An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew

Joel L. Wolf, Daniel M. Dias and Philip S. Yu

P.O. Box 704, Yorktown Heights, N.Y. 10598
IBM Research Division, T. J. Watson Research Center

Abstract:
Parallel processing of relational queries has received considerable attention of late. However, in the presence of data skew, the speedup from conventional parallel join algorithms can be very limited, due to load imbalances among the various processors. Even a single large skew element can cause a processor to become overloaded. In this paper, we propose a parallel sort merge join algorithm which uses a divide-and-conquer approach to address the data skew problem. The proposed algorithm adds an extra scheduling phase to the usual sort, transfer and join phases. During the scheduling phase, a parallelizable optimization algorithm, using the output of the sort phase, attempts to balance the load across the multiple processors in the subsequent join phase. The algorithm naturally identifies the largest skew elements, and assigns each of them to an optimal number of processors. Assuming a Zipf-like distribution for data skew, the algorithm is demonstrated to achieve very good load balancing for the join phase in a CPU-bound environment, and is shown to be very robust relative to the degree of data skew and the total number of processors.

1. Introduction:
As relational database queries become more complex and relations grow larger, performance becomes an increasingly critical issue. Parallel processing of database operations is an attractive approach which might be expected to improve response times and offer the potential for incremental growth. Parallel architectures which exploit a large number of processors have become an area of active research. In recent years there have been several proposals, prototypes, and commercial systems making use of parallel processor architectures for database applications [DEMU85, DEW186, HSIA83, NECH84, VALD84, OZKA84]. All these systems distribute the data across several storage units and deploy the available processing power near each of the storage units. The rationale behind a distributed approach is that data can be accessed in parallel from all the storage units, and further that the data can be processed in parallel by all the processors. Various studies [BIC85, DEW187, SALZ83, QADA85, VALD84] have been made to evaluate the performance of different database machines.

To exploit parallelism, queries are divided into multiple tasks which can run simultaneously on the various processors. The effectiveness of parallel execution depends upon the ability to equally divide the load among the processors while simultaneously minimizing the coordination and synchronization overhead. A factor which can impair the ability to parallelize join queries successfully in a straightforward fashion is the amount of skew present in the data to be joined. In real databases it is often found that certain values for a given attribute occur more frequently than other values [CHRI83, LYNC88, MONR83]. [LYNC88] notes, for example, that for many textual databases the data distribution follows a variant of Zipf's Law [ZIPF49]. Similar distributions for real databases are reported in [WOLF90]. This non-uniformity is referred to as data skew [LAKS88]. It is inherent in the data itself and does not depend on the access pattern. [LAKS88] found that in the presence of data skew, the speedup from conventional join algorithms can be very limited, since the data skew can result in some processors being overutilized while others are underutilized. Even a single large skew element can cause the processor to which it is assigned to become overloaded. The problem is exacerbated for join operations as opposed, say, to sorts, because correlation in the data skew of each relation results in a join output which is quadratic in nature. Previous studies on join performance have largely ignored this phenomenon and assumed uniform distribution of data, thus overestimating the potential benefit of parallel query processing using conventional join algorithms. In [SCHN89] some aspects of data skew on parallel join methods are studied. However, in their study the case where both relations to be joined have data skew (double skew) was explicitly not examined. This case produced a large number of output tuples (368,474) as compared to the case with uniform distributions of data (10,000 output tuples), and the authors could find no way of normalizing the results for the double skew case to meaningfully compare them with the other cases. This is precisely the type of data skew that motivates this paper. Furthermore, we believe that the two single skew cases (i.e., skew in one relation only) that they look at are ones of mild skew, because the number of output tuples generated for the worst case they examine produces almost the same number of tuples (10,036) as that for the uniform case (10,000 tuples). As we will see,
even single skew can have a large effect on performance. Finally, for examining the effect of data skew, they examine the case of 8 processors. We will show that the effect of data skew becomes more pronounced as the number of processors is increased. Obviously, more processors will be utilized in future database machines. Our paper examines cases with up to 128 processors.

The sort merge join [BLAS77] and hash join [KITS83, BRAT84, DEW185] methods are popular algorithms for computing the equijoin of two relations. In this paper we examine the sort merge join method and propose an effective way to deal with the data skew problem. In a typical parallel sort merge join, e.g. [IYER88], each of the relations is first sorted, in parallel, according to the join column. This is called the sort phase. A transfer phase follows, in which the output of the sort phase is shipped to the various processors according to some range algorithm. Finally, in the join phase, the sorted ranges are merged and joined. Each processor handles its own range of data. Conventional parallel join algorithms do not capture the effects of skew distribution in the join column. As indicated, the impact to performance can be devastating [LKS88].

The proposed parallel join algorithm uses a divide-and-conquer approach to address the data skew problem. An extra scheduling phase is introduced after the sort phase in an attempt to balance the load across the multiple processors during the subsequent join phase. Based on output from the sort phase, the scheduling phase divides the join into multiple tasks and attempts to make optimal task assignments to balance the load. Two basic optimization techniques are employed iteratively, one to divide up the tasks, and the other to balance the load. These two algorithms form the basic building blocks of the scheduling phase. The first of these solves a variation of the so-called selection problem, and is due to Galil and Megiddo [GALI79]. The algorithm, henceforth labeled GM, is used to divide up the tasks. (The selection algorithm was originally used to find an $I^k$ smallest element in an $I \times J$ matrix whose columns are non-decreasing. The problem has its roots in convex optimization theory.) GM has the following two pleasant features: First, it can be parallelized easily across the multiple processors. Second, by its nature, the repeated execution of the algorithm naturally identifies the largest skew elements. The second building block heuristically solves the so-called minimum makespan or multiprocessor scheduling problem. The algorithm employed, known as LPT (for longest processing time first), is due to GRAH69. As the name "multiprocessor scheduling" suggests, LPT is used in our algorithm to balance the load across the processors. Although the minimum makespan problem is known to be NP-complete, LPT is a very fast heuristic which has reasonably good worst-case performance and excellent average-case performance. In solving the scheduling phase optimization problem we assume that the system is CPU-bound, so that a CPU pathlength estimate is used in the objective function.

We show that the improvement in the join phase over conventional algorithms is drastic in the high skew case. In fact, the proposed algorithm is demonstrated to achieve very good load balancing for the join phase in all cases, being very robust relative to the degree of data skew and the number of processors. A Zipf-like distribution is used to model the data skew.

The environment and assumptions are described in Section 2. The scheduling phase algorithm is presented in Section 3. In Section 4, a sensitivity analysis is provided to demonstrate the robustness of algorithm and the speed-up in the join phase over conventional algorithms. Finally, in Section 5, we summarize the results and outline our future work. A hash join version of our parallel join algorithm has also been devised [WOLF90].

### 2. Environment and Assumptions:

In this section we outline the system architecture that we assume, elaborate on the overall join algorithm that we employ, and define the data distributions that we use to examine the performance of the algorithms.

We describe our algorithm in the context of the so-called *shared nothing* architecture [STON86], illustrated in Figure 2.1. The architecture has also been referred to as the data partitioning architecture [CORN86]. In this system, there are two sets of processors -- database processors and control processors. There are one or more disks attached to each database processor. Each relation in the database is horizontally partitioned among the database processors by applying a partitioning function to the primary key of the relation. Each database processor has its own memory and operating system, and independently accesses its local disks. The database processors cooperate by sending messages across the interconnection network. The interconnection network may be a point-to-point or a multipoint network. While the particular method used for interconnecting the processors is not crucial to the thrust of our paper, it impacts the performance and overall speedup, particularly in the transfer phase. The control processors interface to users and also send database requests to the database processors. While the methods described here can also be applied to parallel joins in the so-called *data sharing or shared everything* architecture [YU87], we do not address this aspect in the paper.

The parallel sort merge join algorithm that we consider is broadly similar to that in [IYER88], and consists of four phases. In the first phase, each processor $p$ locally sorts its own partition $R_p$ and $R_q$ of each of the relations $R_i$ and $R_j$, respectively, and places the resulting
sorted run on its local disk. This sort phase could be done using an optimal tournament tree external sort as in [IVR88]. The second phase of this algorithm is a scheduling phase that attempts to split the join execution into tasks and assign tasks to the processors in an optimal manner so as to minimize the overall completion time, or makespan. It is this scheduling phase that is the crucial aspect of our paper, and the algorithm is described in detail in Section 3. The third phase is a transfer phase in which the data from different ranges of each of the sorted relations is shipped to the processor(s) assigned during the scheduling phase. Since the scheduling phase partitions the data into ranges or single distinct values (as described later), this transfer phase can be accomplished by a single pass through the data. Finally, in the join phase, the sorted ranges are read from local disk, merged and joined, and the join outputs written to disk.

As described above, the transfer phase involves an additional pass through the sorted runs of the two relations to be joined. It is possible to do this phase without extracting tuples from data blocks provided that the relative byte addresses (RBAs) of the partition boundaries are determined in the scheduling phase. This would considerably reduce the overhead of the phase. Further optimizations are possible, such as combining this transfer step with the join phase. For instance, assuming the join operation is CPU-bound, only one join task need be active on each processor at any time, as described above, the transfer phase involves an additional pass through the sorted runs of the two relations to be joined. It is possible to do this phase without extracting tuples from data blocks provided that the relative byte addresses (RBAs) of the partition boundaries are determined in the scheduling phase. This would considerably reduce the overhead of the phase. Further optimizations are possible, such as combining this transfer step with the join phase. For instance, assuming the join operation is CPU-bound, only one join task need be active on each processor at any time, as described above.

To examine the speedup achievable in the join phase by the algorithm proposed in this paper, we use synthetic data for the values in the join column, based on a Zipf-like distribution [KNUT73] as follows: We assume that the domain of the join column has \( D \) distinct values. The the probability \( p_i \) that the join column value of a particular tuple takes on the \( i \)th value in the domain \( 1 \leq i \leq D \) is \( p_i = c i^{D-\theta} \), where \( c = 1/\sum (1/i^{D-\theta}) \) is a normalization constant. We also assume that each tuple's join column value is independently chosen from this distribution. Setting the parameter \( \theta = 0 \) corresponds to the pure Zipf distribution, which is highly skewed, while \( \theta = 1 \) corresponds to the uniform distribution. We will use \( \theta = 0.5 \) as a case of moderate skew. The Zipf-like distributions corresponding to \( D = 100 \) and \( \theta = 0.0, 0.25, 0.5, 0.75 \) and 1.0 are shown in Figure 2.2. In [LYNCS88], data from large bibliographic databases are used to support models of skewed column value distributions based on Zipf distributions. See also [WOLF90].

Though the data values are assumed to be skewed, we assume that the partitioning function is such that the relations to be joined, \( R_1 \) and \( R_2 \), are more or less uniformly partitioned among the processors, i.e., each processor has a comparable number of tuples of each relation. For example, if the tuples are range partitioned on the primary key, then the ranges can be adjusted to approximately balance the number of tuples in each relation. For most skew distributions and numbers of processors, such range partitioning will lead to good balance.

3. Scheduling Phase Algorithm:

To introduce the algorithm which forms the scheduling phase of the proposed sort merge join approach, suppose that \( v_1 \leq v_2 \) are two values in the domain of the join columns. Let \( P \) denote the number of processors. Given any of the \( 2^P \) sorted runs created during the sort phase, for example the one corresponding to processor \( p \in \{1, ..., P\} \) and relation \( R_1(1,2) \), there is a well-defined (possibly empty) contiguous subset \( R_{1,p} \) consisting of all rows with sort column values in the interval \([v_1, v_2]\). Shipping each of the \( R_{1,p} \) over to a single processor for final merging and joining results in an independent task \( \tau_{i_1, i_2} \) of the total remaining part of the join operation. (The superscript here underscores the fact that a single processor is involved. The significance of this will become apparent shortly.) Assume that we can estimate the time it takes to perform this task, as we shall do in Section 3.4.

Given \( v_1 \leq v_2 \), precisely one of two special cases may occur: Either \( v_1 < v_2 \), or \( v_1 = v_2 \). We call a pair \((v_1, v_2)\) satisfying \( v_1 < v_2 \) a type 1 pair. In the case where \( v_1 = v_2 \), the join output is the cross-product of the two inputs. We shall call a pair \((v_1, v_2)\) satisfying \( v_1 = v_2 \) a type 2 pair. Actually, for type 2 pairs, say with \( v = v_1 = v_2 \), we may wish to consider the additional possibility of partitioning...
one of the two sets $U_{j,<a\lambda}$ and $U_{j,a\lambda}$ as evenly as possible into $MULT$ sets, where $1 \leq MULT \leq n$, and creating still finer independent tasks $\tau_{\mu, \nu}$ of essentially equal task times. In task $\tau_{\mu, \nu}$, $m(1, \ldots, MULT)$, the cross-product of one of the sets and the $m$th partition of the other set is performed on a single processor. Exactly which of the two sets should be partitioned depends on the type of the task from which it arises.

Now we can state our general approach. Suppose we create a sequence of $N$ pairs of values with corresponding multiplicities in the domain of the join columns. This sequence will have the form $\nu_1 \leq \nu_2 < \ldots < \nu_{m+1} \leq \nu_{m+2} < \nu_{m+3} < \nu_{m+4} < \nu_{m+5} < \ldots < \nu_{2m} < \nu_{2m+1}$. Each value in the join columns of $R_1$ and $R_2$ is required to fall within one of the intervals $[\nu_j, \nu_{j+1})$. For $m(1, \ldots, N)$, let $MULT_i$ denote the multiplicity of the pair $(\nu_i, \nu_j)$. We have created $N = \sum MULT_i$ independent tasks $\tau_{\mu, \nu}$ with times $TIME_{\mu, \nu}$'s to be done at the $P$ processors. The total computing time involved can therefore be estimated as $\sum \sum TIME_{\mu, \nu} = \sum \sum MULT_i TIME_{\mu, \nu}$, which we wish to distribute as evenly as possible among the processors. (A "perfect" assignment, not necessarily possible, would have each processor busy for $\sum \sum TIME_{\mu, \nu}$ (P units of time.) Specifically, we would like to assign each task $\tau_{\mu, \nu}$ to a processor $ASSIGN(\tau_{\mu, \nu})$ in such a way that the completion time of the total job,

$$\max_{1 \leq \mu \leq P} \sum_{\nu=1}^{MULT} \sum_{\nu=1}^{TIME_{\mu, \nu}}$$

is minimized. This optimization problem is essentially the so-called minimum makespan or multiprocessor scheduling problem. Although it is known to be NP-complete, there exist a number of very fast heuristic algorithms (LPT, which stands for longest processing time first, due to Graham [GRAH69], and MULTIFIT, due to Coffman, Garey and Johnson [COFF78]) which have reasonably good worst-case performance and excellent average-case performance. For concreteness, we have adopted the LPT algorithm. For completeness, we describe the LPT algorithm in Section 3.1.

The point is that we have control over how the sequence of pairs of values and corresponding multiplicities is created. The goal in the scheduling phase of the proposed algorithm is to create this sequence via a divide-and-conquer approach.

Specifically, we will use an algorithm due to Galil and Megiddo [GAL79] to split large tasks of type 1 into two to three tasks, at least one of which is of type 2. This algorithm, henceforth known as GM, was originally designed to deal with a somewhat different problem. In that context, it was intended to find an $I$th smallest element in an $I \times J$ matrix whose columns were monotone non-decreasing. This problem is known in the literature as the selection problem. It is useful in separable convex resource allocation problems.

In fact, suppose $\nu_1 < \nu_2$ and that $\nu_{i-1,j}$ is a type 1 task. Let $\mu$ denote the median element of the union of both join columns which have values in the range $[\nu_1, \nu_2]$. The GM algorithm will divide each set $\nu_j \cup \nu_{j+1}$ into three contiguous (possibly empty) regions $- \rho_{\mu, \nu_{j+1}, \nu_2}$ consisting of rows with values less than $\mu$; $\rho_{\mu, \nu_{j+1}, \nu_2}$ consisting of rows with values equal to $\mu$; and $\rho_{\mu, \nu_{j+1}, \nu_2}$ consisting of rows with values greater than $\mu$. Thus GM creates three tasks where there had been one before. The first or third task (but not both) might be empty. Either one could be of type 1 or type 2. The second task will not be empty, and will be of type 2. As we shall see in Section 4, large single skew elements are quite likely to be created as "second tasks" during an application of the GM algorithm. (There exists a theoretically faster algorithm, due to Frederickson and Johnson [FRED82, FRED84], for the selection problem. Unfortunately, from our point of view it suffers from two deficiencies: First, it does not automatically provide the three-region partition, which we require. Second,
it does not appear to parallelize easily. In contrast, the Galil and Megiddo algorithm parallels naturally to \( P \) processors, each handling the two sets of sorted runs they created in the first place. We remark, however, that the Frederickson and Johnson algorithm may be the algorithm of choice for handling skews in sorting. (The precise three-region partitioning required for correctness in joining is not needed for sorting.) See [IYER89] for yet another selection problem algorithm applied to that particular problem.) [IBAR88] contains good descriptions of the GM and other selection problem algorithms, but for completeness, and because of the slightly different use to which we put it, we shall describe GM in Section 3.2. See also [TANT88] for a computer science application of a generalization of the selection problem.

Algorithmic descriptions and notes on LPT appear in Section 3.1. Section 3.2 handles GM. Section 3.3 deals with the proposed scheduling algorithm itself, labeled SK EW. SK EW works by repeatedly switching back and forth between LPT and GM. In Section 3.4 we deal with task time estimation.

3.1. LPT:
Procedure: LPT
Input: Number of processors \( P \), number of tasks \( \hat{N} \), and task times \( (TIME_{i}, \forall i = 1, \ldots, \hat{N}) \).
Output: A heuristic assignment of the tasks to the processors which approximately minimizes the makespan.

Sort the tasks (if necessary) in order of decreasing \( TIME_{i} \).
Set \( TOTAL_{p} = 0 \) for each processor \( p \).
Do for \( n = 1 \) to \( \hat{N} \):

Assign task \( n \) to the processor \( p \) for which \( TOTAL_{p} \) is minimum. (Ties are decided in favor of smaller \( p \).)
Add \( TIME_{n} \) to \( TOTAL_{p} \).
End do.
End LPT

Notes on LPT:
- The makespan in the algorithm is represented by \( \max_P TOTAL_{p} \).
- Considerable work has been done on analyzing the worst-case behavior of LPT (and MULTI FIT). We reiterate that the worst-case and average-case behaviors are far apart, worst-case behavior being good, and average-case behavior being excellent.
- The computational complexity of LPT is \( O(\hat{N}\log \hat{N} + \log P) \). The presumably dominant term, \( \hat{N} \log \hat{N} \), comes from the sorting step, for which we employ QUICKSORT. See [AHO74].

3.2. GM:
Procedure: GM
Input: For each \( j(1, \ldots, J) \), a column \( \{a_{i,j}| \forall i = 1, \ldots, \hat{N}, \hat{N} \leq \hat{N} \} \) of non-decreasing elements, where \( \hat{N} \leq \hat{N} \) are indices of the column ranges under consideration.
Output: An \( 1/2 \)th smallest element \( \eta \), and, for each \( j(1, \ldots, J) \), a partition of each range \( \{TOP_{j}, \hat{N}, BOT_{j} \} \) into three new ranges, \( \{TOP_{j}, \hat{N}, BOT_{j} \} \) with values less than \( \eta \), \( \{TOP_{j}, \hat{N}, BOT_{j} \} \) with values equal to \( \eta \), and \( \{TOP_{j}, \hat{N}, BOT_{j} \} \) with values greater than \( \eta \).

Set \( T_{T,j} = TOP_{j} \), and \( B_{H,j} = BOT_{j} \) for each \( j(1, \ldots, J) \). Set \( \hat{i} = 1 \).
Do forever:

Set \( M_{j} = B_{H,j} - T_{T,j} + 1 \) for each \( j(1, \ldots, J) \). Set \( S = \sum_{j} M_{j} \).
For each \( j(1, \ldots, J) \), find the median element \( \eta_{j} \) of the set \( \{a_{i,j}| \forall i \leq \hat{N} \} \). Sort the medians in non-decreasing order, so that \( \eta_{j} \leq \eta_{j+1} \). Compute the value \( k \) such that \( \sum_{i=1}^{k} M_{j} < \frac{S}{2} \) and \( \sum_{i=k+1}^{\hat{i}} M_{j} \geq \frac{S}{2} \). Set \( \eta = \eta_{k} \).
Compute for each \( j(1, \ldots, J) \):
\( T_{T,j} = \min \{ \{ \forall i | TOP_{j} \leq i \leq \hat{N}, BOT_{j} \} \times (a_{i,j} = \eta) \} \), and
\( B_{H,j} = \max \{ \{ \forall i | TOP_{j} \leq i \leq \hat{N}, BOT_{j} \} \times (a_{i,j} = \eta) \} \). Set
\( M_{j}^{B} = \sum_{i=1}^{k} (T_{T,j} - T_{T,j}) \) and \( M_{j}^{H} = \sum_{i=k}^{\hat{i}} (B_{H,j} - T_{T,j} + 1) \).

If \( M_{j}^{B} \leq \hat{i} \leq M_{j}^{H} \) then begin
\( \eta \) is an \( 1/2 \)th smallest element. Set \( TOP_{j} = TOP_{j} \),
\( BOT_{j} = T_{T,j} - 1, TOP_{j} = T_{T,j} \), \( BOT_{j} = B_{H,j} \),
\( TOP_{j} = BB_{j} + 1, \) and \( BOT_{j} = BOT_{j} \) for each \( j(1, \ldots, J) \).
Halt.
End
If \( M_{j}^{B} \geq \hat{i} \) then set \( B_{H,j} = T_{T,j} \) for each \( j(1, \ldots, J) \).
If \( M_{j}^{H} \leq \hat{i} \) then decrement \( \hat{i} \) by \( M_{j}^{H} \) and set \( T_{T,j} = B_{H,j} + 1 \) for each \( j(1, \ldots, J) \).
End do.
End GM

Notes on GM:
- We will always apply GM in the case where \( J = 2P \), twice the number of processors. We will always be looking for the median element.
- For ease of exposition, we have purposely ignored the details of cases in which a column or region therein is (or becomes) empty. The details are somewhat messy, and not essential to understanding the algorithm.
- The \( TT_{j} \) and \( BB_{j} \) values be found by binary search. This can be done in parallel by each of the \( P \) processors. The median elements \( \eta_{j} \) can also be found in parallel.
We are basically following [IBAR88], but with the following modification: When \( S \leq J \), which will occur at some point during the execution of the algorithm, [IBAR88] employs a linear time selection algorithm due to [BLUM72] (and also found in [AH074]) to finish the job. (This particular algorithm is linear in the total number of elements involved, so is not of interest until that number is quite small. Employing it at that point reduces the computational complexity of the entire problem.) However, we soldier on without it, since we also need the three region partition.

[IBAR88] notes that the computation of the value \( k \) can be done without explicitly sorting the medians \( q_i \). Actually, we adopt this improvement as well, but omit it from the description for simplicity. See [IBAR88] for details. With that improvement, [IBAR88] obtains a (serial) computational complexity for GM of \( O(I(\log P)) \).

Precisely one of the three "if" conditions at the end of the algorithm must hold. GM itself works by dividing and conquering.

3.3. SKEW:

Procedure: SKEW

Input: Number of processors \( P \), \( 2P \) sets of sorted runs, \( \{s_{p,r}(i) = 1,...,\text{CARD}_r \} \), one for each processor \( p \in \{1,...,P\} \) and each relation \( r (1,2) \), where \( \text{CARD}_r \) is the cardinality of the sorted run of relation \( r \) at processor \( p \), and \( s_{p,r}(i) \) is the \( i \)-th tuple in this sorted run.

Output: The creation of tasks and a heuristic assignment of those tasks to the processors which approximately minimizes the makespan.

Set the number of tasks \( N = 1 \).

Set the top and bottom of the first task to be \( \text{TOP}_{p,r} = 1 \) and \( \text{BOT}_{p,r} = \text{CARD}_r \), for each processor \( p = 1,\ldots,P \) and each relation \( r = 1,2 \).

Determine the type (1 or 2) of the first task.

Do forever.

Determine the optimal multiplicities \( \text{MULT}_n \) of each type 2 task \( m(1,...,\text{\#\text{tasks}}) \). (Set \( \text{MULT}_n = 1 \) for each type 1 task \( m(1,...,\text{\#\text{tasks}}) \).) Compute the total number of tasks to be \( \bar{N} = \sum \text{MULT}_n \).

Compute the task times \( \text{TIME}_n = \text{MULT}_n \cdot \text{TIME} \).

If \( \bar{N} \geq P \) then apply LPT.

If [\text{solution is unacceptable}]

Then begin

Apply GM to find the median element of the region \( \{ \text{TOP}_{p,r}, \text{BOT}_{p,r}, 1 \leq p \leq P, r = 1,2 \} \) of the largest type 1 task \( n \).

The median element corresponds to a type 2 task with region \( \{ \text{TOP}_{p,r}, \text{BOT}_{p,r}, 1 \leq p \leq P, r = 1,2 \} \). Relabel this new type 2 task as task number \( n \). Determine its optimal multiplicity \( \text{MULT}_n \) and task time \( \text{TIME}_n \).

There also exist (1 or) 2 tasks, most likely of type 1, corresponding to regions \( \{ \text{TOP}_{p,r}, \text{BOT}_{p,r}, 1 \leq p \leq P, r = 1,2 \} \) and \( \{ \text{TOP}_{p,r}, \text{BOT}_{p,r}, 1 \leq p \leq P, r = 1,2 \} \) Increment \( N \) (by 1 or 2) to add these tasks and their optimal multiplicities and task times.

Sort the tasks in order of decreasing task times, so that \( n \leq n_i \) implies \( \text{TIME}_n \geq \text{TIME}_{n_i} \).

End

Else halt with solution from final LPT.

End

End SKEW
Notes on SKEW:

- In the extremely likely event that the first task created is of type 1, the task would correspond to performing the entire join phase on a single processor. If it is of type 2 instead, then the entire join is the join of a single element, so that we are forming a full cross-product of the rows of the two relations. The optimal multiplicity in this case will be determined to be 2, and the algorithm will halt with an essentially perfect solution.

- In general, the optimal multiplicity for a type 2 task n will be that \( m \) with \( 1 \leq m \leq P \) and task time \( T(n) \) for the smallest total time \( m T(n) \) subject to the constraints that \( T(n) \leq (m T(n) + REST)/P \), where \( REST \) is the combined time of all other tasks, and that \( T(n) \geq MINTIME \), where \( MINTIME \), an input variable, is the largest size task which SKEW is not allowed to subdivide. The first constraint has the effect of requiring \( m \) to be greater than some minimum value, while the second constraint has the opposite effect. The first constraint ensures that each individual task must be within one of the \( P \) processors. \( MINTIME \) is used to guard against splitting tasks too finely. (We do not model task initiation times explicitly, but by properly setting \( MINTIME \), we have the same effect. In fact, the algorithm could be made to "throw away" the smallest tasks it creates, by coalescing them with one of their neighbors.) Since the question of whether a type 2 task fits or not depends on the multiplicities of the other type 2 tasks, we cycle through the type 2 tasks in order of size, determining optimal multiplicities, and then repeat the process until the multiplicities remain stable throughout a complete cycle.

- The solution can be unacceptable for several reasons. The most obvious is that the quality of the LPT solution is not within some input variable TOLERANCE. (If \( SOL_{LPT} \) denotes the makespan of the LPT solution, and \( SOL_{EXACT} \) denotes the makespan of a "perfect" solution, the quality of the LPT solution will be acceptable if \( (SOL_{LPT} - SOL_{EXACT})/SOL_{LPT} < TOLERANCE \).) However, the following reasons for failure are also valid: First, it may happen early on in the algorithm that \( \hat{N} < P \), in which case LPT is not even called. Second, it may happen that the time \( T(n) \) of the largest type 1 task \( n \) may satisfy \( T(n) \leq MINTIME \). Finally, it may happen that the number \( \hat{N} \) of tasks already created may satisfy \( \hat{N} \geq MAXT \), where \( MAXT \) is an input variable designed to keep the algorithm from running too long. Generally, setting \( MAXT \) to be on the order of 10 times the number of processors proves quite satisfactory.

- We again use QUICKSORT to perform the sorting.

Figure 3.1 shows a type 1 task being subdivided into three new tasks. The entire 2P sets of sorted runs are shown, with the old type 1 task labeled with Is, 2s and 3s. The new type 2 task corresponding to the median element of the old task is labeled with 2s. The other two new tasks are labeled with 1s and 3s, respectively. These latter two tasks may be of type 1, in which case they may be candidates for subdivision themselves at some further point in the algorithm.

3.4. Task time estimation:

In this section we derive the task time estimation formulas. To begin with, assume that we have a type 2 task \( T(n) \), of multiplicity \( MULT \). (The formulas to handle type 1 tasks will be based on the type 2 formula.) Let \( K_1 \) and \( K_2 \) denote the sizes (measured in blocks) of the two sets of tuples in relations \( R_1 \) and \( R_2 \), respectively, which correspond to the value \( v \). For ease of exposition, let us assume that \( K_1 \) is larger than the two. (The formulas will merely need to be switched if the reverse is true.) Suppose that \( S \) is the memory buffer size (also in blocks) for each processor.

We can either split \( K_1 \) into \( MULT \) equal parts, or we can split \( K_2 \) into \( MULT \) equal parts. Let us label these as Methods 1 and 2, respectively. We will ultimately pick the method which gives the lowest task time. Whichever method we employ, we will let the larger component correspond to the outer loop, and the smaller component correspond to the inner loop. This is provably better than the reverse. The component corresponding to the outer loop will be allocated 1 block in the memory buffer, while the component corresponding to the inner loop will use the remaining \( S - 1 \) blocks. The blocks of the inner loop component cycle through the memory buffer once for each block of the outer loop component, in an alternating forwards and backwards manner. (This approach might accordingly be dubbed the ZIGZAG algorithm.) We thus utilize the memory in a way which minimizes the total number of blocks that need to be read. Let \( \gamma = \gamma K_1 \). By our convention, \( 0 \leq \gamma \leq 1 \).

Method 1: In this case, it is not apparent which of the two values, \( K_1 \) or \( K_2 \), is larger. So we let \( \min = \min(\gamma K_1) \), and \( \max = \max(\gamma K_2) \). A simple analysis then yields a time per processor of...

\[
A[\gamma MULT + K_1^2 + 2 \max K_1 - \max K_1 S + S - 1] + B MULT K_2^2
\]

if \( \min K_1 \geq S - 1 \), and a time per processor of...

\[
A[\gamma MULT + K_1^2 + B MULT K_2^2]
\]

otherwise. Here, \( A \) is a coefficient which equals the per block pathlength overhead of reading in the data, extracting the tuples, merging the sorted runs, and performing the join comparison. \( B \) is a coefficient which equals the pathlength overhead of inserting the output tuples generated by joining one block of tuples (with identical join column values) from each of relations \( R_1 \) and \( R_2 \) into an output file and writing out the data. The second expression corresponds to...
the case where the smaller component fits in the memory buffer, while the first expression corresponds to the case where it does not.

**Method 2**: In this case, it is clear that $\frac{K_i}{\text{MULT}} = \frac{y}{\text{MULT}} K_i \leq K_i$. We thus obtain a time per processor of...

$$A \left[ \frac{y}{\text{MULT}} K_i^2 + 2K_i - K_i(S + S - 1) + B \frac{y}{\text{MULT}} K_i^2 \right]$$

if $\frac{y}{\text{MULT}} K_i \geq S - 1$, and a time per processor of...

$$A \frac{y}{\text{MULT}} K_i + B \frac{y}{\text{MULT}} K_i^2$$

otherwise. Again, the second expression corresponds to the case where the smaller component, in this case $\frac{y}{\text{MULT}} K_i$, fits, while the first expression corresponds to the case where it does not.

Figure 3.2 shows the graph of total task time (which is MULT times the per processor task time corresponding to the lower cost method) as a function of MULT, for a typical choice of parameters. The first local minimum occurs at the point where the smaller component in Method 2 starts to fit in memory. The second local minimum occurs at the point where the smaller component in Method 1 starts to fit in memory. This always happens in exactly the same order, since $\frac{1}{\text{MULT}} \leq \min(1/\text{MULT}, y)$. Method 2 is the method of choice for the smaller multiplicities, and Method 1 is the method of choice for the larger ones. In any event, note the very gradual rise in total task time past each of the local minima. This is because the output term, which is quadratic and identical for both methods, heavily dominates the input term. In fact, note from the y-axis in Figure 3.2 that the difference between the highest total time (which occurs at MULT = 1) and the lowest (at MULT = 5) is less than 2%.

Now we turn to the case of type 1 tasks $v_1 \cdots v_2$. Here we make a fairly simplistic estimate that the individual elements $(v_1 \leq v_2)$ are uniformly distributed over the $D_{v_1 \cdots v_2}$ elements in the underlying domain. Let $K_{v_1}$ and $K_{v_2}$ denote the sizes (measured in blocks) of the two sets of tuples in relations $R_1$ and $R_2$, respectively, which correspond to the range $[v_1, v_2]$. Then we simply estimate that $K_{v_1 \cdots v_2} = \frac{D_{v_1 \cdots v_2}}{D_{v_1}}$ is approximately the size (again in blocks) of that part of relation 2 corresponding to $v$. Applying the type 2 formulas to each such $v$ within the range $[v_1, v_2]$, and summing across all such $v$ gives an estimate of the cost of the type 1 task.

It is important to note that the exact methodology employed to analyze the task times is completely orthogonal to the rest of the paper. The SKEW algorithm will remain unaffected even if these formulas are modified. The method we use to estimate times for type 1 tasks can obviously be improved upon, but as we shall see in the next section, even this simple estimation technique yields good results.

4. **Speedup of the Join Phase**:

In this section we examine the speedup during the join phase that can be obtained by using the proposed scheduling algorithm. We consider the case where the join phase is CPU-bound. While we do not consider it here, the scheduling method can be modified to handle the case where the join phase is I/O bound. We also assume a single query environment (i.e., we do not attempt to optimize the average running time of a number of concurrent queries.) For illustrative purposes, we use synthetic join column values distributed according to the Zipf-like distributions that were described in Section 2. We vary the degree of skew of the relations to be joined and the number of processors involved. We have also examined the effect of different correlations between the skew values in the two relations, different memory buffer sizes, and different sizes of the two relations, and obtained similar results to those reported here.

The parameter values chosen for this illustrative comparison are as follows: The relations $R_1$ and $R_2$ to be joined each have one million tuples. There are ten thousand distinct values ($D = 10000$) in the domain for the join column values of relations $R_1$ and $R_2$, having Zipf-like distributions with parameters $\theta_1$ and $\theta_2$, respectively. Each of the sorted runs and the output of the final join is assumed to have $X = 50$ tuples per 4K block of data. The size of the memory buffer used for buffering tuples during the join phase is taken as 512 K bytes. The overhead of reading in the data, extracting the tuple, merging the sorted runs, and performing the join comparison is assumed to consume 1 unit of CPU time (i.e., per block overhead $A = X$ in Section 3). The join overhead for each output tuple generated, including inserting the results into an output file and writing out the data to disk is also assumed to consume 1 unit of CPU time (i.e., per block overhead $B = X$ in Section 3).
overhead $b = \lambda^2$ in Section 3). We vary the number of processors from 1 to 128, and use combinations of $B$ values of 0 (pure Zipf for the highly skewed case), 0.5 (moderate skew) and 1 (uniform).

Finally, the correlation between the specific skewed values in the two relations is modeled as follows: The $D$ distinct values of relation $R_1$ are arranged in descending order of the number of tuples that have this value in their join column. The correlation is modeled using a single parameter $C$ that takes on integer values from 1 to $D$. Then, corresponding to the descending ordering of relation $R_1$, the value in $R_2$ with the largest number of tuples is placed in a position chosen randomly from 1 to $C$. The next most frequent value of $R_1$ is placed in a randomly chosen position from 1 to $C+1$, except that the position occupied by the previous step is not allowed, and so on. Thus $C = 1$ corresponds to perfect correlation, and $C = D$ to the random case. We choose $C = 50$ for our comparisons, which corresponds to a moderate to high degree of correlation. Preliminary examination of potential join columns in some actual databases supports such a degree of correlation.

We compare the speedups obtained using the proposed algorithm with two heuristics. In the first heuristic, the number of distinct values in the join column values is divided into $P$ range partitions, each with (approximately) the same number of distinct values, and each partition is assigned to one of the $P$ processors. Then tuples from each relation are shipped to the assigned processor and are merged and joined in the final phase. Intuitively, merely dividing the distinct values without regard to the number of tuples with the same value can be expected to lead to poor speedups when the data is highly skewed. For the speedups using this heuristic reported below, the values are idealistic because perfect knowledge of the distinct values in the join column is used in the assignment. In an implementation the heuristic could be approximated by equally dividing the range between the minimum and maximum values of each relation, as determined during the sort phase. Another simple scheme that approximates this heuristic is to use a (uniform) hash partitioning that assigns a distinct value to a particular processor depending on the result of a hash function of the value [DEW87]. However, the results show that even with perfect information, this (naive) heuristic has poor performance in the presence of moderate to high data skew.

The second heuristic used for comparison purposes partitions the two relations into $P$ ranges such that the sum of the number of tuples in each range from the union of both relations to be joined is (approximately) $1/Pm$ the total number of tuples in both relations, and each range is assigned to one of the $P$ processors. The range partitioning can be done using the parallelized version of the Galil-Megiddo algorithm outlined in Section 3. This method is similar to that in [AKL87] in the context of merging two lists, where an algorithm is proposed that breaks each of the two lists into two ranges such that the sum of the number of elements in the first range of the two lists is half the sum of the total number of elements in both lists. A similar algorithm is proposed in [IYER88].

A word about the methodology employed in computing the speedup is in order. We are using the "actual" makespan of the SKEW algorithm rather than the estimated makespan of Section 3. This means that we plug in the actual distributions of values into the formulas of Section 3, but we employ the methods (1 or 2) determined by our estimates, whether they be right or wrong. The actual task times for type 1 elements will thus be higher, in general, than the times obtained if full knowledge of the distribution of values were known beforehand.

We first consider the case where both the relations have highly skewed distributions of values in the join column. This case corresponds to $\theta_1 = \theta_2 = 0$, i.e., pure Zipf distributions for both relations. Figure 4.1 shows the number of processors versus the speedup of the join phase for this case for the proposed scheme and the two heuristics outlined above. In this context, the speedup is the ratio of the CPU time to complete the join phase on one processor to the (actual makespan of the) time on $P$ processors. The figure shows a close to linear speedup for the proposed algorithm, and small speedup for both the heuristics. The same data is displayed as a normalized speedup in Figure 4.2. The normalized speedup for the join phase is defined as the ratio of the speedup to the number of processors. Therefore, a normalized speedup of unity represents the ideal case of perfect speedup. The figure shows that for the proposed scheme, the normalized speedup is for the most part greater than 0.9, and usually close to unity. Virtually the entire reason for the departure of the normalized speedup from unity is the difference between the estimated CPU run time for a task and the actual run time. (LPT never gave a bad solution to the minimum makespan problem.)
discrepancy, in turn, is caused by our simplistic assumption of a uniform distribution within a range in estimating the time for the join operations in a type 1 task. As described in Section 3, a stopping condition for the algorithm is the creation of a fixed number of tasks per processor. Therefore, for a small number of processors, the number of iterations in the algorithm is small. There are some type 1 tasks that have a sizable skew, giving rise to a discrepancy between the estimated and actual run times. As the number of processors increases, so do the number of iterations. Therefore, the estimates of run times get better, leading to better speedups. We expect that better methods of estimating the times of a type 1 task will further improve the speedups using the proposed method. Improved estimation techniques have been devised and will be reported on later. This argument is supported by the bar chart in Figure 4.3. To understand this chart, suppose that all the potential type 2 pairs are ordered by task times. Then Figure 4.3 shows, for each number of processors, the number of the largest potential type 2 pairs that were identified by the algorithm and assigned to separate tasks before a miss occurs—in other words, the next largest potential type 2 pair was not identified, and occurred as part of a type 1 task instead. The chart shows that the number of large type 2 pairs created by the algorithm increases quickly as the number of processors increases. Therefore, the estimates for the run time of the remaining type 1 tasks improves with the number of processors. Notice from Figure 4.2 the sudden improvement in the normalized speedup in going from 8 to 16 processors. The reason for this behaviour can be seen from Figure 4.3, where the number of the largest type 2 pairs assigned separate tasks increases from 6 to 19 with this change. This leads to a significantly better estimate for the type 1 tasks, and therefore to the large improvement in the normalized speedup. The multiplicities of the five largest type 2 pairs as a function of the number of processors is shown in the bar chart of Figure 4.4. Note that while the multiplicities increase with the number of processors, they are still much smaller than the total number of processors available.

Returning to Figure 4.2, we note that the heuristics do poorly for this case because of the high data skew. For the first heuristic, some partitions have a disproportionately large number of tuples, leading to long run times for the processor that is assigned the partition. This effect becomes worse as the number of processors increases because the run time becomes dominated by a few partitions. For the second heuristic, though the number of tuples in each partition is the same, the join output begins to dominate for processors that are assigned the large skew values. Eventually, the largest skew elements determine the makespan of this heuristic, and the speedup converges to that of the first heuristic.

Figure 4.5 shows the normalized speedup for the three algorithms for the case of a high skew on relation \( R_1 (\theta_1 = 0) \) and a medium skew on relation \( R_2 (\theta_2 = 0.5) \). Again, the normalized speedup for the proposed scheme is close to 0.9 for up to 128 processors, for the same reasons as given above. The first heuristic shows no improvement over the previous case. The second heuristic shows some improve-
The normalized speedup for a moderate skew in both relations \((\theta_1 = \theta_2 = 0.5)\) is shown in Figure 4.7. The speedup for the proposed algorithm is slightly worse than in the previous cases. This is likely to be dependent on the specific run, and is not expected to be a general trend. Nevertheless, we expect to be able to improve on these results with better type 1 task time estimation techniques as mentioned earlier. The two heuristics show considerable improvement as compared to the previous cases. The explanation is that the dominating effect of the highly skewed values of the previous cases is now ameliorated. However, for a large number of processors, the proposed scheme still has about twice the speedup of either heuristic.

In Figure 4.8 we examine the normalized speedup for the case of moderate skew for relation \(R_1\) \((\theta_1 = 0.5)\) and uniform distribution for...
relation \( R_y \). For this single (low) skew case, the speedup from the proposed scheme is almost ideal, and the two heuristics are much improved. Even so, the speedup of the proposed scheme for 128 processors is about twice that of the first heuristic, and about a third better than the second heuristic.

Finally, we should point out that even our "actual" task times are really stochastic rather than deterministic in nature. Recent studies in [LAKS89a,b] have shown that the speedup achievable through horizontal growth can be quite sensitive to such variations in task times. One simple way to deal with this issue within the general context of our proposed algorithm is to force the processors to execute their tasks in the same order (largest first) in which they were assigned by LPT. During the course of the join phase the processors could report their progress. If the quality of the SKEW solution degrades past a certain predetermined threshold a new LPT algorithm could be initiated to handle the tasks remaining. Obviously, one would have to modify the timing of the transfer phase somewhat to allow for such a scheme. Slightly more elaborate approaches could also be devised. For example, the type 2 tasks are more deterministic than the type 1 tasks, so some of the smaller of these type 2 tasks might be executed last.

5. Summary:

Conventional parallel join algorithms perform poorly in the presence of data skew. In this paper, we propose a parallel sort merge join algorithm which can effectively handle the data skew problem. The proposed algorithm introduces a scheduling phase in addition to the usual sort, transfer and join phases. During the scheduling phase, a parallelizable optimization algorithm, using the output of the sort phase, attempts to balance the load across the multiple processors in the subsequent join phase. Two basic optimization techniques are employed repeatedly. One solves the selection problem, while the other heuristically solves the minimum makespan problem. Our approach naturally identifies the largest skew elements and assigns each of them to an optimal number of processors. The algorithm is demonstrated to achieve very good load balancing for the join phase in a CPU-bound environment and to be very robust relative to the degree of data skew and the number of processors. A Zipf-like distribution is used to model the data skew. Although we assume that the system is CPU-bound and thus that a CPU pathlength estimate be used in the objective function, we expect other environments can be similarly handled merely by employing a different objective function.

A hash join version of our parallel join algorithm has also been devised [WOLF90]. We are in the process of testing our algorithm against real databases. We have also devised improved type 1 task estimation techniques, and are examining methods to speed up the scheduling phase. Relatively small changes to our current approach substantially enhance the speedups and reduce the overheads due to SKEW, and will be reported on later.

References:


114


