Getting More From Out-of-Core Columnsort

Geeta Chaudhry and Thomas H. Cormen
{geetac,thc}@cs.dartmouth.edu

Dartmouth College Department of Computer Science

Abstract. We describe two improvements to a previous implementation of out-of-core columnsort, in which data reside on multiple disks. The first improvement replaces asynchronous I/O and communication calls by synchronous calls within a threaded framework. Experimental runs show that this improvement reduces the running time to approximately half of the running time of the previous implementation. The second improvement uses algorithmic and engineering techniques to reduce the number of passes over the data from four to three. Experimental evidence shows that this improvement yields modest performance gains. We expect that the performance gain of this second improvement increases when the relative speed of processing and communication increases with respect to disk I/O speeds. Thus, as processing and communication become faster relative to I/O, this second improvement may yield better results than it currently does.

1 Introduction

In a previous paper [1], the authors reported on an out-of-core sorting program based on Leighton's columnsort algorithm [2]. By some resource measures—specifically, disk time and processor time plus disk time—our columnsort-based algorithm was more sorting-efficient than the renowned NOW-Sort program [3].[1] Unlike NOW-Sort, the implementation of our algorithm performed interprocessor communication and disk I/O using only standard, off-the-shelf software, such as MPI [4] and MPI-2 [5].

The present paper explores two improvements to the implementation of our out-of-core program:

1. Overlapping I/O, computation, and communication by means of a threaded implementation in which all calls to MPI and MPI-2 functions are synchronous. Asynchrony is provided by the standard pthreads package. The previous implementation achieved asynchrony by making calls to the asynchronous versions of MPI and MPI-2 functions.

Geeta Chaudhry was supported in part by a donation from Sun Microsystems. This research was supported in part by NSF Grant EIA-98-02068.

[1] For two other measures—specifically, processor time and used-memory time—NOW-Sort was more sorting-efficient than our columnsort-based algorithm. Also, for processor time plus disk time, our algorithm was more sorting-efficient than NOW-Sort in only some cases.

2. Reducing the number of passes over the data from four down to three. An algorithmic observation makes this reduction possible.

We shall refer to our prior implementation as the *non-threaded 4-pass* implementation, and to our new implementations as the *threaded 4-pass* and *threaded 3-pass* implementations. We shall see that the threaded 4-pass implementation can reduce the overall running time down to approximately half of the non-threaded 4-pass implementation's running time. Moreover, the threaded 4-pass implementation allows greater flexibility in memory usage by both the user and the program itself. The effect of reducing the number of passes is more modest; experiments show that the running time of the threaded 3-pass implementation is between 91.5% and 94.6% of that of the threaded 4-pass implementation.

The remainder of this paper is organized as follows. Section 2 summarizes columnsort and presents the out-of-core algorithm originally described in [1]. Section 3 outlines the differences between the non-threaded 4-pass implementation with asynchronous MPI and MPI-2 calls and the threaded 4-pass implementation. In Section 4, we describe our threaded 3-pass implementation. We give empirical results for all implementations. Finally, Section 5 offers some concluding remarks.

2 Background

In this section, we review our non-threaded 4-pass implementation of columnsort from the previous paper [1]. After presenting the basic columnsort algorithm, we describe its adaptation to an out-of-core setting. We conclude this section with a discussion of the performance results.

The Basic Columnsort Algorithm. Columnsort sorts N *records*. Each record contains a *key*. In columnsort, the records are arranged into 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 as follows:

Step 2: Transpose and reshape: 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. 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: We first reshape each set of r/s rows into a single r-element row and then transpose the matrix. This permutation is the inverse of that of step 2.

Step 6: Shift down by $r/2$: We shift each column down by $r/2$ positions, wrapping into the next column. That is, 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 $r/2$: We shift each column up by $r/2$ positions, wrapping around into the previous column. This permutation is the inverse of that of step 6.

Our Out-of-Core Columnsort. Our adaptation of columnsort to an out-of-core setting assumes that the machine has P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ and D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$. 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. We say that a processor *owns* the D/P disks that it accesses.

We use buffers that hold exactly r records. Each processor has several such buffers. For convenience, our current implementation assumes that all parameters (including r) are powers of 2.

There is an upper limit on the number of records N in the file to be sorted. Recalling that $N = rs$ and that each buffer of r records must fit in the memory of a single processor, this limit occurs because of the columnsort requirement that $r \geq 2(s-1)^2$. If we simplify this requirement to $r \geq 2s^2$, we have the restriction that $r \geq 2(N/r)^2$, which is equivalent to $N \leq r^{3/2}/\sqrt{2}$.

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.

Here, we outline the basic structure of each pass of our non-threaded 4-pass implementation along with the key implementation features; for details, see [1]. 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. Each pass is decomposed into s/P *rounds*. Each round processes the next set of P consecutive columns, one column per processor, in five phases:

Read phase: Each processor reads a column of r records from the disks that it owns. We use asynchronous MPI-2 calls to implement this phase.

Sort phase: Each processor locally sorts, in memory, the r records it has just read. Implementation of this phase differs from pass to pass. Pass 1 uses the system `qsort` call. In the other passes, the data starts with some sorted runs, depending on the write pattern of the previous pass. In passes 2 and 3, each processor has s sorted runs of size r/s each, and so we use recursive merging with $\lg s$ levels. In pass 4, each processor starts with two sorted runs of size $r/2$ each, and so local sorting requires just one level of merging to obtain a single sorted run.

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. We use asynchronous MPI calls to perform this communication. In passes 2, 3, and 4, this phase requires every processor to send some data to every other processor. In pass 3, the communication is simpler, since every processor communicates with only two other processors.

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 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. Again, we use asynchronous MPI-2 calls to perform the writes.

Note that the use of asynchronous MPI and MPI-2 calls allows us to overlap local sorting, communication, and I/O. 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 non-threaded 4-pass implementation does not overlap reading with either local sorting or writing.

Performance Results. Table 1 summarizes the performance results of several implementations of out-of-core columnsort.[2] Our performance goal is to sort large volumes of data while consuming as few resources as possible.

The results are for 64 byte records with an integer key at the beginning of each record. The input files are generated using the `drand` function to generate the value of each key.

The results in Table 1 are for two different clusters of SMPs. 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, each an 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 4™.

The second system is also a cluster of 4 Sun Enterprise 450 SMPs. Here, we used two processors on each node. Other than the disks, the components are the same as in the first system. This system has 8 disks, 4 of which are the same IBM disks as in the first system, and the other 4 of which are Seagate ST32171W also spinning at 7200 RPM and also with an average latency of 4.17 msec.

We conclude this section by discussing two additional features of our implementations:

[2] The results in the columns labeled "non-threaded 4-pass" are for more recent runs than in [1].

	non-threaded 4-pass	threaded 4-pass	threaded 3-pass	non-threaded 4-pass	threaded 4-pass	threaded 3-pass
P	4	4	4	8	8	8
D	4	4	4	8	8	8
U (MB)	160	192	256	320	384	512
V (GB)	1	1	1	2	2	2
T (minutes)	4.338	2.161	2.043	5.631	3.217	2.944

Table 1. Parameters and results for three implementations of our out-of-core columnsort algorithm on two clusters of Sun Enterprise 450 SMPs. The rows show number of processors (P), number of disks (D), megabytes of memory used across the entire system (U), gigabytes sorted (V), and time to sort (T) for each system. Times shown are averages of five runs.

- 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. Because the I/O and communication patterns are completely oblivious to the keys, each round has no effect at all on other rounds of the same pass. That is, the rounds of a pass are fully independent of each other and therefore can execute with full asynchrony. We shall take advantage of this property in our threaded implementations.
- Our algorithm produces sorted records in the standard striped ordering used by the Parallel Disk Model (PDM). This ordering affects only the write phase of the last pass, in which each column is striped across all D disks rather than residing solely on the disks of the processor that owns it. 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. To the best of our knowledge, our prior implementation is the first multiprocessor sorting algorithm whose output is in PDM order.

3 Threaded 4-Pass Implementation

In this section, we describe our threaded 4-pass implementation and compare its performance to that of the non-threaded 4-pass implementation.

Motivation. There are two reasons to revise our implementation to use threads. First, threads allow scheduling to be more dynamic. Second, our threaded implementation permits greater flexibility in memory usage.

Our columnsort algorithm uses four types of machine resources: disks, the communication network, processing power, and memory. As described in the previous section, each column goes through five phases, two using I/O (the read and write phases), one using the network (the communicate phase), and the other two using the processing power (the sort and permute phases). All phases make use of memory, of course.

Our non-threaded 4-pass implementation has a single thread of control. In order to permit overlapping of I/O, communication, and processing, it relies on asynchronous MPI and MPI-2 calls in order to schedule the usage of the four types of resources. Due to the single thread of control, this implementation has to decide the overlap mechanism statically at coding time and is not fully flexible. For example, the implementation is unable to adapt automatically to a faster network, faster CPU, or faster I/O. Our threaded 4-pass implementation uses the standard pthreads package to overlap I/O, communication, and processing. It uses only synchronous MPI and MPI-2 calls.

Because the non-threaded 4-pass implementation uses static scheduling, its memory usage is static and has to be known at coding time. Consequently, this implementation is unable to adapt to an increased amount of available memory. Our threaded 4-pass implementation, on the other hand, maintains a global pool of memory buffers, the number of which is set at the start of each run of the program. One can determine the optimum number of buffers for a given configuration by means of a small number of experimental runs.

Basic Structure. In order to overlap the usage of resources in a dynamic manner, we created four threads per processor. One thread is responsible for all the disk I/O functions, one does all the interprocessor communication, one does all the sorting, and the final thread does all the permuting. We shall refer to the four threads as the I/O, communicate, sort, and permute threads, respectively. The threads operate on buffers, each capable of holding exactly r records, and which are drawn from a global pool. The threads communicate with each other via a standard semaphore mechanism. The read and write functions appear in a common thread—the I/O thread—because they will serialize at the disk anyway.

Figure 1 illustrates how the threads work by displaying the history of a column within a given round of a given pass.

1. The I/O thread acquires a buffer b from the global pool and performs the read phase of column c by reading the column from the disk into this buffer. The I/O thread suspends while the read happens, and when the read completes, the I/O thread wakes up.
2. When the I/O thread wakes up after the read completes, it signals the sort thread, which picks up buffer b and performs the sort phase of column c on it.
3. The sort thread signals the communicate thread, which picks up buffer b and performs the communicate phase of column c on it. The communicate phase suspends during interprocessor communication, and after communication completes, it wakes up.
4. The communicate thread signals the permute thread, which picks up buffer b and performs the permute phase of column c on it.
5. Finally, after the permute phase completes, the permute thread signals the I/O thread, which picks up buffer b and writes it out to disk. The I/O thread

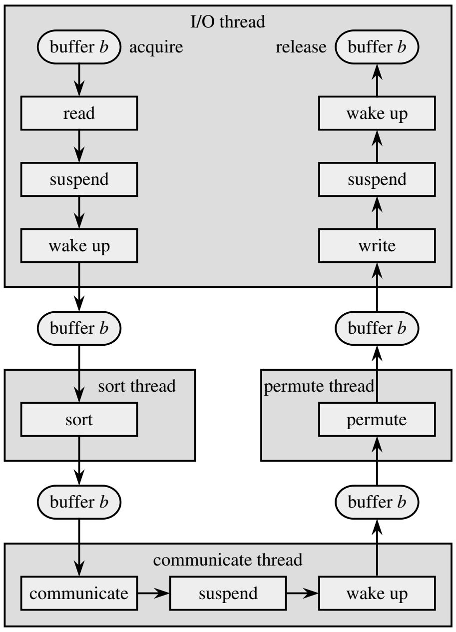

Fig. 1. The history of a buffer b as it progresses within a given round of a given pass. The I/O thread acquires the buffer from the global pool and then reads into it from disk. The I/O thread suspends during the read, and when it wakes up, it signals the sort thread. The sort thread sorts buffer b 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.

suspends during the write, and when the write completes, the I/O thread wakes up and releases buffer b back to the global pool.

The pthreads implementation may preempt any thread at any time. Thus, during the time that a given thread considers itself as active, it might not actually be running on the CPU.

The sort, permute, and communicate threads allocate additional buffers for their own use. Each of these threads allocates one buffer at the beginning of the program and uses it throughout the entire run. Thus, the total memory usage of the threaded 4-pass implementation is three buffers more than are created in the global pool.

Performance Results. From the columns labeled "threaded 4-pass" in Table 1, we see that the threaded 4-pass implementation takes only 49.8% of the time taken by the non-threaded 4-pass implementation on the cluster with 4 processors and 4 disks. On the cluster with 8 processors and 8 disks, the threaded

4-pass implementation takes 57.1% as much time as the non-threaded 4-pass implementation.

What accounts for this significant improvement in running time? Due to the highly asynchronous nature of each of the implementations, we were unable to obtain accurate breakdowns of where any of them were truly spending their time. We were able to obtain the amounts of time that each thread considered itself as active, but because threads may be preempted, these times may not be reflective of the times that the threads were actually running on the CPU. Similarly, the timing breakdown for the non-threaded 4-pass implementation is not particularly accurate.

Our best guess is that the gains come primarily from two sources. First is increased flexibility in scheduling, which we discussed above in regard to the motivation for threaded implementations. The second source of performance gain is that the MPI calls in the threaded 4-pass implementation are synchronous, whereas they are asynchronous in the non-threaded 4-pass implementation. Apparently, asynchronous MPI calls incur a significant overhead. Although there is an overhead cost due to threads, the benefits of scheduling flexibility and synchronous MPI calls in the threaded 4-pass implementation outweigh this cost.

We conducted a set of ancillary tests to verify that a program with threads and synchronous MPI calls is faster than a single-threaded program with asynchronous MPI calls. The first test overlaps computation and I/O, and the second test overlaps computation and communication. We found that by converting a single thread with asynchronous MPI calls to a threaded program with synchronous MPI calls, the computation-and-I/O program ran 7.8% faster and the computation-and-communication program ran 23.8% faster.

Moreover, there is a qualitative benefit of the threaded 4-pass implementation. Because all calls to MPI and MPI-2 functions are synchronous, the code itself is cleaner, and it is easier to modify.

4 Threaded 3-Pass Implementation

This section describes how to reduce the number of passes in the implementation given in the previous section from four to three. The key observation is the *pairing observation* from [1]:

> We can combine steps 6–8 of columnsort 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.

Basic Structure. To take advantage of the pairing implementation, we combine steps 5–8 of columnsort—passes 3 and 4 in a 4-pass implementation—into one pass. Figure 2 shows how. In the 4-pass implementation, the communicate,

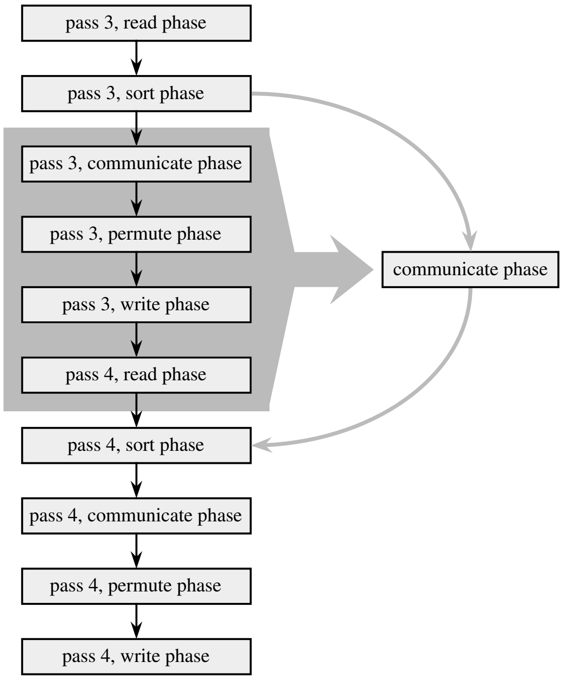

Fig. 2. How passes 3 and 4 of the threaded 4-pass implementation are combined into a single pass of the threaded 3-pass implementation. The communicate, permute, and write phases of pass 3, and the read phase of pass 4 are replaced by a single communicate phase.

permute, and write phases of pass 3, along with the read phase of pass 4, merely shift each column down by $r/2$ rows (wrapping the bottom half of each column into the top half of the next column). We replace these four phases by a single communicate phase.

In the threaded 3-pass implementation, the first two passes are the same as in the threaded 4-pass implementation. To further understand how the last pass works, let us examine a typical round (i.e., neither the first nor the last round) in this pass. At the start of the round, some processor $\mathcal{P}_k$ contains in a buffer $r/2$ records left over from the previous round. When the round completes, some other processor $\mathcal{P}_l$, where $l = (k-1) \bmod P$, will contain $r/2$ leftover records, and it will serve as $\mathcal{P}_k$ in the next round. The round proceeds as follows:

1. Each processor except for $\mathcal{P}_k$ reads in a column, so that $P-1$ columns are read in.
2. Each processor except for $\mathcal{P}_k$ sorts its column locally.
3. With the exception of $\mathcal{P}_k$, each processor $\mathcal{P}_i$ sends the first $r/2$ of its sorted elements to processor $\mathcal{P}_{(i-1) \bmod P}$. After all sends are complete, each processor except for $\mathcal{P}_l$ holds r records ($r/2$ that it had prior to the send, and

$r/2$ that it just received), and processor $\mathcal{P}_l$ holds $r/2$ records, which it keeps aside in a separate buffer to be used in the next round. This communicate phase replaces the four phases shaded in Figure 2.
4. Each processor except for $\mathcal{P}_l$ sorts its column locally.
5. To prepare for the write in PDM order, each processor (except $\mathcal{P}_l$) sends r/P records to every processor (including $\mathcal{P}_l$ and itself).
6. Each processor locally permutes the $r(P-1)/P$ records it has received to put them into the correct PDM order.
7. Each processor writes the $r(P-1)/P$ records to the disks that it owns.

The first and last rounds have minor differences from the middle rounds. The first round processes all P columns, and the last round may process fewer than $P-1$ columns. Because the first round processes P columns, the number of rounds is $1 + \lceil (s-P)/(P-1) \rceil$.

Thread Structure. In the threaded 3-pass implementation, the first two passes have the same thread structure as in the threaded 4-pass implementation, but the third pass is different. As we have just seen, the third pass has seven phases. As before, each phase is assigned to a single thread, except that the read and write phases are assigned to a single I/O thread. Consequently, there is one I/O thread, two communicate threads, two sort threads, and one permute thread. This thread structure raises two additional issues.

First, the number of additional buffers increases. Other than the I/O thread, each thread requires an r-record buffer. Since there are five non-I/O threads, this implementation requires five buffers more than are in the global pool. That is, the threaded 3-pass implementation requires two buffers more than the threaded 4-pass implementation. The figures for memory used in Table 1 reflect the larger number of buffers required for the threaded 3-pass implementation.

Second, because there are two communicate threads, each making calls to MPI, not all MPI implementations are suitable. Some MPI implementations are unreliable when multiple threads perform communication. Fortunately, the MPI implementations that we have used—on both the Sun cluster and Silicon Graphics Origin 2000 systems—support multiple threads that communicate via MPI calls.

Performance Results. By inspection of Table 1, we see that on the 4-processor, 4-disk and on the 8-processor, 8-disk clusters, the threaded 3-pass implementation takes 94.6% and 91.5%, respectively, of the time used by the threaded 4-pass implementation. The improvement due to reducing the number of passes from four to three is not as marked as that from introducing a threaded implementation.

The observed running times lead to two questions. First, what accounts for the improvement that we see in eliminating a pass? Second, why don't we see more of an improvement?

Compared to the threaded 4-pass implementation, the threaded 3-pass implementation enjoys one significant benefit and one additional cost. Not surprisingly, the benefit is due to less disk I/O. The pass that is eliminated reads and writes each record once, and so the 3-pass implementation performs only 75% as much disk I/O as the 4-pass implementation. This reduced amount of disk I/O is the only factor we know of that accounts for the observed improvement.

The added cost is due to more sort phases and more communicate phases. In the 3-pass implementation, each round contains two sort and two communicate phases, for a total of $2+2\lceil(s-P)/(P-1)\rceil$ sort phases and the same number of communicate phases. Together, the two rounds of the 4-pass implementation—which are replaced by the last pass of the 3-pass implementation—perform $2s/P$ sort phases and $2s/P$ communicate phases. Since the problem is out-of-core, we have $s > P$, which in turn implies that $2 + 2\lceil(s-P)/(P-1)\rceil \geq 2s/P$. Thus, the 3-pass implementation always has more sort and communicate phases than the 4-pass implementation.

The degree to which the 3-pass implementation improves upon the 4-pass implementation depends on the relative speeds of disks, processing, and communication in the underlying system. The 25% reduction in the number of passes does not necessarily translate into a 25% reduction in the overall time. That is because although the combined third pass of the 3-pass implementation writes and reads each record only once, all the other work (communication and sorting) of the two passes still has to be done. Therefore, when the last two passes of the 4-pass implementation are relatively I/O bound, we would expect the 3-pass implementation to be significantly faster than the 4-pass implementation. Conversely, when the last two passes are not I/O bound, the advantage of the 3-pass implementation is reduced. In fact, the 3-pass implementation can even be slower than the 4-pass implementation! On the clusters whose results appear in Table 1, the processing and network speeds are not particularly fast. Our detailed observations of the 4-pass implementation reveal that in the last two passes, the I/O and communication times are close. Hence, these passes are only slightly I/O bound, and therefore we see only a modest gain in the 3-pass implementation.

One would expect that technology will evolve so that the speed of processing and communication networks will increase more rapidly than the speed of disk I/O. If this prediction holds, then the last two passes of the threaded 4-pass implementation will become increasingly I/O bound, and so the relative advantage of the threaded 3-pass implementation will become more prominent.

5 Conclusion

We have seen two ways to improve upon our earlier, non-threaded, 4-pass implementation of out-of-core columnsort. This original implementation had performance results that, by certain measures, made it competitive with NOW-Sort. The two improvements make the implementation even faster.

One can characterize the threaded 4-pass implementation as an engineering effort, which yielded substantially better performance on the two clusters on which it was tested.

On the other hand, the threaded 3-pass implementation has both algorithmic and engineering aspects. On the particular clusters that served as our testbed, the performance gains were modest. We note, however, that if we were to run the threaded 3-pass implementation on a cluster with faster processors and a faster network, we would expect to see a more significant performance improvement. Our future work includes such experimental runs.

We intentionally omitted a non-threaded 3-pass implementation, even though that would have completed all four cases in the space of threaded vs. non-threaded and 4-pass vs. 3-pass implementations. The authors chose to not implement that option because, having seen the significant performance improvement from the threaded implementation, the substantial effort required to produce a non-threaded, 3-pass implementation would not have been worthwhile.

We have identified several other directions for future work. Can we bypass the size limitation given by $N \leq r^{3/2}/\sqrt{2}$? We have two approaches that we plan to investigate. One is to spread the r-record buffers across the memories of all processors, thereby increasing the maximum value of r by a factor of P. This change would allow the maximum problem size to increase with the amount of memory in the entire system, rather than being limited by the amount of memory per processor, as it is now. The second approach is to apply ideas used in the Revsort algorithm [6] to increase the exponent of r from 3/2 to 5/3. Another question is how best to remove the power-of-2 restriction on all parameters. Yet another opportunity for improvement lies in the observation that among all passes, the bulk of the sorting time occurs in pass 1. Might it be possible to not fully sort each column in pass 1 yet maintain the correctness of the overall algorithm?

Acknowledgments. The authors thank Len Wisniewski of Sun Microsystems, who was exceptionally helpful in preparing our experimental results. Thanks also to the anonymous reviewers for several helpful remarks.

References

1. Chaudhry, G., Cormen, T.H., Wisniewski, L.F.: Columnsort lives! An efficient out-of-core sorting program. In: Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures. (2001) 169–178
2. Leighton, T.: Tight bounds on the complexity of parallel sorting. IEEE Transactions on Computers **C-34** (1985) 344–354
3. Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H., Culler, D.E., Hellerstein, J.M., Patterson, D.A.: High-performance sorting on networks of workstations. In: SIGMOD '97. (1997)
4. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI—The Complete Reference, Volume 1, The MPI Core. The MIT Press (1998)
5. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI—The Complete Reference, Volume 2, The MPI Extensions. The MIT Press (1998)
6. Schnorr, C.P., Shamir, A.: An optimal sorting algorithm for mesh connected computers. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. (1986) 255–263