A Low Communication Sort Algorithm for a Parallel Database Machine

Raymond A. Lorie and Honesty C. Young

IBM Almaden Research Center
San Jose, CA 95120-6099, USA

Abstract

The paper considers the problem of sorting a file in a distributed system. The file is originally distributed on many sites, and the result of the sort is needed at another site called the "host". The particular environment that we assume is a backend parallel database machine, but the work is applicable to distributed database systems as well.

After discussing the drawbacks of several existing algorithms, we propose a novel algorithm that exhibits complete parallelism during the sort, merge, and return-to-host phases. In addition, this algorithm decreases the amount of inter-processor communication compared to existing parallel sort algorithms. We describe an implementation of the algorithm, present performance measurements, and use a validated model to demonstrate its scalability. We also discuss the effect of an uneven distribution of data among the various processors.

1 Introduction

A database machine based on multiple nodes that share nothing is one way to provide the functionality of a conventional relational database system, with performance that goes far beyond what can be achieved with a single node. In particular, we consider a multiple node backend database machine that is connected to one or more host nodes. The interface to the database machine is SQL-like. This means that a complex query can be executed completely in the backend, and that only the request and the results are exchanged on the host–backend connection. For the sorting problem, however, we assume the host gathers sorted records in such a way that does not require sort key comparison. In other words, we do not carry out any comparison-based global sort or global merge at the host site. We also assume that the host extracts tuples from the communication messages and returns them to the application program.

The backend itself is a collection of n loosely-coupled nodes communicating by message passing on a network. Each node has its own processor(s), memory, channels, disks, and operating system. We assume that the hardware is symmetric in the sense that every node is equivalent to every other node. Besides the network which supports the n-to-n communication between the nodes, each node has a communication channel that it can use to receive a request from the host or send data back to the host. We refer to these communication channels as the n-to-1 network (see Fig. 1). We also assume that n-to-n communication shares the same media where each of the n-to-1 communication has its own media. One or more instances of the DBMS software may run on a single node, and each instance is called a *site*. The scheme of running multiple sites on a single node is called "virtualization" in [9].

We call data sites the sites where the records are originally stored and merge sites the sites where the merge operation is performed. In addition, there is a host site and a coordinator site. To simplify the discussion, we assume a site consists of a data site and a merge site, unless specified otherwise. P_i denotes the i-th site and P_{di} and P_{mi} denote the i-th data, and merge site, respectively. Depending on the algorithm, the sort operation may be performed on either the data sites or the merge sites.

Given a table T, with columns C1, C2, ... Ck, the sorting problem consists of returning to the host, the desired fields of the records that satisfy a predicate, ordered on the value of an expression computed on one or several columns. In order to simplify the analysis, we

Proceedings of the Fifteenth International
Conference on Very Large Data Bases

Amsterdam, 1989

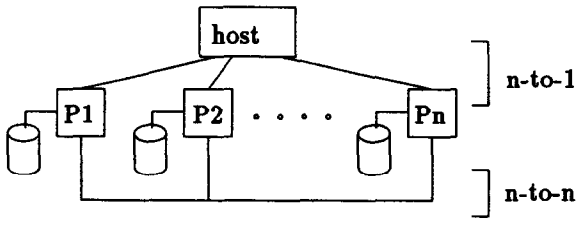

Figure 1: A Generic Backend Database Machine.

assume that the predicate is always true, that all fields are returned, and that the order is defined on a single column, as in the following SQL statement.

```
SELECT * FROM T ORDER BY C1
```

Barring data skew, the performance of a sort depends on the amount of CPU used at each site, the amount of disk I/O done at each site, the amount of n-to-1 communication, and the amount of n-to-n communication, and of course, the degree of parallelism. The cost complexity of n-to-n communication is typically superlinear: n^2 for a full crossbar switch or $n \log n$ for a multistage interconnection network. The cost complexity of n-to-1 communication in contrast is proportional to n[1]. So, a good sort algorithm for a distributed configuration is one that minimizes the amount of n-to-n communication, and exploits high degree of parallelism without substantially increasing the CPU and disk activity.

Section 2 discusses some related work. Section 3 proposes a novel, fully parallel algorithm (FPA) that decreases the amount of communication substantially. Section 4 describes briefly the ARBRE prototype built at the IBM Almaden Research Center, and provides some overall performance for a benchmark sort. Section 5 outlines a model to evaluate the new sort algorithm for a variety of parameter ranges. In section 6, we discuss the effect of an uneven distribution of data on the overall performance of the algorithm. In the conclusion, we summarize the benefits and applicability of the proposed sort algorithm.

2 Related Work

Researchers have studied the parallel sorting problem extensively (see the surveys in [2,8]). Most of the assumptions and results, however, are of theoretical interest (*e.g.*, the number of processors required is a function of the number of records to be sorted). Moreover, practical issues such as the times to read the source records from their data site(s) and to deliver the sorted records to a single host site are often ignored.

The sorting algorithms that are related to our current discussion are sometimes called "limited parallelism sorting algorithms" [2]. Four variations of such algorithms are worth mentioning.

Algorithm A: data sites send all records to a sort site S. S does the sort, then returns the sorted file to the host site. Although the scans are done in parallel, the sort itself is done serially.

Algorithm B: sort the data locally at each data site and merge the sorted file at a single merge site. This is similar to Tandem's FASTSORT [5] except that, in FASTSORT, the input data constitute a single file and a preliminary phase distributes the data before the local sorts take place.

Algorithm C: sort the data locally in each data site, organize the merge sites into a binary tree, and merge the sorted files in a pipelined fashion. This is a variation of the *parallel binary merge algorithm* in [3], which is itself an improved version of an algorithm described in [4] in that it pipelines the merge phase. The sort used in DBC/1012 [11] is similar to algorithm C in that each AMP does a local sort and sends the sorted stream through the Y-net, which carries out the merge operation in hardware.

Algorithm D: data sites use key-range partitioning to redistribute the data to sort sites; sort sites sort the redistributed data locally, and the host site concatenates the sorted file. This is a parallel version of the distributive partitioned sorting proposed in [7]. This algorithm computes, by some method, a partitioning of the key range R into subranges R1, R2, ..., Rn such that a roughly equal number of records falls in each subrange. Any well-known distributed random sampling algorithms may be used here. Each subrange Ri is assigned to one sort site in a one-to-one fashion; for example, Ri is assigned to P_{mi}. Then, each data site sends each of its records to the appropriate sort site that handles the range containing the record's sort key. When all records have been sent, all records with keys in the first range are all in P_{m1}. All records with keys in the second range are in P_{m2}, etc. Each site can then sort the records that it received. When done, P_{m1} returns its sorted records to the host. When it finishes, P_{m2} does the same, etc., until P_{mn}.

Each of the above algorithms has at least one phase where parallelism is not exploited. Algorithm A has a serial sort phase. Algorithm B has a serial (non-pipelined) merge phase. Algorithm C also has a serial (pipelined) merge phase. Algorithm D has the most parallelism, but it sends entire records on the n-to-n communication network, imposing a load of (number_of_records * record_length).

In algorithms B, C, and D, the return of the results

[1] n is the number of nodes in the system

to the host is serialized. There is a single site that sends the sorted records to the host in algorithms B and C. In algorithm D, the sorted records are sent first from P_1, then from P_2, etc. In our experiments, this is responsible for most of the response time.

In the next section, we propose a novel algorithm, similar to algorithm D, that eliminates the drawbacks mentioned above. The technique used to decrease the n-to-n communication is similar to the one used in a semi-join. To our knowledge, the only other research work that sends only the sort keys during an intermediate step of a sorting algorithm is the "key broadcasting" algorithm described in [10], where a parallel sort is executed in a common-bus local network environment.

3 A Fully Parallel Algorithm

Our sort algorithm has parallel sort, merge, and return-to-host phases. Thus, it is called a fully parallel algorithm (FPA). We will exploit the size difference between the sort key and the entire record by sending only sort keys from the data sites to the merge sites. The data sites send sorted files directly to the host which efficiently "merges" them without doing sort key comparisons. As explained below, the host's merge algorithm uses information computed during the merge sites' processing of the sort keys. The merge sites send only this information to the host; the sort keys are discarded as they are merged.

Before presenting the details of our sort algorithm, we describe it at a high level. Each data site sorts locally the portion of the file that it owns; in doing so, it computes the distribution of the keys that it encounters, and sends the distribution to a global coordinator. Based on such information from all data sites, the coordinator assigns each merge site a range of keys for which it is responsible. Each data site sends each of its record keys (in sorted order) to a certain merge site. For each record, the merge site is chosen as the site responsible for the key of that record. Each merge site merges the keys that it receives from all data sites, determining the order that the data sites deliver the next record to the host site.

3.1 Detailed Description

There are four kinds of tasks involved in the sort (see Fig. 2): the host task (always executed on the host machine), the coordinator (executed on any site), the scan tasks (one on each data site), and the merge tasks (one on each merge site). The tasks enclosed by the dotted line box are running on each site. The sort and the scan operations can be done by the same scan task because their operations do not overlap in time. In order to distinguish the work between merging the sort

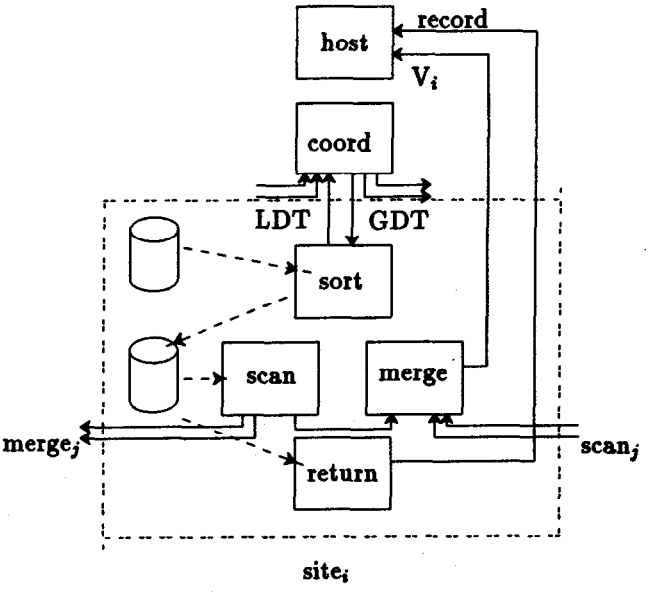

Figure 2: The Dataflow among Tasks.

keys and returning the sorted records to the host, the data sites do not send sorted records to the host site until all the sort keys have been sent to the merge sites. Therefore, the scan task is also responsible for returning the sorted records to the host site.

The host task sends the sort request to the coordinator and waits until it starts receiving a stream of site Ids V_1 from the P_{m1}, etc. At that time, the host considers successively each value in V_1. A value i indicates that it needs to read the next record from the stream of records coming from data site P_{di}.

The coordinator waits for a request from the host. When it receives the request for a sort, it sends the request to all data sites and merge sites. It then waits for all data sites to send their distribution tables. Based on these distribution tables the coordinator constructs the GDT (Global Distribution Table). The GDT is then sent to all data sites. The coordinator needs to be involved no longer.

The sort itself has four phases–local sort, synchronization, merge, and return-to-host.

Phase 1: Local Sort

Phase 1 starts on any P_{di} as soon as it receives its sort request from the coordinator. P_{di} scans its local records and sorts the records using any algorithm. The result of the sort is a file S_i of sorted records. Each record in S_i contains a key and a record body. While the sort is taking place, two objects are being built besides the sorted output: (1) A list that contains the lowest key stored in a single page of S_i and a descriptor for that

page. This list is called the *index list*, because it is used to determine the page that contains the first record of a key range. (2) A small table, called distribution table, containing information on the key distribution; it is derived from the index list mentioned above.

This sorting algorithm and the data structures it uses are best illustrated by an example. Consider a system with three sites. Each record has two fields and the first (single byte) field is the sort key field. Let the contents of the sorted files S1, S2 and S3 be, respectively:

S1	S2	S3
A Alice	C Carl	A Adam
B Barbara	D Debbie	B Brian
G Greg	F Frances	C Christine
H Helen	F Frank	D Dan
H Henry	G Gloria	E Elizabeth
K Kevin	M Mary	E Eric
L Linda	N Nancy	J Jane
M Mark	P Peggy	J John
N Norman	R Ron	K Kathy
S Steve	S Sue	L Larry
	T Teresa	P Peter
	W Walter	R Ruth
		T Tom
		W Wendy

(a)

A particular sampling algorithm may generate the following distribution tables:

B	D	B
H	F	D
K	M	E
M	P	J
S	S	L
	W	R
		W

(b)

Phase 2: Synchronization

As soon as Phase 1 is completed on a P_{di}, this P_{di} sends its distribution table to the coordinator and waits for the GDT from the coordinator. Once the coordinator has received the distribution tables from all sites, it combines their contents and builds a single GDT. The GDT contains the boundaries of n subranges, such that the numbers of records with keys falling in each range are roughly equal. Each range is assigned to a merge site on a one-to-one basis (for example range R1 to site P_{m1}, etc). The coordinator then sends the GDT to all sites.

Let's use the previous example to show how the coordinator computes the GDT. The coordinator receives the data in (b), merges the distribution tables, and partitions the results into 3 roughly equal size groups, to obtain the range definitions that constitute the GDT:

```
B B D D E F   H J K L M M   P R S S W W

---------- F ---------- M ----------
lowest                        highest
```

Phase 3: Merge

During this phase, there are two tasks at each site, working concurrently: a producing scan task and a consuming merge task. Once a data site receives the GDT its producing scan task reads sequentially the records from its sorted file. For each record in the sorted file, the key is extracted and sent to the merge site P_j, where j is such that the key falls in the j-th range defined by GDT. It is received by the consuming merge task of P_j for merging.

A strictly sequential processing of file S_i is not optimal as the following scenario illustrates. Assume all sites start sending only records that fall in the first range. Therefore all traffic is destined for the merge site in charge of the first range, which becomes overloaded. When records in the second range are sent, another merge site becomes overloaded, and the process continues. Our solution to the problem consists of processing only a small portion of the records in the first range (for which the target is P_{m1}). The small portion may be a single page from each data site. While P_{m1} handles the corresponding merge, all producing tasks jump to the beginning of the second range, etc. After the first batch of Rn has been processed, each producing task sends its second batch of R1 and so on, in a round-robin fashion. Clearly, if one looks at a sufficiently large period of time, all data sites produce records that are sent to the appropriate merge sites. So all sites are kept busy, sending, receiving and merging records.

In the following example, we denote a key sent by data site P_{di} to merge site P_{mj} by <key, i>→ P_j [2] for illustrative purpose. P_i starts producing records, sending them to the indicated merge sites. If each data site sends 2 records in a batch, the following communication pattern occurs:

batch1:

<A,1> → P1	<C,2> → P1	<A,3> → P1
<B,1> → P1	<D,2> → P1	<B,3> → P1
<G,1> → P2	<G,2> → P2	<J,3> → P2
<H,1> → P2	<M,2> → P2	<J,3> → P2
<N,1> → P3	<N,2> → P3	<P,3> → P3
<S,1> → P3	<P,2> → P3	<R,3> → P3

[2] The sender's site identifier can be decoded easily from a message and is not included in the message body.

batch2:

<H,1> → P2 <F,2> → P1 <C,3> → P1
<K,1> → P2 <F,2> → P1 <D,3> → P1
...

Since the data sites are not synchronized, the actual communication pattern will differ from the lock-step pattern shown here.

P_1 receives from all sites (including itself):

<A,1>	<C,2>	<A,3>
<B,1>	<D,2>	<B,3>
EOS	...	...
	EOS	EOS

Since the keys are sent to a different consuming merge task in each round, all merge tasks may start at about the same time. Clearly, the merge task of Pmk receives records that have keys that fall into the range for which the site P_{dk} is responsible, and all records received from a single site arrive in order. These ordered streams of keys can thus be merged to produce a single output stream. Each entry in the output stream contains only the identifier of the data site.

For example, consider the first merge site. It merges its records to obtain the following order:

<A,1> <A,3> <B,1> <B,3> <C,2> <C,3> ...

and produces the following V_1 stream:

V1 1 3 1 3 2 3 2 3 2 2 3 3

The other two merge sites produce their output streams in a similar fashion:

V2 1 2 1 1 3 3 1 3 1 3 1 2
V3 1 2 2 3 2 3 1 2 2 3 2 3

Phase 4: Returning Records to the Host
When Phase 3 is finished at site P_i, the corresponding stream V_i and sorted file S_i are sent as streams to the host. If the next entry in V is k, the host knows that it needs to get the next record from S_k sent by data site P_k.

In the example, the host receives a stream of site identifiers (V), which is the concatenation of V_1, V_2 and V_3. It also receives the sorted S's (a) from each site. Following the site id order of V, it retrieves records in that order:

A Alice	from P1
A Adam	from P3
B Barbara	from P1
B Brian	from P3
C Carl	from P2
C Christine	from P3
...	

3.2 Alternative Implementations

The algorithm as explained above is the one that we implemented and evaluated (see section 5 below). Some variations are worth discussing.

1. In the above description, the entire file S is read twice–once to send the keys to the merge tasks, and once to return the records to the host. It may be preferable to have the local sort write two files: S with the entire records and a much smaller file K with keys only. In phase 3, the smaller file K is read, instead of S.
2. When multiple merge sites are running on a single node (because of site virtualization), the merge phase can be divided into several sub-phases. The host site cannot start receiving records in the R2 ranges until it has received all records in R1, which cannot finish before P_{m1} finishes. If the merge phase has several sub-phases, the host can start receiving records in the ranges for which the corresponding merge operations have completed. That is, the host site receiving operation is pipelined with the merge site merging operation. The most important advantage of the pipelined scheduling is that the time to receive the records is expanded, thus, the resource requirements on the host can be smoothed out. Consequently, the same host can accommodate more nodes without incurring response time penalty.
3. Finally, other sampling algorithms may be used to compute the GDT.

4 Prototype

At IBM Almaden Research Center, we have built a prototype of a backend database machine, called ARBRE (Almaden Research Backend Relational Engine), in order to explore the use of parallelism within complex relational queries, and across many independent transactions [9].

A prototype of the database machine is now operational. It runs on three dyadic 4381 nodes providing 6 processors for the backend database machine; a fourth 4381 is used as a host. Each processor is approximately 3.5 MIPS (million instructions per second). The operating system is MVS and a 3088 is used for channel-to-channel communication. The database system is organized as follows: on each of the 4381's, four virtual sites are emulated by running four DBMS's in four address spaces, providing 12 sites in total. Each relation is split horizontally into partitions; each partition is stored in a different DBMS. The access method is borrowed from SQL/DS (DBSS) [6].

Since we have not implemented a query compiler, we

hand-code programs that specify the strategy that would normally be chosen by the optimizer. A strategy is expressed as a series of routines which are generally duplicated at each site. These routines produce or consume streams of data that are passed between disks, nodes, and the host. The execution is controlled by data flow and stream communication is controlled by a pacing mechanism.

We exercised the prototype for various sets of parameters. Here we report the results for one particular set of parameters which we use later to validate our model:

- number of sites : 12 (2 site per processor of 3.5 MIPS, 21 MIPS aggregate)
- number of records: 12 * 16000 records = 192,000 records
- record length : 64 bytes
- key length : 8 bytes (included in the 64)

For each of the four phases, we measured the CPU time, the number of I/O's, the number of messages, and the response time. Results on each site are shown below.

	sort	sync	merge	return	total
CPU (sec)	12.0	-	10.2	6.2	28.4
I/O	-	-	-	-	2884
Comm	2	2	159	509	672
Resp (sec)	44.7	1.6	22.4	23.6	92.3

Note that the CPU for the synchronization is too small to be measured and that we only show the total I/O because some pages logically modified in one phase are written later, due to the buffer page replacement algorithm, blurring the boundaries between phases.

5 Model

In this section, we model every phase separately, and immediately apply the model to the experiment just described.

Let's define the path length of various operations by the following symbols (approximate values from the prototype are given in parentheses):

- to extract a record from a list: Ex (= 950)
- to enqueue a record to a stream: Eq (= 150)
- to dequeue a record from a stream: Dq (= 150)
- to send or receive a message: Cm (= 4000)
- to perform an I/O: Io (= 8000)
- to carry out a key comparison: Cp (= 150)

The system parameters and their values are:

- the page size: P (= 4086[3])
- the message size: M (= 2048)
- the record length: Rl (= 64)
- the key field length: Kl (= 8)
- the number of sites: S (= 12)

The following symbols are also used, and all refer to one site only:

- the number of records: Ni
- the number of disk I/O's: Nio
- number of pages in the base table: Nb
- number of pages in the index list: Nl
- the number of communication messages: Nc

When two sites are sharing the same physical processor, we model the response time to be either the sum of the CPU and I/O times (when it is not CPU bound) or twice the CPU time (when it is CPU bound). Let us analyze what happens at one site.

Phase 1: Local Sort
We developed a model for the CPU and I/O times of local sorts and validated the model by running multiple experiments on our prototype. The detailed description of the local sort model is not relevant to this paper; thus, it will not be explained further. For the chosen parameters, the CPU time is 12 sec and the I/O time is 33 sec. Based on our response time model, the response time for 2 simultaneous sorts is (12 + 33) = 45 sec.

Phase 2: Synchronization
A message is sent from each site to the coordinator and from the coordinator to each site. This is a small linear overhead. Each site has an equal number of records to sort; so all sorts can be expected to terminate roughly at the same time; therefore we will neglect the time required for this phase but will return to the skew problem in a later section.

Phase 3: Merge
Using the symbols defined above, the first order[4] approximation gives us:

- $Nb = \text{ceiling}(Ni/\text{floor}(P/Rl))$
- $Nl = \text{ceiling}(Nb/\text{floor}(P/Kl + 4^5))$
- $Nio = Nb + Nl$
- $Nc = \text{ceiling}(Ni/\text{floor}(M/Kl))$

Consequently, we have

- $Nb = \text{ceiling}(16000/\text{floor}(4086/64)) = 254$
- $Nl = \text{ceiling}(254/\text{floor}(4086/12)) = 1$
- $Nio = 254 + 1 = 255$
- $Nc = \text{ceiling}(16000/\text{floor}(2048/8)) = 63$

The total path length and its breakdown are shown below:

[3] 4096 bytes less 10 bytes page header

[4] In particular, we ignore the path length related to the site identifier manipulation since each site identifier is very small (one to two bytes).

[5] The page descriptor has 4 bytes.

operation	formula	path length (M)
I/O	Nio * Io = 255 * 8000	2.04
Key Extraction	Ni * (Ex + Cp)) = 16000 * (950 + 150)	17.60
Enqueue	Ni * Eq = 16000 * 150	2.40
Send	Nc * Cm = 63 * 4000	0.25
Receive	Nc * Cm = 63 * 4000	0.25
Dequeue	Ni * Dq = 16000 * 150	2.40
Merge	Ni * (ceiling($\log_2$(S)*Cp) = 16000 * (4*150)	9.60
Total		34.5

The order of the operations shown above reflects the sequence for a "typical" record. The Cp term in the formula for key extraction is the path length associated with the partition range check. The CPU time on a 3.5 MIPS processor for 34.5 M instructions is 9.9 sec. Assuming each I/O takes 20 ms, the elapsed time for I/O is 255 * 0.020 = 5.1 sec. The phase is strongly CPU bound. Therefore running simultaneously two such phases on one processor will yield an elapsed time of (2 * 9.9) = 19.8 sec.

These 19.8 sec are somewhat less than the measured 22.4 sec. Part of the difference is due to the fact that the merge needs always a record from a specific incoming stream in order to proceed. Therefore, some waiting time can be expected. A simulation has shown that the relative response time increase due to the deterministic merge varies very little with the increase in the number of sites: 3% for 4 sites to roughly 11% for 64 or 128 sites.

Phase 4: Return

For the return phase, we obtain the following path length:

operation	formula	path length (M)
I/O	Nio * Io = 254 * 8000	2.03
Record Extraction	Ni * Ex = 16000 * 950	15.20
Enqueue	Ni * Eq = 16000 * 150	2.40
Send	Nc * Cm = 500 * 4000	2.00
Total		21.6

The CPU time on a 3.5 MIPS processor for 21.6 M instructions is 6.2 sec. The I/O time is 254 * 0.020 = 5.1 sec. Again, since the phase is CPU bound, the elapsed time for two such phases executed simultaneously is 2 * 6.2 = 12.4 sec. This is true only if the host and the communication network are infinitely fast. In fact one can show that the host CPU requirement is 15.5 sec.

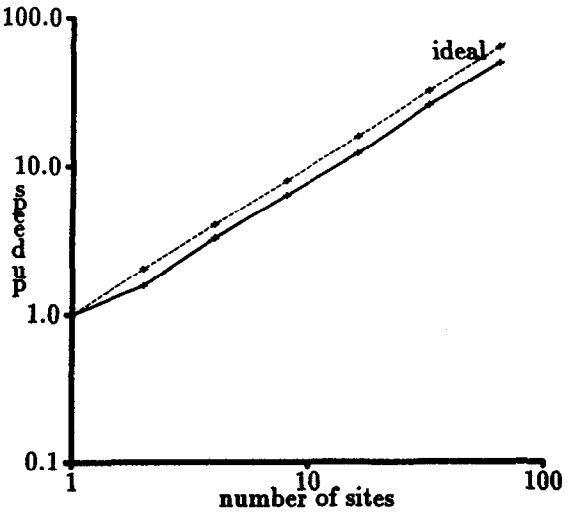

Figure 3: Response Time Reduction.

To summarise, let us compare the results of the model and the actual measurements:

		sort	merge	return
CPU		12.0	10.2	6.2
	(mod)	12.0	9.9	6.2
Resp.		44.7	22.4	23.6
	(mod)	45.0	19.8	>15.5

The response time of the return phase is longer than estimated because the channel-to-channel used in the prototype to implement the n-to-1 communication network becomes the bottleneck.

5.1 Exercising the Model

After having validated the model with actual measurements, we exercised it at will to examine response time reduction and horizontal growth. We adjusted some parameters to values that are more appropriate for high-performance relational DBMS, but difficult to change in our current prototype implementation. We assume that I/O streams read sixteen 4096 byte pages at a time with a latency of (20 + 15 * 16) = 44 ms, and message size of 4KB instead of 2KB. To eliminate the site virtualization effect, we run one site on a processor. We also consider 4 MIPS backend processors, and a 32 MIPS host processor.

We studied the response time reduction by sorting a total of 1 million 64 byte records that are evenly distributed among all sites. We also assume here that the GDT gives the exact partition. For the single site case, the synchronization, and merge phases are not executed. Fig. 3 shows the speed up with respect to the single site case for various configurations.

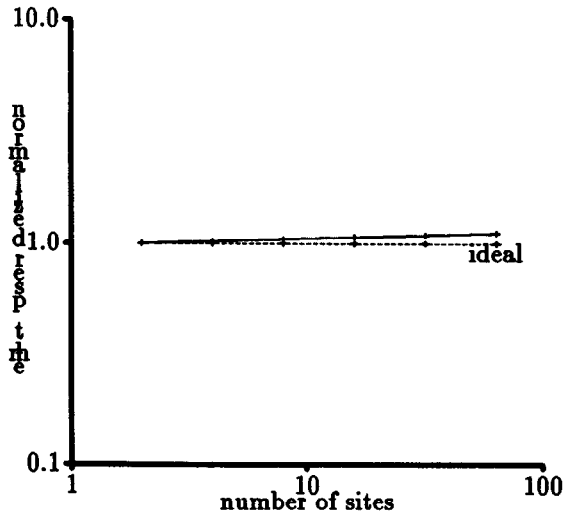

Figure 4: Horizontal Growth.

As the figure indicates, one pays for the distribution overhead when the number of sites goes from 1 to 2; afterwards the speed up is quasi-linear. This is true under the assumptions that the records are evenly distributed initially, and that the GDT ensures a pretty accurate partition. In Section 6, we will address the problem of uneven distribution of initial records.

We also studied horizontal growth by increasing the number of sites and the total number of records to be sorted, in the same proportion. That is, the number of records on each site remains unchanged. In particular, we sort 16K records per site. Since we are interested in multiple site cases, the response time in Fig. 4 is normalized to that of two sites. As expected, when the number of sites in the system increases, the response time increases slightly as well. (Remember that the number of key comparisons needed to produce a record in the merge phase varies with the logarithm of the number of sites.) For the chosen parameters, the response time is increased by about 2.4% when the number of sites doubles.

5.2 Comparison with Other Sort Algorithms

Let's use our model to compare the proposed fully parallel algorithm (FPA) with a variation of FPA (called FPA') without employing the "semi-sort" communication reduction technique and with the algorithms described in Section 2. The work load is to sort 1M 64 byte records using 64 4 MIPS processors. The communication CPU time is added wherever appropriate. The total CPU time is computed as the sum of all non-overlapped CPU times, and the total time is computed as the sum of the total CPU time and the I/O time. For all methods, the n-to-1 communication is 64 MB and is not discussed any further. In the table 5.2, two CPU times are shown for the return phases of both FPA and FPA': the smaller one is the CPU time on a backend processor while the larger one is that on the host processor. These two numbers are carried on in the subtotal and total. The real elapsed time should lie between the total CPU time and the total time.

The major reason for algorithm D performing much worse than FPA is that the sorted results are sent to the host in a serialized fashion. For algorithms B, C, and D, the scan time is not much smaller than the sort time because the particular database system used in the prototype has the property that reading a record for the internal sort is much cheaper (in terms of path length) than returning a record via the record-at-a-time interface. Algorithm C performs better than algorithm B because the merge fan-in of the former is much smaller.

Unless the n-to-n communication has a very high effective bandwidth, algorithms A, C, and D may experience a slow down because the n-to-n communication becomes a bottleneck when all sites exchange records among themselves. The bottleneck is caused by high aggregate communication, bursty communication or both, depending on the sort algorithm. For algorithm C, the amount of n-to-n communication is $64B * 1M * \log_2 (S)$, where S is the number of sites. For a system with 64 sites, the total n-to-n communication is 384 MB. For all algorithms A, B, and D, the amount of n-to-n communication is $64B * 1M = 64$ MB. On the other hand, the amount of n-to-n communication for FPA is $8B * 1M = 8$ MB. Comparing FPA with FPA', we demonstrate that the "semi-sort" technique not only reduces the n-to-n communication but also decreases the total CPU path length.

For algorithms A and D, the n-to-n communication happens in bursts, even though the total response time is much longer than that of FPA. The n-to-n communication in algorithms B and C can be spread over the entire merge phase. However, the entire record is sent $\log_2$ (S) (6 under the chosen parameters) times for algorithm C.

In fact, the sorting algorithm reported in [1] (on which algorithms C is based) is limited by the network data transfer. Also, algorithms B and C were compared in [10], which concluded that algorithm B performs better than algorithm C in the common-bus local network for the same reason.

6 Sensitivity to Data Skews

The FPA automatically balances the load for both the merge and return phases, independently of the initial distribution of data. The local sort phase, however, depends upon this initial distribution: if one site has

algorithm		FPA	FPA'	A	B	C	D
CPU (sec)	scan			4.8	4.8	4.8	4.8
	sort	5.1	5.1	436.0	5.1	5.1	6.0
	merge	9.5	10.5		370.3	173.7	
	return	0.4/7.0	0.4/7.0	22.9			22.9
subtotal		15.0/21.6	16.0/22.6	464	380	184	32.7
I/O (sec)	scan			0.7	0.7	0.7	0.7
	sort	1.8	1.8	160.1	1.8	1.8	1.8
	merge	0.8	0.8				
	return	0.7	0.7	45.1			45.1
subtotal		3.3	3.3	206	2.5	2.5	47.6
CPU+I/O (sec)		18.3/24.9	19.3/25.9	670	383	187	80
n-to-n comm (MB)		8	64	64	64	384	64

Table 1: Algorithm Comparisons

more data than another, the local sort phase will take more time. The total elapsed time is thus a function of the maximum local sort time.

We propose the following technique to reduce the effect of skew. Suppose each data site periodically informs the other data sites where it is in the scan (in particular, if it has finished or not). Then, an overloaded site can send pages to other sites that have finished. The overloaded site still needs to read the pages and send them on the communication network; the receiving sites need to receive and extract the data from the page. If records are short, then scanning a page is time consuming; if records are long the path length for communication becomes more important and this load balancing technique is not as efficient. Of course having several sites being able to access the same disk would help in this case.

In order to study the effect of an uneven distribution of the data, let us consider one site that has A records while all other sites have B records. Assuming A > B, we define the skew as Z= A/B. It is easy to calculate the response time degradation of the initial scan as a function of Z. The corresponding curve is plotted as a dotted line in Fig. 5. The figure also shows how the proposed technique may improve the situation for different record lengths. Note again that the degradation applies only the pre-scan phase; its effect should be amortized over the whole duration of the global sort.

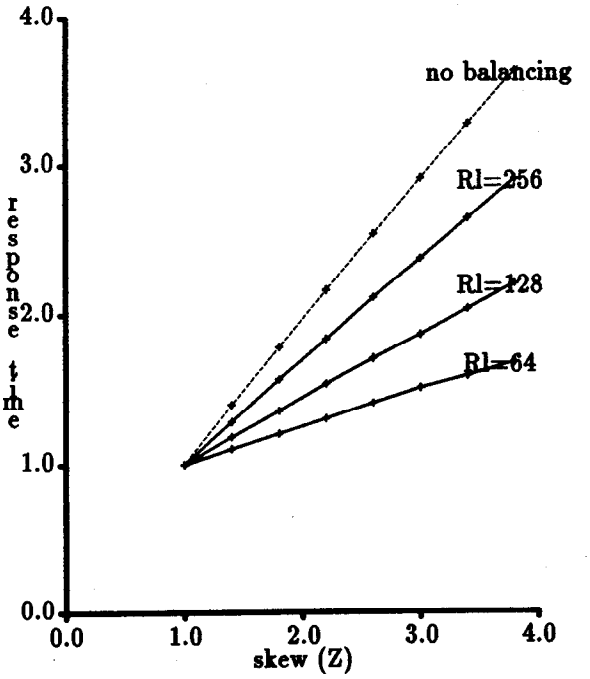

Figure 5: Effect of Redistributing Excess Records.

7 Conclusions

There is increased interest in using multiple nodes to improve the throughput and response time of database operations. Sort is just one of these operations; but it may be the most important one since it also plays a role in other operations such as duplicate elimination, grouping, joining, etc.

We have proposed an efficient parallel sort algorithm that assumes only a "shared nothing" architecture. In such an architecture, the nodes do not share main memory; nor do they share disks. Instead, they communicate among themselves, and with the host, by message passing.

The algorithm, evaluated by exercising a model validated by an actual prototype, exhibits the following nice properties:

1. All sites (except for very short intervals) are performed simultaneously, yielding high utilization of resources.
2. It scales very well. It decreases response time practically linearly when more processors are working on the same problem; it also supports horizontal growth, where 2n processors sort 2N records in roughly the same time that n processors sort N records. The primary reason for such good performance is that the sorted records are returned from all sites to the host site concurrently.
3. It reduces the amount of n-to-n communication and the path length to send/receive communication messages by using a "semi-sort" algorithm, where only keys have to be sent on the n-to-n communication network.

The efficiency of the algorithm is affected by the amount of skew in the data (the uneven distribution of data on the sites). Since skew only affects the initial scan of the data, and not the other phases of the sort, adverse effect is diluted. In any case, we discuss a load-balancing technique that can often be used to reduce the effect of skew on the initial scan itself. This technique is very efficient if the work needed to process a page is significantly larger than the work required to read and send/receive it. If this is not the case, the same technique can be used if several processors are connected to the same disk (which is generally needed in some small amount for availability, anyway); then the data can be read through the I/O channel with large prefetch rather than through communication.

Acknowledgements

We acknowledge the important contribution of Jean-Jacques Daudenarde, Gary Hallmark, and Jim Stamos to the development of the ARBRE prototype. We also thank Jim for useful discussions on the performance analysis, and for many valuable comments on the draft, which improved the presentation of this paper very substantially.

References

[1] M. Beck, D. Bitton, and W. K. Wilkinson. Sorting large files on a backend multiprocessor. *IEEE Transactions on Computers*, C-37(7):769–778, July 1988.

[2] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *ACM Computing Surveys*, 16(3):287–318, September 1984.

[3] D. Bitton-Friedland. *Design, Analysis and Implementation of Parallel External Sorting Algorithms.* PhD thesis, Computer Sciences Department, University of Wisconsin—Madison, January 1982.

[4] S. Even. Parallelism in tape-sorting. *Commun. ACM*, 17(4):202–204, April 1974.

[5] J. Gray, M. Stewart, A. Tsukerman, S. Uren, and B. Vaughan. FASTSORT: an external sort using parallel processing. *Tandem Systems Review*, 2(3):40–47, December 1986.

[6] IBM Corporation. *SQL/Data System General Information.* IBM Form No. GH24-5012.

[7] P. J. Janus and E. A. Lamagna. An adaptive method for unknown distributions in distributive partitioned sorting. *IEEE Transactions on Computers*, C-34(4):367–372, April 1985.

[8] S. Lakshmivarahan, S. K. Dhall, and L. L. Miller. Parallel sorting algorithms. *Advances in Computers*, 23:295–354, 1984.

[9] R. Lorie, J. Daudenarde, G. Hallmark, J. Stamos, and H. Young. Adding intra-transaction parallelism to an existing dbms: early experience. *IEEE Data Engineering Bulletin*, 12(1):2–8, March 1989.

[10] K. P. Mikkilinent and S. Y. W. Su. An evaluation of sorting algorithms for common-bus local networks. *Journal of Parallel and Distributed Computing*, 5(1):59–81, February 1988.

[11] P. M. Neches. The anatomy of a data base computer system. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 102–104, 1987.