# A Parallel Sort Merge Join Algorithm for Managing Data Skew

Joel L. Wolf, *Member, IEEE*, Daniel M. Dias, *Member, IEEE*, and Philip S. Yu, *Senior Member, IEEE*

*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, low cost 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, and is shown to be very robust relative, among other things, to the degree of data skew and the total number of processors.**

*Index Terms*—**Algorithms, databases, data skew, joins, optimization, parallel processing, scheduling.**

## I. 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 [6], [11], [13], [19], [30], [31], [37]. 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 [3], [14], [32], [33], [37] 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 [7], [28], [29]. [28] notes, for example, that for many textual databases the data distribution follows a variant of Zipf's Law [41]. Similar distributions for real databases are reported in [38]. This nonuniformity is referred to as data skew [27]. It is inherent in the data itself and does not depend on the access pattern. [27] 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 [34] 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 our 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, in examining the effects of data skew, they consider the case of 8 processors. We will show that the effects of data skew become 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 [4] and hash join [12], [24], methods are popular algorithms for computing the equi-join of two relations. In this paper we examine the sort merge join method

Manuscript received May 11, 1990; revised February 18, 1991.
The authors are with IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.
IEEE Log Number 9202068.

and propose an effective way to deal with the data skew problem. In a typical parallel sort merge join, e.g., [21], 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 partitioning algorithm. Finally, in the join phase, the sorted ranges are merged and joined. Each processor handles its own range of data. But such a conventional parallel join algorithm does not capture the effects of skew distribution in the join column, and as indicated, the impact to performance can be devastating [27].

The proposed parallel join algorithm uses a divide-and-conquer approach to address the data skew problem. An extra (low overhead) 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 [17]. The algorithm, henceforth labeled GM, is used to divide up the tasks. (The selection algorithm was originally used to find an $I$th smallest element in an $I \times J$ matrix whose columns are nondecreasing. 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 [18]. 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 initially assume that the system is CPU-bound, so that a CPU pathlength estimate is used in the objective function. We then outline the differences for the I/O-bound case.

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 examined, being very robust relative, among other things, 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 II. The scheduling phase algorithm is presented in Section III. In Section IV, a sensitivity analysis is provided to demonstrate the robustness of the algorithm and the speedup in the join phase over conventional algorithms. Finally, in Section V, we summarize our results. A hash join version of our parallel join algorithm has also been devised [38].

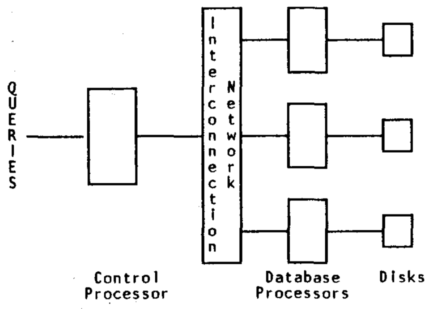


Fig. 1. Data partitioning architecture.

## II. 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 [35], illustrated in Fig. 1. The architecture has also been referred to as the *data partitioning* architecture [10]. In this system, there are two sets of processors—database processors and a control processor. 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 processor interfaces to users and also sends 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 the *shared everything* architecture, we do not address this aspect in the paper.

The parallel sort merge join algorithm that we consider is broadly similar to that in [21], and consists of four phases. In the first phase, each processor $p$ locally sorts its own partition $R_{1,p}$ and $R_{2,p}$ of the relations $R_1$ and $R_2$, respectively, and places the resulting sorted run on its local disks. This sort phase could be done using an optimal tournament tree external sort as in [21]. 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 III. 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 (RBA's) 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. This task is assigned to merge ranges (or a single value) from both relations. These ranges can be read at each processor, shipped to the assigned processor, merged with the corresponding ranges from the other processors, and joined, all in one pass through the data. The penalty of such a scheme is that each task now has a setup overhead of starting the range reads on all the other processors. The need for additional data buffers and synchronization delays in reading data from different processors increase the complexity of such an optimization. In a data sharing environment, all processors have direct access to the disks. Therefore, the transfer phase can be eliminated for such an architecture.

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 [25] as follows: We assume that the domain of the join column has $D$ distinct values. 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 $p_i = c/i^{(1-\theta)}$, where $c = 1/\sum_{i=1}^{D}(1/i^{(1-\theta)})$ 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 medium skew. The Zipf-like distributions corresponding to $D = 100$ and $\theta = 0, 0.25, 0.5, 0.75$ and 1 are shown in Fig. 2. In [28], data from large bibliographic databases are used to support models of skewed column value distributions based on Zipf-like distributions. See also [38].

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 partition. For most skew distributions and numbers of processors, such range partitioning will lead to good balance.

## III. 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 $2P$ sorted runs created during the sort phase, for example the one corresponding to processor $p \epsilon \{1, \cdots, P\}$ and relation $r \epsilon \{1, 2\}$, there is a well-defined (possibly empty) contiguous subset $\rho_{p,r,v_1,v_2}$ consisting of all rows with sort column values in the interval $[v_1, v_2]$. Shipping each of the $\rho_{p,r,v_1,v_2}$ over to a single processor for final merging and joining results in an independent task $\tau^1_{v_1,v_2}$ 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 III-D.

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 shall 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 just 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 $\cup_{p=1}^{P} \rho_{p,1,v,v}$ and $\cup_{p=1}^{P} \rho_{p,2,v,v}$ as evenly as possible into *MULT* sets, where $1 \leq MULT \leq P$, and creating still finer independent tasks $\tau^1_{v,v}, \cdots, \tau^{MULT}_{v,v}$ of essentially equal task times. In task $\tau^m_{v,v}, m \epsilon \{1, \cdots, MULT\}$, 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 time estimate of each alternative, and will be deferred until Section III-D. The goal here is to avoid *swamping* one of the $P$ processors with too much work. We do not insist that each of the *MULT* tasks be performed on a different processor, though in practice this is likely to be the case. Performing $MULT > 1$ tasks would, on its surface, appear less efficient than performing one, since the input from one of the relations must be shipped to each of the processors involved. If this were the case, we would want to make use of this approach only to handle excessive skew. However, as will be seen, the amount of additional work created may be counterbalanced by savings due to available processor memory, and will in all likelihood be negligible compared to the overall task time. We will say that the type 2 pair $(v, v)$ divided in this manner has *multiplicity MULT.* (A type 1 pair $(v_1, v_2)$ will be said to have multiplicity 1.) A task $\tau^m_{v_1,v_2}$ is said to have the same type and multiplicity as the pair $(v_1, v_2)$ 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 $v_{1,1} \leq v_{1,2} < \cdots < v_{n-1,1} \leq v_{n-1,2} < v_{n,1} \leq v_{n,2} < v_{n+1,1} \leq v_{n+1,2} < \cdots < v_{N,1} \leq v_{N,2}$. Each value in the join columns of $R_1$ and $R_2$ is required to fall within one of the intervals $[v_{n,1}, v_{n,2}]$. For $n \epsilon \{1, \cdots, N\}$, let $MULT_n$ denote the multiplicity of the pair $(v_{n,1}, v_{n,2})$. We have created $\hat{N} = \sum_{n=1}^{N} MULT_n$ independent tasks $\tau^m_{v_{n,1},v_{n,2}}$ with times $TIME^m_{v_{n,1},v_{n,2}}$ to be done at the $P$ processors. The total computing time involved can therefore be estimated as $\sum_{n=1}^{N} \sum_{m=1}^{MULT_n} TIME^m_{v_{n,1},v_{n,2}} = \sum_{n=1}^{N} MULT_n TIME^{MULT_n}_{v_{n,1},v_{n,2}}$, which we wish to distribute as evenly as possible among the processors. (A "perfect" assignment, not necessarily possible, would have each pro-

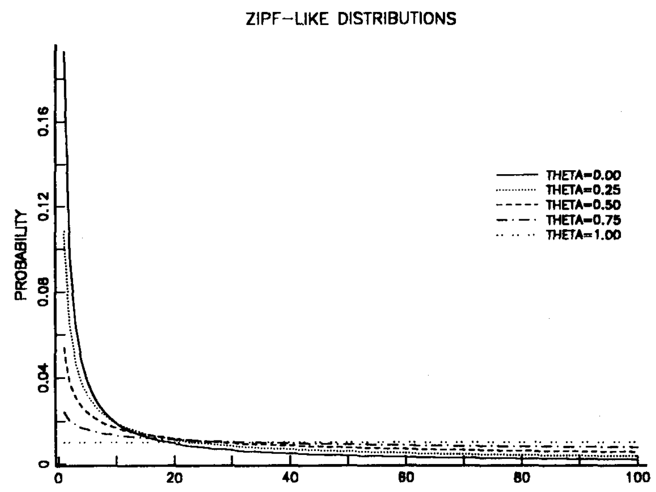


Fig. 2. Zipf-like distributions for various parameter choices.

cessor busy for $\sum_{n=1}^{N}\sum_{m=1}^{MULT_n} TIME^m_{v_{n,1},v_{n,2}}/P$ units of time.) Specifically, we would like to assign each task $\tau^m_{v_{n,1},v_{n,2}}$ to a processor $ASSIGN(\tau^m_{v_{n,1},v_{n,2}})$ in such a way that the completion time of the total job, or makespan, is minimized. The makespan corresponds to the maximum of the completion times for the individual processors, and thus is given by

$$\max_{1\le p\le P}\left\{\sum_{n=1}^{N}\sum_{\substack{m=1\\ ASSIGN(\tau^m_{v_{n,1},v_{n,2}})=p}}^{MULT_n} TIME^m_{v_{n,1},v_{n,2}}\right\}.$$

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 [18], and MULTIFIT, due to Coffman, Garey and Johnson [9] being prime examples) 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 III-A.

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 [17] 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 $I$ by $J$ matrix whose columns were monotone nondecreasing. This problem is known in the literature as the *selection* problem. It is useful in separable convex resource allocation problems.

In fact, suppose that $v_1 < v_2$, and $\tau^1_{v_1,v_2}$ is a type 1 task. Let $\mu$ denote the median element of the union of both join columns which have values in the range $[v_1, v_2]$. The GM algorithm will divide each set $\rho_{p,r,v_1,v_2}$ into three contiguous (possibly empty) regions—$\rho^1_{p,r,v_1,v_2}$, consisting of rows with values less than $\mu$; $\rho^2_{p,r,v_1,v_2}$, consisting of rows with values equal to $\mu$; and $\rho^3_{p,r,v_1,v_2}$, 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 IV, 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 [15], [16], for the selection problem. Unfortunately, from our point of view it suffers from two deficiencies: First, and most importantly, it does not automatically provide the three-region partition, which we require. Second, it does not appear to parallelize easily. In contrast, while the issue has not to our knowledge been studied in the literature, the Galil and Megiddo algorithm parallelizes naturally to $P$ processors, each handling the two sets of sorted runs they created in the first place. Communications between the processors and a central coordinator can be done in an obvious tree-like fashion, leading to a communications time logarithmic in the number of processors. 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 [22] for yet another selection problem algorithm applied to that particular problem. [20] 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 III-B. See also [36] for a computer science application of a generalization of the selection problem, and [39] for an application of the selection problem to nested block joins.

Algorithmic descriptions and notes on LPT appear in Section III-A. Section III-B handles GM. Section III-C deals with the proposed scheduling algorithm itself, labeled SKEW.

SKEW works by repeatedly switching back and forth between LPT and GM. In Section III-D we deal with task time estimation.

## A. LPT

```
Procedure: LPT
Input: Number of processors $P$, number of tasks $\hat{N}$, and task times $\{TIME_n \mid n = 1, \cdots, \hat{N}\}$.
Output: A heuristic assignment of the tasks to the processors which approximately minimizes the makespan.
    Sort the tasks (if necessary) in order of decreasing $TIME_n$.
    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 $\max_{1 \leq p \leq P} TOTAL_p$.
- Considerable work has been done on analyzing the worst-case behavior of LPT (and MULTIFIT). 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 [1].

## B. GM

```
Procedure: GM
Input: For each $j \epsilon \{1, \cdots J\}$, a column $\{\alpha_{i,j} \mid i = TOP_j, \cdots, BOT_j\}$ of nondecreasing elements, where $TOP_j \leq BOT_j$ are indexes of the column ranges under consideration. A number $I$ between 1 and the total number of elements $\sum_{j=1}^{J}(BOT_j - TOP_j + 1)$ in the columns.
Output: An $I$th smallest element $\eta$, and, for each $j \epsilon \{1, \cdots, J\}$, a partition of each range $\{TOP_j, \cdots, BOT_j\}$ into three new ranges, $\{TOP_j^1, \cdots, BOT_j^1\}$ with values less than $\eta$, $\{TOP_j^2, \cdots, BOT_j^2\}$ with values equal to $\eta$, and $\{TOP_j^3, \cdots, BOT_j^3\}$ with values greater than $\eta$.
    Set $T_j = TOP_j$ and $B_j = BOT_j$ for each $j \epsilon \{1, \cdots, J\}$.
Set $\hat{I} = I$.
    Do forever
        Set $M_j = B_j - T_j + 1$ for each $j \epsilon \{1, \cdots, J\}$. Set $S = \sum_{j=1}^{J} M_j$.
        For each $j \epsilon \{1, \cdots, J\}$, find the median element $\eta_j$ of the set $\{\alpha_{i,j} \mid i = T_j, \cdots, B_j\}$. Sort the medians in nondecreasing order, so that $\eta_{j_1} \leq \cdots \leq \eta_{j_J}$.
        Compute the value $k$ such that $\sum_{i=1}^{k-1} M_{j_i} < S/2$ and $\sum_{i=1}^{k} M_{j_i} \geq S/2$. Set $\eta = \eta_{j_k}$.
        Compute for each $j \epsilon \{1, \cdots, J\}$, $TT_j = \min\{i \mid (TOP_j \leq i \leq BOT_j) \wedge (\alpha_{i,j} = \eta)\}$, and
        $BB_j = \max\{i \mid (TOP_j \leq i \leq BOT_j) \wedge (\alpha_{i,j} = \eta)\}$.
        Set $M^1 = \sum_{j=1}^{J}(TT_j - T_j)$ and
        $M^2 = \sum_{j=1}^{J}(BB_j - T_j + 1)$.
        If $[M^1 < \hat{I} \leq M^2]$
        Then begin
            $\eta$ is an $I$th smallest element. Set $TOP_j^1 = TOP_j$,
            $BOT_j^1 = TT_j - 1, TOP_j^2 = TT_j$,
            $BOT_j^2 = BB_j, TOP_j^3 = BB_j + 1$, and
            $BOT_j^3 = BOT_j$ for each $j \epsilon \{1, \ldots J\}$. Halt.
        End
        If $[M^1 \geq \hat{I}]$ then set $B_j = TT_j$ for each $j \epsilon \{1, \cdots J\}$.
        If $[M^2 < \hat{I}]$ then decrement $\hat{I}$ by $M^2$ and set
        $T_j = BB_j + 1$ for each $j \epsilon \{1, \cdots J\}$.
    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. Thus, $I = \lceil \sum_{j=1}^{J}(BOT_j - TOP_j + 1)/2 \rceil$.
- 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 can 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 [20], but with the following modification: When $S \leq J$, which will occur at some point during the execution of the algorithm, [20] employs a linear time selection algorithm due to [5] (and also found in [1]) 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.
- [20] notes that the computation of the value $k$ can be done without explicitly sorting the medians $\eta_j$. Actually, we adopt this improvement as well, but omit it from the description for simplicity. See [20] for details. With that improvement, [20] obtains a (serial) computational complexity for GM of $O(I(\log J)^2)$.
- Precisely one of the three "if" conditions at the end of the algorithm must hold. GM itself works by dividing and conquering.

## C. SKEW

```
Procedure: SKEW
Input: Number of processors $P$, $2P$ sets of sorted runs, $\{\alpha_{i,p,r} \mid i = 1, \cdots, CARD_{p,r}\}$, one for each processor $p \epsilon \{1, \cdots, P\}$ and each relation $r \epsilon \{1, 2\}$, where $CARD_{p,r}$ is the cardinality of the sorted run of relation $r$ at processor $p$, and $\alpha_{i,p,r}$ 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 $TOP_{N,p,r} = 1$ and $BOT_{N,p,r} = CARD_{p,r}$ for each processor $p = 1, \cdots, P$ and each relation $r = 1, 2$.
Determine the type (1 or 2) of the first task.
Do forever
    Determine the optimal multiplicities $MULT_n$ of each type 2 task $n \epsilon \{1, \cdots, N\}$. (Set $MULT_n = 1$ for each type 1 task $n \epsilon \{1 \cdots, N\}$.) Compute the total number of tasks to be $\hat{N} = \sum_{n=1}^{N} MULT_n$.
    Compute the task times $\{TIME_n^{MULT_n} \mid n = 1, \cdots, N\}$.
    If $\hat{N} \geq P$ then apply LPT.
    If *[solution is unacceptable]*
    Then begin
        Apply GM to find the median element $\mu^{(n)}$ for the region $\{TOP_{n,p,r}, \cdots, BOT_{n,p,r} \mid p = 1, \cdots, P, r = 1, 2\}$ consisting of the largest type 1 task $n$.
        The median element corresponds to a type 2 task with region
        $\{TOP_{n,p,r}^2, \cdots, BOT_{n,p,r}^2 \mid p = 1, \cdots, P, r = 1, 2\}$
        Relabel this new type 2 task as task number $n$.
        Determine its optimal multiplicity $MULT_n$ and task time $TIME_n^{MULT_n}$.
        There also exist (1 or) 2 tasks, most likely of type 1, corresponding to regions
        $\{TOP_{n,p,r}^1, \cdots, BOT_{n,p,r}^1 \mid p = 1, \cdots, P, r = 1, 2\}$
        and $\{TOP_{n,p,r}^3, \cdots, BOT_{n,p,r}^3 \mid p = 1, \cdots, P, r = 1, 2\}$. 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 $n_1 \leq n_2$ implies $TIME_{n_1}^{MULT_{n_1}} \geq TIME_{n_2}^{MULT_{n_2}}$.
    End
    Else halt with solution from final LPT.
End do
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 $P$, 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 $TIME_n^m$ with the smallest total time $mTIME_n^m$ subject to the constraints that $TIME_n^m \leq (mTIME_n^m + REST)/P$, where $REST$ is the combined time of all other tasks, and that $TIME_n^m \geq MINTIME$, 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. Said differently, the first constraint ensures that each individual task must *fit* within one of the $P$ processors, while $MINTIME$ 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 $SOL_{PERFECT}$ denotes the makespan of a "perfect" solution, the quality of the LPT solution will be acceptable if $(SOL_{LPT} - SOL_{PERFECT})/SOL_{LPT} < TOLERANCE$.) 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 $TIME_n^1$ of the largest type 1 task $n$ may satisfy $TIME_n^1 \leq MINTIME$. 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, since it results in a modest number of tasks but is seldom the actual stopping criterion. Similarly, $MINTIME$ rarely results in stopping the algorithm. In contrast, a TOLERANCE of about 1% nearly always serves as the SKEW algorithm stopping criterion.
- We again use QUICKSORT to perform the sorting. Note that the cost of sorting in both LPT and SKEW itself is always quite modest, because of the incremental manner in which the sorts are performed. The pre-sorted items are nearly in order to begin with, since they are little changed from the output of the last sort iteration.

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

Fig. 4 shows four "snapshots" of the SKEW algorithm in operation. The first diagram shows a single type 1 task, corresponding to the entire set of sorted runs. The second diagram shows a type 2 task, labeled with 1's, being created by the first GM iteration, together with two new type 1 tasks. The third diagram shows a second iteration of GM, creating a

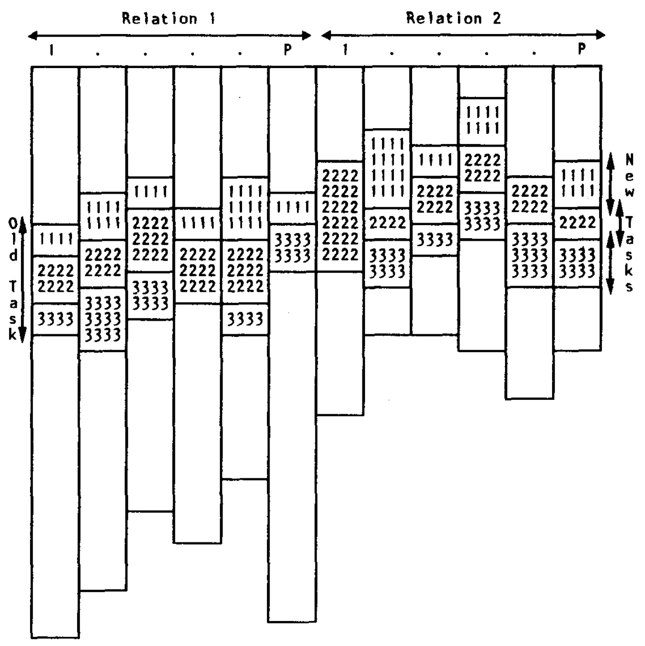


Fig. 3. Using GM to subdivide a type 1 task.

type 2 task (labeled with 2's) from one of the two new type 1 tasks. Similarly, the fourth diagram shows a third iteration of GM, creating a type 2 task (labeled with 3's) from the *other* of the type 1 tasks. At the end of this iteration, three type 2 tasks and four type 1 tasks have been created, and the process continues in this manner. Statistically, it is extremely likely that the largest skew elements will be discovered during one of these GM iterations. (We will give quantitative evidence to confirm this statement in Section IV.)

## D. Task Time Estimation

In this section we derive task time estimation formulas. For concreteness, we shall assume that the join phase is accomplished by an algorithm which we label ZIGZAG. This algorithm is broadly similar to the algorithm described in [23] in the context of nested block joins. We point out that the exact algorithm employed by the join phase is orthogonal to the rest of our paper. Similarly, we have modeled the ZIGZAG algorithm by formulas which we feel are at the appropriate level of detail, but this, too, is orthogonal to our main thrust. To handle different join phase algorithms or levels of modeling detail, one needs merely to replace our task time estimation formulas. The SKEW algorithm is otherwise unaffected.

We consider a CPU-bound environment first. To begin with, assume that we have a type 2 task $\tau^m_{v,v}$ 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 the larger of 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 block will use the remaining $S - 1$ blocks. We make the assumption that the blocks of the inner loop component cycle through the memory buffer once for each block of the outer loop component, in an alternatingly forward and backward 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 $\gamma = K_2/K_1$. By our convention, $0 \leq \gamma \leq 1$.

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

$$A[\frac{\gamma}{MULT}K_1^2 + 2\mathcal{M}K_1 - \mathcal{M}K_1S + S - 1] + B\frac{\gamma}{MULT}K_1^2 \tag{3.1}$$

if $mK_1 \geq S - 1$, and a time per processor of

$$A[\frac{1}{MULT} + \gamma]K_1 + B\frac{\gamma}{MULT}K_1^2 \tag{3.2}$$

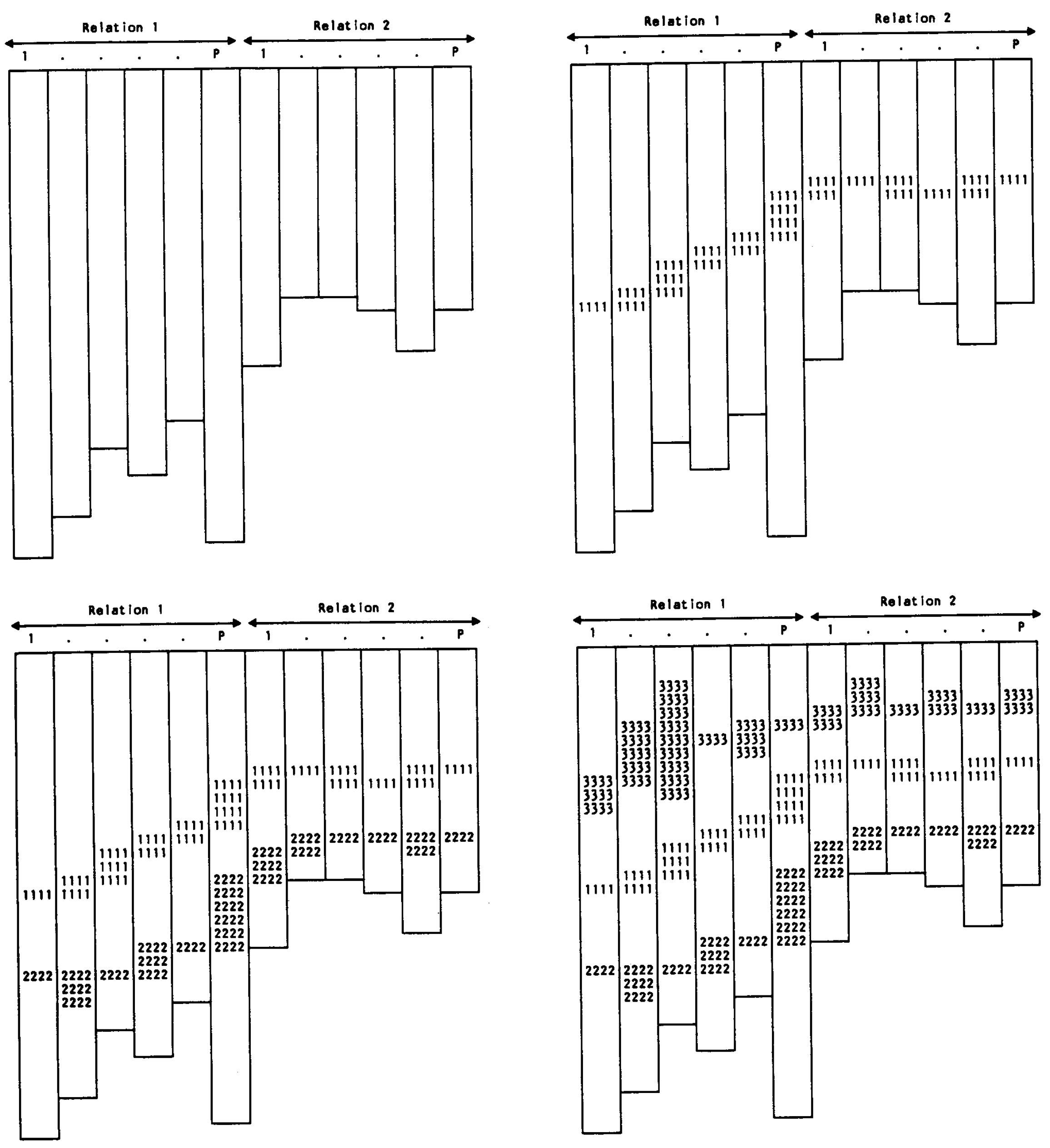


Fig. 4. SKEW algorithm snapshots.

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 $K_2/MULT = K_1\gamma/MULT \leq K_1$. We thus obtain a time per processor of

$$A[\frac{\gamma}{MULT}K_1^2 + 2K_1 - K_1S + S - 1] + B\frac{\gamma}{MULT}K_1^2 \quad (3.3)$$

if $K_1\gamma/MULT \geq S - 1$, and a time per processor of

$$A[\frac{\gamma}{MULT} + 1]K_1 + B\frac{\gamma}{MULT}K_1^2 \quad (3.4)$$

otherwise. Again, the second expression corresponds to the case where the smaller component, in this case $K_1\gamma/MULT$, fits, while the first expression corresponds to the case where it does not.

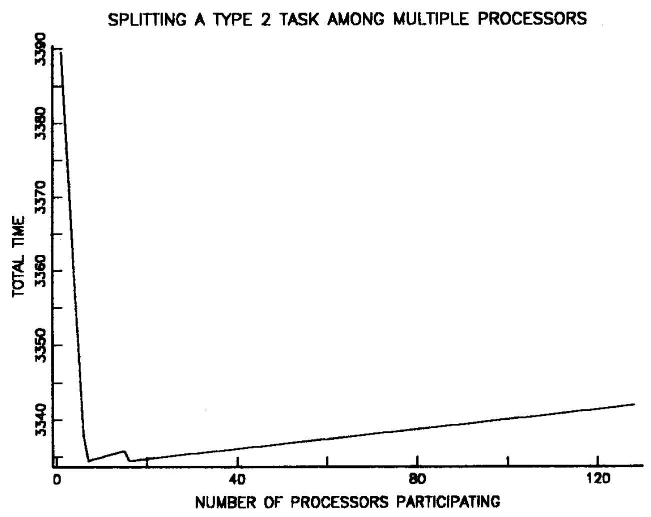


Fig. 5. Total task time as a function of multiplicity.

Fig. 5 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 $MULT$, in an illustrative example. (Different examples will yield graphs with qualitatively the same general structure.) 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 $\gamma/MULT \leq \min(1/MULT, \gamma)$. 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 Fig. 5 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 $\tau^1_{v_1,v_2}$. Here we make a fairly simplistic estimate that the individual elements $\{v \mid v_1 \leq v \leq v_2\}$ are uniformly distributed over the $D_{v_1,v_2}$ elements in the underlying domain. 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 range $[v_1, v_2]$. Then we simply estimate that $K_{r,v} = K_r/D_{v_1,v_2}$ is approximately the size (again in blocks) of that part of relation $r$ 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.

In the above analysis we have assumed that the join phase is CPU-bound. Note that many relational database systems implement an asynchronous prefetch, where the processing of a block is overlapped with the I/O for the next block. Thus, the above CPU-bound case corresponds to the situation where the time to process a block of each of relations $R_1$ and $R_2$ and write out the join result exceeds the corresponding I/O time. The converse is true for the I/O-bound case, which we deal with next. Several possibilities exist, depending on precisely how the two input relations and the output relation are handled by the processors' I/O subsystems. As an example, first assume that both the relations to be joined and the join output relation are placed on the same physical disk. Then the time to read a block of the two relations and to write the join output is the sum of the times for each of the individual I/O operations. Thus, (3.1) through (3.4) continue to hold, though with a change in the interpretation of the constants $A$ and $B$: The coefficient $A$ now represents the average I/O time to read a block of either relation $R_1$ or $R_2$, and, likewise, $B$ is now the average I/O time to write out all the blocks generated by joining one block of tuples (with identical join column values) from each of relations $R_1$ and $R_2$. As a second example, assume that the relations to be joined are on the same physical disk, but the join output relation is on a separate disk. In this case, the reading of the input relations can go on concurrently with the writing of the join output. Thus, to a good approximation, the join time becomes the maximum of the read and the write times. For example, (3.1) now becomes

$$\max(A[\frac{\gamma}{MULT}K_1^2 + 2\mathcal{M}K_1 - \mathcal{M}K_1S + S - 1], B\frac{\gamma}{MULT}K_1^2), \qquad (3.1)'$$

again with revised interpretation of $A$ and $B$. (This is an approximation to the extent that the first block of the input relations are read before the output of the join operation begins. Therefore the reads and the writes are not completely overlapped.) In a similar manner, (3.2) through (3.4) can be modified to be the maximum of the two terms. For completeness, we include them here:

$$\max(A[\frac{1}{MULT} + \gamma]K_1, B\frac{\gamma}{MULT}K_1^2) \qquad (3.2)'$$

$$\max(A[\frac{\gamma}{MULT}K_1^2 + 2K_1 - K_1S + S - 1], B\frac{\gamma}{MULT}K_1^2) \qquad (3.3)'$$

$$\max(A[\frac{\gamma}{MULT} + 1]K_1, B\frac{\gamma}{MULT}K_1^2). \qquad (3.4)'$$

As indicated, we have used an implementation of a ZIGZAG algorithm similar but not identical to that introduced in [23] in the context of nested block joins. Implementation of that particular variant would result in modestly different replacement equations for (3.1)–(3.4) and (3.1)′–(3.4)′. Similarly, it is not essential that a ZIGZAG algorithm be employed. In fact, while slightly more efficient, ZIGZAG is also somewhat more difficult to implement than a more straightforward algorithm such as the (unidirectional) one described in [39]. The SKEW algorithm will not be affected.

Clearly, the time estimates for type 2 tasks will be excellent, since the SKEW algorithm has perfect knowledge of the number of input tuples involved. The (uniform distribution) method we use to estimate times for type 1 tasks is obviously less precise, but as we shall see in the next section, even this simple estimation technique yields good results.

## IV. Speedup of the Join Phase

In this section we examine the speedup during the join phase that can be obtained by using the proposed SKEW scheduling algorithm. We focus on the case where the join phase is CPU-bound, but will report on the I/O-bound case as well. We 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 II. We vary the degree of skew of the relations to be joined and the number of processors involved. We also vary the correlation between the skew values in the two relations and the number of distinct values.

The base case 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 ($D = 10\,000$) in the domain for the join column values of relations $R_1$ and $R_2$, having pure Zipf distributions (Zipf-like, with parameters $\theta_1 = \theta_2 = 0$, 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 III-D). 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 $B = X^2$ in Section III-D). 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 defined using a single parameter $C$ that takes on integer values from 1 to $D$. 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 $C = 500$, which corresponds to a moderate degree of correlation. Preliminary examination of potential join columns in some actual databases supports such a degree of correlation.

Starting with this base case, we examine the effects of varying, in turn, the values of

- $\theta_1$ and $\theta_2$, using combinations of $\theta$ values of 0 (pure Zipf for the highly skewed case), 0.5 (medium skew), and 1 (uniform, or zero skew). These examples are studied in the most detail.
- $C$, using values of $C = 100$ (high correlation) and $C = 2500$ (low correlation).
- $D$, using a value of $D = 100\,000$ distincts.

These examples apply in a CPU-bound environment, but, given different interpretations of the coefficients $A = B$, would apply also in an I/O-bound environment where input and output I/O is directed to the same disk. We also examine the base case example in an I/O-bound environment where input and output I/O are directed to different disks. For all examples, we vary the number of processors $P$ between 1 and 128 in powers of 2. We remark that we have also examined the effect of different memory buffer sizes, relation sizes, and so on. Results are similar to those reported here.

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 [14]. However, the results show that even with perfect information, this (naive) heuristic has poor performance in the presence of medium 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) $1/P$th 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–Meggido algorithm outlined in Section III-B. This method is similar to that in [2] in the

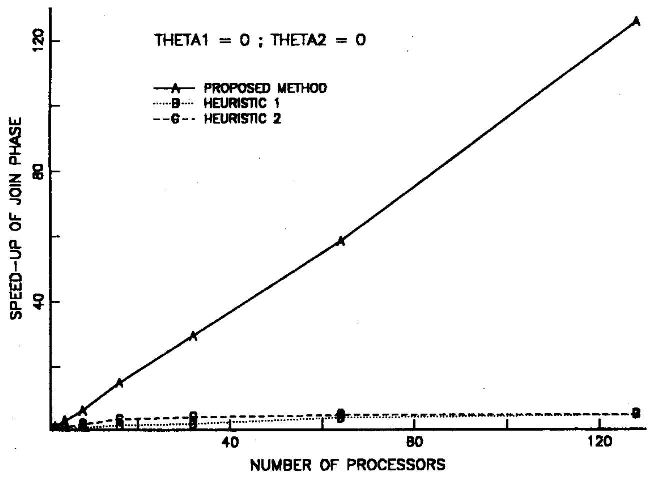


Fig. 6. Speedup of the join phase—high/high skew case.

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 [21].

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 III. This means that we plug in the *actual distributions* of values into the formulas of Section III, 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 base 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. Also, we have $C = 500$, $D = 10\,000$, and a CPU-bound environment. Fig. 6 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) time on $P$ processors. The figure shows a close to linear speedup for the proposed algorithm, and small speedup for both heuristics. The same data is displayed as a normalized speedup in Fig. 7. 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.) This discrepency, 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 III-C, 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 sizeable skew, giving rise to a discrepency 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. This argument is supported by the bar chart in Fig. 8. To understand this chart, suppose that all the *potential* type 2 pairs are ordered by task times. Then Fig. 8 shows, for each value of $P$, 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 Fig. 7 the sudden improvement in the normalized speedup in going from 8 to 16 processors. The reason for this behavior can be seen from Fig. 8, 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 Fig. 9. Note that while the multiplicities increase with the number of processors, they are still much smaller than the total number of processors available.

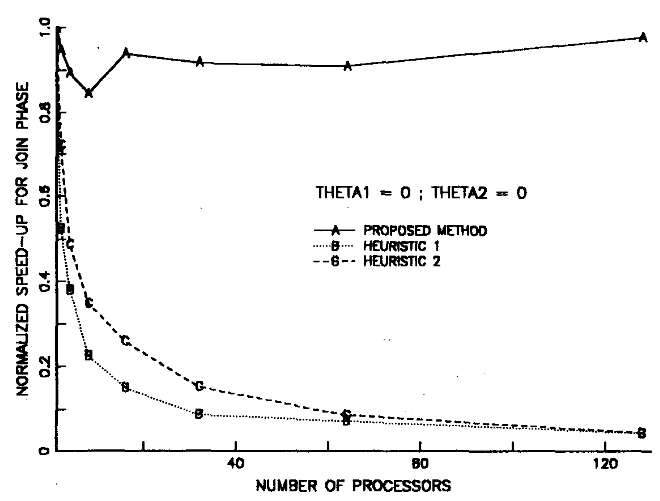


Fig. 7. Normalized speedup of the join phase—high/high skew case.

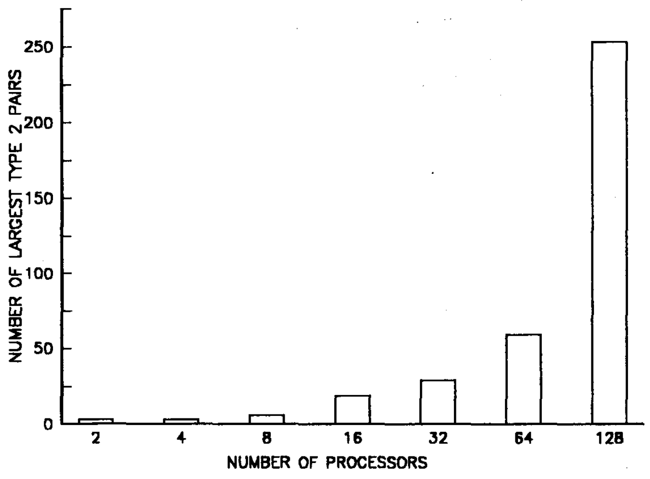


Fig. 8. Number of largest type 2 pairs assigned to tasks for the high/high skew case.

Returning to Fig. 7, 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.

Fig. 10 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 improvement, but continues to be an order of magnitude worse than the proposed scheme for a large number of processors.

We next examine the speedup for the case of a single skew. Fig. 11 shows the normalized speedup with high skew for relation $R_1$ ($\theta_1 = 0$) and a uniform distribution for relation $R_2$ ($\theta_2 = 1$). The speedup for the proposed algorithm now is close to ideal. This is because the estimates for the run times are now close to the actuals. Both heuristics, on the other hand, show marginal improvement over the previous case. The explanation is that for both heuristics the processors that are assigned the large skew element of relation $R_1$ dominate the join phase run time.

The normalized speedup for a medium skew in both relations ($\theta_1 = \theta_2 = 0.5$) is shown in Fig. 12. The speedup for the proposed algorithm is slightly worse than in the previous

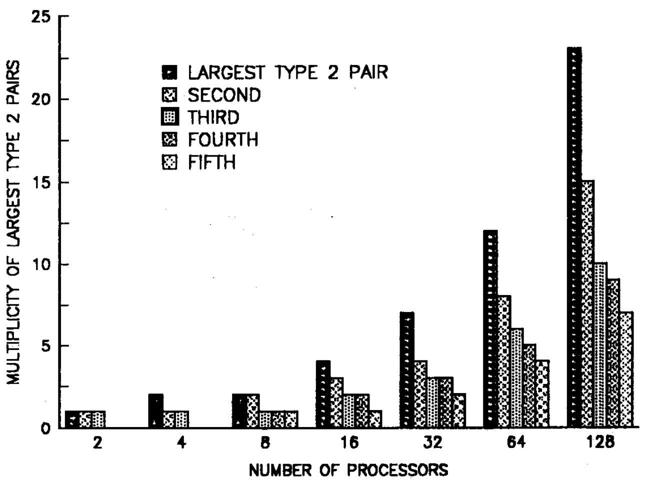


Fig. 9. Multiplicity of 5 largest type 2 pairs for the high/high skew case.

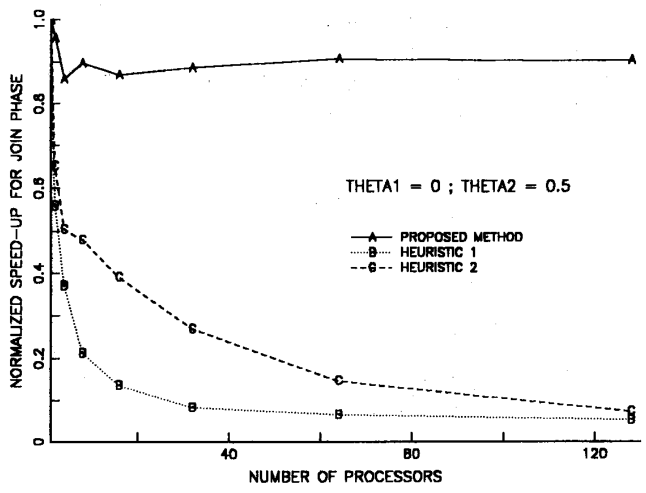


Fig. 10. Normalized speedup of the join phase—high/medium skew case.

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 Fig. 13 we examine the normalized speedup for the case of medium skew for relation $R_1$ ($\theta_1 = 0.5$) and uniform distribution for relation $R_2$ ($\theta_2 = 1$). 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.

As indicated, the curves described thus far all correspond to cases where the correlation $C = 500$, the number of distinct values $D = 10\,000$, and the system is CPU-bound. As also indicated, our algorithm is quite robust, performing well for every set of parameters we have examined. In Fig. 14 we show sample results for some of these parameter changes. Using the High/High skew case as a base, we vary $C$ to be 100 and 2500 in the top two graphs, labeled “high” and “low” correlation, respectively. We vary $D$ to be 100 000 in the bottom left graph. In the bottom right graph we examine the I/O-bound case with source and output relations on separate disks, so that (3.1)′–(3.4)′ apply. (We still assume that $A = B$, though the interpretation of these coefficients is now different.) Each of the four graphs is similar to the base case of Fig. 7. However, we do point out that the proposed algorithm performs better in the high correlation case than in the low correlation or

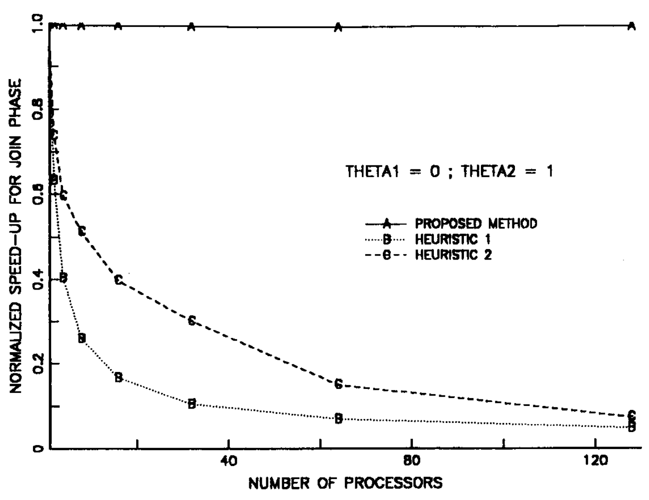


Fig. 11. Normalized speedup of the join phase—high/zero skew case.

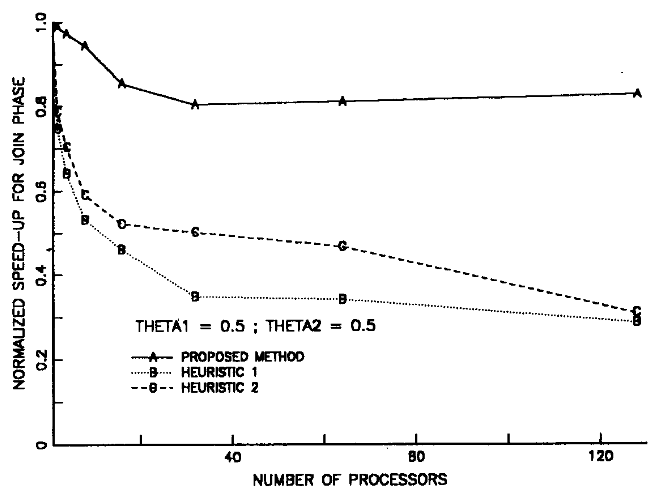


Fig. 12. Normalized speedup of the join phase—medium/medium skew case.

100 000 distincts cases, since the type 1 task time estimates will be better.

It is true, of course, that the total join phase costs under these various situations are quite different. In Fig. 15 we have plotted relative uniprocessor join phase costs for each of the 8 cases we have analyzed. (The I/O-bound case is not shown, since the $A$ and $B$ coefficients in (3.1)–(3.4) and (3.1)′–(3.4)′ are incompatible.) The first 5 are for our base case of Fig. 7 and the lesser skew scenarios of Figs. 10–13; these are labeled HH, HM, HZ, MM, and MZ, where H means "high," M means "medium," and Z means "zero" skew. The remaining 3 correspond to the first 3 cases in Fig. 14. These are labeled HIGH C and LOW C (for "correlation"), and HIGH D (for "distincts"). These uniprocessor join phase costs are largely dependent on the size of the corresponding output relations. (The largest cost corresponds to the HIGH C case, which has been normalized to 1.) Comparing the first 5 cases, one sees the effects of different skew parameters. Obviously, the larger the data skew, the more costly the join. Note that HZ and MZ are close in cost, since their output relations will be equal in size. Comparing HH with HIGH C and LOW C one sees the dramatic quadratic effect due to correlation. Comparing HH with HIGH D, one sees a lesser effect due to the number of distinct values. Fig. 15 shows that the relative *importance* of achieving near linear speedup in the join phase can vary considerably with different join relations.

We have pointed out that our type 2 task time formulas are probably very good, but not perfect, and that our type 1 task time estimates are somewhat less accurate. Recent studies in [27] have shown that the speedup achievable through horizontal growth can be quite sensitive to 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

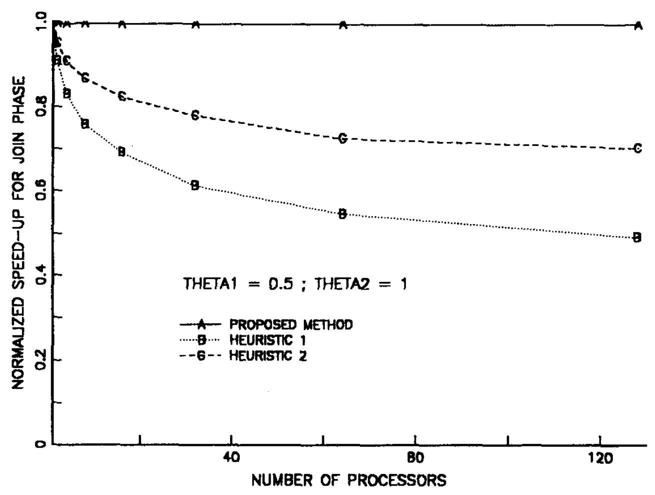


Fig. 13. Normalized speedup of the join phase—medium/zero skew case.

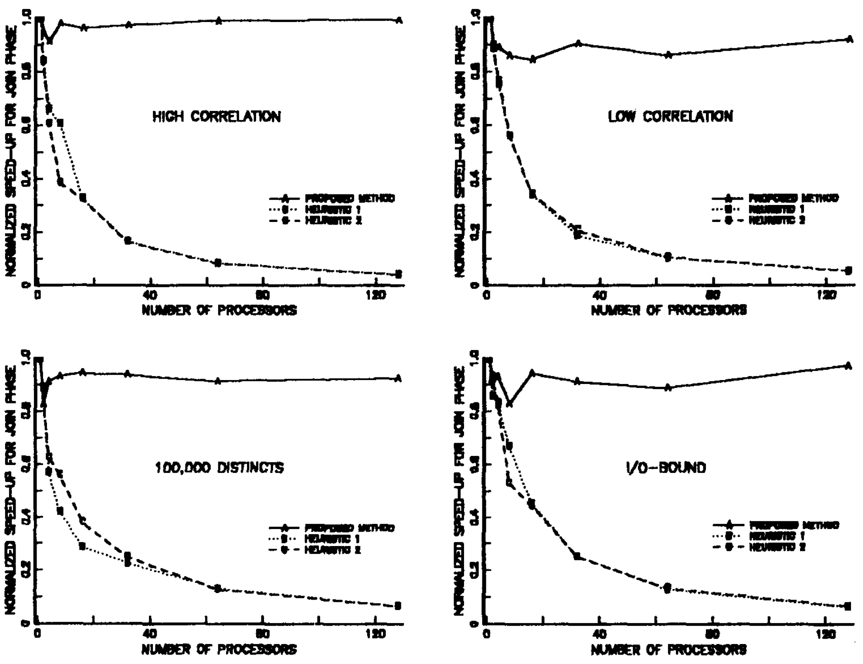


Fig. 14. Normalized speedup of the join phase—miscellaneous cases.

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.

Finally, we should comment that the cost of the scheduling phase algorithm we have proposed is, indeed, quite low relative to the potential join phase savings. To quantify this, we have written both code which carefully emulates the scheduling phase algorithm, and also an analytic SKEW overhead model based on instruction count estimates, number of LPT and GM runs, and so on. If, for example, we assume processor speeds of 10 mips, a disk access time of 25 ms per track, and communications via a shared token ring capable of transferring 100 mbits per second, the scheduling phase time

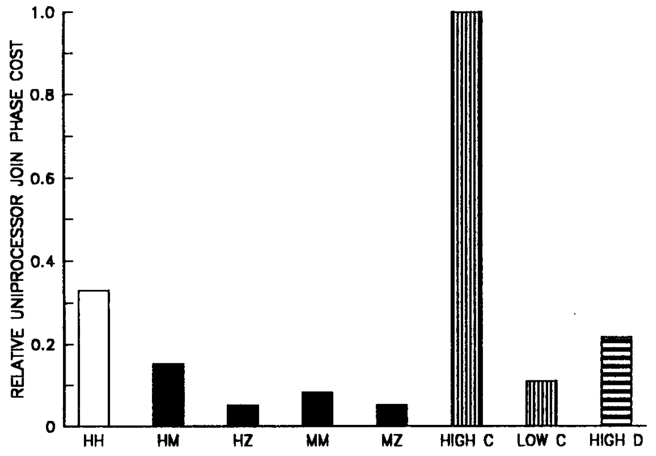


Fig. 15. Relative uniprocessor join phase cost.

is less than 1% of the SKEW join phase time for our base case with 128 processors.

## V. 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 low overhead 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 and to be very robust relative to the degree of data skew, the number of processors, and many other parameters. Our algorithm works well in either a CPU- or I/O-bound environment. A hash join version of our parallel join algorithm has also been devised [38], and a paper comparing these two approaches appears in [40].

## Acknowledgment

We thank G. Flatow of IBM Research for providing us access to real data quantifying the degree of data skew. We also thank the anonymous reviewers for many useful comments and suggestions.

## References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[2] S. G. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *IEEE Trans. Comput.*, vol. C-36, no. 11, pp. 1367–1369, 1987.

[3] L. Bic and R. L. Hartman, "Hither hundreds of processors in a database machine," in *Proc. 1985 Int. Workshop Database Machines*, Springer-Verlag, 1985.

[4] M. Blasgen and K. Eswaran, "Storage and access in relational databases," *IBM Syst. J.*, vol. 4, p. 363, 1977.

[5] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, pp. 448–461, 1972.

[6] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, A highly parallel database system," *IEEE Trans. Knowledge Data Eng.*, vol. 2, no. 1, pp. 4–24, Mar. 1990.

[7] S. Christodoulakis "Estimating record selectivities," *Inform. Syst.*, vol. 8, no. 2, pp. 105–115, 1983.

[8] E. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[9] E. Coffman, M. Garey, and D. S. Johnson, "An application of bin packing to multiprocessor scheduling," *SIAM J. Comput.*, vol. 7, pp. 1–17, 1978.

[10] D. W. Cornell, D. M. Dias, and P. S. Yu, "On multisystem coupling through function request shipping," *IEEE Trans. Software Eng.*, vol. SE-12, no. 10, pp. 1006–1017, 1986.

[11] S. A. Demurjian, D. K. Hsiao, D. S. Kerr, J. Menon, P. R. Strawser, R. C. Tekampe, J. Trimble, and R. J. Watson, "Performance evaluation of a database system in multiple backend configurations," in *Proc. 1985 Int. Workshop Database Machines*, Springer-Verlag, 1985.

[12] D. J. DeWitt and R. H. Gerber "Multiprocessor hash-based join algorithms," in *Proc. 11th Int. Conf. Very Large Databases*, 1985.

[13] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen, "The GAMMA database machine project," *IEEE Trans. Knowledge Data Eng.*, vol. 2, no. 1, pp. 44–62, Mar. 1990.

[14] D. J. DeWitt, M. Smith, and H. Boral, "A single-user performance evaluation of the Teradata database machine," MCC Tech. Rep. DB-081-87, 1987.

[15] G. Frederickson and D. B. Johnson "The complexity of selection and ranking in X+Y and matrices with sorted columns," *J. Comput. Syst. Sci.*, vol. 24, pp. 197–209, 1982.

[16] ——, "Generalized selection and ranking: Sorted matrices," *SIAM J. Comput.*, vol. 13, pp. 14–30, 1984.

[17] Z. Galil and N. Megiddo "A fast selection algorithm and the problem of optimum distribution of effort," *J. ACM*, vol. 26, pp. 58–64, 1979.

[18] R. Graham "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Mathemat.*, vol. 17, no. 2, pp. 416–429, 1969.

[19] D. K. Hsiao, *Advanced Database Machine Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

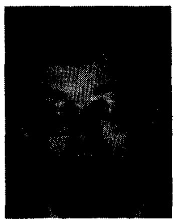
[20] T. Ibaraki and N. Katoh, *Resource Allocation Problems*. Cambridge, MA: M.I.T. Press, 1988.

[21] B. R. Iyer and D. M. Dias, "System issues in parallel sorting for database systems," in *Proc. 6th Int. Conf. Data Eng.*, 1988.

[22] B. R. Iyer, G. R. Ricard, and P. J. Varman, "Percentile finding algorithm for multiple sorted runs," in *Proc. 15th Int. Conf. Very Large Databases*, 1989.

[23] W. Kim, "A new way to compute the product and join of relations," in

*Proc. ACM SIGMOD Conf.*, Santa Monica, CA, 1980.
[24] M. Kitsuregawa, H. Tanaka, and T. Motooka, "Application of hash to data base machine and its architecture," *New Generation Comput.*, vol. 1, no. 1, 1983.
[25] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
[26] S. Lakshmi and P. S. Yu, "Effectiveness of parallel processing in database systems," *Comput. Syst. Sci. Eng.*, vol. 5, no. 2, pp. 73–81, 1990.
[27] ——, "Effectiveness of parallel joins," *IEEE Trans. Knowledge Data Eng.*, vol. 2, no. 4, pp. 410–424, 1990.
[28] C. A. Lynch, "Selectivity estimation and query optimization in large databases with highly skewed distributions of column values," in *Proc. 14th Int. Conf. Very Large Databases*, 1988.
[29] A. Y. Montgomery, D. J. D'Souza, and S. B. Lee, "The cost of relational algebraic operations on skewed data: Estimates and experiments," in *Inform. Processing 83*, IFIP, 1983.
[30] P. M. Neches and J. E. Shemer, "The genesis of a database computer," *IEEE Comput. Mag.*, vol. 17, no. 11, pp. 42–56, 1984.
[31] E. Ozkarahan, *Database Machines and Database Management*. Englewood Cliffs, NJ: Prentice-Hall, 1986.
[32] G. Z. Qadah, "The equi-join operation on a multiprocessor database machine: Algorithms and the evaluation of their performance," in *Proc. 1985 Int. Workshop Database Machines*, Springer-Verlag, 1985, pp. 35–67.
[33] S. Salza, M. Terranova, and P. Velardi, "Performance modeling of the DBMAC architecture," in *Proc. 1983 Int. Workshop Database Machines*, Springer-Verlag, 1983, pp. 74–90.
[34] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," in *Proc. ACM SIGMOD Conf.*, Portland, OR, 1989, pp. 110–121.
[35] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng.*, vol. 9, no. 1, 1986.
[36] A. N. Tantawi, D. Towsley, and J. Wolf, "Optimal allocation of multiple class resources in computer systems," in *Proc. ACM SIGMETRICS Conf.*, Santa Fe, NM, 1988, pp. 253–260.
[37] P. Valduriez and G. Gardarin, "Join and semi-join algorithms for a multiprocessor database machine," *ACM Trans. Database Syst.*, vol. 9, no. 1, pp. 133–161, 1984.
[38] J. L. Wolf, D. M. Dias, and P. S. Yu, "An effective algorithm for parallelizing hash joins in the presence of data skew," in *Proc. 7th Int. Data Eng. Conf.*, Kobe, Japan, pp. 200–209.
[39] J. L. Wolf, B. Iyer, K. Pattipati, and J. Turek, "Optimal buffer partitioning for the nested block join algorithm," in *Proc. 7th Int. Data Eng. Conf.*, Kobe, Japan, pp. 510–519.
[40] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek, "Comparative performance of parallel join algorithms," in *Proc. 1st Int. Conf. Parallel Distributed Syst.*, Miami, FL, 1991, pp. 78–88.
[41] G. K. Zipf, *Human Behavior and the Principle of Least Effort*. Reading, MA: Addison-Wesley, 1949.



**Joel L. Wolf** (M'91) received the Ph.D. degree from Brown University in 1973, and the Sc.B. degree from the Massachusetts Institute of Technology in 1968, both in mathematics.

He is currently a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY, with interest in mathematical optimization. He has also been an Assistant Professor of Mathematics at Harvard University, as well as a Distinguished Member of Technical Staff and manager at Bell Laboratories. He is the author of numerous papers and patents.

In 1988, he won an IBM Outstanding Innovation Award for his work on the Placement Optimization Program, a practical optimization technique to solve the Disk File Assignment Problem. He is a member of the Association for Computing Machinery and the Operations Research Society of America.

**Daniel M. Dias** (M'88) received the B.Tech. degree from the Indian Institute of Technology, Bombay, and the M.S. and Ph.D. degrees from Rice University, Houston, TX, all in electrical engineering.

He is a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. His current research interests include database systems, transactions and query processing, parallel and distributed systems, interconnection networks, and performance analysis.

**Philip S. Yu** (S'76–M'78–SM'87) received the B.S. degree in E.E. from National Taiwan University, Taipei, Taiwan, Republic of China, in 1972, the M.S. and Ph.D. degrees in E.E. from Stanford University, in 1976 and 1978, respectively, and the M.B.A. degree from New York University in 1982.

Since 1978 he has been with the IBM T. J. Watson Research Center, Yorktown Heights, NY. Currently, he is manager of the Architecture Analysis and Design group, which focuses on the design and analysis of clustering multiple processors for transaction and query processing. His current research interests include database management systems, parallel and distributed processing, computer architecture, performance modeling, and workload analysis. He has published more than 110 papers in refereed journals and conferences and has published over 70 research reports and 50 technical disclosures.

Dr. Yu is a member of the Association for Computing Machinery.