PPS - a Parallel Partition Sort Algorithm for Multiprocessor Database Systems

X. Zhao, N.J. Martin, R.G. Johnson
Department of Computer Science
Birkbeck College
University of London

Abstract

In this paper a new algorithm, Parallel Partition Sort (PPS), is proposed which is an improved range partition sort algorithm to handle the problem of workload imbalances, especially partition imbalance and the heterogeneous imbalance, in a shared-nothing multiprocessor database environment. The new algorithm partitions the key range into a number of range intervals, then a fast internal sorting method is applied on each range interval combined with a dynamic mechanism to handle different interval sizes. A dynamic mathematical model approach is used to balance the workload among processing nodes by estimating data distribution during the sorting process. Experimental results demonstrate that the new algorithm performs better than existing parallel range partition sorting algorithms in a shared-nothing database environment for a wide degree of skew.

1 Introduction

It is often said that 25-50% of all the work performed by computers consists of sorting data. Hence, the performance of sorting algorithms is of major importance in practice. Sorting algorithms are frequently used by relational database systems for building indexes on tables, ordered retrieval, duplicate elimination, non-equal join, grouping, and aggregation. As highly parallel database systems are becoming increasingly used for large database and transaction processing tasks, the development of efficient parallel sorting algorithms for parallel database systems is particularly important.

In a shared-nothing parallel database system, each processing node has its own local memory and disk drives and all communication goes through an interconnection network. In a typical sorting problem, initially the data to be sorted is on disk and distributed throughout all the processing nodes, and after sorting, data is sorted into many non-overlapping runs and all the sorted runs are either stored back on distributed disks or sent to a single sink site. Two major approaches have been taken to deal with this sorting problem. One is that each processing node sorts its local data and creates sorted temporary runs on its local disk, then these sorted runs are merged in parallel on all processing nodes. The problem of this approach is that splitting the merge task into a number of sub-merges in order to be carried out in parallel is not a trivial thing. The other approach is that a range partition is calculated and each processing node applies this range partition to divide the sorting task into a number of sub-tasks, then each processing node distributes the data and then processes its own sub-tasks.

In [Dewitt 91], a comparison of these two approaches is investigated and the results demonstrate that the approach which breaks the sorting task early by a range partition performs better than that which firstly sorts the local data and then splits the merge task later.

The PPS algorithm presented in this paper improves on existing range partition sort algorithms through the use of a dynamic model to balance the workload among processing nodes by estimating data distribution during the sorting process.

The paper is organized as follows. In Section 2, we briefly review parallel range partition sort algorithms as well as other research results which are utilised in the PPS algorithm. The new PPS algorithm is presented in Section 3, with an analysis and comparison with a conventional range partition sort algorithm being given in Section 4. Experimental results from an implementation of the PPS algorithm are given in Section 5. Finally, Section 6 presents conclusions.

2 Background

As noted in Section 1, [Dewitt 91] analyzed the major approaches to parallel sort algorithms, and showed that

Proceedings of the 11th International Workshop on Database and Expert Systems Applications (DEXA'00)
0-7695-0680-1/00 $10.00 © 2000 IEEE

those based on range partition performed best. In this section, we briefly review such algorithms and more recent related work as well as research results which are utilised in the PPS algorithm.

There are a number of papers concerned with parallel sorting including [Helman 98], [Dachraoui 96], [Young 95], [Li 94], [Dewitt 91], [Lorie 89], [Quinn 88], and [Yamane 88]. A number of papers, including [Olariu 99], [Fernandez 97] and [Hsu 97], have investigated the application of specialised architectures, such as meshes, to parallel sorting. Since the motivation for our work has been efficient parallel sorting within database management systems, and such systems are required on as wide a variety of parallel architectures as possible, our work is based on a multiprocessor architecture with no specialised interconnection network. Hence, we only consider work below which is not based on a specialised network architecture.

A typical range partition sort described in [Dewitt 91] is as follows:

1. Sampling Phase: Determine a "splitting vector" $v[i]$ by sorting sampled keys, where $0 \leq i < s$ and s is the number of processing nodes, and an interval is formed by two successive elements such as $(v[i], v[i+1])$. Assign a processing node address to a data interval such as all records on processing node i have sort key value greater than or equal to $v[i]$ but less than $v[i+1]$ where $0 \leq i < s$. Records on processing node 0 have sort key value less than $v[0]$, while records on processing node $s-1$ have sort key value greater than or equal to $v[s-1]$.

2. Redistribution Phase: Based upon this assignment, redistribute the records on all processing nodes so that each record is at the appropriate processing node. When the redistributed records fill local memories, sort the in-memory records and store them on disk as sorted runs.

3. Local Sorting Phase: After this redistribution, merge the sorted runs locally on each processing node to produce the final result.

Although [Dewitt 91] has analysed the problem quite thoroughly, the algorithm itself is not an optimal algorithm for parallel sorting. To let the number of the intervals be the number of processing nodes creates a number of problems. The first problem is that the interval sizes vary and this variation causes workload imbalance among processing nodes due to the inaccuracy of the sample phase. The second problem is that these unpredictable interval sizes impair the ability of applying fast internal sort algorithms for sorting each interval because some interval sizes may exceed memory size bounds on processing nodes.

[Li 94] has improved the algorithm above by partitioning the key range into a large number of intervals. To some degree, this overpartitioning approach eases the problem of interval size exceeding local memory size since interval sizes are likely to be reduced by increasing the number of intervals. But overpartitioning does not tackle the real issue of the problem, that is the inaccuracy of the sample phase. By overpartitioning, the overheads of managing large number of intervals are increased and the load imbalance is still unsolved since overpartitioning does not take into consideration of interval data distribution and uniform output cannot be guaranteed without knowing input distribution information [Knott 73].

[Young 95] tries to improve the performance in the case of sending sorted data to a single sink site, but it does not tackle the workload imbalance problem. [Dachraoui 96] calculates the ranks of keys before sorting data. The rank represents the distribution of the key value. So [Dachraoui 96] establishes the key distribution of sorting records and then balances the sorting phase by applying this information. The price to pay is the phase of calculating key ranks which can be very expensive since it checks through all the key values a number of times.

Although such parallel range partition sort algorithms exploit high degrees of parallelism, they are afflicted with the problem of unbalanced workloads among processing nodes. The unbalanced workloads curtail scalability and degrade the performance. To take full advantage of parallel processing, a workload balancing mechanism is necessary.

Workload imbalance may be caused by data skew. The placement imbalance is from unbalanced initial data placement. The partition imbalance arises when the workload inherent in a sort is not balanced among processing nodes. Another kind of workload imbalance is caused not by data skew itself but rather by the unbalanced processing power and resource availability among processing nodes: this is the heterogeneous imbalance.

In addition to using dynamic methods to improve the range partition algorithm, the PPS algorithm also exploits non-conventional sorting techniques as pioneered by E Isaac and R Singleton. While conventional sorting algorithms are based on key comparisons, non-conventional sorting algorithms partition the key's range by applying key distribution information. In [Isaac 56] and [Kron 65], sorting by address calculation is presented which assigns a preliminary address to each record. It is an internal sorting technique which has two stages. In the first rough focusing stage, an address calculation function is used to place a new record among the records which have already been inserted. The purpose of the second fine focusing stage is solely to adjust for imperfections due to the roughness of the address calculation function.

Distributive Partitioned Sort (DPS) is another internal

sorting technique described in [Dobos 78]. DPS passes through n records once to find the maximum, minimum, and median records, divides the range of data between the maximum and minimum into n buckets with $n/2$ equal length intervals on one side of the median and another $n/2$ equal intervals on the other side. Each record is put into its bucket and the DPS recurses for the buckets with more than one record. In [Janus 85], an adaptive method is proposed to improve DPS for a non-uniform distribution. The adaptive DPS also scans through n records to find the maximum and minimum, and divides the range into n buckets. In order to spread records more evenly on n buckets, the adaptive DPS uses the following steps to map a record to a bucket: 1) split the range into m equal length cells, 2) find the cumulative probabilities $p_1, \ldots, p_m$ of m cells, 3) fit a line between each adjacent pair of cumulative probabilities and save the slope of y-intercept of each line, 4) distribute each of the n records by first determining to which cell it belongs, say k, and then use the kth line equation to find the bucket into which the record falls. The DPS sorting algorithm and the adaptive version give the considerable faster running time $O(n)$. Furthermore the adaptive DPS demonstrates significant improvement over the DPS for highly skewed data.

3 The PPS Parallel Partition Sort Algorithm

As with a conventional range partition sort algorithm, the PPS algorithm creates an interval partition table to control the allocation of key range intervals to processing nodes. A high level view of the PPS algorithm is as follows.

1. Sampling Phase:
   Initialize the interval partition table by filling entries with $\{(x_{i-1}, x_i)\}, i = 1, 2, \ldots, n$, where x_i are ordered values of the key attribute range. n is set near to the ratio of the total size of data to the total distributed memory size. Then assign intervals evenly among the processing nodes. So each entry in the partition table has a low interval range value, a high interval range value, and a processing node id that the interval is assigned to.

2. Partitioning Phase:
   Based upon the assignment, all processing nodes redistribute the records so that each record is sent to the appropriate processing node. Each processing node maintains the data distribution information, such as current sizes of the intervals distributed from the processing node. At some timepoints, all processing nodes exchange their data distribution information. So the entire current interval distribution pattern can be formed. The mathematical data distribution estimation methods described in [Zhao 95] are applied. Based on the estimated final interval distribution, a balance process can begin to re-assign the intervals from overloaded processing nodes to underloaded processing nodes. Interval allocation assignments can be adjusted. For some intervals, if the final estimated sizes are going to exceed the memory requirement of its processing node, these intervals are split in half into two intervals based on their median values and the interval allocation assignments will be adjusted to accommodate this change. The interval allocation assignments will be broadcasted to all the other processing nodes. All processing nodes receives the distributed records and organizes them based on the interval definitions. Then the first 3 steps of the adaptive DPS internal sorting method is applied.

3. Local Sorting Phase:
   After data redistribution, each processing node sorts its intervals by employing the last two steps of the adaptive DPS internal sorting method and stores the sorted result on its disk. Each processing node maintains the processing information, such as the number of sorted intervals. At some timepoints, all the processing nodes exchange the processing distribution information. So overall sorting progress distribution information can be established. The same mathematical data distribution estimation methods described in [Zhao 95] are applied to estimate the final outcome of the processing distribution. If any serious processing imbalance occurs among processing nodes, some intervals are reassigned from the underperformed processing nodes to the overperformed processing nodes.

4. Merging Phase:
   If the sorted results are needed at a single node, all the processing nodes sends their sorted data to this single node. On this single node, no actual merge action is needed. Data are organized sequentially based on the order of interval ids.

Suppose there are C processing nodes with local disk space and M bytes of local memory and all processing nodes are connected via a network in a parallel database system. Relation R to be sorted is originally equally partitioned among the processing nodes and let R_c be the portion on the processing node c where c is one of the processing node ids, $0, \ldots, C-1$. The key attribute is the attribute to be sorted and each key has K bytes. Let N be the cardinality of the relation and N_c be the cardinality of the portion on the processing node c. We have $N_c = N/C$. A range partition partitions R_c on the key attribute into B number of intervals, $R_c^0, R_c^1, \ldots, R_c^{B-1}$. The ids of intervals are $0, 1, \ldots, B-1$. Each tnterval are assigned to a processing node during the first phase of PPS. The intervals on its assigned processing nodes are merged by receiving

thier data from other processing nodes during the data partitioning phase. Let intervals be $R^0, R^1, \ldots, R^{B-1}$, where $R^b = \sum_{c=0}^{C-1} R_c^b$. The $\sum$ here signifies that all the interval data with the same interval identifier are merged. The initial attempt to make the size of each interval approximate to the memory size is by setting B, the number of the intervals, to the ceiling of $\frac{NL}{M}$ where L is the record length in bytes.

A common way to initialize the partition table on the key range is by sampling. Take a simple random sample without replacement of size Bs from the N keys and sort the keys in the sample to be $x_1, \ldots, x_{Bs}$. s is a number that indicates the sample rate. Choose $x_s, x_{2s}, \ldots, x_{Bs}$ as the interval divider. The key values lying between two successive dividers $(x_{si}, x_{s(i+1)})$ form an interval where $i = 1, \ldots, B-1$. Two special cases are that the first interval includes the keys smaller than x_0 and the last interval includes the keys greater than x_{Bs}. The interval partition table is built as

$$\{(x_1, x_s], (x_s, x_{2s}], \ldots, (x_{(B-1)s}, x_{Bs})\}$$

with an assigned processing node id column and a counter column that holds interval sizes.

Each interval is initially assigned a processing node id on which its records for the interval are gathered. Each processing node allocates $\frac{B}{C}$ intervals. Let X_i be $\{x_{is}, x_{is+1}, \ldots, x_{is+s-1}\}$, the ordered samples which compose the interval i, where i is one of $0, 1, \ldots, B-1$. Let y_i^k be the number of the samples in X_i taken from the processing node k where k is one of $0, 1, \ldots, C-1$. The interval i can be allocated to the processing node j when $y_i^j = \max_{k=0}^{C-1} y_i^k$. It means that an interval is allocated to the processing node on which the most of the sampled keys are taken for this interval.

The expected size in bytes of each interval is $\frac{NL}{B}$. The average size on each processing node is $\frac{NL}{C}$. Let $q\frac{NL}{C}$ be the skewed data on a processing node. where q is some constant greater than 1. Let $\hat{s}$ be the average number of samples taken on each processing node. We have $\hat{s}C = sB$. The upper bound on the probability that any processing node has the data size of all intervals greater than $q\frac{NL}{C}$ is given in [Blel 91]. It is

$$p = Ce^{-(1-1/q)^2 q\hat{s}/2}$$

or

$$\hat{s} = \frac{2\ln(\frac{C}{p})}{(1-\frac{1}{q})^2 q}$$

The sample of $\hat{s}C$ keys guarantees that a processing node has skewed data at most $q\frac{NL}{C}$ with probability $1-p$.

As we will see that the employed dynamic mechanism can handle any overflowed intervals, now we focus our attention on the skewed data on a processing node during the data partitioning phase. With the initial allocation of intervals, each record is put into one of the intervals based on its key value and sent to the assigned processing node. Each processing node maintains the current sizes of all the intervals to which it sends records. So let O_{db}^{tp} be the number of records in the interval b on the processing node d, which is assigned to processing node p at point t, where t is defined as the percentage of records processed. Let function f map interval identifiers to assigned processing node identifiers and $F(x)$ represent the set of the interval identifiers with the same processing node identifiers x.

At a timepoint before the partition phase finishes, each processing node has its own distribution information of $O_{dj}^{tf(j)}$, where j is $0, \ldots, B-1$. Each processing node broadcasts its distribution information to all the other processing nodes, thus allowing the entire current interval bucket distribution pattern, $O_j^{tf(j)} = \sum_{i=0}^{C-1} O_{ij}^{tf(j)}$, where j is $0, \ldots, B-1$, to be organized on all the processing nodes. The $\sum$ signifies that all O values with the same interval identifier are added together. This distribution information is a measure of the workloads when $O^{tk} = \sum_{j \in F(k)} O_j^{tk}$, where k is $0, \ldots, C-1$.

A value t is used to measure the progress of the partition phase. t represents the ratio of the amount of data that has been partitioned on a processing node to the total amount of data that is going to be partitioned on this processing node. When t reaches particular threshold values for the first time, a 'checkpoint' occurs and the processing node sends its distribution information O^t to all the other processing nodes. Threshold values can be set up initially as $5\%, 10\%, \ldots, 50\%$. At a checkpoint each processing node has finished approximately the same amount of work and the merged O^{tk} gives the processing node workloads, where k is $0, \ldots, C-1$.

In [Zhao 95], mathematical approaches for estimating data distribution are studied. By applying these approaches, a new distribution estimation method is developed for the PPS algorithm. The method views data from the output of data distribution on a range partition as a random sampling process without replacement. So the data can then be used to estimate the distribution of the range partition. To predict the final bucket distribution characteristics, the probability distribution of intervals is calculated from the O values known at point t of the partition phase. The final estimated sizes for the intervals can be calculated by multiplying the total size of data with their interval probabilities. The procedure is as follows:

1. Test if a uniform distribution fits the distribution data. If it fits, use the uniform distribution function. Else go to next step.

2. Test if a binomial distribution fits the distribution data. If it fits, use the binomial distribution function. Else go to next step.

3. Test if a normal distribution fits the distribution data.

0-7695-0680-1/00 $10.00 © 2000 IEEE

If it fits, use the normal distribution function. Else go to next step.

4. Apply the Fourier method.

Further details can be found in [Zhao 95].

If differences among estimated final interval sizes are significant, a balancing process begins. The processing node that has the biggest workload ships some intervals to the processing node that has the smallest workload. Each processing node independently adjusts its interval allocation assignments and its distribution information. Then the incoming data for these intervals is sent to the correct destination. If an estimated interval size is too big to fit into local memory, the processing node that the interval resides on splits the interval into two intervals through its median physically. Again, each processing node independently adjusts its interval allocation assignments.

Another dynamic feature in PPS is to handle the heterogeneous imbalance. During the local sorting phase after the partitioning phase, when a node completes an interval sorting, this processing node broadcasts the information with the sorted interval id to the other nodes. Since all processing nodes maintain all interval final assignments and interval sizes information, each processing node can calculate sorting progress information independently for all processing nodes. Let N^i_{node} be the total number of records on the processing node i, i is one of $0, 1, \ldots, C-1$. And let $N^i_{node_sorted}$ be the total number of records have been sorted at the t moment on the processing node i, i is one of $0, 1, \ldots, C-1$ and t is the percentage of the total sorted records against the total number of records on that processing node. Let $N^i_{node_unsorted}$ be $N^i_{node} - N^i_{node_sorted}$. Suppose $N^m_{node_max} = \max_{i=0}^{C-1}(N_{node_sorted})$ and $N^n_{node_min} = \min_{i=0}^{C-1}(N_{node_sorted})$. So the processing node m is the most powerful node and the node n has the least processing power. Let R^i_{node} be $\frac{N^i_{node_sorted}}{N^i_{node}}$. If $R^m_{node}N^m_{node_unsorted} - R^n_{node}N^n_{node_unsorted}$ is bigger than one of the intervals on the processing node n, one of the intervals on the node n will be chosen to move to the processing node m and all processing nodes will adjust their interval assignment tables and workload information to refect this change. The above process can continue until there is no more discrepancy among processing nodes.

Some notations are given in Table 1. The PPS algorithm is described as follows:

1. Sampling Phase:

   - Based on probability requirements, s can be calculated. In parallel, each processing node takes $s\frac{B}{C}$ samples and the cost is $s\frac{B}{C}I_{sample}$. Each processing node sorts the sampled keys. The cost is $s\frac{B}{C}\log_2(s\frac{B}{C})(I_{keymove} + I_{keycomp})$ when an in-memory heapsort is employed.

I_{sample}:	cost of taking a random sample into memory.
$I_{keycomp}$:	cost of comparing keys in memory.
$I_{keymove}$:	cost of moving a key into a location.
$I_{keycalc}$:	cost of calculating a cell location for a key.
$I_{recmove}$:	cost of moving a record into a location.
I_{net}:	cost of transferring a record on network.
I_{IO}:	cost of reading or writing a record on disks.

Table 1. Notations

M:	Memory capacity in bytes for each processing node.
C:	Number of processing nodes.
K:	Number of bytes in a key.
B:	Number of intervals.
L:	Size in bytes of each record.
s:	Sample rate. sB is the number of samples
$\lvert P_s \rvert$:	Size in records of skewed intervals.
$\lvert P_u \rvert$:	Size in records of remaining unskewed intervals.
q:	Degree of skewed intervals. It is $\frac{\lvert P_s \rvert}{\lvert P_u \rvert}$.
N_{cpu}:	CPU processing rate in instructions per second.
ω_{io}:	I/O bandwidth of secondary storages.
ω_{comm}:	Effective communication bandwidth.
I_{cpu}:	pathlength for processing a record in any operations.

Table 2. Description

   - The locally sorted sets of sampled keys are gathered and merged at a designated processing node. Each sampled key is sent with an extra byte to indicate the original processing node id. Let $x_0, x_1, \ldots, x_{Bs}$ be the total sorted keys. The interval dividers are $x_0, x_s, \ldots, x_{Bs}$. The interval i contains the keys between the key x_{is} and $x_{(i+1)s}$ and includes the keys equal to x_{is} where i is one of $0, 1, \ldots, B-1$. Two special cases are that the first interval includes the keys smaller than x_0 and the last interval includes the keys greater than x_{Bs}.
   - For each interval the numbers of sampled keys associated with a processing node id are counted. The processing node id which contributes the most sampled keys for an interval is found. Let V be the set of triplets of (i, j, n) for each interval, where i is an interval id, j is the processing node id which contributes the most keys for the interval i, and n is the number of the keys from the processing node j in the interval i. Let Z be the set of pairs of the interval id and its allocated processing node id. Z is called the interval allocation table. Z can be formed as follows: Choose the triplet with the largest number n in V and put the

0-7695-0680-1/00 $10.00 © 2000 IEEE

other two items as a pair into Z. Delete this triplet in V and repeat this process until V is empty. When a processing node has the number of allocated intervals equal to $\frac{B}{C}$, the process stops assigning any more intervals to this processing node, instead the subconsequent intervals are allocated to the processing node which has the next largest number of the sampled keys in the intervals.

- When a binary tree merge is applied to merge the sorted local keys, the parents of the leaves merge two sequences of size $s\frac{B}{C}$ at a cost of $2s\frac{B}{C}(I_{keycomp} + I_{keymove})$, and the grandparents of the leaves merge two sequences with size $2s\frac{B}{C}$ at a cost of $4s\frac{B}{C}(I_{keycomp} + I_{keymove})$, and so on. Sum them together, the total cost of the merge is $2s\frac{B}{C}(C-1)(I_{keycomp} + I_{keymove})$. The total keys moved on the network are $s\frac{B}{C}(C-1)$. The cost is $s\frac{B}{C}(C-1)(K+1)I_{net}$. The cost to broadcast the interval partition table and the interval allocation assignment is $(K+2)BI_{net}$ when all ids are one byte long.

2. Partitioning Phase:

- In parallel, processing nodes read their portion of the records and extract keys. Each key is searched through the interval partition table to find which interval it belongs to and its destination processing node id, then the record of this key is placed into a proper buffer. When the buffer for the intervals is full, the records in the buffer are sent to the target processing node. The cost for disk I/O is $\frac{N}{C}I_{IO}$. The cost of searching and placing records is $\frac{N}{C}(log_2(B)I_{keycomp} + I_{recmove})$. The cost of sending data is $\frac{N}{C}I_{net}$. Each processing node maintains its own O_{db}^{tp}.
- In parallel, processing nodes receive the records and apply the first stage of the internal adaptive distributive partitioned (DPS) sort algorithm described in [Janus 85] to the intervals. Suppose that each interval is divided into m equal length cells which are $x_{ks}, x_{ks}+\frac{x_{(k+1)s}-x_{ks}}{m}, \ldots, x_{ks}+m\frac{x_{(k+1)s}-x_{ks}}{m}$ where x_{ks} and $x_{(k+1)s}$ are the interval dividers for the interval k and k is one of $0, 1, \ldots, B-1$. The cumulative probabilities of m cells are calculated as $p_i = c_i/c$ where c_i is the number of the records whose keys are smaller or equal than $x_{ks} + i\frac{x_{(k+1)s}-x_{ks}}{m}$ and c is the total number of the records in the interval k where i is one of $0, 1, \ldots, m-1$. Since there is not enough memory for all intervals to do a complete internal sort, the records are stored back on local disks accompanied with the cumulative probability figures. At this stage, the size of each interval is evaluated against the memory requirement of internal sorting. For any interval which cannot fit in memory for an internal sort, the interval is split into two equal sub-intervals so the first $\frac{m}{2}$ cells compose a sub-interval and the remaining of $\frac{m}{2}$ cells form another sub-interval. The cost of calculating the cumulative probabilities is $\frac{N}{C}(log_2(m)I_{keycomp} + I_{keycalc})$. The cost for writing to disks is $\frac{N}{C}I_{IO}$.
- Suppose under a skew condition, data are evenly distributed among processing nodes except one which has excess data. It has $\delta \times 100\%$ more records than each of the remaining processing nodes. We have $|P_s| = \frac{N}{1+(C-1)(1-\delta)}$ and $|P_u| = \frac{(1-\delta)N}{1+(C-1)(1-\delta)}$, where $|P_S|$ and $|P_u|$ are the cardinalities of the skewed partition and unskewed partitions respectively. Suppose the checkpoint t is 50%. The cost of the balancing process equals the cost of moving the data of re-assigned intervals to their new destinations. Since the balancing process happens before the partition phase completes, only a portion of the re-assigned intervals is moved. For $t = 50\%$, only 50% data are moved. The total cost for a balancing process is $\frac{(|P_s|-|P_u|)t}{I_{net}} + \frac{(|P_s|-|P_u|)t}{I_{IO}}$ plus the overhead of managing distribution information, say about 1% of the partitioning phase communication cost.

3. Local Sorting Phase:

- In parallel, processing nodes read the intervals and their cumulative probabilities from local disks, apply the internal adaptive sort to the intervals, and write the sorted intervals back. The range of an interval is divided into buckets where the number of buckets is the number of the records in the interval. The buckets are ordered in the range of an interval. When each record is mapped into a bucket according to its key's value, this record is assigned the order of the bucket. For a key a in the interval k, its bucket id j is computed as $j = (a - \hat{a}u)\frac{(c_{i+1}-c_i)}{u} + c_i$ where $u = \frac{x_{(k+1)s}-x_{ks}}{m}$ and $\hat{a} = \lfloor \frac{a}{u} \rfloor$. If each bucket has one record, the interval is sorted. For the buckets with more than one and less than 6 records, a simple sequential sort can be applied to sort these buckets. For the buckets with more than 6 items, the QuickSort algorithm [Sedge 78] can be applied on these buckets. The cost for reading and writing the intervals is $2\frac{N}{C}I_{IO}$. The cost of internal sorting is $\frac{N}{C}(I_{keycalc} + 2I_{recmove}) + T$ where T is the cost for sorting the buckets with more than one record. T is $\frac{N}{C}p_e(I_{keycomp} + I_{recmove})$ and p_e is the expected percentage of buckets with more than one record. p_e is 26.42% according to [Janus 85]. Suppose one processing node is slower than the rest of nodes on sorting records at the R_{slow} rate. It means that for every record sorted on this node, $1 + R_{slow}$ number

0-7695-0680-1/00 $10.00 © 2000 IEEE

$$L = 64$$
$$N = 1000000$$
$$M = 2048000$$
$$N_{cpu} = 20000000$$
$$\omega_{io} = 4096000$$
$$\omega_{comm} = 4096000$$
$$I_{cpu} = 1000$$

Table 3. Parameters

of records are sorted on any other processing node. The cost for the heterogeneous balancing process is: $(\frac{NR_{slow}}{C}(1-\frac{1}{C})I_{io} + \frac{NR_{slow}}{C}(1-\frac{1}{C})I_{net}$.

4 Analysis and Comparison

In this section we compare our PPS algorithm with the algorithm presented in [Dewitt 91], which is called the ProS algorithm in the following text. In [Dewitt 91], no overlap between I/O operations, CPU operations, or network traffic is assumed, and no contention in the network are assumed. To be able to make a fair comparison the same assumptions are made here, we call it the assumption one. In order to do a more realistic analysis, we make another assumption that is that disk I/O operations, CPU operations, and network communication operations run in parallel. This is called the assumption two. While a page is being read from disk, CPU can process the data already in memory or send or receive data from the network. Table 3 gives values to some parameters.

Firstly we discuss the sample phase. The PPS algorithm partitions the relation into B intervals and the ProS algorithm divides the relation into C intervals. The PPS algorithm can split overflowed intervals into smaller ones without degrading the performance of the interval sorting algorithm, and consequently the PPS algorithm can carry out the same number of samples as the ProS without sacrificing performance. Therefore the PPS takes the same number of samples as the ProS and we have $\hat{s}C = sB$ where $\hat{s}$ is the number of samples taken per processing node and s is the number of samples per interval.

There are four sequential steps in the sampling phase in both algorithms. In the sampling step, the sampling keys are taken from disk and sorted in parallel. In the merging step, the sorted sampled keys are merged together. Then there is a building step which builds an interval partition table, and finally an informing step communicates the table to all the processing nodes. The disk I/O cost in the sampling step of the PPS is

$$T_{s_io} = s\frac{B}{C}I_{sample}$$

and the CPU cost is

$$T_{s_sort} = s\frac{B}{C}\log_2(s\frac{B}{C})(I_{keymove} + I_{keycomp})$$

When CPU and disk operations run in sequential under assumption one, the cost of the sampling step is

$$T^1_{s_s} = T_{s_io} + T_{s_sort}$$

When CPU and disk I/O run in parallel under assumption two, the cost of the sampling step is

$$T^2_{s_s} = \max(T_{s_io}, T_{s_sort})$$

Similarly the cost of the merging step is

$$T^1_{s_m} = 2s\frac{B}{C}(C-1)(I_{keycomp} + I_{keymove}) + s\frac{B}{C}(C-1)(K+1)\frac{1}{C}I_{net}$$

$$T^2_{s_m} = \max(2s\frac{B}{C}(C-1)(I_{keycomp} + I_{keymove}), s\frac{B}{C}(C-1)(K+1)\frac{1}{C}I_{net})$$

The cost of the building and forming interval table is

$$T_{s_b} = sB(I_{keycomp} + I_{keymove})$$

The cost of the informing step is

$$T_{s_i} = 2(K+1)\frac{B}{C}I_{net}\log_2(C)$$

The total cast of the sampling phase of the PPS algorithm is

$$T^1_s = T^1_{s_s} + T^1_{s_m} + T_{s_b} + T_{s_i}$$

$$T^2_s = T^2_{s_s} + T^2_{s_m} + T_{s_b} + T_{s_i}$$

Respectively the cost of the sampling phase of the ProS algorithm in [Dewitt 91] is

$$\hat{T}_{s_s} = \hat{s}I_{sample} + \hat{s}\log_2(\hat{s})(I_{keymove} + I_{keycomp}$$

$$\hat{T}_{s_m} = 2\hat{s}(C-1)(I_{keycomp} + I_{keymove}) + \hat{s}(C-1)I_{net}$$

$$\hat{T}_{s_b} = CI_{keymove}$$

$$\hat{T}_{s_i} = \frac{KC}{D}I_{net}\log_2(C)$$

$$\hat{T}_s = \hat{T}_{s_s} + \hat{T}_{s_m} + \hat{T}_{s_b} + \hat{T}_{s_i}$$

In the partitioning phase of the PPS algorithm, there are two categories of work running in parallel. They are the sending part and the receiving part. The sending part reads records from disk and sends them to appropriate processing nodes by examining key values against the interval partition table. The receiving part receives records, organizes them into the cells and stores cells and their cumulative probability back on disk. These two parts compete for all available resources on a processing node. Two parts of disk I/O operations work sequentially and two parts of CPU also work sequentially on a processing node node. For the sending part of the PPS algorithm, the I/O cost is

$$T_{r_s_io} = \frac{N}{C}I_{IO}$$

0-7695-0680-1/00 $10.00 © 2000 IEEE

the CPU cost is

$$T_{r_s_cpu} = \frac{N}{C}(\log_2(B) I_{keycomp} + I_{recmove})$$

For the receiving part of the PPS algorithm, the CPU cost is

$$T_{r_r_cpu} = \frac{N}{C}(\log_2(B) I_{keycomp} + I_{recmove})$$

the I/O cost is

$$T_{r_r_io} = \frac{N}{C} I_{IO}$$

The network communication connects all processing nodes together. Its cost is the cost of shipping all data across network. It is

$$T_{r_net} = N(1 - \frac{1}{C}) I_{net}$$

The extra cost for balancing process based on $t = 50\%$ is

$$T_{extra_net} = ((|P_s| - |P_u|) I_{net} + T_{r_net} \frac{1}{100}) t$$

$$T_{extra_IO} = ((|P_s| - |P_u|) I_{IO} + T_{r_net} \frac{1}{100}) t$$

The total cost is

$$\begin{aligned} T_r^1 = & T_{r_s_io} + T_{r_r_io} + T_{extra_IO}) \\ & + (T_{r_s_cpu} + T_{r_r_cpu}) \\ & + T_{r_net} + T_{extra_net} \end{aligned}$$

$$\begin{aligned} T_r^2 = \max\ (& (T_{r_s_io} + T_{r_r_io} + T_{extra_IO}), \\ & (T_{r_s_cpu} + T_{r_r_cpu}), \\ & (T_{r_net} + T_{extra_net})) \end{aligned}$$

In the partitioning phase of the ProS algorithm, the sending part reads data from disks and sends them away according to a splitting vector, and the receiving part receives records into memory, sorts them when memory is full, and stores the sorted records into disks as runs. Since there is no memory available while the received records fill local memory, no other operations can be executed until these records are sorted and stored back on disks. The redistribution phase of the ProS algorithm has the sequential steps: the sending and receiving step and the internal sorting step. To simplify the formula that we assume the same fast internal sorting algorithm is applied and no interval overflow occurs. The cost of the sending part is

$$\hat{T}_{r_s_io} = \frac{N}{C} I_{IO}$$

$$\hat{T}_{r_s_cpu} = \frac{N}{C}(\log_2(C-1) I_{keycomp} + I_{recmove})$$

The cost of the receiving part dominated by the skewed interval is

$$\hat{T}_{r_r_io} = q\frac{N}{C} I_{IO}$$

$$\hat{T}_{r_r_cpu} = q\frac{N}{C} I_{recmove}$$

The communication cost is

$$\hat{T}_{r_net} = N(1 - \frac{1}{C}) I_{net}$$

The total cost is

$$\hat{T}_{r_s}^1 = (\hat{T}_{r_s_io} + \hat{T}_{r_r_io}) + (\hat{T}_{r_s_cpu} + \hat{T}_{r_r_cpu}) + \hat{T}_{r_net}$$

$$\hat{T}_{r_s}^2 = \max((\hat{T}_{r_s_io} + \hat{T}_{r_r_io}), (\hat{T}_{r_s_cpu} + \hat{T}_{r_r_cpu}), \hat{T}_{r_net})$$

The cost for the internal sorting step is

$$\hat{T}_{r_i} = q\frac{NL}{CM}(\frac{M}{L}\log_2(\frac{M}{L})(I_{keycomp} + I_{recmove}))$$

The total cost of the partitioning phase of the ProS algorithm is

$$\hat{T}_r^1 = \hat{T}_{r_s}^1 + \hat{T}_{r_i}$$

$$\hat{T}_r^2 = \hat{T}_{r_s}^2 + \hat{T}_{r_i}$$

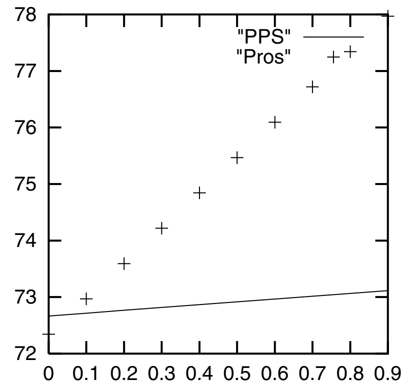

Figure 1. Data Skew Estimation 1

For the last phase of the algorithms, the local sorting phase, the cost of the PPS algorithm is

$$T_{l_io} = \frac{N}{C}(2 + \frac{R_{slow}}{C}) I_{IO}$$

$$T_{l_s} = \frac{N}{C}(1 + \frac{R_{slow}}{C})(I_{keycalc} + 2I_{recmove})(1 + p_e)$$

the extra cost of the balancing process is

$$T_{l_io_extra} = \frac{N}{C} R_{slow}(1 - \frac{1}{C}) I_{io}$$

$$T_{l_net_extra} = \frac{N}{C} R_{slow}(1 - \frac{1}{C}) I_{net}$$

the total costs of the local sorting phase are

$$T_l^1 = T_{l_io} + T_{l_io_extra} + T_{l_s} + T_{l_net_extra}$$

Proceedings of the 11th International Workshop on Database and Expert Systems Applications (DEXA'00)
0-7695-0680-1/00 $10.00 © 2000 IEEE

$$T_l^2 = \max\ (T_{l_io} + T_{l_io_extra}),\ T_{l_s},\ T_{l_net_extra})$$

The cost of the local sorting phase of the ProS algorithm is

$$\hat{T}_{l_io} = 2\frac{N}{C}(1 + R_{slow})I_{IO}$$

$$\hat{T}_{l_s} = \frac{N}{C}(1 + R_{slow})(I_{keycomp} + I_{recmove})$$

$$\hat{T}_l^1 = \hat{T}_{l_io} + \hat{T}_{l_s}$$

$$\hat{T}_l^2 = \max\ \hat{T}_{l_io},\ \hat{T}_{l_s})$$

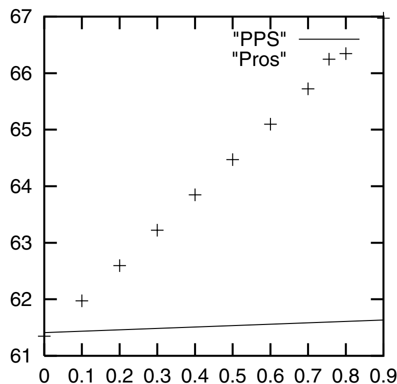

Figure 2. Data Skew Estimation 2

The Figures, 1 and 2, display the comparison of the PPS algorithm and ProP algorithm under data skew conditions. The Figure 1 shows the performance estimation under the assumption one while figure 2 shows the estimation under the assumption two. In Figure 1 and 2, the solid line represents the PPS algorithm performance and the dotted line represents the performance of the ProS algorithm. The described cost models are used. The vertical axis is the execution time in seconds, the horizontal line represents the percentage of excess data portion on a processing node to the average size of data portion on the other nodes. The figures show the PPS algorithm reacts better when the degree of the data skew is large.

5 Experimental Results

The PPS algorithm is architecture independent. It can be applied not only in a parallel database environment, but also in other general parallel computing environments. With advances on LAN technologies, such as fast Ethernet and Gigabit Ethernet, this algorithm can be used as a general parallel sorting algorithm on a LAN environment.

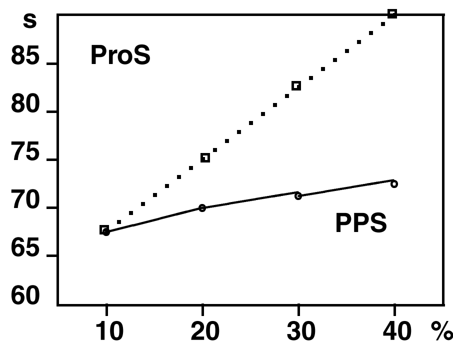

Figure 3. Data Skew Influence

The current PPS implementation is in ANSI C++. It has a communication layer that isolates networking specific functions. The purpose is to make it portable to various platforms. Figure 3 shows the data skew influence. There are four computers involved and they all run Linux RedHat 6.5. The network is a fast Ethernet environment. Computers are powered by Pentium III 500 CPUs. The vertical axis is the execution time in seconds, the horizontal line represents the percentage of the largest data portion on a processing node to the average size of data portion on the other nodes. Again the PPS algorithm is compared with the ProS. The solid line represents the PPS algorithm performance. The dotted line represents the performance of the ProS algorithm. The figure shows the PPS algorithm reacts better when the degree of data skew is large.

6 Conclusion

As most unbalanced distributions can be balanced by relocating intervals, the PPS algorithm aims to achieve a balanced output with minimum cost. Firstly, this algorithm combines the distribution process with the partitioning phase to cut the preprocessing cost. Secondly, the algorithm generates the distribution information on every processing node to avoid the need for processor synchronization. Thirdly, the algorithm performs the balancing process as part of the partitioning phase to send remaining data to the correct destinations. Finally, the algorithm takes heterogeneous imbalance into consideration and employs a dynamic mechanism to balance resource differences and processing power differences.

The PPS algorithm needs no knowledge of input distribution information, and relies on no conditions and limitations on the distribution. Therefore it is an independent component which can be applied with any data distribution. This algorithm could also be incorporated into other balancing algorithms as an adjustment phase. In view of

0-7695-0680-1/00 $10.00 © 2000 IEEE

its independence of distribution information, only a small amount of work is required to incorporate this method. When preprocessing-based balancing algorithms encounter an unacceptable level of performance, this method provides a good replacement. Given the importance of sorting in many database operations, the algorithm enables the exploitation of parallelism at a number of stages during the execution of database queries.

References

[Blel 91] G. Blelloch, C. Leiserson, B. Maggs, C. Plaxton, S. Smith, and M. Zagha, "*A Comparison of Sorting Algorithms for the Connect Machine CM-2*", In 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, South Carolina, July 1991.

[Dachraoui 96] T. Dachraoui and L. Narayanan, "*Fast Deterministic Sorting on Large Parallel Machines*", Eighth IEEE Symposium on Parallel and Distributed Processing, 1996, Page(s):273-280.

[Dewitt 91] D. J. Dewitt, J. F. Naughton, and D. A. Schneider, "*Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting*", Proceedings of the First International Conference on Parallel and Distributed Information Systems, Page:280-291, 1991.

[Dobos 78] W. Dobosiewicz, "*Sorting by Distributive Partitioning*", Information Processing Letters, Vol 7, No 1, 1978.

[Fernandez 97] A. Fernandez and K. Efe, "*Generalized Algorithm for Parallel Sorting on Product Networks*", Transactions on Parallel and Distributed Systems, Vol 8, No 12, 1997.

[Helman 98] D. R. Helman, "*Sorting on clusters of SMPs*", Parallel Processing Symposium, 1998, IPPS/SPDP, Page(s):561-567

[Hsu 97] D. F. Hsu and D. S. L. Wei, "*Efficient Routing and Sorting Schemes for de Bruijn Networks*", Transactions on Parallel and Distributed Systems, Vol 8, No 11, 1997.

[Isaac 56] E. J. Isaac, R. C. Singleton, "*Sorting by Address Calculation*", Journal ACM, Vol 3, July 1956.

[Janus 85] P. J. Janus and Edmund A. Lamagna, "*An Adaptive Method for Unknown Distributions in Distributive Partitioned Sorting*", IEEE Transactions on Computers, Vol c-34, No 4, April 1985.

[Knott 73] G. D. Knott, "*Hashing Function*", The Computer Journal, Vol. 18, No 3, 1973.

[Kron 65] R. A. Kronmal, M. E. Tarter, "*Cumulative Polygon Address Calculation Sorting*", Proceedings of the 20th National Conference of the ACM, 1965.

[Li 94] Hui Li and Kenneth C. Sevcik, "*Parallel Sorting by Overpartitioning*", Technical report CSRI-295, CSRI, University of Toronto, February, 1994.

[Lorie 89] R. A. Lorie, H. C. Young, "*A Low Communication Sort Algorithm for a Parallel Database Machine*", Proceedings of the 15th International Conference on VLDB, Page:125-134, Amsterdam, 1989.

[Olariu 99] S. Olariu, C. Pinotti, and S. Q. Zheng, "*How to Sort N Items Using a Sorting Network of Fixed I/O Size*", Transactions on Parallel and Distributed Systems, Vol 10, No 5, 1999.

[Quinn 88] Michael J. Quinn, "*Parallel Sorting Algorithms for Tightly Coupled Multiprocessors*", Parallel Computing, Vol 6, 1988.

[Sedge 78] R. Sedgewick, "*Implementing Quicksort Programs*", Communication ACM, Vol 21, No 10, 1978.

[Yamane 88] Y. Yamane and R. Take, "*Parallel Partition Sort for Database Machines*", Database Machines and Knowledge Base Machines, Page:117-130, Kluwer Academic Publishers, 1988.

[Young 95] H. C. Young and A. N. Swami, "*The Parameterized Round-Robin Partitioned Algorithm for Parallel External Sort*", Proceedings of Parallel Processing Symposium, 1995, Page(s):213-219,

[Zhao 95] X. Zhao, R. G. Johnson and N. J. Martin, "*Dynamic Distribution Estimation for Hash Partitioning in Multiprocessor Database Systems*", High Performance Parallel Processing Symposium, 1995, Singapore.

0-7695-0680-1/00 $10.00

© 2000 IEEE