Oblivious vs. Distribution-based Sorting: An Experimental Evaluation

Geeta Chaudhry*
Thomas H. Cormen†
Dartmouth College Department of Computer Science
{geetac,thc}@cs.dartmouth.edu

Abstract

We compare two algorithms for sorting out-of-core data on a distributed-memory cluster. One algorithm, Csort, is a 3-pass oblivious algorithm. The other, Dsort, makes three passes over the data and is based on the paradigm of distribution-based algorithms. In the context of out-of-core sorting, this study is the first comparison between the paradigms of distribution-based and oblivious algorithms. Dsort avoids two of the four steps of a typical distribution-based algorithm by making simplifying assumptions about the distribution of the input keys. Csort makes no assumptions about the keys. Despite the simplifying assumptions, the I/O and communication patterns of Dsort depend heavily on the exact sequence of input keys. Csort, on the other hand, takes advantage of predetermined I/O and communication patterns, governed entirely by the input size in order to overlap computation, communication, and I/O. Experimental evidence shows that, even on inputs that followed Dsort's simplifying assumptions, Csort fared well. The running time of Dsort showed great variation across five input cases, whereas Csort sorted all of them in approximately the same amount of time. In fact, Dsort ran significantly faster than Csort in just one out of the five input cases: the one that was the most unrealistically skewed in favor of Dsort. A more robust implementation of Dsort—one without the simplifying assumptions—would run even slower.

1 Introduction

This paper demonstrates the merit of oblivious algorithms for out-of-core sorting on distributed-memory clusters. In particular, we compare the performance of *Csort*, a 3-pass oblivious algorithm that makes no assumptions about the input distribution, to that of *Dsort*, a 2-pass distribution-based algorithm that makes strong simplifying assumptions about the input distribution. This difference makes Csort a more robust algorithm for sorting real out-of-core data. Because Csort is oblivious, its I/O and communication patterns are not affected by the input distribution or the exact input sequence. In Dsort, on the other hand, the I/O and communication patterns show a strong sensitivity to the input sequence, even when it adheres to the assumed input distribution. An added benefit of Csort's predetermined I/O and communication patterns is that it is much simpler to implement. We ran experiments with five 128-GB datasets, ranging from the more favorably skewed (for Dsort) to the less favorably skewed. There was no significant variation in the running

*Supported by National Science Foundation Grant IIS-0326155 in collaboration with the University of Connecticut.

†Supported in part by DARPA Award W0133940 in collaboration with IBM and in part by National Science Foundation Grant IIS-0326155 in collaboration with the University of Connecticut.

time of Csort for these inputs, demonstrating that its running time is determined primarily by the input size. Dsort ran no faster than Csort, except in the two most-favorably biased cases; the difference was marginal in one of these two cases. One downside of using Csort is that the maximum problem size that it can handle is often smaller than what Dsort can handle.

The problem of sorting massive data comes up in several applications such as geographical information systems, seismic modeling, and Web-search engines. Such *out-of-core* data typically reside on parallel disks. We consider the setting of a distributed-memory cluster, since it offers a good price-to-performance ratio and scalability. The high cost of transferring data between disk and memory, as well as the distribution of data across disks of several machines, makes it quite a challenge to design and implement efficient out-of-core sorting programs. In addition to minimizing the number of parallel disk accesses, an efficient implementation must also overlap disk I/O, communication, and computation.

Along with merging-based algorithms, distribution-based algorithms form one of the two dominant paradigms in the literature for out-of-core sorting on distributed-memory clusters. An algorithm of this paradigm proceeds in three or four steps. The first step samples the input data and decides on $P - 1$ splitter elements, where P is the number of processors in the cluster. The second step, given the splitter elements, partitions the input into P sets $S_0, S_1, \ldots, S_{P-1}$, such that each element in S_i is less than or equal to all the elements in S_{i+1}. The third step sorts each partition. The sorted output is just the sequence $\langle R_0, R_1, \ldots, R_{P-1} \rangle$, where R_i is the sorted version of S_i. We refer to the first, second, and third steps as the *sampling* step, the *partition* step, and the *sort* step, respectively. There is often a fourth step to ensure that the output is perfectly load-balanced among the P processors. In order to give Dsort every possible advantage when comparing it to Csort, its implementation omits the sampling and load-balancing steps.

In previous work, we have explored a third way—distinct from merging-based and distribution-based algorithms—of out-of-core sorting: oblivious algorithms. An *oblivious* sorting algorithm is a compare-exchange algorithm in which the sequence of comparisons is predetermined [Knu98, Lei92]. For example, algorithms based on sorting networks [Knu98, CLRS01] are oblivious algorithms.

A distribution-based algorithm, when adapted to the out-of-core setting of a distributed-memory cluster, generates I/O and communication patterns that vary depending on the data to be sorted. Due to this input dependence, the programmer ends up spending much of the effort in handling the effects of data that might lead to "bad" I/O and communication patterns. We know of two implementations of out-of-core sorting on a cluster [ADADC+97, Gra90]. Both are distribution-based, and both sidestep one of the principal challenges of the sampling step of a distribution-based sort: locating the kth smallest of the input keys for k equal to $N/P, 2N/P, \ldots (P-1)N/P$ (a classical problem in order statistics). We refer to these k elements as *equi-partition splitters*. Solving this problem in an out-of-core setting is cumbersome [Vit01]. Not surprisingly, therefore, both of the existing implementations assume that the $P - 1$ equi-partition splitters are known in advance, eliminating the sampling and load-balancing steps altogether.

An oblivious algorithm, when adapted to an out-of-core setting, generates I/O and communication patterns that are entirely predetermined, depending only on the input size. In previous work, we have developed several implementations based on the paradigm of oblivious algorithms [CCW01, CC02, CCH, CC04].

The comparison between Csort and Dsort is novel. To the best of our knowledge, this work is the first experimental evaluation that compares these two paradigms in the context of out-of-core sorting on distributed-memory clusters.[1] Moreover, both Csort and Dsort run on identical hardware, and both use similar software: MPI [SOHL+98, GHLL+98] for communication and UNIX file system calls for I/O. Both are implemented in C and use the standard pthreads package for overlapping I/O, communication, and

[1] There are some existing comparisons of parallel sorting programs for an in-core setting, e.g., [BLM+98].

	P_0				P_1				P_2				P_3			
	D_0		D_1		D_2		D_3		D_4		D_5		D_6		D_7	
stripe 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
stripe 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
stripe 2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
stripe 3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Figure 1: The layout of $N = 64$ records in a parallel disk system with $P = 4$, $B = 2$, and $D = 8$. Each box represents one block. The number of stripes is $N/BD = 4$. Numbers indicate record indices.

computation.

There are several reasons that we did not compare Csort to NOW-Sort [ADADC+97, ADADC+98], the premier existing implementation of a distribution-based algorithm:

- NOW-Sort is built on top of active-messages and GLUnix, and we wanted to use software that is more standard. (Our nodes run Red Hat Linux and use MPI for communication.)
- There are several differences in the hardware that NOW-Sort targets and the hardware of modern distributed-memory clusters.
- Finally, NOW-Sort does not produce output in the standard striped ordering used by the Parallel Disk Model (PDM) [VS94]. As Figure 1 shows, the PDM stripes N records[2] across D *disks* $D_0, D_1, \ldots D_{D-1}$, with N/D records stored on each disk. The records on each disk are partitioned into *blocks* of B records each. Any disk access (read or write) transfers an entire block of records between the disks and memory. We use M to denote the size of the internal memory, in records, of the entire cluster, so that each processor can hold M/P records.

  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 the resulting algorithm can be used as a subroutine in other PDM algorithms. Implementing our own version of a distribution-based sort removes all these differences, allowing for a fairer comparison.

Experimental results on a Beowulf cluster show that Csort, even with its three disk-I/O passes, sorts a 128-GB file significantly faster than Dsort in two of the five cases. There is a marginal difference in the running times in two of the remaining three cases, one in the favor of Dsort and the other in Csort's favor. In the last case, Dsort is distinctly faster than Csort. These results show that the performance of Csort is competitive with Dsort. Csort, therefore, would fare even better compared to a generalized version of Dsort, that is, one that does not assume that the equi-partition splitters are known in advance and that does not ensure load-balanced output.

The remainder of this paper is organized as follows. Section 2 describes the original columnsort algorithm and summarizes Csort. Section 3 presents the design of Dsort, along with notes on the 2-pass implementation. Section 4 analyzes the results of our experiments. Finally, Section 5 offers some final comments.

[2]The data being sorted comprises *N records*, where each record consists of a key and possibly some satellite data.

2 Csort

In this section, we briefly review the columnsort algorithm. After summarizing a 4-pass out-of-core adaptation, we briefly describe Csort, our 3-pass implementation. Our previous papers [CC02, CCW01] contain the details of these implementations.

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

- *Step 2: Transpose and reshape:* Transpose the $r \times s$ matrix into an $s \times r$ matrix. Then "reshape" it back into an $r \times s$ matrix by interpreting each row as r/s consecutive rows of s entries each.
- *Step 4: Reshape and transpose:* This permutation is the inverse of that of step 2.
- *Step 6: Shift down by $r/2$:* Shift each column down by $r/2$ positions, wrapping the bottom half of each column into the top half of the next column. Fill the top half of the leftmost column with $-\infty$ keys, and create a new rightmost column, filling its bottom half with ∞ keys.
- *Step 8: Shift up by $r/2$:* This permutation is the inverse of that of step 6.

A 4-pass implementation

In our adaptation of columnsort to an out-of-core setting on a distributed-memory cluster, we assume that D, the number of disks, equals P, the number of processors.[3] We say that a processor *owns* the one disk that it accesses. The data are placed so that each column is stored in contiguous locations on the disk owned by a single processor. Columns are distributed among the processors in round-robin order, so that P_j, the jth processor, *owns* columns j, $j + P$, $j + 2P$, and so on.

Throughout this paper, we use buffers that hold exactly β records. For out-of-core columnsort, we set $\beta = r$. We assume that each processor has enough memory to hold a constant number, g, of such β-record buffers. In other words, $M/P = g\beta = gr$, implying that $\beta = r = M/Pg$. In both Csort and Dsort, each processor maintains a global pool of g memory buffers, where g is set at the start of each run of the program.

Each pass reads records from one part of each disk and writes records to a different part of each disk.[4] Each pass performs two consecutive steps of columnsort. That is, pass 1 performs steps 1 and 2, pass 2 performs steps 3 and 4, pass 3 performs steps 5 and 6, and pass 4 performs steps 7 and 8. Each pass is decomposed into s/P *rounds*. Each round processes the next set of P consecutive columns, one column per processor, through a pipeline of five stages. This pipeline runs on each processor. In each round on each processor, an r-record buffer travels through the following five stages:

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

Sort stage: Each processor locally sorts, in memory, the r records it has just read.

[3] Our implementation of Csort can handle any positive value of D as long as D divides P or P divides D; we assume that $D = P$ in this paper. In our experimental setup, each node has a single disk, so that $D = P$.

[4] 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.

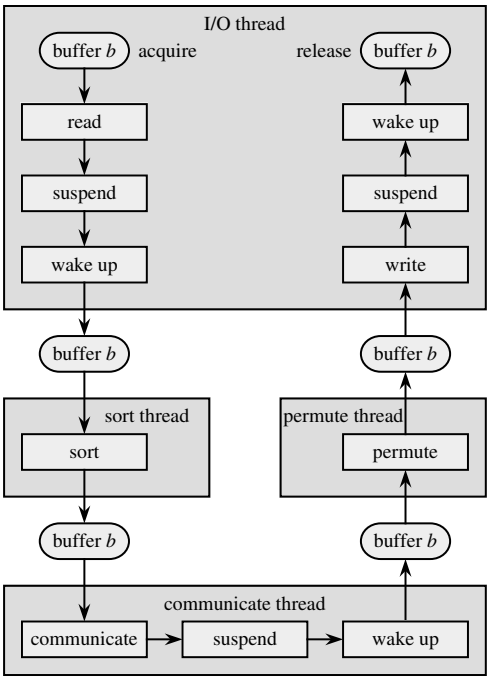

Figure 2: The history of a buffer b as it progresses within a given round of a given pass. The I/O thread acquires the buffer from the global pool and then reads into it from disk. The I/O thread suspends during the read, and when it wakes up, it signals the sort thread. The sort thread sorts buffer 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.

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 buffer from a different round. We used threads in order to provide flexibility in overlapping I/O, computation, and communication. In the 4-pass implementation, there were four threads per processor. As Figure 2 shows, the sort, communicate, and permute stages each had their own threads, and the read and write stages shared an I/O thread. The threads operate on r-record buffers, and they communicate with one another via a standard semaphore mechanism.

Csort: The 3-pass implementation

In Csort, we combine steps 5–8 of columnsort—passes 3 and 4 in the 4-pass implementation—into one pass. In the 4-pass implementation, the communicate, permute, and write stages of pass 3, along with the read stage 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 stages by a single communicate stage. This reduction in number of passes has both algorithmic and engineering aspects; for details, see [CC02, CCW01].

3 Dsort

In this section, we outline the two passes of Dsort: Pass 1 executes the partition step, and pass 2 executes the sort step, producing output in PDM order. We continue to denote the number of records to be sorted as N, the number of processors as P, and the amount of memory per processor as M/P. As in Csort, g is the number of buffers in the global pool of buffers on each processor and $\beta = M/Pg$ denotes the size of each buffer, in records.

3.1 Pass 1 of Dsort: Partition and create sorted runs

Pass 1 partitions the N input records into P *partitions* of N/P records each. After pass 1 completes, each processor has the N/P records of its partition. Similar to NOW-Sort, Dsort requires that the $P-1$ equi-partition splitters are known in advance.

The five stages of each round

Similar to the passes of Csort, pass 1 of Dsort is decomposed into rounds. Each round processes the next set of P buffers, one β-record buffer per processor, through a pipeline of five stages. In our implementation, each of the five stages has its own thread. Below, we describe what each processor (P_i, for $i = 0, 1, \ldots, P-1$) does in the five stages.

Read stage: Processor P_i reads β records from the disk that it owns into a buffer. Hence, a total of βP records are read into the collective memory of the cluster.

Permute stage: Given the splitter elements, processor P_i permutes the just-read β records into P sets, $U_{i,0}, U_{i,1}, \ldots, U_{i,P-1}$, where the records in set $U_{i,j}$ are destined for processor P_j. Note that, even though the entire input is equi-partitioned, these P sets need not be of the same size.

Communicate stage: Processor P_i sends all the elements in set $U_{i,j}$ to processor P_j, for $j = 0, 1, \ldots, P-1$. At the end of this stage, each processor has received all the records that belong to its partition, out of the total of βP records that were read in collectively.

The communicate stage is implemented in three substages:

- In the first substage, processor P_i sends the size of set $U_{i,j}$ to each processor P_j, so that P_j knows how many records it will receive from P_i.
- In the second substage, each processor sends P messages, one per processor. The message destined for processor P_j contains all the records in set $U_{i,j}$.

- After the second substage, it may be true that a processor needs to receive significantly more records than it can hold in its internal memory. The third substage, therefore, proceeds in a loop over the following three steps: allocate a buffer, keep receiving messages until the buffer fills up, and send the buffer on to the next stage. The performance of this stage depends on the input data, even with the assumption that the overall input is equi-partitioned. Since a processor may fill a non-integral number of buffers, it might be left with a partially full buffer at the end of this final substage. This partially full buffer is then used as the starting buffer (for the third substage of the communicate stage) of the next round.

Sort stage: For each buffer that it fills up in the communicate stage, processor P_i sorts the elements in the buffer.

Write stage: Processor P_i writes out the β records in each sorted buffer to the disk that it owns. If P_i fills no buffers in this round, no data is written out; the write stage just sends the buffer back to the read stage for subsequent rounds. If, on the other hand, P_i received several buffers' worth of records, the write stage writes out the full buffers. Only one buffer, however, returns to the read stage; the rest are released back into the memory of the processor.

At the end of pass 1, each processor has the N/P records that belong to its partition. Furthermore, these N/P records are stored on the disk of the corresponding processor as $N/\beta P$ sorted runs of β records each.

3.2 Pass 2 of Dsort: Merge sorted runs and create striped output

Pass 2 of Dsort executes the sort stage of distribution-based sorting. Each processor uses an $(N/\beta P)$-way merge sort to merge the $N/\beta P$ runs of β records each, and it communicates with the other processors to produce output in striped PDM order. Our implementation of pass 2 has four threads: read, merge, communicate, and write. Unlike pass 1, in which each thread received buffers from the thread of the preceding stage and sent buffers to the thread of the succeeding stage, the threads of pass 2 are not as sequential in nature. We describe how each thread operates.

Read thread: Each processor starts out with $N/\beta P$ sorted runs on its disk. Each run has β records, or β/B blocks, assuming that B, the block size, divides β, the buffer size.

- Wait for a request of the form (r_x, h_y, loc), meaning that the yth block h_y of the xth sorted run r_x is to be read into the memory location loc.
- Read in the yth block of the xth run, and copy the B records to the specified memory location loc.
- Signal the merge thread.

Since the $N/\beta P$ runs are being merged using an $(N/\beta P)$-way merge sort, the order in which the various blocks of the sorted runs are required depends on the rate at which each run is being consumed by the merge sort. For example, it is possible that on a given processor, the first run starts with a record whose key value is greater than all the key values in the second run. In this scenario, all blocks of the second run are brought into memory before the first run has exhausted even a single record. Feedback from the merge thread, therefore, is essential for the read thread to bring in blocks in an efficient sequence.

Merge thread: The merge thread initially requests some fixed number, say l, of blocks from each of the $N/\beta P$ sorted runs. This number l depends on the amount of memory available on each processor. It is essential to always have more than one block per run in memory, so that the merging process is not repeatedly suspended because of delays in disk reads. The main task of the merge thread is to create one single sorted run out of the $N/\beta P$ sorted runs produced after pass 1. It starts out by acquiring an output buffer, say $buff_0$, and starting an $(N/\beta P)$-way merge that puts the output into $buff_0$. This merging procedure continuously checks for the following two conditions:

- The output buffer $buff_0$ is full. In this case, the merge thread sends the buffer to the communicate thread and acquires another buffer from the pool of buffers.
- The current block, say h_y, of records from some run r_x has been exhausted. The merge thread sends a request to the read thread to get the next block of run r_x. Note that, since we always keep l blocks from each run in memory, the block that is requested is h_{x+l}. After issuing a request for block h_{x+l}, the merge thread waits to make sure that block h_{x+1}, the next block of r_y needed to continue the merging, is in memory.

Communicate thread: For each buffer that this thread receives, it spreads out the contents to all other processors, for striped PDM output. In other words, blocks $h_0, h_P, h_{2P}, \ldots$ are sent to processor P_0, blocks $h_1, h_{P+1}, h_{2P+1}, \ldots$ are sent to processor P_1, and so on, until processor P_{P-1}, which receives blocks $h_{P-1}, h_{2P-1}, h_{3P-1}, \ldots$. After the communicate thread receives one buffer's worth of data from all the other processors, it sends the buffer to the write thread.

Write thread: The write thread writes out the β records in the buffer out to the disk owned by the processor and releases the buffer back into the pool of buffers.

4 Experimental results

This section presents the results of our experiments on *Jefferson*, a Beowulf cluster that belongs to the Computer Science Department at Dartmouth. We start with a brief description of our experimental setup. Next, we explain the five types of input sequences on which we ran both Csort and Dsort. Finally, we present an analysis of our experimental runs.

4.1 Experimental setup

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

In all our experiments, the input size is 128 GB, the number of processors is 16, the block size is 256 KB; the record size is 64 bytes; and g, the size of the global pool of buffers, is 3. Since the record size is 64 bytes, N is 2^{31} records and B is 2^{12} records. For all cases but one, we use 128-MB buffers (i.e., $\beta = 2^{21}$ records). For Dsort, the memory requirement of the worst-case input type was such that we had to use 64-MB buffers; the experiment crashed if we used 128-MB buffers. In the case of Dsort, we set the parameter l of pass 2 to be 3.[5]

[5]The parameters g and l were set experimentally, to elicit the best performance.

4.2 Input generation

As we mentioned before, even with equi-partition splitters known in advance, the I/O and communication patterns of Dsort depend heavily on the exact input sequence. More specifically, as explained in Section 3.1, the performance of each round of pass 1 of Dsort depends on how evenly the βP records of that round are split among the P processors. In the best case, in each round, each processor would receive exactly β records. In the worst case, all βP records in each round would belong to the partition of a single processor.

In each of our five input types, the following is true: In each round, all βP keys are such that q out of the P processors receive $\beta P/q$ records each. In each round, therefore, $P - q$ processors receive no records. The five types of inputs are characterized by the following five values of q: 1, 2, 4, 8, and 16. For each round of any input type, the q processors that receive the βP records are chosen randomly, subject to the constraint that over all $N/\beta P$ rounds, each processor receives N/P records. The smaller the value of q, the worse the load balance of each round and the longer Dsort takes to sort the input. Note that $q = P$ represents a highly unrealistic input sequence, one where the data are perfectly load balanced across the P processors in every round, in addition to the overall input being perfectly balanced across the processors. Even values of q strictly less than P represent unrealistic scenarios in which the data destined for the q processors are perfectly load balanced across those q processors every time. Thus, our experiments are, if anything, tilted in favor of Dsort.

4.3 Results

Figure 3 shows the running times of Dsort and Csort for 128 GB of input data. Each plotted point in the figure represents the average of three runs. Variations in running times were relatively small (within 5%). The horizontal axis is organized by the five values of q, with $q = 16$ and $q = 1$ being the best and worst cases for Dsort, respectively.

As mentioned before, the running time of Csort exhibits negligible variation across the various values of q, demonstrating that Csort's predetermined I/O and communication patterns do indeed translate into a running time that depends almost entirely on the input size. Dsort shows much variation across the various values of q. As Figure 3 shows, Csort runs much faster for two of the five inputs ($q = 1$ and $q = 2$). In two of the remaining three cases ($q = 4$ and $q = 8$), the difference in the running times is marginal. Dsort runs significantly faster in the most favorable case ($q = 16$).

The variation in the running times of Dsort is due to differences in the performance of Pass 1. As explained in Section 3.1, the performance of the communicate and write stages is highly sensitive to the exact input sequence. Specifically, the lower the value of q, the more uneven are the I/O and communication patterns of each round, and therefore, the longer each round takes to complete. We ran Dsort for four values of q, and for each run, we observed the running time of each of the two passes. Figure 4 demonstrates that, across all four values of q, there is little variation in the performance of pass 2. The performance of pass 1, on the other hand, varies greatly with the value of q, thus substantiating our claim that pass 1 is responsible for the variation in the running times of Dsort.

5 Conclusion

This paper presents the first study that compares the paradigm of distribution-based algorithms to that of oblivious algorithms, in the context of out-of-core sorting on distributed-memory clusters. We do so by

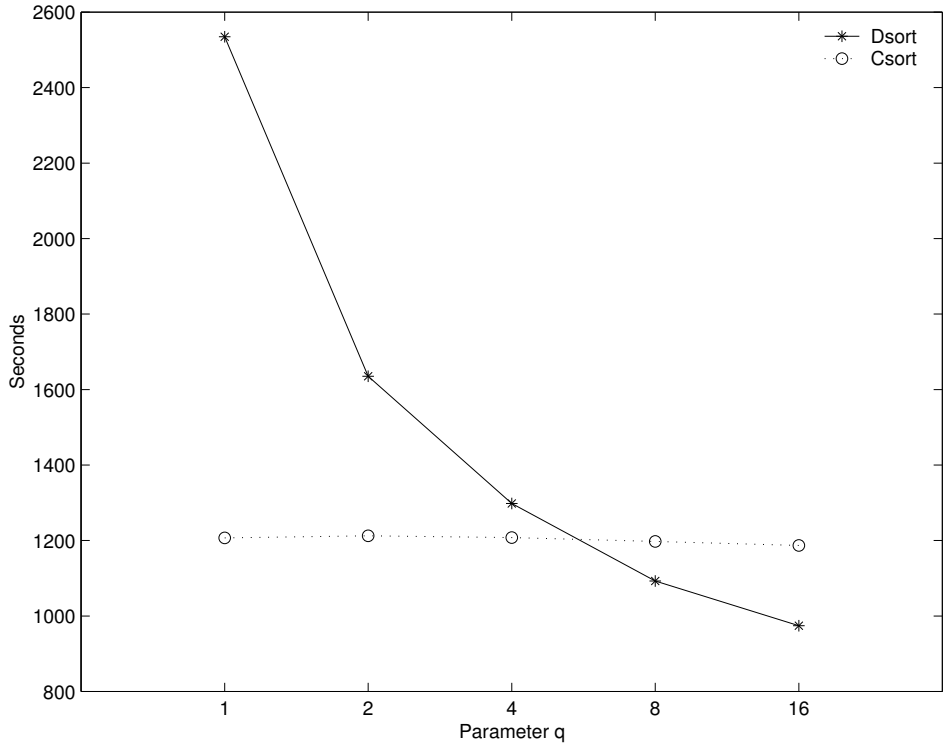

Figure 3: Observed running times of Dsort and Csort for the five values of q: $q = 16$ is the most favorable case for Dsort, and $q = 1$ is the least favorable. These timings are for the following setting of parameters: $P = 16$, $N = 2^{31}$ records or 128 GB, and $\beta = 2^{21}$ records or 128 MB, for 64-byte records.

comparing the performance of two algorithms: Csort and Dsort. Below, we summarize the main differences between Csort and Dsort:

- Csort is a 3-pass oblivious algorithm, and Dsort is a 2-pass distribution-based algorithm.
- Dsort assumes that the the equi-partition splitters are known in advance, thereby obviating the need for the sampling and load balancing steps. To sort all inputs, a distribution-based sort must remove this assumption, necessitating at least one more pass over the data. Csort makes no assumptions about the input distribution.
- The running time of Dsort, even when the input follows the above mentioned assumption, varies greatly with the exact input sequence. This variation arises because the I/O and communication patterns of Dsort are sensitive to the input sequence. The running time of Csort, on the other hand, varies negligibly with the input sequence. Our experimental results demonstrate this difference. This difference also makes Csort much simpler to implement.
- Dsort can handle problem sizes larger than those that Csort can handle. Csort can sort up to $\beta^{3/2}\sqrt{P}/2$ records, whereas Dsort can sort up to $\beta^2 P/B$ records.[6]

[6]Given the values of these parameters in our experimental setup, this difference translates as follows: Dsort can sort up to 2 TB of data, whereas Csort can only handle up to 512 GB of data.

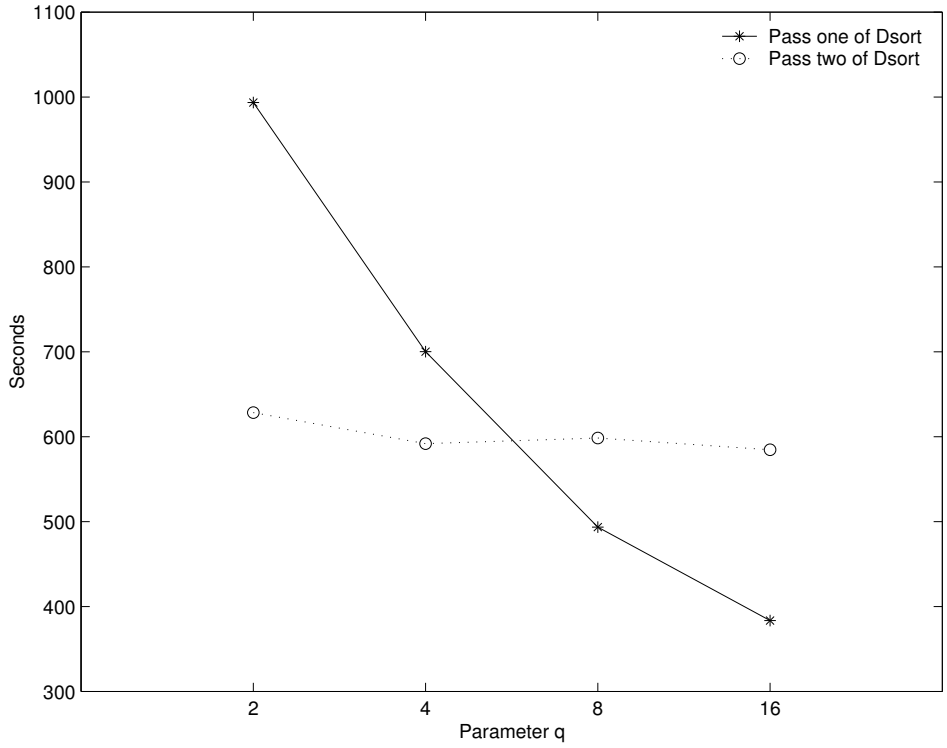

Figure 4: Observed running times of the two passes of Dsort for four values of q: 2, 4, 8, and 16. These timings are for the following setting of parameters: $P = 16$, $N = 2^{31}$ records or 128 GB, and $\beta = 2^{21}$ records or 128 MB, for 64-byte records.

Both Csort and Dsort are implemented using identical software, and they run on identical hardware. The results of our experiments show that Csort fares well compared to Dsort. On three out of five inputs, Csort runs faster. On one of the remaining two inputs, the difference between the running times of Csort and Dsort is marginal. Dsort runs significantly faster on the other remaining input, the one which represents the rather unrealistic case of $q = 16$.

In future work, we would like to implement a distribution-based algorithm that makes no assumptions about the input distribution and compare its performance with that of an oblivious algorithm. We also plan to continue our efforts toward designing new oblivious algorithms and engineering efficient implementations of them.

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*, 1997.

[ADADC+98] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. Searching for the sorting record: Experiences in tuning NOW-Sort. In *1998 Symposium on Parallel and Distributed Tools (SPDT '98)*, August 1998.

[BLM+98] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems*, 31:135–167, 1998.

[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.

[CC04] Geeta Chaudhry and Thomas H. Cormen. Slabpose columnsort: A new oblivious algorithm for out-of-core sorting on distributed-memory clusters. Submitted to *Algorithmica*, 2004.

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

[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.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 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.

[Gra90] Goetz Graefe. Parallel external sorting in Volcano. Technical Report CU-CS-459-90, University of Colorado at Boulder, Department of Computer Science, March 1990.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley, 1998.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.

[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.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[VS94] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2/3):110–147, August and September 1994.