Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting

by

David J. DeWitt
Jeffrey F. Naughton
Donovan F. Schneider

Computer Sciences Technical Report #1043
August 1991

Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting

David J. DeWitt* Jeffrey F. Naughton* Donovan A. Schneider†

Abstract

We consider the problem of external sorting in a shared-nothing multiprocessor. A critical step in the algorithms we consider is to determine the range of sort keys to be handled by each processor. We consider two techniques for determining these ranges of sort keys: *exact splitting*, using a parallel version of the algorithm proposed by Iyer, Ricard, and Varman; and *probabilistic splitting*, which uses sampling to estimate quantiles. We present analytic results showing that probabilistic splitting performs better than exact splitting. Finally, we present experimental results from an implementation of sorting via probabilistic splitting in the Gamma parallel database machine.

1 Introduction

In this paper we consider the problem of external sorting in a shared-nothing parallel database system. "Shared-nothing" means that the database system is implemented on top of a multiprocessor in which each processor has its own local memory and disk, and all communication between processors must take place through an interconnection network. The specific sorting problem we address is "multiple-input multiple-output" sorting. In this sorting problem, initially the data in the file to be sorted is on disk, distributed throughout the multiprocessor, and unsorted. At the termination of the sorting algorithm, the file must again be on disk, but partitioned into approximately equal sized non-overlapping sorted runs, one at each processor.

At the top level, an algorithm for this problem is as follows:

1. Determine a "splitting vector" $v[i]$, where $1 \leq i < k$, such that in the final sorted order, all records on processor p_1 have sort key value less than $v[1]$, all records on processor p_2 have sort key value greater than $v[1]$ but less than $v[2]$, and so on, until all records on processor p_k have sort key value greater than $v[k-1]$.

2. Based upon this splitting vector, redistribute the records in the file so that each record is at the appropriate processor.

3. After this redistribution, locally sort the records on each processor to produce the final result.

*Computer Sciences Department, University of Wisconsin at Madison
†Hewlett Packard Labs, Palo Alto, CA

The performance of the sorting algorithm is critically dependent upon Step 1. In this paper we consider two fundamentally different approaches to determining the splitting vector. The first approach we consider is due to Iyer et al. (An earlier paper due to Yamane and Take [YT88] proposes a similar algorithm with the same asymptotic running time as the algorithm of Iyer et al. However, since the description in Yamane and Take is not sufficiently detailed to estimate anything other than rough asymptotic running times, we chose the algorithm due to Iyer et al. for this study.) We call their method "exact splitting", since it deterministically computes an exact splitting vector, that is, one such that each processor is allocated the same number of records (plus or minus one). The second approach we consider, which we call "probabilistic splitting," computes an approximate splitting vector by sampling the unsorted file. As we discuss below, probabilistic splitting and similar techniques have apparently been discovered and rediscovered many times in the sorting literature, but to our knowledge have never been carefully studied in the context of external parallel sorting.

We show, using an analytical model, that on shared-nothing parallel systems, probabilistic splitting beats exact splitting. Furthermore, we present scaleup, speedup, and sizeup experimental data from an implementation of sorting using probabilistic splitting on the Gamma database machine.

In related work on parallel external sorting, Salzberg et al. [STG+90] present an algorithm for single-input single-output sorting. They demonstrate that by using a carefully engineered parallel version of sort-merge, the internal sort and merge phases can be made fast enough that the single input and the single output sites become the bottleneck. A parallel sort-merge has the benefit that it is not necessary to compute a splitting vector; however, sort-merge is probably not appropriate in a multiple-input multiple-output sort, since the final merge cannot be accomplished faster than a sequential pass through the entire file. Beck et al. [BBW88] describes a similar sort. Other earlier algorithms that also used a final sequential merge phase include Bitton et al. [BBDW83], and Valduriez and Gardarin [VG84].

Graefe [Gra90] presents a thorough description of the implementation and evaluation of a multiple-input, multiple-output sorting algorithm for a shared-memory multiprocessor. This algorithm uses a splitting vector. Graefe suggests that this splitting vector might be computable by sampling, but he does not investigate this alternative. His experimental results assume that the splitting vector is known exactly before the algorithm is run.

There is a huge body of literature dealing with parallel internal sorting in abstract computational models like the PRAM. This work is not directly relevant to our work, because of 1) the unrealistic abstract computational model, and 2) the unrealistic assumption that the number of processors is proportional to the number of records to be sorted.

As noted previously, the idea of sorting using an approximate splitting vector obtained by sampling has been invented and re-invented many times in the sorting literature.

Frazer and McKellar [FM70] proposed a uniprocessor internal sort based upon sampling. They showed that asymptotically, given some restrictions on the distribution of the data being sorted, their algorithm runs in linear time.

Distributive Sorting, a recursive bucket sort based upon sampling, has some similarities with sorting via probabilistic splitting as well. Dobosiewicz [Dob78] originally proposed the algorithm and proved that if the data is uniformly distributed, it runs in time $O(n)$. Later,

Janus and Lamagna [JL85] proposed a fix to distributive sorting that attempts to provide better performance for non-uniform data distributions. They use sampling to compute an approximate cumulative distribution function for the data, and experimentally evaluated the algorithm's performance using 31 partitions. The results showed that if the distribution was smooth, the algorithm performed well. They did not investigate a parallel implementation of the algorithm.

Quinn [Qui88] has suggested implementing a parallel quicksort as follows: to sort on k processors, choose $k-1$ pivot elements at random, and run the recursive quicksorts in parallel. His experiments showed zero speedup beyond 10 processors, because of the skew in the sizes of the arguments to the recursive quicksorts.

Huang and Chow [HC83] consider parallel external sorting using approximate partitioning based upon sampling. In their algorithm, the samples are used for two purposes: first, to compute the approximate splitting vector; second, within each local quicksort after partitioning the samples are used as pivot elements. Using combinatorial analysis they present analytic formulas showing good asymptotic performance, but did not implement their approach or compare it to any other.

Baugstø and Greipsland [BG89] also consider parallel external sorting using approximate partitioning based upon synchronized sampling of sorted subfiles. In more detail, in their algorithm, they implement sampling as follows: first, each processor sorts its local file fragment. Next, each processor samples some fraction of the fragment by looking at every kth record for some predetermined k. The resulting set of samples is then sorted and a partitioning vector is formed. Baugstø and Greipsland do not discuss how to analyze the quality of the resulting partitioning vector or how to determine what fraction of the file should be sampled.

Lorie and Young [LY89] describe an algorithm for multiple-input, single-output sorting. A novel feature of their algorithm is that the final output phase is heavily overlapped with the internal sorting phase, which reduces the time from the initiation of the algorithm until the first sorted tuple appears at the output. Their algorithm uses a splitting vector, which they suggest might be computable by some statistical technique. In an example they appear to be using a technique virtually identical to that of Baugstø and Greipsland, but the technique is not further described. In the analytic data presented, Lorie and Young assume that the splitting vector is known exactly before the algorithm begins.

Most recently, in work done concurrently with and independently from the work presented in this paper, Blelloch et al. [BLM+91] investigated internal sorting algorithms for a 2048 processor CM-2. They concluded (through experimental and analytic comparisons) that probabilistic splitting followed by local radix sorts provided the highest performance, beating a parallel bitonic sort and a parallel radix sort.

The remainder of this paper is organized as follows. Sections 2 and 3 describes the probabilistic and exact partitioning algorithms, and gives a cost model for each. Section 4 uses these cost models to compare the performance of the two. Section 5 describes our implementation of probabilistic splitting in the Gamma parallel database machine, and Section 6 presents the results of experiments on that implementation.

2 Probabilistic Splitting

Algorithm Description

Probabilistic splitting is based upon a simple idea: to find an approximate splitting vector for the file, compute a random subset of the file; find an exact splitting vector for the random subset; finally, use this exact splitting vector for the subset as an approximate splitting vector for the whole file. While this idea is simple, some care must be taken in evaluating the quality of this approximate splitting vector. Also, there is an interesting tradeoff between the added cost of taking a larger sample, which produces a better approximation, and the improved performance that results from the better approximation.

An error in the approximate splitting vector translates directly to skew in the number of records sent to the various processors in the parallel database system. In the following we derive the number of samples necessary to guarantee a given skew for a given number of processors.

Let s, for skew, denote the ratio of the maximum number of records at any site to the average number of records at a site. If we have k processors, and N records in the file to be sorted, then the average number of records per processor is N/k. Then by definition of s, the maximum number of records at any site is sN/k. By Theorem 7.1 from Seshadri and Naughton [SN91], if we take a total of kn samples, then the probability p that any processor contains more than sN/k keys is at most

$$p = ke^{-(1-1/s)^2 sn/2}$$

Solving this for n gives

$$n = \frac{2\ln(k/p)}{(1-1/s)^2 s} \tag{1}$$

as the number of samples required to guarantee a skew of at most s with probability $1-p$.

Trading Samples for Skew

To complete the description of probabilistic splitting, we need to describe how to find the optimal value of the skew s. Setting s too small will result in poor performance due to the cost of too many samples in the sampling phase; setting s too large will result in poor performance due to the uneven sizes of the subproblems sent to the processors in later phases of the algorithm. Since the function relating the skew and the number of samples depends upon the number of processors k (Equation 1), for each value of k there will be a different optimal s. Section 6 includes experiments illustrating this tradeoff in our implementation of probabilistic splitting.

Finding this optimal s is an optimization problem, and to set up and solve this optimization problem, we need cost estimates for the various phases of the algorithm. The cost model developed below is extremely simple, and as such should not be interpreted as an attempt to predict absolute numbers for running times for the algorithm. Our intent is that although the model is not sufficiently precise to yield accurate absolute running times, it will be good enough to predict which choice among a set of alternatives can be expected to give the best performance. This is analogous to the situation with query optimizers, where optimizers are

often very successful in choosing a good evaluation plan even though their cost estimates are not accurate in an absolute sense.

The main simplifying assumptions we make are that there is no overlap between I/O operations, CPU operations, or network traffic, that there is no contention in the network, and that the work of the individual nodes of the machine (including I/O, CPU, and network) is done completely in parallel with the other nodes of the machine. These assumptions allow us to consider that the time for a single node to perform its segment of the computation is the total time for the multiprocessor to perform the full computation.

COMP	0.012	ms. to compare keys
KS	0.036	ms. to exchange two keys
MOVE	0.053	ms. to move a record
IO_S	44.000	ms. to do a sequential IO
IO_R	50.000	ms. to do a random IO
MSG	1.000	ms. to send and receive a message
SAMPLE	120.000	ms. to take a random sample

Table 1: Parameters for the cost model.

We used the set of parameters in Table 1 in the cost model. These parameters are based upon measurements from the Gamma parallel database machine. Many of these times are higher than we had guessed before measuring them in the system. The COMP time is high because for comparisons we used the general purpose Gamma `compare_key()` routine, which works for keys of arbitrary type at arbitrary positions within a tuple. The MOVE times are actually measurements for moving a tuple from one slotted page to another slotted page, which incurs additional overhead when compared to a simple Unix `bcopy()`. The I/O times were for 32K byte pages. Finally, in general each sample involved two random IOs: one for the index page to locate the data page for the randomly selected tuple, and another for the data page itself.

Furthermore, we use the following notation:

B	number of records per page
P	number of bytes per page (memory and disk)
M	number of pages of available main memory
N	number of records in the file
k	number of processors

The most interesting case to consider here is the case where the multiprocessor can sort the file in two passes. This requires only that each processor's buffer space exceeds the square root of the number of file pages it is allocated, which allows very large files. For example, if each processor has 100 pages available for sorting, then each processor can sort a 10000 page subfile. With 32 processors, and 8K byte pages, the multiprocessor can sort a two gigabyte file. If the number of memory pages per processor is increased to 1000 (which is just 8 megabytes), the 32-node multiprocessor can sort a 200 gigabyte file in just two passes. For more numbers about just how large a file can be sorted in two passes, see the table on page 95 in [STG+90].

In this case (the two-pass case), at the top level the algorithm works as follows:

1. In parallel, the processors sample their fragment of the disk-resident file.
2. In parallel, the processors sort their samples, then send the samples to a single processor, where the sorted sets of samples are merged. Then an approximate splitting vector is computed from the total sorted set of samples, and this splitting vector is broadcasted to the other processors.
3. In parallel, the processors read their fragment of the file, and using the splitting vector, redistribute the records to the appropriate processor.
4. When a processor's memory has filled with incoming records, the processor sorts these records, writes a sorted run to disk, then continues reading incoming records.
5. In parallel, the processors merge the sorted runs from the disk and back onto the disk.

We now consider the cost for each step in turn. For concreteness, we will set

$$n = \frac{2\ln(k/0.99)}{(1-1/s)^2 s}$$

that is, n is the expected number of samples taken per processor to guarantee a skew of s with 99% certainty. Then

1. Sampling takes time $n * \text{SAMPLE}$.
2. Sorting each local set of sampled keys, assuming that we use an in-memory heapsort to do so, takes

   $$n * \log_2(n) * (\text{KS} + \text{COMP})$$

   CPU time. Next, these local sets of sampled keys must be gathered at a single node and sorted there. The simplest way to do this is to have each processor send a message to the designated processor, then to have this processor do a total sort. A more efficient way is to embed a binary tree in the hypercube. Then initially the leaves send their sorted samples to their parents, where the two sets of samples are merged. Next, the parents send these sets of samples to their parents, and so on, until the root of the tree has the complete sorted set of samples. The total time from the initiation of the sort at the leaves until the completion of the final merge at the root of the tree is given by $n\log_2(n) * (\text{KS} + \text{COMP})$ to sort the initial sets of samples at each processor, plus $2n(k-1) * (\text{COMP} + \text{MOVE})$ to do the merges up the tree. The time for the merges is computed as follows. The parents of the leaves merge two files of size n, at a cost of $2n * (\text{COMP} + \text{MOVE})$; the grandparents of the leaves merge two files of size $2n$, and so on, until the root merges two files of size $2^{\log_2(k)-1}n$. The sum of these quantities is $2n(k-1) * (\text{COMP} + \text{MOVE})$. Also, the network cost to gather the samples is $\log_2(k) * \text{MSG}$ seconds, and the network cost to broadcast the resulting splitting vector to all k processors will take another $\log_2(k) * \text{MSG}$ seconds.

3. While actually partitioning the source file, we will assume that, for each record r, a processor does a binary search of the splitting vector to determine to which processor r belongs, then copies r to a buffer page for tuples bound for that processor. The cost for this step is

$$\frac{N}{kB} * \mathrm{IO_S} + (N/k) * (\log_2(k-1) * \mathrm{COMP} + \mathrm{MOVE})$$

Finally, to ship the data will take time $\frac{N}{kB} * \mathrm{MSG}$.

4. For the sort of incoming records, note that each processor can hold MB records in its memory, and it must sort this many records $s\frac{N}{kBM}$ times, where the extra factor of s comes from the skew. Assuming that the sort is done by heapsorting (key,ptr) pairs, then copying records to their final position, the cost is

$$s\frac{N}{kBM}((MB) * (\log_2(MB)) * (\mathrm{KS} + \mathrm{COMP}))$$

to sort the keys and

$$s\frac{N}{kBM}((MB) * \mathrm{MOVE} + M * \mathrm{IO_S})$$

to move the records to the output buffers and write them to disk.

5. In the final merge, the processor with the heaviest load will have $s\frac{N}{kBM}$ runs of MB records each, so the time spent will be

$$s\frac{N}{kBM}M * \mathrm{IO_R}$$

to read in the runs and

$$s\frac{N}{kBM}((MB) * \log_2(s\frac{N}{kMB})(\mathrm{KS} + \mathrm{COMP}))$$

to do the compares during the merge and finally

$$s\frac{N}{kBM}((MB) * \mathrm{MOVE} + M * \mathrm{IO_S})$$

to copy the records to the output pages and then write them to disk.

Gathering together all the terms that depend on the skew s, we get an equation for the cost as a function of the skew. This expression involves terms in s, $1/s$, $1/s^2$, and $\log_2(s)$. To determine an optimal skew (and hence an optimal number of samples) we could use a numeric method to find roots of the derivative of this equation; however, it is simpler to just compute $c(s)$ for a reasonable range of skews (say $s = 1.0$ to 2.0 by increments of 0.01) and return the value of s that gave the smallest $c(s)$. This value of s can then be used to determine the appropriate number of samples to take.

We implemented this approach of searching for the best value of s, and found that over a wide range of numbers of processors and a wide range of problem sizes, taking 100 samples per processor was close to optimal. This result is supported by our experiments with the implementation, reported in Section 6. Note that by Equation 1, by keeping the number of samples per processor constant, we are letting the skew grow (slowly) as processors are added.

This makes intuitive sense, since as the number of processors grows while keeping the size of the source file constant, the amount of work exclusive of sampling that must be done by each processor decreases. Even keeping the number of samples constant implies that sampling takes a larger and larger fraction of the execution time as the number of processors increases; if we increase the number of samples per processor as the number of processors grow, the increase in the fraction of the total running time due to sampling quickly swamps any gains obtained by reducing the skew.

Although our goal was not to predict absolute performance numbers, a comparison of the predicted numbers and the measured numbers in our implementation show that the two are in fairly good agreement. Table 2 compares the predicted vs. measured numbers for varying numbers of processors, in each case sorting one million 100 byte tuples and using 100 samples/processor.

num processors	predicted time (sec)	measured time (sec)	ratio
5	243	271	0.90
10	128	143	0.89
15	90	103	0.87
20	70	80	0.88
25	58	67	0.87
30	51	58	0.88

Table 2: Analytic model predictions vs. measured performance.

3 Exact Splitting

In this section we consider the algorithm proposed by Iyer et al. for finding an exact splitting vector. The complete algorithm is rather complex; see Iyer [IRV89] for details. We have adapted some of the steps for a shared-nothing multiprocessor, since the original algorithm was designed with shared memory in mind. Our goal in this section is to present enough of the algorithm to give the intuition behind it and also to justify our model of its cost.

As in the probabilistic splitting section, assume that we have k processors and that the file has N records. The first step of the exact splitting algorithm is that each processor fully sorts its fragment of the file, producing k sorted runs, one per processor. Recall that our overall goal is to compute a $k-1$-element splitting vector that divides the entire file into k equally sized segments.

The algorithm can perhaps best be explained in terms of how it computes a single splitting element. Suppose that we wish to find the sort-key value that partitions the entire file into two segments, one containing the initial fN records of the entire sorted file, the other containing the last $(1-f)N$ records of the entire sorted file. Furthermore, assume that $f \leq 0.5$ (the case $f > 0.5$ is symmetric.) Then the algorithm proceeds as follows:

1. Each processor selects $(1-f)/f$ equally spaced elements from its sorted run. (Note that

these are not randomly chosen elements; they are chosen at equal intervals from the sorted run.)

2. Each of the processors sends its $(1-f)/f$ elements to a single processor, which merges the elements and then computes the exact f splitting value for these $k(1-f)/f$ elements.

3. This exact f splitting value is broadcasted to all k processors.

4. Each processor determines the pair of elements from the $(1-f)/f$ elements chosen in Step 1 that bracket the broadcasted splitting value.

5. The processors do a form of coordinated binary search on the records in the intervals between these "bracketing" elements to determine the exact splitting value for the whole file. In more detail, suppose that on the previous iteration, the splitting value was determined to be v, and that for $1 \leq i \leq k$, processor p_i has determined that records u_i and l_i in sorted order bracket this value v. On the current iteration, for $1 \leq i \leq k$, each processor reads element $(u_i + l_i)/2$ in the sorted order, and sends this element to one coordinating processor, which then determines a new splitting value v'. This process continues until $u_i = l_i$ for $1 \leq i \leq k$, at which point the current v is guaranteed to be an exact f splitting value for the entire file.

The last step of this description is actually greatly simplified — it is not guaranteed that at any time in the algorithm, the true final splitting value will indeed be bracketed by the current u_i and l_i. In general, the algorithm must search forward or backward outside of this interval. However, what we have given, the case in which the exact splitting value is always bracketed by u_i and l_i, is the best case in that it results in the fewest number of records read. (In terms of the description in Iyer et al. [IRV89], we are assuming the cases 2 and 3 on pages 138–139 never occur.)

Recall that this just describes how to find a single element of the splitting vector $v[i]$. To compute the full vector, the previous algorithm is repeated for $f = 1/k, 2/k, \ldots, (k-1)/k$. These k iterations make the work per processor in exact splitting proportional to the number of processors in the system.

We now develop a simple analytic model for the performance of a sorting algorithm that uses exact splitting. As was the case in the model for probabilistic splitting, our goal is not to predict absolute numbers. Rather, we want to make a relative comparison between probabilistic and exact splitting, and also to identify the key parameters that determine how the two algorithms relate.

Again, as in the case for probabilistic splitting, we assume that the file can be sorted in two passes. In this case, the algorithm at the top level looks like:

1. Each processor sorts its local segment of the file.

2. The processors cooperate to determine an exact splitting vector.

3. The processors redistribute the data and write sorted incoming runs to disk.

4. Each processor merges the runs from disk.

We now consider the cost of each step in turn.

1. The initial sort of the file takes IO time $\frac{N}{kB} * \mathrm{IO_S}$ to read the local segment of the file in order to form the initial sorted runs, and also $\frac{N}{kB} * \mathrm{IO_S}$ to write these runs to disk. The cpu to form these initial sorted runs is just $(MB) * (\log_2(MB)) * \mathrm{KS} + \mathrm{MOVE})$ per run, and there will be $\frac{N}{BkM}$ such runs. The IO to merge these runs is $\frac{N}{kB} * (\mathrm{IO_S} + \mathrm{IO_R})$ since the reads are sequential while the writes are random. The CPU for the merge will be $\frac{N}{k} * (\log_2(\frac{N}{BkM}) * \mathrm{KS} + \mathrm{MOVE})$.

2. Analyzing the cost of determining the splitting vector is again more complicated. We will divide the cost into two parts, one for the initialization, one for the iterations. Furthermore, recall that this process must determine $k-1$ splitting elements; we first consider the cost of finding the single splitting element at quantile $i/(k-1)$ (the whole set consists of the quantiles $i/(k-1)$ for $i = 1, \ldots, k-1$.)

   - **Basis:** Initially, we need to read $(1 - \frac{i}{k})/\frac{i}{k} = \frac{k}{i} - 1$ pages from disk. This will cost $(\frac{k}{i} - 1) * (\mathrm{IO_R})$. Next, each processor must send these initial keys to some processor to be sorted. As we did for probabilistic splitting, we assume that the keys are gathered by being passed up a binary tree of processors, sorting them as they go. The initial sort will take $(\frac{k}{i} - 1)\log(\frac{k}{i} - 1) * (\mathrm{KS} + \mathrm{COMP})$, while the merges up the tree will take a total of $2(\frac{k}{i} - 1)(k-1) * (\mathrm{KS} + \mathrm{COMP})$, while the network cost is $\log_2(k) * \mathrm{MSG}$ to gather the keys, and another $\log_2(k) * \mathrm{MSG}$ to distribute the splitting constant afterwards. The cost for the sum of the basis operations for all the quantiles will be the sum of the preceding quantities, from $i = 1$ to $k-1$.
   - **Iterations:** Now consider the cost of the iterations for the quantile i/k. First, as explained above, the number of iterations is just $\log_2(\frac{N}{k}/(\frac{k}{i} - 1))$. Each iteration consists of reading the "midpoint" between two consecutive elements (for a cost of $\mathrm{IO_R}$), sending it to a designated processor ($\log_2(k) * \mathrm{MSG}$, assuming we pass the elements up a binary tree of processors), then sorting these k elements ($k(\log_2(k) * (\mathrm{KS} + \mathrm{COMP}))$), and finally sending a message back to each processor ($log_2(k) * \mathrm{MSG}$). Again, the total cost is the sum of this quantity for $i = 1$ to $k-1$.

   This completes the cost for the quantile determination.

3. After the quantiles have been computed, the computation proceeds as in the case for probabilistic splitting. To redistribute the data, we again assume that, for each record r, a processor does a binary search of the splitting vector to determine on which processor r belongs, then copies r to a buffer page for tuples bound for that processor. The cpu cost for this step is $(N/k) * (\log_2(k-1) * \mathrm{COMP} + \mathrm{MOVE})$ while the IO cost is just $\frac{N}{kB} * \mathrm{IO_S}$. The network cost to redistribute the data is $\frac{N}{kB} * \mathrm{MSG}$. Next, each processor must write the incoming tuples to disk, which will cost $\frac{N}{kB} * \mathrm{IO_S}$.

4. Finally, the processor must merge these runs, which will entail IO cost $\frac{N}{kB} * \mathrm{IO_S} + \frac{N}{kB} * \mathrm{IO_R}$ and cpu cost $(N/k) * (\log_2(\frac{N}{kB}/M) * \mathrm{KS} + \mathrm{MOVE})$. Note that there is no skew here.

4 Analytic Results and Comparison

It is perhaps most clear to consider probabilistic splitting and exact splitting in terms of what they both add over an ideal (and usually unrealistic) situation where a perfect splitting vector is known a priori at zero cost. Probabilistic splitting adds the explicit cost of sampling and the implicit cost due to skew in the later phases of the algorithm. Exact splitting adds the explicit costs of each processor sorting its original segment of the file and of the iterative algorithm to compute the exact splitting vector. In broad terms, our analytic model predicts that in a shared-nothing multiprocessor, it is better to sample and tolerate some skew than to compute an exact splitting vector. Furthermore, the relative difference in the performance of the two algorithms increases as the number of processors applied to the sort grows.

Figure 1 shows the performance predicted for the two algorithms on sorting one million 100 byte tuples. Each processor was allocated 25 buffer pages (32K bytes each) for a total of 800K bytes. This sounds like a small number of pages, and it is, but as we discuss in Section 5, we found that as long as there was enough memory to sort the file in two passes, the performance of the sorting algorithm was relatively insensitive to the amount of memory allocated. At 25 pages, and 320 tuples per page, each processor can sort $25 * 25 * 320 = 200000$ tuples in two passes. Even with just five processors (the smallest number of processors in the plot in Figure 1), the multiprocessor can sort up to a one million tuple relation in two passes.

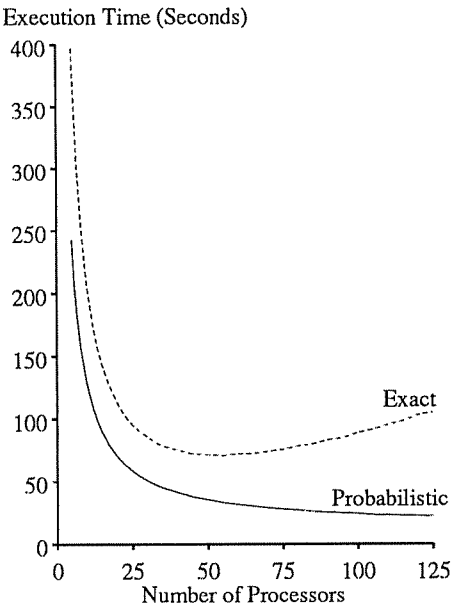

Figure 1: Analytic model predictions for sorting 1M tuples.

Over the complete range of processors considered, probabilistic splitting is faster than exact splitting. For large numbers of processors, the execution time for exact partitioning actually begins to increase as more processors are added. This is due to the fact that if there are k processors in the system, the algorithm to find an individual partitioning element must be repeated $k-1$ times. As mentioned in the sections that developed these models, we regard the relative positions of the two curves and their shapes as more important than the absolute numbers. For

this reason, in the remainder of this section we will present graphs of the ratios of the execution times of these two algorithms instead of the execution times themselves. Specifically, we show the ratio of the time for exact splitting over the time for probabilistic splitting. Hence, the line $y = 2$ on the the graph corresponds to probabilistic splitting being twice as fast as exact splitting.

The next factor we investigated was the dependence of performance of the two algorithms on the record size. Sampling in general is more expensive as tuple sizes decrease, since each random sample must pull in a complete page. This means that as tuple sizes decrease, each random sample represents a larger percentage of the time to scan the whole file. Figure 2 shows the difference between the relative performance of the two algorithms in sorting one million 50 byte tuples, one million 100 byte tuples, and one million 200 byte tuples. The graph indicates that the relative performance of the two algorithms remains approximately constant for all three tuple sizes. The reason for this is that exact splitting is also adversely impacted by small record sizes, since in much of the binary search phase of the algorithm, each probe of the relation must pull in a complete page in order to examine a single tuple. The curve for 50 byte tuples stops at 60 processors because with one million 50 byte tuples, and 25 memory pages of 32K bytes each per processor, if there are more than 60 processors the entire relation fits into the aggregate memory of the multiprocessor. (When the entire relation fits in memory, our cost model, which assumed two-passes, becomes invalid.)

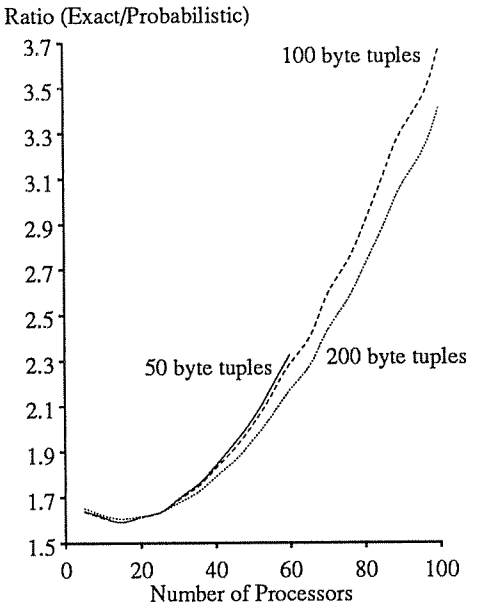

Figure 2: Analytic model predictions for sorting, 50, 100, and 200 byte tuples.

We also wanted to investigate the dependence of the two algorithms on problem size. Figure 3 shows the predicted relative performances of the algorithm on files of 1M, 10M, and 100M 100 byte tuples. The shape of these curves can be explained by noting that, roughly, the number of page reads per processor in the sampling portion of probabilistic splitting is 100 – 200 (the precise number depends upon the hit ratio for index pages in the buffer pool during the sampling.) The number of page reads per processor in the percentile determination phase of

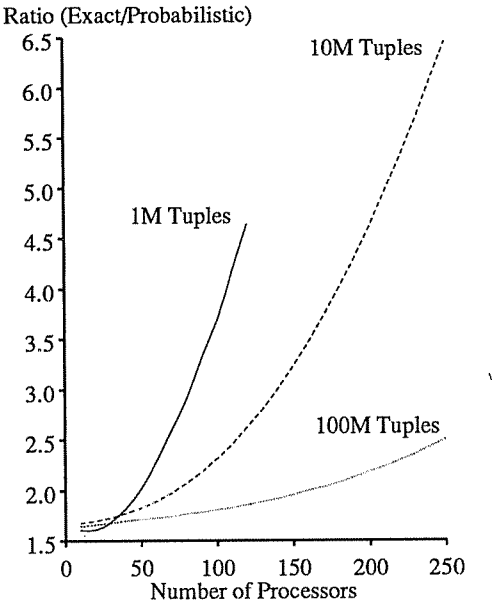

Figure 3: Analytic predictions for ratios of algorithms on 1M, 10M, and 100M tuple files.

exact splitting is $O(k^2)$, where k is the number of processors. As k grows, these k^2 I/O's become a significant factor in the running time of exact splitting, so the ratio of the running time of exact splitting to that of probabilistic splitting grows. However, as n grows, the fraction of the total running time represented by these k^2 I/Os is hidden by the running time for the sorting portion of the algorithm. In essence this means that if we fix n and let k grow, then the ratio of the running times should grow; if we compare curves for different values of n, say n_1 and n_2, with n_2 larger than n_1, this effect is delayed (it takes more processors before probabilistic dominates.) To produce the curves in the figure, we scaled the memory per processor with the problem size: 25 pages for 1M tuples, 66 pages for 10M tuples, and finally 180 pages for 100M tuples. Also, we started at 10 processors rather than 5. This ensured that at every data point, the sort could be completed in two passes.

Finally, we anticipated that as the size of the file grows, probabilistic splitting becomes more attractive in an absolute sense (not just in comparison with exact splitting.) Figure 4 shows the speedups for probabilistic partitioning for various file sizes. In the figure, the basis point is 10 processors, so speedups are ratios with respect to the 10 processor time. This implies that perfect speedup is 10, not 100. (We used 10 processors instead of 1 as the basis because with only a single processor, the files cannot be sorted in two passes.) Figure 4 implies that the larger the file size, the more processors probabilistic splitting can apply effectively to the sorting problem. Intuitively, the reason for this is that as we add more processors, the amount of sorting work per processor decreases, but the amount of sampling work per processor remains constant. For large file sizes, even at 100 processors the sampling time is a tiny fraction of the total execution time (e.g., 12 seconds out of 1400 seconds at 100M tuples.) However, for smaller files the sampling time at 100 processors becomes a significant portion fraction of the total execution time (e.g., 12 seconds out of 25 seconds for 1M tuples.)

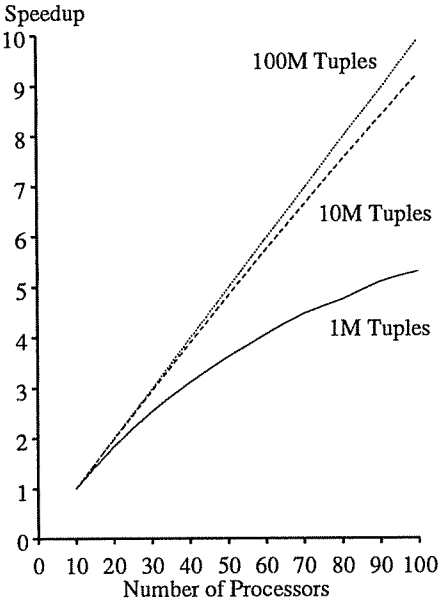

Figure 4: Analytic model predictions of speedup for 1M, 10M, and 100M tuple files.

5 Description of Implementation

In order to investigate the performance of our parallel sorting algorithm based on probabilistic splitting, we implemented the algorithm using the Gamma Database Machine [DGS+90] as our experimental vehicle. Gamma falls into the class of *shared-nothing* [Sto86] architectures. The hardware consists of a 32 processor Intel iPSC/2 hypercube. Each processor is configured with a 80386 CPU, 8 megabytes of memory, and a 330 megabyte MAXTOR 4380 (5 1/4 in.) disk drive. Each disk drive has an embedded SCSI controller which provides a 45 Kbyte RAM buffer that acts as a disk cache on sequential read operations. The nodes in the hypercube are interconnected to form a hypercube using custom VLSI routing modules. Each module supports eight full-duplex, serial, reliable communication channels operating at 2.8 megabytes/sec.

Gamma is built on top of an operating system designed specifically for supporting database management systems. NOSE provides multiple, lightweight processes with shared memory. A non-preemptive scheduling policy is used to help prevent convoys [BGMP79] from occurring. NOSE provides communications between NOSE processes using the reliable message passing hardware of the Intel iPSC/2 hypercube. File services in NOSE are based on the Wisconsin Storage System (WiSS) [CDKK85].

The services provided by WiSS include sequential files, byte-stream files as in UNIX, B^+ tree indices, long data items, an external sort utility, and a scan mechanism. A sequential file is a sequence of records that may vary in length (up to one page) and that may be inserted and deleted at arbitrary locations within a file. Optionally, each file may have one or more associated indices that map key values to the record identifiers of the records in the file that contain a matching value. One indexed attribute may be designated to be a clustering attribute for the file.

In Gamma, relations are *horizontally partitioned* [RE78] (also known as declustering [LKB87])

across all disk drives in order to increase the aggregate I/O bandwidth provided by the hardware. The query language of Gamma provides the user with several alternative declustering methods. For the experiments described below, the user determined which tuples reside on each site based on a range predicate applied to the partitioning attribute of each tuple of the relation. A collection of tuples stored on a processor is referred to as a *fragment* of the relation.

Our parallel sorting algorithm based on probabilistic splitting operates as follows. First, each processor randomly samples its local fragment of the relation to be sorted and sends the sort attribute values of the sampled tuples to a central coordinator. The coordinator sorts all the sampled values and determines the partitioning elements such that the relation will be divided into as many partitions as there are processors. The coordinator then sends a vector of these partitioning elements to each processor whose disk contains a fragment of the relation.

During the second phase of the algorithm the relation is redistributed according to the elements of the splitting vector. This phase begins by initiating scan, sort, and I/O processes on each processor. After being initiated, each scan process reads its local fragment of the relation and re-distributes it over the network using the elements of the partitioning vector (in Gamma terminology, a split table) to determine to which processor each tuple should be routed to. Tuples are redistributed using 8K byte network packets.

As each packet of tuples arrives at a processor, the sort process adds the packet to a memory resident sort buffer (without incurring a copy). Each time this sort buffer fills, an array of pointers to the tuples is sorted using the WiSS quicksort algorithm. Finally, the sorted run is given to the I/O process, which materializes the sorted run (by copying each tuple from its network packet to an output page) and writes it to disk.

In the final phase of the algorithm, each of the sort processes merges its sorted runs into a single run, which is then written back to disk. As currently implemented, the algorithm terminates with the sorted relation partitioned across all the disks in the system. That is, in a configuration with k processors, the first $1/k$th (approximately) of the relation is stored on the disk attached to processor 1, the next $1/k$th on processor 2, etc. Modifying the implementation so that the entire relation is materialized in sorted order on the host processor or in an application program would be straightforward.

In order to minimize the number of random seeks performed, pages of sorted runs are read and written in blocks composed of two 32K byte pages. Such random seeks occur in both phases of the algorithm. During the first phase of the algorithm, while the scan process at a node is reading blocks of the unsorted relation for redistribution, the I/O process on the same node is concurrently writing sorted runs to disk. Since these two processes read/write from/to different regions of the disk, each I/O operation involves a random seek. The same problem occurs during the merge phase of the sort as each page that is read from an input run or written to an output run almost always incurs a random seek.

While performing I/O operations in blocks of pages does provide some improvement, since the I/O system of the iPSC-2 hypercube does not support IBM-style channel programs, the improvement is not dramatic. In particular, once a process has issued an I/O request, the operating system determines whether another process is runnable. If so, a context switch is performed. If the process selected to run next immediately performs an I/O operation then by the time the first process is allowed to run again, the disk heads will have been moved, negating

any benefits provided by blocking I/O operations.

Our initial implementation used 8K byte pages. Since this configuration did not provide "adequate" performance, we added the multi-block I/O feature described above. This change did not, however, improve performance as much as we had expected. For example, doing I/O four pages at a time improved performance by only 5-10% over a single page. On the other hand, switching to a single 32K byte page, improved performance by almost 30%. In general (for database system operations other than sorting), using such large pages can actually decrease performance; for example, with index operations. Instead of long pages, a much better long-term solution would be to modify Gamma's operating system to accept vector's of I/O operations that would always be executed in order.

One interesting thing that we observed while running some preliminary experiments was that using the maximum buffer space available did not generally minimize the execution time for a sort. When memory was a scarce resource, obtaining long runs was important in order to minimize the number of passes through the file. However, as long as the amount of memory available is between the square root of the data file size and the size of the entire data file, the file can always be sorted into two passes. Since the CPU cost of the sort is independent of the size of each run, producing longer runs tends to reduce the I/O cost while making the sort CPU bound. (Graefe made a similar observation in [Gra90].) Since all our tests required two passes through the file, through experimentation we found that using a buffer size of about 100 pages balanced the CPU and I/O time and tended to minimize the overall execution time of the sort operation. We also implemented a second version of the algorithm in which the sorted runs were produced using a tournament sort. Despite the fact that the runs produced were twice as long as the ones produced using quicksort, this version actually ran slower.

One problem to overcome with this parallel algorithm is how to correctly and efficiently sample the relation in parallel in order to determine the partitioning elements. For correctness, the relation must be sampled as if it were stored on a single processor. That is, each tuple, regardless of the processor that it is stored on, must be equally likely to be sampled. (In statistical terms, this is a "simple random sample.") Note that if we wish to take n samples, it is not acceptable to have each processor take $1/n$ samples, since this will not result in a truly random sample. To see this, note that if each processor takes $1/n$ samples, then we will never get a set of n samples in which more than $1/n$ tuples come from any single processor's portion of the database.

To take a truly random sample while still making use of the parallelism available, in our implementation each processor attempts to sample n tuples from its local fragment of the relation (each processor uses the same random number generator with the same seed). However, for efficiency, each processor checks the local catalog information to determine if the tuple to sample is indeed stored on its local disk. If so, the tuple is retrieved from disk and its sort attribute value is sampled. If the tuple is not stored locally, the sample can be ignored. In terms of disk I/O, the effect is the same as if some central processor generated n random keys, then sent to each processor p only the keys that for tuples in the partition stored at p. A B-tree index is used to efficiently retrieve the tuple to sample.

Note that this optimization does not require that the sort attribute of the relation be identical to the attribute used to partition the relation during relation creation. Instead, it

only requires that the attribute used to fetch a random tuple is the same attribute used to partition the relation during relation creation. If no such attribute is available, the algorithm still works correctly, it will only suffer a performance degradation due to unsuccessful searches of the index for tuples stored on other processors.

6 Results

Scaleup and speedup are useful metrics for evaluating the performance of a parallel algorithm on a multiprocessor database machine [DG90]. Scaleup is an interesting metric for multiprocessor database machines as it indicates whether a constant response time can be maintained as the workload is increased by adding a proportional number of processors and disks. Speedup is an interesting metric because it indicates whether additional processors and disks result in a corresponding decrease in the response time of a query. A similar set of experiments were reported in [EGKS89] for equi-join queries on Release 2 of Tandem's NonStop SQL system, in [DGS+90] for equi-join queries in Gamma, and in [DNS91] for non-equijoin queries in Gamma. Scaleup and speedup results for several parallel sorting algorithms are contained in [Gra90] and in [STG+90]. A third interesting metric for algorithm performance is what we will call "sizeup," in which the number of processors is held constant and the size of the problem instance is varied. Sizeup tests indicate the growth rate of the execution time as a function of the problem size. Salzberg et al. also reported sizeup numbers for sorting in [STG+90].

For every configuration for which results are reported in this section, each relation to be sorted was evenly distributed during relation creation amongst all the processors by applying a range predicate to the `unique1` attribute (whose values range from 0 to the relation cardinality minus 1). The sort was performed on the unique2 attribute of each relation. The values of this attribute also range from 0 to the relational cardinality minus 1. However, since the values of the `unique1` and `unique2` attributes for a tuple are not correlated with one another, sorting a relation on its `unique2` attribute on k processors results in $(k-1)/k$th of the tuples in the relation being redistributed during the repartitioning phase of the algorithm.

Randomness and Sample Size

We will return to scaleup, speedup, and sizeup later in this section; initially, we present results illustrating the dependence of the execution time on the sample size. Briefly, up to a certain point, taking more samples significantly reduces skew and hence the total execution time; however, beyond a certain point, the time spent in the sampling itself outweighs the savings due to reducing the skew. Another aspect of the dependence between execution time and the number of samples taken has to do with the probabilistic nature of probabilistic splitting. If we take two different sets of samples of the same size, the quality of the splitting vector generated from the two samples will in general not be the same, since some random samples are more representative of the actual population than others. This means that if we run the sorting algorithm using two different random samples of the same size, the running time of the sort using the two sets of samples will also differ. The more samples taken, the lower the variance in the running time between different sets of samples.

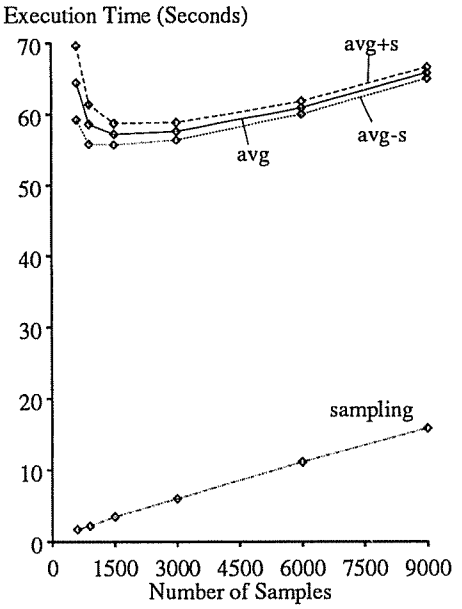

Figure 5: Sorting time as a function of sample size.

Figure 5 and 6 illustrate these two aspects of the relationship between the number of samples and the execution time. The two graphs both represent the same set of experiments — a sort of one million 100 byte tuples on 30 processors using varying sample sizes. For each sample size, we ran 30 trials, initializing the random number generator with a different seed each time.

In the graph in Figure 5 there are four lines. For a point (x, y) on the lowest line, labeled "sampling," y is the average elapsed time (over thirty trials) to take x samples. For a point (x, y) on the middle of the upper three lines, y is the average total execution time for the sort (over the thirty trials) using x samples. For a point (x, y) on the uppermost of the upper three lines, y is the average total execution time for the sort plus one standard deviation, using x samples, where the standard deviation is computed over the same thirty trials. Similarly, for a point (x, y) on the lowest of the upper three lines, y is the average total execution time for the sort minus one standard deviation, using x samples, where the standard deviation is computed over the same thirty trials.

The graph in Figure 6 expands the scale for the top three lines of the graph in Figure 5. Also, two additional lines have been added, one showing the maximum execution time observed over all 30 trials at a given number of samples, the other showing the minimum.

The final point we wish to emphasize is that the performance of probabilistic splitting does not depend on the distribution of the data in the sort attribute. To demonstrate this experimentally, we generated an instance of the relation to be sorted in which the sort attribute was drawn from a highly skewed normal distribution (one million values drawn from a normal distribution with mean 500,000 and variance 10,000.) Table 3 shows the performance of the sorting algorithm for both the skewed and the uniformly distributed sort attribute. Both lines of that table refer to sorting one million 100 byte tuples using 30 processors and 3000 samples; the average, min, max, and variance reported are over 30 trials, each trial with a different random seed.

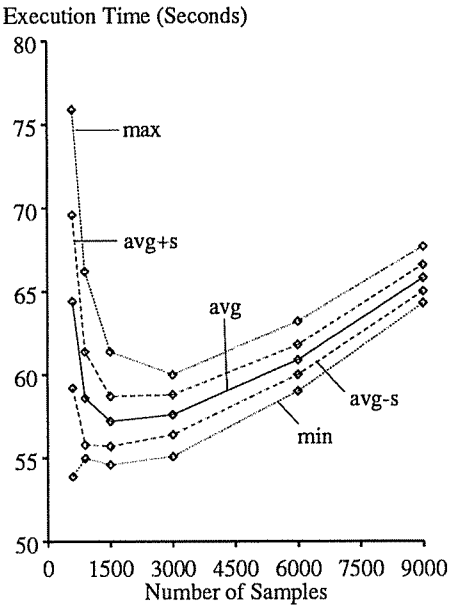

Figure 6: The variance of sorting time as a function of sample size.

distribution	average time (sec)	max (sec)	min (sec)	std. deviation (sec)
uniform	57.6	60.6	54.3	1.34
skewed	57.0	60.0	55.1	1.22

Table 3: Performance of sorting on skew vs. uniform distribution.

Scaleup

For the scaleup experiments, we varied the number of processors with disks from 5 to 30, in steps of 5, sorting an average of 160K, 100 byte tuples per processor. Thus, with 5 processors, a 0.8 million, 100 byte tuple relation was sorted. At 30 processors, the size of the relation sorted was 4.8 million tuples. As described in Section 2, in our implementation for k processors we took a total of $100k$ samples. This means that with 5 processors, we took 500 samples; with 30 processors, 3000. All timings presented are averages over 5 trials, where each trial used a different random seed.

Table 4 presents the scaleup results we obtained. Ideally the algorithm should exhibit a constant response time as the relation size and hardware configuration is scaled. Three factors contribute to the slight increase in response times. First, as additional processors are added, the skew in the sizes of the portions of the file allocated to each processor grows. Second, the task of initiating four processes at each site (a sampling operator, a relation scan, a sort operator, and a store operator) is performed by a single processor. Finally, as the number of processors increase, the effects of short-circuiting [DGS+90] messages during the execution of the query diminishes. For example, in the 5 processor configuration, approximately 1/5th of the tuples of the input relation end up being sent to the sort process on the same processor, thereby short-circuiting the communications network. As the number of processors is increased, the number of these short-circuited packets decreases to the point where, with 30 processors,

only 1/30th of the packets will be short-circuited. Because these intra-node packets are less expensive than their corresponding inter-node packets, smaller configurations will benefit more from short-circuiting.

no. of proc.	relation size (tuples)	execution time (sec)	actual/ideal
5	0.8M	217	1.00
10	1.6M	227	1.05
15	2.4M	235	1.08
20	3.2M	244	1.12
25	4.0M	249	1.15
30	4.8M	257	1.18

Table 4: Scaleup results.

Speedup

For the speedup experiments, we fixed the size of the relation being sorted at one million tuples while varying the number of processors from 5 to 30. We again held constant (at 100 samples per processor) the number of samples used to determine the partitioning elements. The `unique2` attribute was again used as the sort attribute. Again, all timings presented are averages over 5 trials, where each trial used a different random seed.

no. of processors	execution time (sec)	speedup	% parallel efficiency
5	270.7	1.0	100
10	143.5	1.9	95
15	103.2	2.6	87
20	80.4	3.4	85
25	67.2	4.0	80
30	58.3	4.6	77

Table 5: Speedup results.

The response time and speedup for the one million tuple relation sort are shown in Table 5. It is obvious that adding additional processors significantly reduces the execution time of the query. Several factors prevent the system from achieving perfectly linear speedups. (It is important to note that since the base case was 5 processors a perfect speedup factor for 30 processors would be 6.0 and not 30.0!) As was the case in the scaleup experiments, performance is limited by the overhead of scheduling the operators of the query tree, the effects of short-circuiting, and the effects of skew in the size of the sort tasks allocated to each processor.

To demonstrate the effect of errors in the approximate splitting vector, we measured the number of tuples distributed to each node. We then took the maximum of these values and measured how far it differed from the optimal value (assuming a perfectly uniform distribution). In the 5 processor configuration, the maximum skew was approximately 5%. In the 30 processor

configuration, though, the maximum skew was found to be 18% above optimal. Since in a multiprocessor, performance is limited by the slowest site, the increase in skew as processors are added results in sublinear speedups.

Sizeup

For the sizeup experiments, we fixed the number of processors at 30, and sorted relations of various sizes ranging from 1.2M tuples to 4.8M tuples. In all cases, the number of samples was fixed at 3000 (100 per processor), and the times presented are averages over 5 trials.

problem size (tuples)	execution time (sec)	ratio (running times)
1.2M	70.4	1.0
2.4M	133.0	1.89
3.6M	196.1	2.79
4.8M	257.0	3.65

Table 6: Sizeup results.

Table 6 shows the somewhat surprising result that our implementation achieved sublinear sizeup. That is, sorting 4.8M tuples took less than four times as long as sorting 1.2M tuples. The reason this occurs is that the portion of the running time due to sampling takes an amount of time that is independent of the size of the problem. Hence, for smaller problem sizes, the sampling overhead is a more significant component of the running time than it is for larger problem sizes. Table 7 presents the same data as Table 6 except that the sampling overhead is subtracted from each of the running times. This table shows that if we ignore the sampling component, the speedup is approximately linear. This highlights an important point: the larger the problem size, the more effective is probabilistic splitting.

problem size (tuples)	execution time - sample time (sec)	ratio (running times)
1.2M	58.4	1.00
2.4M	121.0	2.07
3.6M	184.1	3.15
4.8M	245.0	4.20

Table 7: Sizeup results exclusive of sampling time.

7 Conclusion

Partitioning the file being sorted is a critical step in multiple-input multiple-output external sorting. Our analytic results suggest that for this problem, probabilistic splitting dominates a previously proposed deterministic method; our experimental results prove that for up to thirty processors (the maximum we could test), probabilistic splitting achieves good speedup and scaleup performance.

This speedup and scaleup will not continue indefinitely. In order to maintain constant expected skew in the sizes of the subfiles produced by probabilistic splitting, the number of samples taken must grow with the number of processors. This means that as the number of processors scales, eventually one must choose between large skews or large numbers of samples, neither of which will give good performance in general. However, it is important to note that the speedup achieved by probabilistic splitting improves as the size of the file to be sorted grows (recall Figure 4.) For sorting one million record files on systems with tens of processors, our implementation proves that probabilistic splitting is a highly effective technique. Although we are unable to prove it on our current hardware, we expect that for files of one billion records and beyond, probabilistic splitting will be effective even on systems with a few hundreds of processors.

Acknowledgements

This research was supported by donations from DEC, IBM, NCR, and Tandem. We would also like to thank Rick Rasmussen for his assistance in dealing with the Gamma hardware and software.

References

[BBDW83] Dina Bitton, Haran Boral, David J. Dewitt, and W. Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.

[BBW88] Micah Beck, Dina Bitton, and W. Kevin Wilkinson. Sorting large files on a backend multiprocessor. *IEEE Transactions on Computers*, 37(7):769 – 778, 1988.

[BG89] Bjørn Arild W. Baugstø and Jarle Fredrik Greipsland. Parallel sorting methods for large data on a hypercube database computer. In *Proceedings of the Sixth International Workshop on Database Machines*, pages 127 – 141, Deauville, France, June 1989. Springer-Verlag.

[BGMP79] M. W. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. *Operating System Review*, 13(2), 1979.

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, Hilton Head, North Carolina, July 1991.

[CDKK85] H-T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the Wisconsin Storage System. *Software—Practice and Experience*, 15(10):943–962, October 1985.

[DG90] D. DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad. *ACM SIGMOD Record*, 19(4), December 1990.

[DGS+90] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. A comparison of non-equijoin algorithms. In *Proceedings of the Eighteenth International Conference on Very Large Databases*, Barcelona, Spain, August 1991. To appear.

[Dob78] W. Dobosiewicz. Sorting by distributive partitioning. *Information Processing Letters*, 7(1):1–6, 1978.

[EGKS89] S. Englert, J. Gray, T. Kocher, and P. Shah. A benchmark of Nonstop SQL Release 2 demonstrating near-linear speedup and scaleup on large database. Technical Report 89.4, Tandem Part No. 27469, Tandem Computers, 1989.

[FM70] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496 – 507, 1970.

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

[HC83] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the Seventh International Computer Software and Applications Conference*, pages 627 – 631, Chicago, Illinois, November 1983.

[IRV89] Balakrishna R. Iyer, Gary R. Ricard, and Peter J. Varman. Percentile finding algorithm for multiple sorted runs. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 135 – 144, Amsterdam, The Netherlands, August 1989.

[JL85] Philip J. Janus and Edmund A. Lamagna. An adaptive method for unknown distributions in distributive partitioned sorting. *IEEE Transactions on Computers*, C-34(4):367–372, April 1985.

[LKB87] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proceedings of 1987 SIGMETRICS Conf.*, May 1987.

[LY89] Raymond A. Lorie and Honesty C. Young. A low communication sort algorithm for a parallel database machine. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 125–134, Amsterdam, The Netherlands, August 1989.

[Qui88] M. J. Quinn. Parallel sorting algorithms for tightly coupled multiprocessors. *Parallel Computing*, 6:349 – 367, 1988.

[RE78] D. Ries and R. Epstein. Evaluation of distribution criteria for distributed database systems. Technical Report UCB/ERL Tech. Rep. M78/22, UC-Berkeley, May 1978.

[SN91] S. Seshadri and Jeffrey F. Naughton. Sampling issues in parallel database systems. Submitted for publication., June 1991.

[STG+90] Betty Salzberg, Alex Tsukerman, Jim Gray, Susan Uern, and Bonnie Vaughan. FastSort: A distributed single-input single-output external sort. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 94–101, Atlantic City, New Jersey, May 1990.

[Sto86] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[VG84] Patric Valduriez and Georges Gardarin. Join and semijoin algorithms for a multiprocessor database machine. *ACM Transactions on Database Systems*, 9(1):133–161, March 1984.

[YT88] Y. Yamane and R. Take. Parallel partition sort for database machines. In Masaru Kitsuregawa and Hidehiko Tanaka, editors, *Database Machines and Knowledge Base Machines*, pages 117 – 130. Kluwer Academic Publishers, 1988.