1998

Skew-Insensitive Parallel Algorithms for Relational Join

Khaled Alsabti
Syracuse University, School of Computer and Information Science, kaalsabt@top.cis.syr.edu

Sanjay Ranka
University of Florida, School of CISE, ranka@cise.ufl.edu

Skew-Insensitive Parallel Algorithms for Relational Join

Khaled Alsabti
School of CIS
Syracuse University
kaalsabt@top.cis.syr.edu

Sanjay Ranka
Department of CISE
University of Florida
ranka@cise.ufl.edu

Abstract

Join is the most important and expensive operation in relational databases. The parallel join operation is very sensitive to the presence of the data skew. In this paper, we present two new parallel join algorithms for coarse grained machines which work optimally in presence of arbitrary amount of data skew. The first algorithm is sort-based and the second is hash-based. Both of these algorithms employ a preprocessing phase (prior to the redistribution phase) to equally partition the work among the processors. The proposed algorithms have been designed for memory resident-data. However, they can be extended to disk resident-data. These algorithms are shown to be theoretically as well as practically scalable. Experimental results are provided on the IBM SP-2.

1 Introduction

Join is the most important and expensive operation in relation database [13]. Natural join (equ-join), the most popular form of join, of relation R on attribute x with relation S on attribute y is the set of all tuples t such that t is the concatenation of a tuple r belonging to R and a tuple s belonging to S and $r.x = s.y$. Parallel join has been a widely studied problem in the literature. Most of the parallel join algorithms are based on the uniprocessor join algorithms. The uniprocessor join algorithms can be categorized into three major paradigms: nested-loop, hash-based, and sort-based based [13]. Further, these algorithms can be roughly divided into two groups. One group of the algorithms is *skew-sensitive* where the performance significantly deteriorates with the presence of data skew, while the other group is *skew-insensitive* which alleviates the presence of data skew to some degree. Database research shows that the data skew exists in many real and realistic data sets [4, 16, 15].

For designing a scalable parallel algorithm, load balancing should be achieved and the overhead for achieving the load balancing should be low. The presence of the data skew affects the load balancing achieved by the conventional parallel join algorithms. It also affects the structure and the amount of the interprocessor communication. We present two new parallel join algorithms which work optimally in presence of arbitrary amount and any type of skew. The first algorithm is sort-based while the second is hash-based algorithm.

Both of these algorithms employ a preprocessing phase (prior to the redistribution phase) to equally partition the work among the processors using *perfect information* of the join attribute distribution. The cost of this preprocessing step is relatively small in case of uniform distribution. Further, it is shown to generate perfect or near-perfect load balancing for data sets with a varying degree of data skew. These algorithms are shown to be theoretically as well as practically scalable. We empirically compare our algorithms with two conventional join algorithms used for handling skew. Experimental results are provided on the IBM SP-2. Our algorithms are relatively architecture independent and are designed for memory-resident (in-core) data.

The proposed algorithms have been designed for memory resident-data. In the new generation of coarse grained machines, the main memory size can be as large as 1 GBytes/processor. For a 128 processor machine, the aggregate memory available can be as large as a few hundred gigabytes. This can accommodate relations of reasonable sizes in today relational database applications. Further, our algorithms can be easily extended for disk-resident relations.

The rest of this paper is organized as follows. Section 2 describes the parallel machine model and a set of communication primitives which form the building blocks for our algorithms. Sections 3 presents the notations and assumptions which are used in the following sections. Section 4 presents and analyses the conventional join algorithms in the absence of data skew. We review some of the proposed parallel join algorithms and discuss some of the important characteristics which are used in classifying these algorithms in section 5. Section 6 presents and analyses the two new algorithms. Experimental results are presented in section 7. Conclusions are presented in section 8.

2 Coarse-grained Parallel Machine

Coarse Grained Machines (CGMs) consist of a set of processors (tens to a few thousand) connected through an interconnection network. The memory is physically distributed across the processors. Interaction between processors is either through message passing or through a shared address space. Popular interconnection topologies are buses (SGI Challenge), 2D meshes (Paragon, Delta), 3D meshes (Cray T3D), hypercubes (nCUBE), fat tree (CM5) and hierarchical networks (cedar, DASH).

CGMs have cut-through routed networks which will be the primary thrust of this paper and will be used for modeling the communication cost of our algorithms. For a lightly loaded network, a message of size m traversing d hops of a cut-through (CT) routed network incurs a communication delay given by $T_{comm} = \tau + t_h d + \mu m$, where τ represents the handshaking costs, t_h represents the signal propagation and switching delays and μ represents the inverse bandwidth of the communication network. The startup time τ is often large, and can be several hundred machine cycles or more. The per-word transfer time μ is determined by the link bandwidth. μ is often higher (an order to two orders of magnitude is typical) than δ, the cost of a unit computation local to a processor. The per-hop component $t_h d$ can often be subsumed into the startup time τ without significant loss of accuracy. This is because the diameter of the network, which is the maximum of the distance between any pair of processors, is relatively small for most practical

sized machines, and t_h also tends to be small.

The above expressions adequately model communication time for lightly loaded networks. However, as the network becomes more congested, the finite network capacity becomes a bottleneck. Multiple messages attempting to traverse a particular link on the network are serialized. A good measure of the capacity of the network is its cross-section bandwidth (also referred to as the bisection width). For p processors, the bisection width is $p/2$, $2\sqrt{p}$, and 1 for a hypercube, wraparound mesh and for a shared bus respectively. Join algorithms are cross-section bandwidth limited. Hence, it is important to analyze these algorithms for cross-section bandwidth. We have performed our analysis for two popular interconnection networks: hypercubes and two dimensional meshes. The analysis for permutation networks and hypercubes is the same in most cases. These cover nearly all commercially available machines. A permutation network is one for which almost all of the permutations (each processor sending and receiving only one message of equal size) can be completed in nearly the same time (e.g. CM-5 and IBM SP Series).

Parallel Primitives Parallelization of applications requires distributing some or all of the data structures among the processors. Each processor needs to access all the non-local data required for its local computation. This generates aggregate or collective communication structures. Several algorithms have been described in the literature for these primitives and are part of standard textbooks [6, 12]. The use of collective communication provide a level of architecture independence in the algorithm design. It also allows for precise analysis of an algorithm by replacing the cost of the primitive for the targeted architecture.

In the following, we briefly describe some important parallel primitives that are repeatedly used in our algorithms and implementations. For commonly used primitives, we simply state the operation involved. The analysis of the running time is omitted. For other primitives, a more detailed explanation is provided. Table 1 describes the collective communication primitives used in the development of our algorithms and their communication time requirements on cut-through routed hypercubes and meshes. In what follows, p refers to the number of processors. We assume that sending a message form one node to another when no other traffic is present can be completed in $O(\tau + \mu m)$, where m is the size of the message.

The primitives we have used in this paper are as follows:

1. **All-to-All Broadcasting:** In all-to-all broadcast, every node has a message of size m to be sent to all other processors. For more details see [12].

2. **Global combine and prefix scans:** Each processor has a vector of size m. In the global-combine operation, an element-wise sum (or some other operation) is computed on the input vector such that the resultant vector will be stored on all the processors. Whereas in the global vector prefix-sum, an element-wise prefix-scan is used instead of the sum. For more details see [12].

3. **Transportation primitive:** It performs many-to-many personalized communication with possibly high variance in message size. Let r be the maximum of outgoing or incoming traffic at any processor The transportation primitive breaks down the communication into two all-to-all communication phases

where all the messages sent by any particular processor have uniform message sizes [23]. If $r \geq p^2$, the running time of this operation is equal to two all-to-all communication operations with a maximum message size of $O(\frac{r}{p})$. For more details see [23].

4. **Non-order maintaining data movement:** Each processor i has m_i elements. The objective is to redistribute the elements such that each processor will be assigned approximalety equal number of elements ($\overline{m}$). Let m be the maximum difference between the m_i and $\overline{m}$. For more detials see [1].

5. **Random access write:** Let M be the number of elements distributed across p processors. Each processor is initially assigned approximately $m = \frac{M}{p} elements$. In a Random Access Write (RAW) each of the M elements may need to write data to another element [17]. Each element has, in array P, the index of the element to which it has to send its data. It is possible to have collisions during a RAW. This happens when two or more data elements are written to the same destination. When collisions occur, one of the following [1] can be done: (i) choose one of the colliding values using a pre-defined rule (ii) combine the colliding data values using a pre-defined binary associative operator. In the following example of a RAW collisions are resolved using a binary associative operator (shown as a +).

Element index	0	1	2	3	4	5	6	7
Pointer P	7	.	0	7	1	6	3	0
After RAW	D(2)+D(7)	D(4)	-	D(6)	-	-	D(5)	D(0)+D(3)

   Details of the algorithm for n writes on an array of size n are given in [24, 3] .

6. **Merging two sorted lists:** Merging of globally sorted lists has been widely studied problem in the literature. We have chosen a merging algorithm presented in [19]. The size of first list R is N elements and the size of second list S is M. Each processor has approximately the same number of elements,$n = \frac{N}{p}$ and $m = \frac{M}{p}$, of lists R and S respectively. The computation time needed by the algorithm is $O(\delta(n+m))$ when n and m are sufficiently large.

7. **Parallel sort:** In the parallel sort algorithm, each processor initially has $m = \frac{M}{p}$ elements. The objective is to globally sort all the elements across all the processors such that each processor will be assigned approximately equal number of elements. There are several well-known algorithms for sorting on coarse grain parallel machines. We have chosen a parallel sampling-based sort for our problem [12]. The total computation time required is $O(\delta(m \lg m + p^2 \lg p + p \lg m + m \lg p))$. For $m \geq p^2$, this reduces to $O(\delta(m \lg m + p^2 \lg p))$. The communication time required is given in Table 1. For more details see [2].

[1] There are several other combining methods.

Primitive	Hypercube	Mesh (wraparound,square)
all-to-all Broadcast	$O(\tau \lg p + \mu m(p-1))$	$O(\tau(\sqrt{p}-1) + \mu m(p-1))$
Prefix-sum	$O(\tau \lg p + \mu m)$	$O(\tau\sqrt{p-1} + \ mum)$
Global-combine	$O(\tau \lg p + \mu m)$	$O(\tau(\sqrt{p}-1) + \mu m)$
Transportation	$O(\tau p + \mu m)$	$O(\tau + \mu m)\sqrt{p}$
Non-Order Maintaining Data Mov.	$O(\tau p + \mu m)$	$O((\tau + \mu m)\sqrt{p})$
RAW	$O(\tau p + \mu m)$	$O(\tau + \mu m)\sqrt{p})$
Circular q-shift	$O(\tau + \mu m)$	$O((\tau + \mu m)(\sqrt{p}+1))$
Merge two lists	$O(\tau p + \mu(m+n))$	$O((\tau + \mu(m+n))\sqrt{p})$
Sample sort	$O(\tau p + \mu(plog^2 p + m))$	$O(\tau\sqrt{p} + \mu(p^{1.5} + m\sqrt{p}))$

Table 1: Time complexity of communication time of the primitives on different interconnection networks

3 Notations and Assumptions

The following notations and assumptions are used for the presentation and the analysis of algorithms described in the next few sections:

- p - the number of processors in the system
- There are two relations R and S where R is the smaller relation
- n - the number of tuples in relation R
- t_R - the tuple size of relation R
- m - the number of tuples in relation S
- t_S - the tuple size of relation S
- $\frac{1}{\mu}$ - the data transfer rate
- τ - the communication start-up overhead
- h - the time taken by the hash function
- F - the fudge factor which is used in estimating the average number of comparisons required per tuple in probing the hash table. F is somewhat greater than 1 due to potential collisions
- δ - the cost of a unit computation local to a processor
- O_R - the largest ratio of the cumulative size of outgoing messages of relations R ($O_R \leq 1$)
- O_S - the largest ratio cumulative size of outgoing messages of relations S ($O_S \leq 1$)
- Q_R - the largest ratio of the relation R which is assigned to some processor to perform the join ($Q_R \leq 1$)

- Q_S - the largest ratio of the relation S which is assigned to some processor to perform the join ($Q_S \leq 1$)
- J - the total join output size produced by all the processors
- J_{max} - the maximum join output size produced by some processor

We assume that each processor has approximately $\frac{n}{p}$ and $\frac{m}{p}$ tuples of relations R and S, respectively. This is not necessarily realistic, especially if some other database operation is performed prior to the join operation. However, this nonuniform tuple distribution can always be handled by the non-order maintaining data movement primitive described in section 2.

In analyzing the join algorithms, we assume that the local partitions of the two relations at each node are memory resident. Since the total memory capacity in a large parallel system is expected to be high, reasonably large relation partitions can be accommodated in the main memory. Also, no CPU and communication overlap is considered.

4 Conventional Parallel Join Algorithms

In this section, we briefly present the parallelization of the three conventional join algorithms: *nested-loops*, *sort-based* and *hash-based*. All of the three algorithms are sensitive to the presence of the data skew, skew-sensitive algorithms. The main purpose of this section is to illustrate the communication costs inherent in the join algorithms independent of the data skew. Moreover, we analyze these algorithms under the assumption that no data skew is present; the join attribute has a uniform distribution.

Each parallel join algorithm consists of a global and local join method. The global join method refers to the implementation of the join operation across all the processors. The local join method refers to the join method which is used locally to carry out the join operation between the local fragments of both relations. Potentially, we can obtain nine different methods to implement the join operation using the three known join methods. We restrict ourselves to three algorithms in which the global and the local join methods are the same method.

Nested-Loops Algorithm In nested-loops algorithm each tuple of relation R is tested for a potential join with every tuple of relation S. Each processor joins its local fragment of relation R with relation S using the local join method. A simple parallelization can be achieved by circulating the S fragments (of size $t_S \frac{m}{p}$) $p-1$ times. The total memory requirement is $O(t_R \frac{n}{p} + t_m \frac{m}{p})$. One advantage of the nested-loops method is that it is not sensitive to the *redistribution* [2] skew.

The time requirement of the nested-loops algorithm is the sum of the time required by the local join algorithm and the inter-processors communication. The time required by local join is $O(\delta(\frac{n}{p}m + (t_R +$

[2] We will discuss different types of skew in the next section.

$t_S)J_{max}))$. Assuming that the tuples of both relations are randomly assigned among the processors, we expect that J_{max} to be approximately equal to $\frac{J}{p}$. In case this assumption is not valid, J_{max} can be as large as J. This means that one of the processors will perform almost the entire join process. For this reason, we will not consider this algorithm in the next section.

The time required by the communication is the cost of $p-1$ operations of circular 1-shift of size $t_S\frac{m}{p}$. The cost of circular 1-shift is given in Table 1. The total time requirements of the nested-loops algorithm is shown in Table 2 for the two interconnection networks.

Network	The time requirement of nested-loops
Hypercube	$O(\tau p + \mu t_S m + \delta(\frac{n}{p}m + (t_R + t_S)J_{max}))$
Mesh	$O((\tau p + \mu t_S m(\sqrt{p}+1) + \delta(\frac{n}{p}m + (t_R + t_S)J_{max}))$

Table 2: The total time requirements of the nested-loops algorithm on different interconnection networks

Sort-Based Algorithm We adopt the following version of the sort-based method. This algorithm consists of two phases: sorting and merging phases. The purpose of the first phase of the algorithm is to sort both relations $(R+S)$ as one big relation such that any tuple of processor i has a join attribute value less than the join attribute value of any tuple of processor j, where $i < j$. This phase is carried out by the sample sort algorithm which has been described in section 2.

As a result of the sorting phase, processor i will receive $O(\frac{n}{p})$ and $O(\frac{m}{p})$ tuples of relations R and S, respectively. The received tuples of relations R and S are independently merged to obtain sorted lists for the local fragments of both relations. Each processor then produces the join output by merging its local fragments of both relations.

Assuming that the join attribute in both relations has a uniform distribution, the time required by the sorting phase is given in Table 1 (m needs to be replaced by $\frac{t_R n + t_S m}{p}$) and the time required by the merge phase is $O(\delta(\frac{n+m}{p}\lg p + t_R\frac{n}{p} + t_S\frac{m}{p} + (t_R+t_S)J_{max}))$. The total time requirements of the sort-based algorithm is given in Table 3. We expect that J_{max} to be very close to $\frac{J}{p}$ with high probability. This algorithm is highly parallel since it achieved a good load-balancing (under the uniformity assumption) in two ways. Firstly, the join output is produced almost equally by all the processors. Secondly, the sizes of the local fragments (after sorting) are approximately equal.

However, the sort-based method is very sensitive to the data skew. The above algorithm will perform poorly in the presence of the data skew due to the following two reasons. Firstly, j_{max} might be as high as J and secondly the variant of the sizes of the local fragments (after sorting) can be very high.

Hash-Based Algorithm There is reasonable consensus that parallel hash-based algorithm is the most efficient algorithm for the join operation in case that the join attribute has a uniform distribution [13]. The

Network	Complexity of sort-based
Hypercube	$O(\tau p + \mu(plog^2 p + \frac{t_R n + t_S m}{p}) + \delta(\frac{m}{p}\lg\frac{m}{p} + \frac{n}{p}\lg\frac{n}{p} + \frac{n+m}{p}\lg p + t_R\frac{n}{p} + t_S\frac{m}{p} + (t_R + t_S)J_{max}))$
Mesh	$O(\tau\sqrt{p} + \mu(p^{1.5} + \frac{t_R n + t_S m}{\sqrt{p}}) + \delta(\frac{m}{p}\lg\frac{m}{p} + \frac{n}{p}\lg\frac{n}{p} + \frac{n+m}{p}\lg p + t_R\frac{n}{p} + t_S\frac{m}{p} + (t_R + t_S)J_{max}))$

Table 3: The time total requirements of the sort-based algorithm on different interconnection networks

hash-based has two phases: the partition and the join phases.

In the partition phase, each processor applies a common hash function on the join attribute values for its local fragments of relations R and S and determines the destination processors for the tuples based on predetermined assignment of the hash values into processors number. This step takes $O(\frac{n+m}{p}) \times (h + \delta)$ time. The expected partition (fragments) sizes of relations R and S are $\frac{n}{p}$ and $\frac{m}{p}$, respectively. Generating these partitions requires $O(\delta(t_R\frac{n}{p} + t_S\frac{m}{p}))$ of time. Each processor may have a set of tuples to send to every other processors. The communication phase can be performed by using the transportation primitive for the two relations with $O(t_R\frac{n}{p})$ and $O(t_S\frac{m}{p})$ as the maximum outgoing/incoming message sizes, respectively.

In the join phase, each processor builds a local hash table for its local fragment of one of the relations, i.e. R, using different hash function. Then, each processor probes its local hash table for its local fragment of the other relation, i.e. S. The join phase can be done in :

$$O(\frac{n}{p} \times (h + \delta) + \frac{m}{p} \times (h + F\delta + \delta) + \delta(t_R + t_S)J_{max})$$

The total time requirements of the hash-based algorithm is given in Table 4.

Like the sort-based algorithm, the hash-based method is very sensitive to the data skew and it is expected to perform poorly in the presence of the data skew. As it is noted in the literature, its performance significantly deteriorates with the presence of the data skew (single or double skew) [22, 13].

Network	The time requirement of hash-based
Hypercube	$O(\tau p + \mu(\frac{t_R n + t_S m}{p}) + \delta(\frac{t_R n + t_S m}{p}) + \frac{n}{p}(h + \delta) + \frac{m}{p}(h + F\delta + \delta) + \delta(t_R + t_S)J_{max})$
Mesh	$O(\tau + \mu(\frac{t_R n + t_S m}{\sqrt{p}}) + \delta(\frac{t_R n + t_S m}{p}) + \frac{n}{p}(h + \delta) + \frac{m}{p}(h + F\delta + \delta) + \delta(t_R + t_S)J_{max})$

Table 4: The total time requirements of the hash-based algorithm on different interconnection networks

5 Join Algorithms with Data Skew

In this section, we describe different types of data skew and review some of the proposed parallel join algorithms and their characteristic. As it is observed by the database research, data skew exists in several real or realistic data sets [4, 16, 15]. The simple parallelization schemes described in the previous section will have poor performance in the presence of data skew mainly due to resulting load imbalances in the amount

of local computation required.

There are four main characteristics which can be used to classify the methods used for achieving load balance in parallelization of join methods [13]. These characteristics are as follows:

1. Types of data skew: The data skew may exist in one relation (*single skew*) or in both relations (*double skew*). The data skew and different types of data skew have been defined and modeled in [25]. The data skew types are *tuple placement skew*, *selectivity skew*, *redistribution skew* and *join product skew*. The tuple placement skew occurs when the initial partitions of a relation have different sizes. The selectivity skew results from performing other database operations prior to the join operation. Whereas, the redistribution skew occurs a result of the redistribution phase of the join algorithm which may generate partitions with high variance in term of sizes. The product skew occurs when the variance of the output sizes produced by each processor is high. The first two types of the data skew can be handled by the *non-order maintaining data movement* primitive which is presented in section 2. In the rest of this paper, we assume that both of the relations are approximately partitioned among the p processors. We also define *work skew*. This skew combines the redistribution and product skews which can be measured by the variance of the amount of work performed by each processor.

2. Load Metrics: Load metric is the criterion that is used to balance the load across the processors. There are mainly two criterion which have been used as a load metrics:

   - *Cardinality*: This load metric uses the partition/bucket sizes of one or both relations to balance the load across the processors. An approach which can be used for load-balancing is to ensure that partition sizes are approximately of the same size. Another approach, which uses the cardinality of the output as a load metric, assigns tuples among processors such that each processor will approximately produce equal number of output tuples.
   - *Estimated execution-time* : The join operation is divided into tasks. The time required to perform each task is estimated using some cost model. The tasks are then assigned to processors such that each processor will finish its tasks in approximately the same time.

3. Statistical Measures: Several statistical measures have been used for load balancing:

   - *Bucket-based:* One or both relations are decomposed into buckets. The sizes of the buckets of one or both relations are used in the assignment process.
   - *Class-based:* Join attribute values are organized into *equivalence* classes using some deterministic function. For each class, a set of statistics are maintained; i.e. the number of distinct join attribute values and the number of tuples from both relations.
   - *Perfect information:* This method is an extreme case of class-based method when each class contains only one distinct join attribute value.

4. Task Allocation: There are mainly two allocation strategies: *static* and *adaptive*. In the former, a task is statically assigned to one of the processor for the entire computation. The latter allows for immigration of the tasks to other processors during the join process.

Several parallel join algorithms have been proposed to alleviate the presence of the data skew; i.e [10, 11, 7, 18, 26, 14, 27, 9, 20, 5, 21]. Most of the proposed algorithms are hash-based algorithms. In the rest of this section, we review a few of the proposed algorithms.

Bucket tuning was introduced in [10]. In this strategy, the number of the buckets of one of the relations is chosen to be very large. In the later phases, smaller buckets are combined to form large size join buckets. In *Bucket Spreading* strategy [11], buckets are spread across all processors. These are then reassigned to appropriate processors based on their sizes using a special *Omega* network. A similar algorithm to bucket spreading strategy which uses a software control instead of Omega network has been designed in [7]. A *Partition tuning* strategy was presented in [7]. This strategy organizes a relation as a set of data cells, and reassigns these data cells from overflow processors to underflow processors using a *best fit decreasing* strategy to balance the load among processors. Three algorithms which use the *partition tuning* and *best fit decreasing* strategies have been presented in [8]. Based on their simulation results, they recommended that the *adaptive load balancing parallel hash* (*ABJ*) is the algorithm of choice for most the cases. The bucket tuning, bucket spreading and partition tuning strategies use the *cardinality* of the partitions/buckets as a load metric. The above algorithms are sensitive to the output skew and expected to perform poorly in the presence of mild or high output skew.

An incremental hash-based algorithm has been proposed and improved in [27] and [9], respectively. In this approach, the join process proceeds in several steps. A *check point* strategy is used after each step to either evaluate the load degree (at the end of the first step) or apply partition tuning using the cardinality of the partially full buckets to change the buckets assignment.

A sampling-based approach has been proposed in [5]. Their approach uses a random sample to estimate the degree of the data skew. Based on this estimation, an appropriate join algorithm is invoked. The invoked algorithm is one of four hash-based algorithms which have been proposed in [5] along with the conventional hash-based algorithm. Two of these algorithms have been designed to deal with the presence of the redistribution skew and the other two deal with the presence of the product skew. Further, the random sample is, once again, used in the partitioning phase of the join algorithms. Their main assumption is that the skew degree is not very high. Based on their experiments, the *virtual processor range partitioning* (*VPP*) is the algorithm of choice in case of mild skew.

Two algorithms have been proposed which use an *estimated execution time* as a load metric to alleviate the presence of double or single skew [26]. The first algorithm is sort-based and the second algorithm is hash-based . The sort-based algorithm uses a divide-and-conquer approach to address the data skew and a heuristic scheduling phase to balance the load across the processors. The hash-based algorithm uses a two-level hierarchical hashing. The results from the hierarchical hashing are used in a heuristic scheduling phase to balance the load across processors.

A hash-based algorithm *HISH* which uses a histogram-based technique to estimate the data distribution and the amount of work has been proposed in [20]. A cost model has been designed to estimate the amount of work contributed by a join attribute value. The estimated work is then used in the partitioning

phase to balance the work among the processors. They also use *virtual processors* approach in assigning the work among the processors. Their histogramming technique, which is based on sampling, produces an approximation of the frequency distributions of both relations and the output sizes. In most real database systems, the histograms are generally precomputed which makes the preprocessing step of this algorithm is of negligible cost. This algorithm has been compared against *VPP* algorithm [5], conventional hash-based and *ABJ* algorithm [8]. For mild product skew, the performance of the *ABJ* and *VPP* are comparable. However, *HISH* is superior to all the three algorithms.

A PRAM algorithm which is similar to our new algorithms in spirit has been proposed in [21]. The proposed algorithm uses the exact total join output size as well as the join output size contributed by each join attribute value to balance the load across the processors.

6 Our algorithms

In this section, we present and analyze two new parallel algorithms which deal with arbitrary amount of skew as well as different types of skew. One of these algorithms is a sort-based while the other is a hash-based. Both of these algorithms employ a preprocessing phase (prior to the redistribution phase) to collect *perfect information* of the join attribute distribution. We later discuss how to extend them to disk-resident data.

The main idea of the new algorithms is to compute a weight for each distinct join attribute value. These weights are generated using the *perfect information* of the join attribute distribution. In the partitioning phase of the join algorithm, p partitions of approximately equal weights are generated. These partitions are then assigned among the processors using static allocation strategy. Further, these set of weights can be defined in different ways to alleviate different types of skew, i.e. define a weight function for each skew type.

For an in-core parallelization of the join operation, we expect that the product skew can affect the performance of the algorithm more than the redistribution skew. For this reason, we will investigate two weight functions; *output* function (for the output skew) and *work* function (for the work skew).

The proposed algorithms have been designed using a set of primitives by which they are relatively architecture independent. The sort-based algorithm is presented in subsection 6.1, while the hash-based algorithm is presented in subsection 6.2.

6.1 The Sort-Based Algorithm

The sort-based algorithm presented in section 4 is expected to perform poorly with the presence of data skew. The new sort-based algorithm has been designed to alleviate the effect of the presence of the data skew (double or single). As we discussed above, this algorithm uses a *perfect information* of the join attribute value to generate p partitions of approximately equal weights.

The algorithm consists of several phases:

- **Sorting phase:** To sort the local fragments of both relations locally, followed by sorting the join attribute values globally
- **Preprocessing phase:** To collect the perfect information of the join attribute and generate the set of weights
- **Splitters phase:** To decide the decomposition strategy and generate a set of splitters
- **Redistribution phase:** To create the partitions and redistribute them among the processors
- **Merging phase:** Similar to the conventional sort-based algorithm

The new sort-based algorithm first sorts relations R and S using parallel sample-based algorithm. In the sample sort, each processor first sorts its local fragments of both relations using a sequential sorting algorithm. However, in the subsequent steps of the sorting phase, the join attribute values are projected and used instead of the whole record.

In the preprocessing phase, the perfect information of the join attribute is collected as follows. Each processor scans its local fragments (of the join attribute) of both relations and counts the number of duplicates of each distinct value of the join attribute. The last and the first elements of the local lists might cause an interprocessor communication. In case that the largest value of some processor i is the same as the smallest value of processor $i+1$, a group-combine of unit size with addition will be performed to obtain the total number of duplicates of that value. Let $Hist_R$ and $Hist_S$ be the results of the counting step for both relations R and S, respectively. Each element of these lists consists of two fields: (1) the value of the join attribute and (2) the number of duplicates. It should be noted that the sizes of $Hist_R$ (n_H) and $Hist_S$ (m_H) are smaller than or equal to n and m, respectively. Merging primitive is performed on $Hist_R$ and $Hist_S$ to obtain a combined list $Hist$ of both relations . The merging primitive has been discussed in section 2. The implementation of the merging primitive guarantees that all the elements of the same values are assigned to one processor.

In the next step, the set of weights is generated using some weight function. We define two functions: *work* and *output* weight functions. These functions have been defined to assign a weight to some join attribute value using only the information of that value; i.e. the frequency of the join attribute values in each relation (f_R and f_S). These two functions are defined as follows [3] :

- *output* weight function F_O:

$$F_O(f_R, f_S) = f_R \times f_S \tag{1}$$

- *work* weight function F_W [4] :

$$F_W(f_R, f_S) = f_R \times f_S + f_R + f_S \tag{2}$$

[3] We can define a weight function to alleviate the redistribution skew as $F_D(f_R, f_S) = f_R + f_S$.

[4] This function has been defined in [5]. However, they use it to estimate the cost of join buckets.

Potentially, One can define more complicated functions which include the (exact or estimated) cost of the next phases of the join algorithm; i.e. the cost of the interprocessor communication and cost of processing a tuple during the local join method.

One advantage of the above weight functions (F_O and F_W) is that the weights set can be computed locally. This can be achieved by scanning $Hist$ list and applying the weight function for each distinct value.

Let W be a list of the weights. It can be easily shown that the size of W is less than or equal to $\min(n_H, m_H)$. During the previous step, each processor keeps track of its local sum of its weights ω_i. The total sum of the weights ω is computed by performing a global-combine-sum primitive of unit size on ω_i.

Our assignment technique needs the ranks of the weights. These ranks $Rank$ can be computed by performing global exclusive-prefix-sum on W list in two steps. In the first step, the rank of the first weight of each processor ($Rank_0$) is computed by performing global exclusive-prefix-sum of unit size on ω_i's. The remaining ranks are computed locally by each processor using sequential prefix-sum on local W list with $Rank_0$ as the starting value.

Our objective is to generate p partitions of approximately equal weights. There are two approaches which have been used for assigning tuples to processors: *full-fragmentation* and *fragmentation-replication*. In the full-fragmentation approach, both relations are partitioned into disjoint fragments; these fragments are then assigned among the processors. The fragmentation-replication approach might partition one or both relations into non-disjoint fragments; i.e. replicates some of the data among more than one fragments. Ideally, one would like to use the full-fragmentation approach because it incurs less overhead than the fragmentation-replication approach. However, the full-fragmentation approach is not applicable for some cases. This can happen if the maximum value of $W_i's$, call it ω_{max}, has value greater than $c\frac{\omega}{p}$, for some constant c. We call c a *load factor* . In that case, we switch to the fragmentation-replication approach. ω_{max} is found by finding the local ω_{max} of each processor followed by performing global-combine on the local ω_{max}'s using max operation as the combine operation. To decide between the two approaches, we use the *Approach* function:

$$Approach(\omega_{max}) = \begin{cases} full\text{-}fragmentation & : \quad \omega_{max} < c\frac{os}{p} \\ fragmentation\text{-}replication & : \quad Otherwise \end{cases}$$

In case of full-fragmentation approach, the algorithm processed as follows. $p-1$ splitters are chosen to create p partitions. Each partition is assigned to different processor. These splitters are selected such that the sum of the weights of each partition is $c\frac{\omega}{p} + \epsilon$ and all the join attribute values in partition i are smaller than the values in partition $i+1$. Each processor locally determines which element of its local $Hist$ is a splitter by using its ranks $Rank$ and weights W lists. Element i is a splitter j if its rank less than $Rank_0$ of processor j and its rank plus its weight greater than or equal to the $Rank_0$ of processor j. After Finding the splitters, many-to-all broadcast is performed (with potentially different message sizes) on the splitters.

In the redistribution phase, the local fragments of relations R and S are partitioned using the splitters list using binary search (the local fragments are already sorted). The required inter-processors communication

for both relations is performed using the transportation primitive.

The merging phase is exactly similar to the conventional sort-based (section 4).

In case of fragmentation-replication approach, a more complicated assignment procedure is needed. Each processor locally finds a set of splitters as discussed above. Each splitter might be assigned to multiple adjacent processors. For each splitter i with join attribute value t, we need to determine the replicated relation, first destination, the number of destinations and a set of weights (these are another set of weights) which are used in the redistribution phase, call these w^{dist}. The replicated relation is the relation having smaller number of tuples with join attribute value t. This choice will generally have less communication overhead. The first destination (d_i^1) is computed using the splitter's rank ($Rank^i$); i.e. $Rank^i \ div \ \frac{\omega}{p}$, and the number of destinations (n_i) is computed using the splitter weight W^i and $Rank^i$; i.e. $(Rank^i + W^i) \ div \ \frac{\omega}{p} - d_i^1 + 1$.

w_{ij}^{dist} is used by all the processors to determine how many number of tuples having the join attribute value t (the value of splitter i) of the fragmented relation to be sent to processor j. These weights are computed locally as follows.

$$w_{ij}^{dist} = \begin{cases} \frac{\frac{(j+1)\times\omega}{p} - Rank^i}{W^i} & : \quad \frac{j\times\omega}{p} < Rank^i \\ \frac{\omega}{p \times W^i} & : \quad Rank^i < \frac{j\times\omega}{p} < Rank^{i+1} \\ \frac{Rank_{l+1} - \frac{(j)\times\omega}{p}}{W^i} & : \quad Otherwise \end{cases} \tag{3}$$

After finding the splitters , many-to-all broadcast (with potentially different message sizes) is performed on the splitters along with other information. It can be easily shown that the size of all the weights set is at most $2p$.

In the redistribution phase, a destination processor for each tuple is determined as follows. A tuple with join attribute value less than the value of splitter i and greater the value of splitter $i-1$ is assigned to processor with address equals to d_{i-1}^1. For tuples having a join attribute values equal to the value of splitter i, we perform the following. Assigns those tuples belonging to the replicated relation to all processors with addresses equal to $d_i^1, d_i^1+1, \ldots, d_i^1+n_i-1$. For those tuples belonging to the fragmented relation, computes the number of duplicates of that value, n_{dup}, and assigns $n_{dup} \times w_{ij}^{dist}$ tuples to processor with address equals to $d_i^1 + j$. The cost of counting the number of duplicates is linear since the local fragments of both relations are already sorted. The required inter-processors communication and the merging phase are exactly the same as in the full-fragmentation approach.

Time Requirements The overall time requirement of the new sort-based algorithm is the sum of the time required by all the phases. The time taken by the sample sort is given in Table 1. However, the local sorting step takes $O(\delta(t_R \frac{n}{p} + \frac{n}{p} \lg \frac{n}{p} + t_S \frac{m}{p} + \frac{m}{p} \lg \frac{m}{p}))$. The subsequent steps of the sample sort are performed on the join attribute values only. Counting the number of duplicates and generating $Hist_R$ and $Hist_S$ lists

takes $O(\delta\frac{n+m}{p})$ plus the time taken by the group-combine. The time required by the merging primitive for merging $Hist_R$ and $Hist_S$ lists is given in Table 1. The worst case for generating W list is $O(\delta\frac{n+m}{p})$ time. Computing the total sum of the weights ω takes time as much as a global-combine-sum of unit size. To compute $Rank$ list, we need to spend the time taken by exclusive-prefix-sum of unit size plus $O(\delta\frac{n+m}{p})$ time. To choose between full-fragmentation and fragmentation-replication approaches, we need to spend the time taken by the global-combine-sum of unit size.

Finding the splitters takes $O(\delta\frac{n+m}{p})$ time in the worst case. Since the size of the weights set (of the splitters) is at most $2p$, computing all the weights set can be done in $O(p)$. Broadcasting the splitter takes $O(\mu p)$ time in the worst case. Assigning tuples to processors takes $O(\delta p(\lg\frac{n}{p} + \lg\frac{m}{p}))$ time in case of full fragmentation, and $O(\delta\frac{n+m}{p})$ in case of fragmentation-replication. The communication requirements for both relations take time as much as the time taken by the transportation primitive with messages sizes equal to $t_R n \max(Q_R, O_R)$ and $t_S m \max(Q_S, O_S)$ respectively, where Q_R, O_R, Q_S, and O_S are defined in section 3.

Merging the received R's tuples and S's tuples take $O(\delta(nQ_R \lg p + t_R nQ_R))$ and $O(\delta(mQ_S \lg p + t_S mQ_S))$, respectively. The final merging and producing the output tuples takes at most $O(\delta(nQ_R + mQ_S + (t_R + t_S)\frac{J}{p}))$.

The overall time requirement of the algorithm can be simplified to the time taken by the sorting phase, transportation primitive, merging the $R's$ tuples, merging the $S's$ tuples and the final merging. The computation requirement is

$$O(\delta(\tfrac{n}{p}\lg\tfrac{n}{p} + t_R\tfrac{n}{p} + \tfrac{m}{p}\lg\tfrac{m}{p} + nQ_R\lg p + t_R nQ_R + mQ_S\lg p + t_S mQ_S + (t_R + t_S)\tfrac{J}{p})).$$

The communication requirement is given in Table 5 for the two interconnection network.

Network	requirement of the sort-based
Hypercube	$O(\tau p + \mu(plog^2 p + t_R n \max(Q_R, O_R) + t_S m \max(Q_S, O_S)))$
Mesh	$O(\tau\sqrt{p} + \mu\sqrt{p}(p + t_R n \max(Q_R, O_R) + t_S m \max(Q_S, O_S)))$

Table 5: The communication time requirement for the sort-based algorithm on different interconnection networks

6.2 The Hash-Based Algorithm

The new hash-based algorithm is very similar to the new sort-based algorithm in the sense that they both collect the same types of information and generate the same set of splitters. However, they mainly differ in the following:

1. Counting the number of duplicates of each distinct join attribute value is done differently. In the hash-based algorithm, a Random Access Write (RAW) primitive with the addition as a collision resolution strategy is used in this process.

2. The local join methods are different. The hash-based algorithm uses a hash-based method as opposed to sort-based method in the sort-based method.

Our hash-based algorithm consists of several phases:

- **Preprocessing phase:** To collect the perfect information of the join attribute using RAW and generate the set of weights
- **Splitters phase:** To decide the decomposition strategy and generate a set of splitters
- **Redistribution phase:** To create the partitions and redistribute them among the processors
- **Join phase:** Similar to the conventional hash-based algorithm

The distributed memory is *viewed* as a global shared memory with addresses in the range $[0..b]$ where $b = m \times p$ and m is the size of the local available memory of each processor and p is the number of the processors. Location i of this global shared memory resides at processor $\lfloor \frac{i}{m} \rfloor$ and it corresponds to location $i \ mod \ m$ of the local memory of that processor . First, the algorithm hashes [5] the join attribute values to integers using some hash function $hash$. The result of $hash$ is used as an address of the global shared memory. As necessary requirement of the hash function $hash$, its range should be less than or equal to the size of the global shared memory. We choose to use the RAW algorithm of [24]. This algorithm is very scalable as it shown in Table 1.

Another requirement of the hash function $hash$, to ensure that the number of duplicates computed in the counting step is accurate, is that the hash function $hash$ should satisfy the following condition[6] :

- for all join attribute values x and y in both relations, $hash(x) = hash(y) \Leftrightarrow x = y$ [21].

We apply the random access write RAW on both relations where the addresses are the hash values and the values are ones. The result of RAW operation is the $Hist_R$ and $Hist_S$ lists of both relations R and S, respectively. For the details about RAW algorithm see section 2. Computing the set of weights W is straightforward. Since $Hist_R$ and $Hist_S$ have same size, this process does not need any interprocessor communication and it can be done by applying the weight function on each pair of $Hist_R$ and $Hist_S$ lists.

Computing the rank list $Rank$, total sum of the weights set ω and finding the splitters phase are exactly similar to the sort-based algorithm (section 6.1). However, the redistribution phase is quite different because the local fragments of the relations R and S are not sorted. In case of full-fragmentation, the assignment of tuples are done by searching the splitters list for each tuple using binary search. While in the fragmentation-replication case, instead of counting the number of duplicates of the tuples having a join attribute value

[5] We might not need to perform this hashing process. This might happen when the join attribute has an integer type and its domain is less than the size of the global shared memory.

[6] In case that this condition is not satisfied, our hash-based algorithm is still complete (produce all the join-able tuples). Using a function which does not satisfy the condition affects the performance of the algorithm but it does not affect its complicity.

equals to some splitter value as in the sort-based algorithm, we use a weighted round-robin method to assign those tuples belonging to the fragmented relation to destination processors. The required inter-processors communication is carried out by the transportation primitive.

The join phase is very similar to the conventional hash-based algorithm.

Time Requirements The overall time requirement of the new hash-based is sum of the time required by all the phases. The hashing step (if needed) requires $O(h\frac{n+m}{p})$ time. The time taken by RAW is given in Table 1. The time required for computing W, $Rank$, ω and splitters is the same as in the sort-based algorithm. Assigning tuples to processors takes $O(\delta(\frac{n+m}{p}\lg p + t_R\frac{n}{p} + t_S\frac{m}{p}))$ time for both decomposition strategies. The communication requirements for both relations takes time as much as the time taken by the transportation primitive with messages sizes equal to $t_R n \max(Q_R, O_R)$ and $t_S m \max(Q_S, O_S)$, respectively. Building the local hash table takes $O(nQ_R(h+\delta))$ and probing it and producing the result takes $O(mQ_S(h+F\delta) + \delta(t_R + t_S)\frac{J}{p})$

The overall time requirement of the hash-based algorithm can be simplified to the time taken by the RAW primitive, transportation primitive, assigning phase, building and probing the hash table and producing the output tuples. The computation time is

$$O(\tfrac{n+m}{p}(\delta\lg p + h) + \delta(t_R\tfrac{n}{p} + t_S\tfrac{m}{p}) + nQ_R(h+\delta) + \delta mQ_S F + \delta(t_R+t_S)\tfrac{J}{p} + mQ_S h).$$

The communication time is given in Table 6 for the two interconnection network.

Network	Time requirement
Hypercube	$O(\tau p + \mu(t_R n \max(Q_R, O_R) + t_S m \max(Q_S, O_S)))$
Mesh	$O(\tau\sqrt{p} + \mu\sqrt{p}(t_R n \max(Q_R, O_R) + t_S m \max(Q_S, O_S)))$

Table 6: The communication time requirement of the hash-based algorithm on different interconnection networks

6.3 Scalability Considerations

In real database applications, the sizes of the tuples t_R and t_S are generally a few hundreds of bytes. In the two new algorithms, the total cost of the preprocessing step is proportional to the cardinality of the relations times the size of the join attribute. Whereas, the over all costs of the both algorithms are proportional to the cardinality of the relations times the size of the tuples. The size of the join attribute is generally smaller than the size of the tuples by an order to two orders of magnitude. Hence, we expect that the cost of the preprocessing step is relatively small in case of uniform distribution. Further, the preprocessing cost can be reduced by drawing a sample from join attribute values of both relations and applying the preprocessing phase only on this sample. We will empirically investigate this option in the next section.

The conventional hash-based and sort-based algorithms (Section 4) are scalable for uniformly distributed data. For this case, a perfect or near-perfect load-balancing is generated by these algorithms for arbitrary number of processors. However, their performance deteriorate significantly in the presence of data skew.

For uniform distribution, our algorithms converge to their counterpart conventional algorithms because the values of Q_R, O_R, Q_S and O_S are approximately equal to $\frac{1}{p}$ and the cost of the preprocessing step is relatively small comparing to the overall cost.

For cases in which the cost of producing the join output dominates the overall cost of the required work, the complexity of our algorithms is proportional to $O(\frac{J}{p})$ which is optimal (within a constant factor). For the case where the required work is dominated by the size of both relations, the complexity of our algorithms are proportional to $O(t_R\frac{m}{p} + t_S\frac{m}{p})$. These time requirements remain the same (within a small constant factor) even in the the presence of arbitrary amount of skew. Hence, we conclude that the new algorithms are scalable and indeed robust. Our experimental results presented in the next section verify these observations.

7 Experimental Results

We have implemented the four algorithms, namely the conventional hash-based (*SSH*) and sort-based (*SSS*) algorithm, the new hash-based (*SIH*) and the new sort-based (*SIS*) algorithms, on an IBM SP-2 with 16 processors. The clock speed of the processors is 66.7 MHz, the memory size is 256 MB per processor, and the operating system is AIX version 4.1.4. Our experiments were targeted to study the effect of the following:

1. The weight functions
2. The load factor
3. The tuple size
4. The size of the relations
5. The effect of the sampling ratio

Data sets We have evaluated the algorithms for data set generated using three distributions:

1. Uniform distribution: The join attribute values has a uniform distribution in $[0..256K]$.
2. Scalar skew distribution: This distribution has two parameters (one_R and one_S). Relation R (S) has one_R (one_S) tuples with join attribute of value "1" and the rest of the tuples are generated randomly from $[2..n]$ ($[2..m]$) [5]. The default value for one_R and one_S is 1000.
3. Zipf distribution: The Zipf distribution has two parameters which determine the degree of the skew of the data [28]. The first parameter z is between zero and one. The data set corresponds to a uniform

distribution when z is set to zero. The level of skew increases as the value of this parameter increases. The second parameter determines the number of distinct values d. The probability density function $P(x = i)$ for each distinct value i in $[1..d]$ is given by:

$$P(x = i) = \frac{1}{i^z \sum_{j=1}^{d} \frac{1}{j^z}} \tag{4}$$

For all experiments, d is fixed to $128K$. The default value of z is 0.75.

The default values of the sizes of both relations and the tuple size are $256K$ and 100 bytes, respectively.

For each experiment, the algorithms was executed three times and the median is reported. In the first experiment, we ran the new algorithms using the two weight functions. Table 7 shows the overall execution time in seconds of the new algorithms for different data sets. Clearly, the work function captures the cost more effectively as it results in better load balance. We have set the weight function to the work function for the rest of our experiments.

Distribution	Hash-Based		Sort-Based	
	Output	Work	Output	Work
Uniform	0.425	0.423	0.794	0.798
Scalar	1.043	0.701	2.150	1.294
Zip-f	6.940	5.936	7.443	6.818

Table 7: The overall execution time (in seconds) for the two new algorithms using different weight functions for the three distributions on 16 processors

We also ran the new algorithms using different load factors (1, 1.25 and 1.5); the overall performance is almost independent of the load factor for the three data sets. The load factor is fixed to 1 for the rest of the experiments.

Figures 1, 2 and 3 show the total execution times for the four algorithms using different tuple sizes (52, 100 and 200 bytes). We can draw the following conclusions from these figures:

1. For uniform distribution the absolute cost of the preprocessing phase is independent of the tuple size. However, its relative cost decreases with the increase of the tuple size. Further, the preprocessing step is relatively small comparing to the overall cost.

2. Our algorithms substantially outperform the conventional algorithms for mild (Scalar Skew) and high (Zip-f) skews. Further, we expect the improvement to be significantly better for large number of processors. This is due the fact that that the conventional algorithms are not scalable, whereas our algorithms are.

3. The new hash-based algorithm is the clear winner for small levels of skew. For high degree of skew, the new sort-based outperforms all the other algorithms. This is due to the fact that the sequential

sort-based (in-core version) outperforms the other algorithms for large and highly skewed relations. This can attributed to the high cost of probing the hash table.

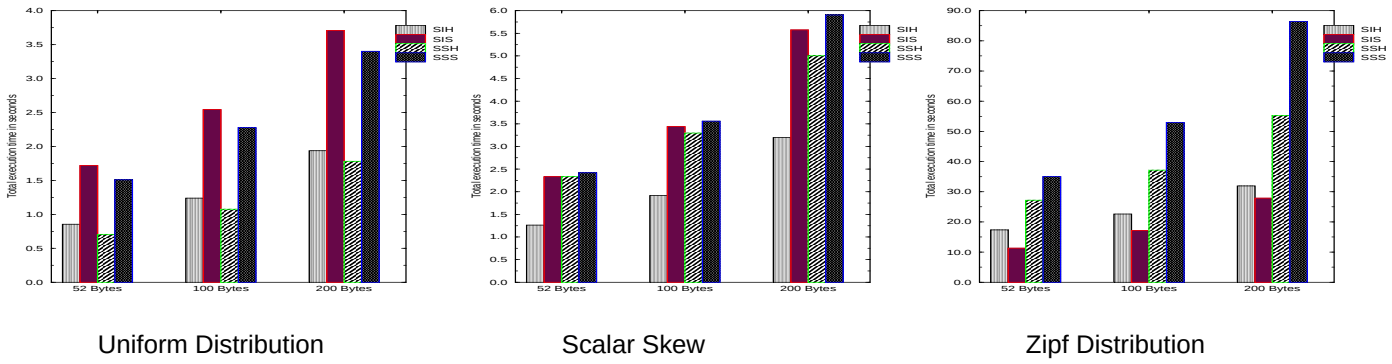

Figure 1: Comparison of different algorithms for different sizes of tuple on 4 processors

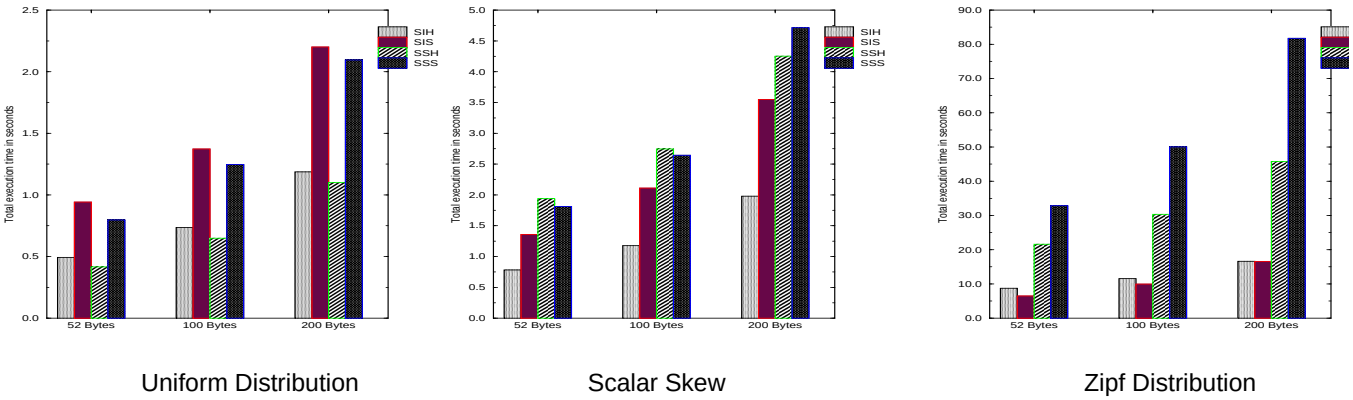

Figure 2: Comparison of different algorithms for different sizes of tuple on 8 processors

Figure 4 shows that our algorithms have excellent size-up properties.

The speed-up [7] of the four algorithms on 4, 8 and 16 processors for datasets and tuple sizes of 256K and 100 bytes, respectively, are shown in Figure 5. When the amount of the work required is not high, our algorithms do not achieve any speed-up. For example, the hash-based algorithms do not achieve any speed-up for uniform distribution and small number of processors. This is because that the amount of the required work is about 1.11 seconds. This is comparable to the overhead of these algorithms. However, our algorithms achieved almost similar speed-up as of the conventional algorithms for these cases.

For mild and high skew, the speed-up achieved by our algorithms is significantly better than the conventional algorithms. Further, the new algorithms achieved almost linear speed-up.

[7] The speed-up of the hash-based (sort-based) algorithms are measured against the sequential hash-based (sort-based) algorithm.

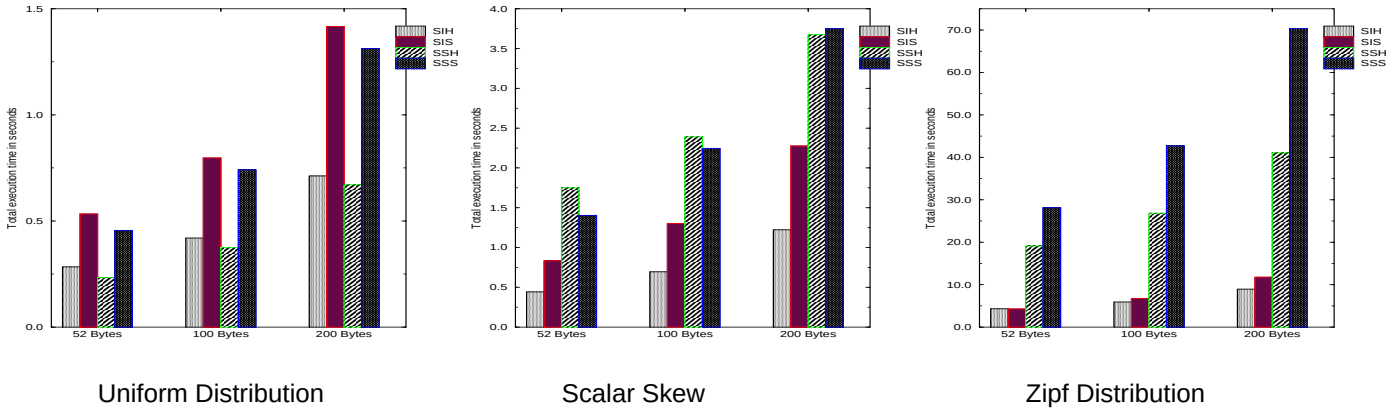

Figure 3: Comparison of different algorithms for different sizes of tuple on 16 processors

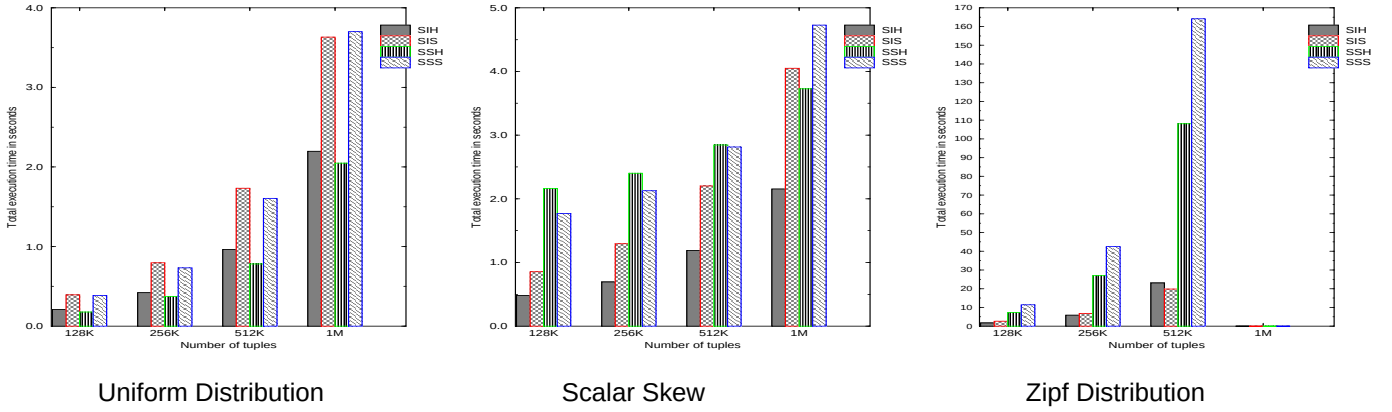

Figure 4: Comparison of different algorithms for different sizes of relations on 16 processors

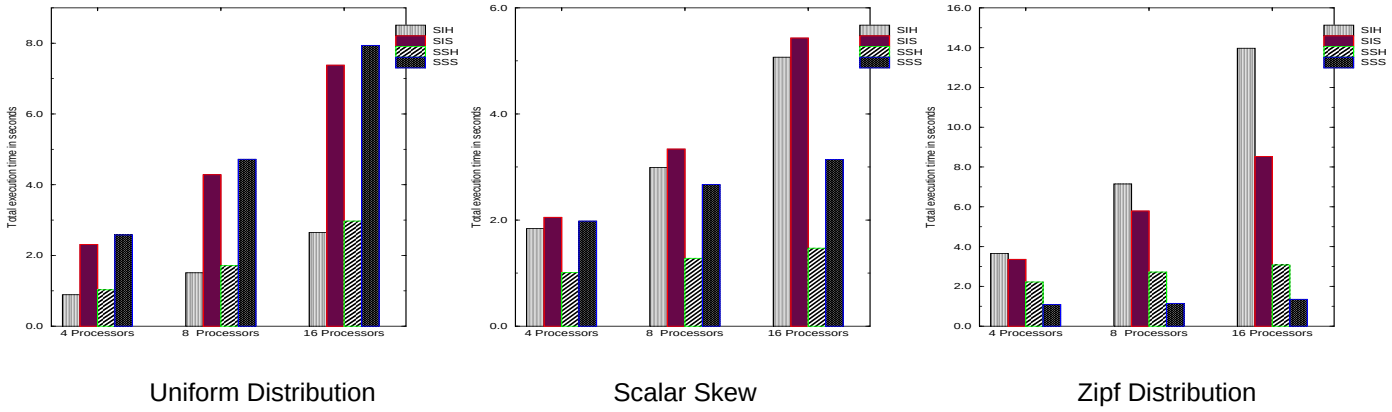

Figure 5: Speed-up of different algorithms on data sets of size 256K and tuples of size 100 bytes

We investigated the effect of utilizing sampling in the preprocessing phases on our algorithms. We ran the preprocessing phase on different sample sizes ($6.25\%, 12.5\%, 25\%, 50\%$ and 100% (the entire relation)) of both relations for the two algorithms. These samples were generated using random and regular sampling for the hash-based and sort-based, respectively. Figure 6 shows the execution times of the two algorithms using different sampling ratios on 16 processors. The sampling has a small effect on the overall time requirements for data set with uniform distribution. This is because the cost of the preprocessing phase is relatively small.

Using a large sampling ratio improves the performance of hash-based algorithms when the skew is modest to large. This is because it captures the skew more effectively and produces better load balancing. On the other hand, the performance of the sort-based algorithm improves for smaller sample sizes especially for highly skewed data sets. This can partly be attributed to our weight function. For Zip-f distribution, the sort-based algorithm with 100% and 6.25% sampling ratios achieved 1.02 and 1.25, respectively, as the degree of the join output skew; i.e. the maximum number of produced output joins by some processor divided by the average number of output joins. However, the partition expansion [8] for both relations were around 12.5 and 4.5 using 100% and 6.25%, respectively, as sampling ratio. The partition expansion affects the redistribution cost as well as the merging cost of the sort-based algorithm. The hash-based is more robust against the partition expansion because the probing cost is less sensitive to the partition expansion. These experiments suggest that a more complicated weight function is needed for the sort-based approach. We are currently investigating different weight functions.

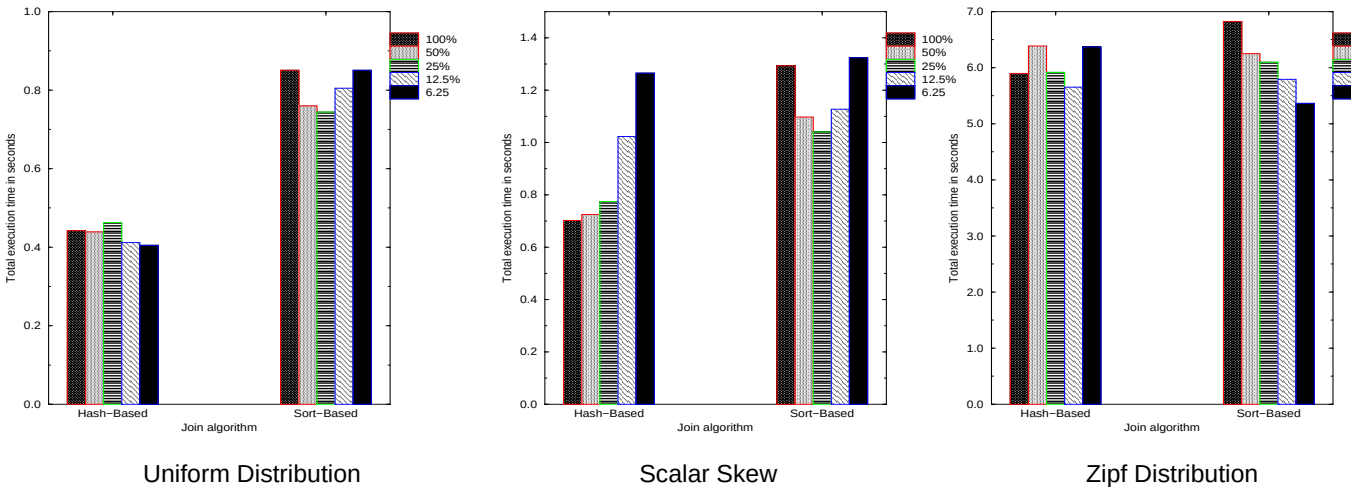

Figure 6: Comparison of different sampling ratios on 16 processors

[8] We define the partition expansion by the maximum size of the partition of some processor divided by the average partition size.

8 Conclusions

We have presented two new parallel join algorithms for coarse grained machines which work optimally in presence of arbitrary amount of data skew. The first algorithm is sort-based and the second algorithm is hash-based. Both of these algorithms employ a preprocessing phase (prior to the redistribution phase) to equally partition the work among the processors using *perfect information* of the join attribute distribution. The cost of this preprocessing phase is relatively small in case of uniform distribution.

These algorithms are shown to be theoretically as well as practically scalable. The hash-based algorithm achieved almost perfect speed-up for highly skewed data on different number of processors. It was also better than the other algorithms except for high degree of skew and large relations for which the new sort-based algorithm performs slightly better. Clearly, one can design a hybrid algorithm which estimates the amount of skew to trigger the appropriate join algorithm.

The proposed algorithms can be easily extended to disk-resident relations. In case that the join attribute values of both relations can be accommodated in the main memory, one can project these values and apply our techniques on them to compute the set of splitters. Then, one can apply the state of the art sequential join algorithm for the local disk-resident fragments in the join/merge phase. This will require one extra sequential I/O (Read) of both relations. As we discussed earlier, the total size of the aggregate main memory across coarse grained machines can be as large as few hundred gigabytes. Assuming that the tuple size is larger than the join attribute size by a factor of more than 25, this technique can handle relations with sizes proportional to a few terabytes. In case that the aggregate main memory can not accommodate the join attribute values, one can sample these values and apply our techniques on them to compute the splitters set. We will investigate these extensions in the future.

The performance of our algorithms can be further improved by

1. Using the collected information of the join attributed values to tune the performance of the algorithms, i.e. estimate the skew degree to choose the relation with less skew to build the hash table in the hash-based algorithm.
2. Employing a filtering phase prior to the redistribution phase to filter out all the tuples which do not contribute to the join output.

We are currently exploring the impact of these improvements on the overall performance.

References

[1] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel Construction of Multidimensional Binary Search Trees. *ICS'96 Philadelphia, PA.*, 1996.

[2] K. Alsabti and S. Ranka. Integer Sorting Algorithms for Coarse-Grained Parallel Machines. *In Proc. of Int'l Conference on High Performance Computing HiPC'97*, 1997.

[3] S. Bae, K. Alsabti, and S. Ranka. Array Combining Scatter Functions on Coarse-Grained Machines. *Submitted for potential publication.*

[4] S. Christodoulakis. Estimating Record Selectivities. *Inform. Syst.*, 8,No. 2:105–115, 1983.

[5] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. *In Proceedings of the 18th VLDB Conference*, 1992.

[6] G. Fox et al. *Solving Problems on Concurrent Processors: Vol. 1.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[7] K. A. Hua and C. Lee. Handling Data Skew in Multiprocessors Database Computers Using Partition Tuning. *Proc. 17th Int'l Conf. Very Large Data Bases, Morgan Kaufman, San Mateo, Calif*, pages 525–535, 1991.

[8] K. A. Hua, C. Lee, and C. M. Hua. Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning. *IEEE Transactions on Knowledge and Data Engineering*, 7, No 6, December 1995.

[9] K. A. Hua, W. Tavavapong, and H. Young. A Performance Evaluation of Load Balancing techniques for Join Operations on Multicomputer Database Systems. *IEEE 11th Int'l Conference on Data Engineering*, 1995.

[10] M. Kitsureqawa, M. Nakayama, and M. Takagi. Query Execution for Large Relations on Functional Disk System. *in Proc. IEEE fifth Int. Conf. data Eng.*, pages 159–167, 1989.

[11] M. Kitsureqawa and Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Methods for Data Skew in the Super Database Computer (SDC). *Proc. 16th Int'l Conf. Very Large Data Bases, Morgan Kaufman, San Mateo, Calif*, pages 210–221, 1990.

[12] V. Kumar et al. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc, 1994.

[13] H. Lu, B. Ooi, and K. Tan. *Query Processing in Parallel Relational Database Systems.* IEEE Computer Society Press, 1994.

[14] H. Lu and K.L Tan. Dynamic and Load-Balanced Task-oriented Database Query Processing in Parallel Systems. *Proc. Third Int'l Conf Extending Data Base Technology*, pages 357–372, 1992.

[15] C. Lynch. Selectivity Estimation and Query Optimization in Large Databases with Highly Skews Distribution of Columns Values. *in Proc. 14th Int. Conf. Very Large Data Bases*, pages 240–251, 1988.

[16] A. Montgomery, D. D'Souza, and S. Lee. The Cost of Relational Algebraic Operations on Skew Data: Estimates and Experiments. *in Proc. Inform. Process*, pages 235–241, 1983.

[17] D. Nassimi and S. Sahni. Data Broadcasting in SIMD Computers. *IEEE Transactions on Computers*, C-30(2):101–107, 1981.

[18] E. Omiecinki and E. T. Lin. The Adaptive-Hash Join Algorithm for a Hypercube Multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, 3,No. 3:334–349, May 1992.

[19] Chao-Wei Ou and Sanjay Ranka. Parallel Remapping Algorithm for Adaptive Problems. *Journal of Parallel and Distributed Computing*, 1997.

[20] V. Poosala. *Histogram-Based Estimation Techniques in Database Systems.* Ph.D. thesis, University of Wisconsin-Madison, 1997.

[21] R. Raman and U. Vishkin. Parallel Algorithms for Database Operations and a Database Operation for Parallel Algorithms. *In Proc. 9th IEEE International Parallel Processing Symposium (IPPS)*, 1995.

[22] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *Proc. ACM Int'l Conf. Management Data, Assoc for Computing Machinery*, pages 110–121, 1989.

[23] R. Shankar, K. Alsabti, and S. Ranka. Many-to-Many Personalized Communication With Bounded Traffic. *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, February 1995.

[24] R. Shankar and S. Ranka. Random Data Accesses on a Coarse-Grained Parallel Machine II. One-to-Many and Many-to-One Mappings. *Journal of Parallel and Distributed Computing*, July 1997.

[25] C. B. Walton and A. G. Dale. Data Skew and the Scalability of Parallel Joins. *IEEE transactions on*, 36,No. 5:44–51, 1991.

[26] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. New Algorithms for Parallelizing Relational Database Joins in the Presence of Data Skew. *IEEE Transactions on Knowledge and Data Engineering*, 6,No. 6:990–997, December 1994.

[27] X. Zaho, R. G. Johnson, and N. J. Martin. DBJ- a Dynamic Balancing Hash Join Algorithm in Multiprocessor Database Systems. *Information Systems Journal*, 19(1), 1994.

[28] G.K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Reading, MA, 1949.