorting algorithm has problems. Producing the output naturally in PDM order is infeasible. That is because the natural way to distribute records is into buckets whose size is the striping unit B. Unfortunately, there are so many striping units of data altogether that the number of buckets becomes unmanageable. During distribution, each processor must maintain one buffer for each of its B-record buckets. If we could keep each such buffer in memory at the same time, then we would have N/B blocks in memory, and so the problem would fit in memory!

Instead of producing the output naturally in PDM order, a distribution sorting algorithm can produce its output in processor-major order and then permute the data into PDM order. This reordering is a permutation of blocks, which requires one additional pass.

Columnsort, unlike distribution sorting, is oblivious. That is, the exact I/O and communication patterns depend only on the system parameters and the problem size. In distribution sorting, both I/O and communication are data dependent.

Future work

At this time, we envision three directions in which to proceed.

First, as mentioned above, we can combine the last two passes of out-of-core columnsort into a single pass. The key idea is the pairing observation. The last pass would work as follows. Each processor reads a column, sorts it locally, sends the bottom half to the next processor (receiving half a column from the previous processor), sorts locally again, communicates in a similar way to our current pass 4 communication, and writes to disk. The first and last processors work a little differently. We omit the details here.

Second, we plan to revise our implementation to use threads in order to overlap local computation, communication, and I/O. Our current implementation uses asynchronous I/O and communication calls, and sorting and communication are tied together. In a threaded implementation, all calls can be synchronous, with asynchrony provided by the thread mechanism. It will be interesting to see how a fully threaded implementation performs relative to our current code.

Third, we would like to improve on the restriction (1). Currently, the maximum problem size is proportional to $(M/P)^{3/2}$. We conjecture that we can incorporate some ideas of the Revsort mesh-sorting algorithm [10]. Our hope is that the resulting algorithm would allow the maximum problem size to be proportional to $(M/P)^{5/3}$ yet be implementable.

Acknowledgments

We thank David Kotz, Wayne Cripps, and Arne Grimstrup for their cooperation in helping us set up experimental testbeds.

References

[1] 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.

[2] 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*, 1997.

[3] Rakesh D. Barve and Jeffrey Scott Vitter. A simple and efficient parallel disk mergesort. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 232–241, June 1999.

[4] 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.

[5] 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.

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

[7] 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.

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

[9] Sanguthevar Rajasekaran. A framework for simple sorting algorithms on parallel disk systems. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 88–97, 1998.

[10] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 255–263, May 1986.

[11] 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.