Relaxing the Problem-Size Bound for Out-of-Core Columnsort

Dartmouth College Department of Computer Science, Technical Report TR2003-445

Geeta Chaudhry
Elizabeth A. Hamon
Thomas H. Cormen

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

Contact author: Tom Cormen, 6211 Sudikoff Laboratory, Hanover, NH 03755.

Abstract

Previous implementations of out-of-core columnsort limit the problem size to $N \leq \sqrt{(M/P)^3/2}$, where N is the number of records to sort, P is the number of processors, and M is the total number of records that the entire system can hold in its memory (so that M/P is the number of records that a single processor can hold in its memory). We implemented two variations to out-of-core columnsort that relax this restriction. Subblock columnsort is based on an algorithmic modification of the underlying columnsort algorithm, and it improves the problem-size bound to $N \leq (M/P)^{5/3}/4^{2/3}$ but at the cost of additional disk I/O. M-columnsort changes the notion of the column size in columnsort, improving the maximum problem size to $N \leq \sqrt{M^3/2}$ but at the cost of additional computation and communication. Experimental results on a Beowulf cluster show that both subblock columnsort and M-columnsort run well but that M-columnsort is faster. A further advantage of M-columnsort is that it handles a wider range of problem sizes than subblock columnsort.

This research was supported in part by NSF Grant EIA-98-02068.

1 Introduction

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. For example, geographical information systems, seismic modeling programs, and web-search engines store and search through enormous amounts of data. In earlier papers [CCW01, CC02], the authors have reported on various programs that sort out-of-core data on distributed-memory clusters. All of these programs are based on Leighton's 8-step columnsort algorithm [Lei85]. In our adaptation of the columnsort algorithm to an out-of-core setting, we end up with a restriction on the maximum problem size that we can sort. We shall refer to this restriction as the *problem-size restriction*. The maximum number N of records[1] that we can sort on a cluster with P processors and M records of memory overall is

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

Here, M/P is the number of records that a single processor can hold in its memory.[2] There are two sources of this restriction:

1. *The height restriction.* Columnsort sorts N values in an $r \times s$ matrix, subject to some restrictions. One of the restrictions is $r \geq 2s^2$, so that the matrix is tall and thin.[3]

2. *The height interpretation.* In our prior implementations of columnsort, we require each column of the $r \times s$ matrix to fit in the internal memory of a processor. Setting r to be M/P, setting s to be N/r and substituting these values of r and s into the height restriction gives us the problem-size restriction (1).

In the present paper, we explore two approaches to relax the problem-size restriction. These two approaches stem from attacking separately the height restriction and the height interpretation.

Subblock columnsort: We relax the height restriction by adding two new steps to columnsort. The resulting algorithm, which we call *subblock columnsort*, relaxes the height restriction by a factor of $\sqrt{s}/2$, to $r \geq 4s^{3/2}$. With the same height interpretation of $r = M/P$, we get a problem-size restriction of

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

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.

Our best previous adaptation of columnsort to an out-of-core setting, which we call *threaded columnsort*, is structured into three passes, where a *pass* consists of reading each record once from disk, doing some computation, and writing it back to disk. The two additional steps of subblock columnsort add an extra pass, leading to additional disk I/O, communication, and computation. In certain special cases, this pass involves no communication.

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

[2] In reality, M/P is smaller than the actual size of the physical memory of a processor since we need some auxiliary buffers for in-core computation and interprocessor communication.

[3] Leighton's original paper [Lei85] has the restriction $r \geq 2(s-1)^2$. We choose to ignore the low-order terms and use the simpler and more stringent $r \geq 2s^2$.

M-columnsort: As expressed in restrictions (1) and (2), the maximum problem size depends on the amount of memory per processor, or M/P, even with the relaxed height restriction achieved by subblock columnsort. Therefore, if the number of processors in the cluster increases, but the amount of memory per processor stays fixed, the maximum problem size remains unchanged. This lack of scalability is due to the height interpretation. *M-columnsort* changes the height interpretation from $r = M/P$ to $r = M$, thus leading to the improved problem-size restriction

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

On a cluster with 16 processors, with $M/P = 2^{19}$ records, this change will allow us to sort up to one terabyte of data, assuming a record size of 64 bytes. M-columnsort does not add any extra passes compared to the original columnsort. As we shall see in Section 4, however, it does incur substantial amounts of communication and additional computation.

Experimental results on a Beowulf cluster with fast processors and a Myrinet interconnect show that the primary determinants of out-of-core execution time for a given algorithm are the amount of data per processor to be sorted and the amount of memory used on each processor. The dependence on data per processor is a consequence of the system being I/O-bound. Since subblock columnsort has four passes to threaded columnsort's three, subblock columnsort takes approximately one-third longer. M-columnsort, though it has exactly three passes, takes longer than threaded columnsort due to its increased computation and communication. Although M-columnsort takes longer than threaded columnsort, it runs faster than subblock columnsort in all cases.

The remainder of this paper is organized as follows. Section 2 describes the original columnsort algorithm and summarizes earlier implementations. Sections 3 and 4 present the design of subblock columnsort and M-columnsort, respectively, along with notes on their implementations. Section 5 analyses experimental results of the various columnsort programs. Finally, Section 6 offers some final comments and discusses future work.

2 Columnsort

In this section, we review the columnsort algorithm and earlier adaptations of it to an out-of-core setting. We conclude this section 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 $r \ge 2s^2$. 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:* 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. For example, in a 6×3 matrix, 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:* 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 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 earlier adaptations of columnsort to an out-of-core setting on a distributed-memory cluster, we assume that the cluster has P processors $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{P-1}$ and D disks $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{D-1}$, where $D \geq P$. A processor *owns* the D/P disks that it accesses.[4] Buffers hold exactly r records. 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 further 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 [CCW01, CC02]. 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. A round progresses through a pipeline with the following five stages:

Read stage: Each processor reads a column of r records from the disks that it owns.

Sort stage: Each processor locally sorts, in memory, the r records it has just read. Implementation of this stage differs from pass to pass.[5]

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

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

Because we implemented the stages asynchronously, at any one time each stage could be working on a different round.

The key features of our previous work are as follows:

- The first implementation [CCW01] used asynchronous I/O and asynchronous communication to overlap I/O, computation, and communication. This implementation had performance results that, by certain measures, made it competitive with the NOW-Sort program [ADADC+97].

[4] 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 $D \geq P$.

[5] In a given pass p, the data might start with some sorted runs, depending on the write pattern of pass $p-1$. The implementation takes advantage of the sorted runs to sort by merging.

- The second implementation [CC02] used threads in order to provide greater flexibility in overlapping I/O, computation, and communication. Experimental results showed that this improvement reduced the running time to about half of that of the first implementation. In this implementation, there were four threads per processor. The sort, communicate, and permute stages each had their own threads, and the read and write stages shared an I/O thread.
- The third implementation reduced the number of passes from four down to three by combining the last two passes into a single pass. The pipeline for the first two passes is unchanged. For the last pass, the pipeline had seven stages, two of which were sort stages and two of which were communicate stages; for details see [CC02]. Subblock columnsort and M-columnsort use this implementation as the starting point. We refer to this 3-pass implementation as the *threaded columnsort program*.
- All the implementations use only standard, off-the-shelf software, such as MPI [SOHL$^+$98] and MPI-2 [GHLL$^+$98] for communication and I/O.
- There are no assumptions required about the keys. In fact, our algorithm's I/O and communication patterns are oblivious to the keys.
- The output appears in the standard striped ordering used by the Parallel Disk Model (PDM).[6]

Recalling the problem-size restriction

We conclude this section by recalling 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 set r to be 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). 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.

3 Subblock columnsort

In this section, we present subblock columnsort and discuss some aspects of its implementation.

The algorithm

To describe subblock columnsort, we need to define what a *subblock* is. We will be working with $\sqrt{s} \times \sqrt{s}$ subblocks of the matrix, where each subblock is a contiguous set of $\sqrt{s}$ rows and $\sqrt{s}$ columns. Subblocks are aligned to the matrix, meaning that the indices of the top row and leftmost column of each subblock must be multiples of $\sqrt{s}$. We need $\sqrt{s}$ to be an integer, which, combined with the power-of-2 assumption, constrains s to be a power of 4. The main idea behind subblock columnsort, inspired by the Revsort algorithm [SS86], is to add two extra steps after step 3.

Subblock columnsort consists of the following ten steps:

[6]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, the implementations in [CCW01, CC02] are the first multiprocessor sorting algorithms whose output is in PDM order.

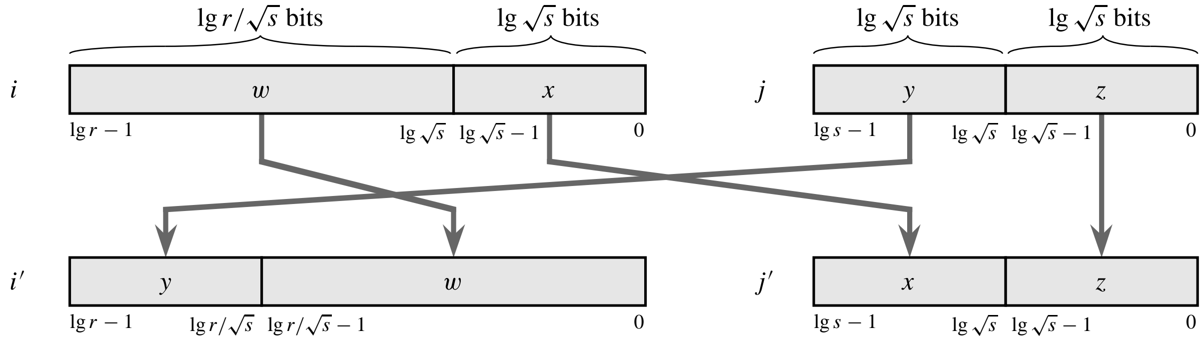

Figure 1: The subblock permutation as a bit permutation. Each entry in the $r \times s$ matrix has a row number, expressed in $\lg r$ bits, and a column number, expressed in $\lg s$ bits. The subblock permutation permutes the element in row i and column j to row i' and column j'. We number the bits starting from 0 as the least significant bit, and we use the notation "$..$" to denote ranges of consecutive bits. The permutation maps bits $w = i_{\lg \sqrt{s} .. \lg r - 1}$ to $i'_{0 .. \lg r/\sqrt{s} - 1}$, $x = i_{0 .. \lg \sqrt{s} - 1}$ to $j'_{\lg \sqrt{s} .. \lg s - 1}$, $y = j_{\lg \sqrt{s} .. \lg s - 1}$ to $i'_{\lg r/\sqrt{s} .. \lg r - 1}$, and $z = j_{0 .. \lg \sqrt{s} - 1}$ to $j'_{0 .. \lg \sqrt{s} - 1}$. Because x determines the source row number within an element's subblock and z determines the source column number within the subblock, this mapping ensures that the bits forming an element's target column number come from the bits that determine where in a source subblock the element started.

- Do steps 1–3 of columnsort.
- Step 3.1 performs any permutation that moves all the values in each $\sqrt{s} \times \sqrt{s}$ subblock into all s columns. We shall refer to this property as the *subblock property*.
- Step 3.2 sorts each column.
- Do steps 4–8 of columnsort.

In [CC03], it is shown that as long as s divides r, s is a power of 4, and $r \geq 4s^{3/2}$, subblock columnsort sorts any input correctly. This modified height restriction implies the problem-size restriction (2).

As discussed in [CC03], there are several permutations that have the subblock property. The one we use here, which we call the *subblock permutation*, is based on permuting sets of bits within the row and column numbers. Because we assume that r and s are powers of 2, each row number is a sequence of $\lg r$ bits and each column number is a sequence of $\lg s$ bits. We shall express the subblock permutation in terms of the source row and column numbers and the corresponding target row and column numbers of each matrix element.

To ensure that the subblock property holds, we only need to show that two distinct elements in the same source subblock will map to two different column numbers. We do so by ensuring that the $\lg s$ bits that determine the target column number come from source bits that determine where in a $\sqrt{s} \times \sqrt{s}$ subblock a matrix element resides. Figure 1 shows the idea. If we look at the source row and column numbers of a given element, the least significant $\lg \sqrt{s}$ bits of each—denoted by x and z, respectively, in the figure—determine the row and column numbers of that element within its subblock. (The most significant bits—w and y—determine which subblock the element is in, but not where in the subblock it resides.) The subsequences x and z form the bits of the target column number, with z forming the least significant half and x forming the most significant half. Thus, the subblock permutation has the subblock property. As an arithmetic formula,

the subblock permutation maps the (i, j) entry to position (i', j'), where

$$\begin{aligned} i' &= \left\lfloor \frac{j}{\sqrt{s}} \right\rfloor \frac{r}{\sqrt{s}} + \left\lfloor \frac{i}{\sqrt{s}} \right\rfloor , \\ j' &= j \bmod \sqrt{s} + (i \bmod \sqrt{s})\sqrt{s} . \end{aligned}$$

It may seem strange that the target row number is formed by using w as the least significant bits, when w started out as the most significant bits of the source row number. The advantage of permuting in this way is that it creates sorted runs of $r/\sqrt{s}$ elements in each column. To see why, first observe that entering the subblock permutation, each column is sorted. Now consider two elements e_1 and e_2 that start in the same column (so that their y and z bits are the same) and are permuted into the same target column (so that their x bits are also the same). There are $r/\sqrt{s}$ elements in a source column that have fixed values of the x, y, and z bits, and they vary in their w bits. Let the w bits of e_1 and e_2 be w_1 and w_2, respectively, and assume that $w_2 = w_1 + 1$ (so that e_2 is $\sqrt{s}$ rows below e_1 in the source column and $e_1 \leq e_2$). Because e_1 and e_2 have the same y bits, which become the most significant bits of the target row, and because the w bits become the least significant bits of the target row, e_2's target row is 1 greater than e_1's target row. Since e_1 and e_2 are any two elements that start out $\sqrt{s}$ rows apart in their source column, we see that all $r/\sqrt{s}$ elements that start in the same source column and are permuted to the same target column appear as one sorted run of size $r/\sqrt{s}$ in the target column.

Implementation notes

Since subblock columnsort differs from columnsort only in the two additional steps, our implementation of subblock columnsort started with the 3-pass threaded columnsort program and integrated these two steps as one extra pass, which we call the *subblock pass*. The subblock pass performs steps 3 and 3.1 of subblock columnsort. The overall thread structure of subblock columnsort is the same as that of the threaded columnsort program.

Like the first two passes in threaded columnsort, the subblock pass is decomposed into s/P rounds, where each round processes the next set of P consecutive columns, one column per processor, in a five-stage pipeline. The only stage of the subblock pass that differs substantially from the corresponding stage of passes 1 and 2 of threaded columnsort is the communicate stage. In each round's communicate stage within passes 1 and 2 of threaded columnsort, each processor sends P messages. Each message consists of r/P records. One of these messages goes back to the sending processor, in which case the message does not need to go over the network. For the subblock pass, however, we shall show three properties:

1. In the communicate stage of each round, each processor sends only $\lceil P/\sqrt{s} \rceil$ messages, each of size $r/\lceil P/\sqrt{s} \rceil$.

2. When $\sqrt{s} \geq P$ so that $\lceil P/\sqrt{s} \rceil = 1$, the one message is always destined for the sending processor, and therefore no communication over the network occurs.

3. Any permutation that achieves the subblock property must send at least $\lceil P/\sqrt{s} \rceil$ messages per round, and so the communication pattern of our subblock pass is optimal.

To see that the first two properties hold, we again refer to Figure 1. Let us examine the processors to which column j's elements are mapped by the subblock permutation. Column j is owned by processor p, where $p = j \bmod P$. This processor number is the $\lg P$ least significant bits of the column number j.

If $P \le \sqrt{s}$, then the bits that determine the processor number are entirely within the z field in Figure 1. Since these bits are the same in the target column as they are in the source column, the target processor number must be the same as the source processor number. Thus, since $\lceil P/\sqrt{s} \rceil = 1$, we have proven property 2.

If $P > \sqrt{s}$, then the $\lg \sqrt{s}$ least significant bits of the source and target column numbers for a given element are the same. Therefore, only $\lg P - \lg \sqrt{s}$ of the bits determining the processor number can differ. These bits come from the x field, which is part of the row number. All combinations of these bits will occur in a given source column, and so all combinations will occur in the target processor number. There are $2^{\lg P - \lg \sqrt{s}} = P/\sqrt{s}$ such combinations. By the power-of-2 assumption, therefore, we have $\lceil P/\sqrt{s} \rceil = P/\sqrt{s}$, and so we have proven property 1.

To show property 3, we shall show that if the subblock property holds and any source column maps to fewer than $P/\sqrt{s}$ target processors, then a contradiction arises. We start by noting that every processor must own exactly s/P columns. Now let us suppose that some source column, say column j, maps to k target processors, where $k < P/\sqrt{s}$. Therefore, column j maps to fewer than $k(s/P)$ target columns. Since $k < P/\sqrt{s}$, we have that $k(s/P) < \sqrt{s}$, so that column j maps to fewer than $\sqrt{s}$ target columns. Because the subblock property holds, any $\sqrt{s}$ entries within a given subblock must map to $\sqrt{s}$ different columns. If we consider the intersection of any subblock with column j, we have $\sqrt{s}$ entries of the subblock, and hence this portion of column j (not even considering the rest of the column) must map to $\sqrt{s}$ different target columns. This fact contradicts our earlier conclusion that column j maps to fewer than $\sqrt{s}$ target columns. Thus, we have proven property 3.

4 *M*-columnsort

In order to achieve the problem-size restriction (3), $N \le \sqrt{M^3/2}$, 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.

Implementation notes

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

We use in-core columnsort for the distributed-memory multiprocessor sort. We implemented three in-core multiprocessor sorting algorithms: bitonic sort, radix sort, and columnsort. We found that in-core columnsort, with an $(M/P) \times P$ matrix, was consistently faster than bitonic sort on problem sizes representative of those we encounter in the sort stage. Radix sort was competitive with in-core columnsort over a wide range of problem sizes, but we decided to use in-core columnsort because radix sort has a high dependence on the key format and because columnsort's communication patterns are independent of the values in the keys.

Our implementation of in-core columnsort is multithreaded. In particular, there are two threads: one for local sorting (steps 1, 3, 5, and 7 of the in-core sort) and one for communication (steps 2, 4, 6, and 8 of the

in-core sort). These threads are in addition to the four non-sort threads inherited from threaded columnsort. Wherever threaded columnsort's pipeline had a single stage for sorting, M-columnsort's pipeline has eight stages. Each stage in a pipeline sends a buffer to its successor, and the additional threads in M-columnsort require the allocation of four additional buffers.

The pipeline for each of the first two passes has 11 stages: read, the eight from in-core columnsort, permute, and write. These 11 stages occupy only four threads: one for both read and write, one for permute, one for the four local sorting steps of in-core columnsort, and one for the four communication steps of in-core columnsort. We are able to eliminate the communicate stage from the out-of-core pipeline because each column is shared among all the processors. After the in-core sort, each processor has its own portion of the sorted data. Depending on the permutation stage of the out-of-core sort, this data is destined for a certain set of target columns. Since each processor owns a portion of each column, we were able to design the in-core sort to ensure that each processor can write out its data into its portion of each of the target columns.

The pipeline for the last pass is rather different. We eliminate one communicate stage, but each of the two sort stages turns into eight in-core sort stages. Although the resulting pipeline has 20 stages, they occupy only seven threads: one for both read and write, one for permute, one for the remaining communicate stage, and two for each of the two in-core sorts.

5 Experimental results

The experimental results came from runs on a Beowulf cluster that has 16 dual 1.5-GHz P4 Xeon nodes, each with 1 GB of RAM. The nodes run Redhat Linux 7.2 and are connected with a high-speed Myrinet network which has a peak speed of 250 MB per second. Each node has an Ultra-160 10,000-RPM SCSI-3 hard drive, with about 10 GB of available disk space for user files. For disk I/O, we use the C `stdio` interface. The nodes communicate using standard synchronous MPI [SOHL+98] calls within asynchronous threads. Our threads are implemented using the pthreads package of Linux. Not all MPI implementations work correctly in the presence of multiple threads. We found that MPI/Pro, which allows multiple threads to issue MPI calls, worked well.

Although each node has two processors, we found no advantage to running more than one process per node. Thus, we treat each node as if it had only one processor, and we use the terms "node" and "processor" interchangeably.

Our experimental runs were for combinations of the following:

Algorithm: We ran threaded columnsort, subblock columnsort, and M-columnsort. For a baseline, we also ran just the I/O portions of three and four passes of columnsort. In this way, we can determine how much time of each run was spent waiting for non-I/O activity.

Buffer size: For threaded columnsort, subblock columnsort, and M-columnsort, we varied the buffer size (r). We report here results for buffer sizes of 2^{24} and 2^{25} bytes. 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. Record sizes varied between 64 and 128 bytes, but we found that the buffer size mattered more than the record size.

Number of processors and volume of data: We ran various combinations with 4, 8, and 16 processors and with an amount of data varying from 4 GB up to 32 GB. We did not run any experiments with

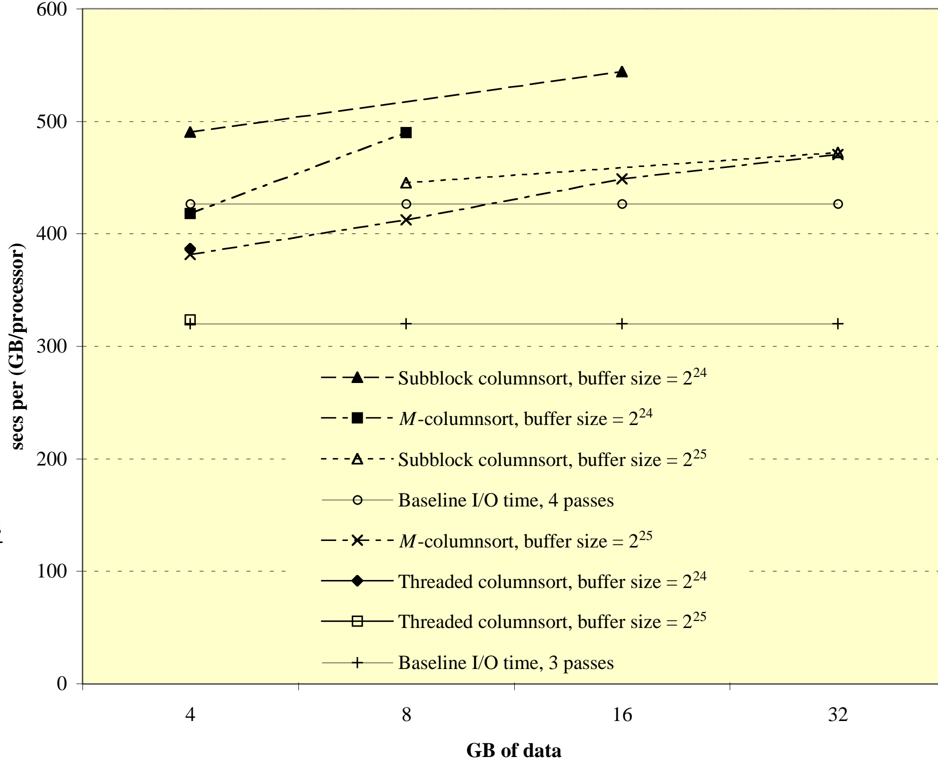

Figure 2: Execution times, in seconds, for the three versions of columnsort plus baseline I/O times for three and four passes.

less than 1 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 2 GB of data per processor due to disk-space limitations.[7] All runs, therefore, were for either 1 GB or 2 GB of data per processor.

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. We couch our results in terms of GB of data per processor.

Figure 2 summarizes our experimental results. Each plotted point in the figure represents the average of multiple runs of an algorithm with a given buffer size, but with the number of processors and the record size varying. Variations in running times were relatively small (within 10%). The horizontal axis is organized by total number of GB of data sorted across all processors.

Due to the problem-size restriction of equation (1), threaded columnsort could not handle more than 4 GB of data. Results for threaded columnsort appear as single points in Figure 2. For a buffer size of 2^{25} bytes, the total time is just barely above the baseline 3-pass I/O time; in other words, threaded columnsort is almost purely I/O-bound. When we halve the buffer size, to 2^{24} bytes, there are more frequent switches

[7] We did not overwrite the original data files so that we could verify the correctness of the output files. Moreover, our implementation requires a temporary file. Therefore, the input, output, and temporary files together induce a disk-space requirement of three times that of the input file size. The cluster on which we ran our experiments did not permit us to use much more than 6 GB of disk space per node.

between pipeline stages and so the I/O-boundedness diminishes somewhat. We found that with only one exception, larger buffer sizes resulted in faster execution. We could not make these buffers arbitrarily large because there is a limit on how large these buffers can get until demand paging starts slowing down the execution.

Due to the power-of-4 restriction on s in subblock columnsort, not all problem sizes were eligible to be run for a given value of r (i.e., for a given buffer size). That is why the two lines plotted for subblock columnsort in Figure 2 cover disjoint problem sizes, and why each line covers problem sizes that differ by a factor of 4. With a buffer size of 2^{25} bytes, the running times are only slightly above the baseline 4-pass I/O time. (Recall that subblock columnsort has one pass more than threaded columnsort.) Thus, subblock columnsort is substantially I/O-bound. With the smaller buffer size of 2^{24} bytes, execution times increase for the same reason as in threaded columnsort. Observe that each of the subblock columnsort lines rises only slightly as the volume of data sorted increases, thus demonstrating our earlier claim that the volume of data per processor is the most salient characteristic in determining execution time.

Figure 2 shows one particular advantage of M-columnsort over the other two algorithms: we were able to run it at all four problem sizes. In fact, had sufficient disk space been available, we could have run M-columnsort on up to one terabyte total on 16 processors with 2^{25}-byte buffers and 64-byte records. Execution times are well above the baseline 3-pass I/O time, and so M-columnsort is not nearly as I/O-bound as the other two algorithms. This fact is of course no surprise, since M-columnsort has a much more complicated in-core sort stage and also uses more memory (due to the extra buffers required by the additional threads).

In all our runs, M-columnsort was clearly superior to subblock columnsort. According to the performance results in Figure 2, M-columnsort was always at least as fast as subblock columnsort. The primary reason that M-columnsort was faster is that it makes only three passes over the data rather than four passes. For a given buffer size, subblock columnsort can handle only problem sizes that are powers of 4, whereas M-columnsort can handle any power-of-2 problem size. Furthermore, for most realistic configuration sizes, M-columnsort can sort larger problem sizes than subblock columnsort. By restrictions (2) and (3), M-columnsort can handle a larger number of records than subblock columnsort if $M^{3/2}/\sqrt{2} > (M/P)^{5/3}/4^{2/3}$ or, equivalently, if $M < 32P^{10}$. For example, if $P = 8$ processors, then as long as the total memory in the system holds fewer than 2^{35} records, M-columnsort will sort more records than subblock columnsort.

6 Conclusion

We have seen two ways to increase the problem-size bound in out-of-core columnsort. Subblock columnsort modifies the original columnsort algorithm by adding extra steps and altering the exponent in the height restriction. In the out-of-core implementation, subblock columnsort requires one additional pass. Like threaded columnsort, subblock columnsort's maximum problem size increases only with the amount of memory per processor (though the increase is superlinear). M-columnsort leaves the original columnsort algorithm intact, but it uses a different height interpretation in the out-of-core setting. By setting the column height to the amount of memory across the entire system, M-columnsort's maximum problem size increases superlinearly in the amount of memory in the system. Experiments on a Beowulf cluster show that both subblock columnsort and M-columnsort run well but that M-columnsort is superior. Moreover, M-columnsort handles a wider range of problem sizes than subblock columnsort.

There are several directions for our future work:

- We plan to combine subblock columnsort and M-columnsort into one four-pass algorithm which has a problem-size bound of $N \leq M^{5/3}/4^{2/3}$, i.e., restriction (2) but with M/P replaced by M. This

algorithm would have a larger problem-size bound than either subblock columnsort or M-columnsort alone.

- The closer the height interpretation is to $r = M/P$, the less communication overhead is incurred during the sort stages. We will develop an implementation that allows for values of r between M/P and M, depending on the problem size N for a given run.
- Our current implementations are I/O-bound, and their I/O bandwidth is below the sustainable rate of the disks. There is only so much that we can do about I/O rates while keeping the I/O interface at a reasonably high level. We do expect, however, to investigate memory-mapped I/O to eliminate unnecessary copying of data.
- We would like to eliminate the power-of-2 requirement from as many parameters as possible.
- There may be other algorithms for the in-core sort stages of M-columnsort that are superior to in-core columnsort. In particular, we will try a distribution-based sorting method.

Acknowledgments

The Beowulf cluster upon which we performed experiments is located at the Institute for Security Technology Studies (ISTS) at Dartmouth College. We are grateful to Garry Davis, Michel Gray, and David Nicol for granting us access to the cluster and to the researchers at ISTS for accommodating our schedule.

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

[ADADC+97] 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*, May 1997.

[CC02] 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.

[CC03] Geeta Chaudhry and Thomas H. Cormen. Stupid columnsort tricks. Submitted to SPAA 2003, January 2003.

[CCW01] 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.

[GHLL+98] 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.

[Lei85] Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985.

[SOHL+98] 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.

[SS86] 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.