

Handling Data Skew in Parallel Hash Join Computation Using Two-Phase Scheduling *

Xiaofang Zhou

Maria E. Orlowska

Division of Information Technology
CSIRO
GPO Box 664, Canberra, ACT 2601, Australia
xiaofang.zhou@dit.csiro.au

Department of Computer Science
The University of Queensland
St Lucia, QLD 4072, Australia
maria@cs.uq.oz.au

Abstract

A large number of parallel join algorithms has been proposed to maintain load-balancing in the presence of data skew. However, one important type of data skew – join product skew (JPS) – has been little studied. In this paper, a dynamic parallel join algorithm, which employs a two-phase scheduling procedure, is designed to handle the JPS problem. Two sets of scheduling heuristics are studied against various parameters. It is shown that many of the existing algorithms can be regarded as a special case of our algorithm, whose cost is based on the nature of data skew. While it can cope with JPS which other algorithms cannot approach, it can be as efficient as most existing algorithms when JPS does not exist.

1 Introduction

Parallel database research centres on processing database operations in very large databases using general-purpose parallel architectures to improve database performance. As join is one of the most time-consuming operations in the relational model, substantial research has been done towards designing efficient parallel join algorithms [2, 7, 8, 13, 16, 19, 21].

It is a consensus that hash-based parallel join algorithms outperform other parallel join algorithms. Let $P = \{p_i : i = 1..n\}$ be a set of processors, and r_1 and r_2 be two relations with a common attribute A defined on domain D . There are basically two phases to perform $r_1 \bowtie r_2$ in parallel by using hash function $f : D \rightarrow (1..n)$. The *hash partitioning phase* is employed to obtain two sets of buckets $r_{1i} = \{t \in r_1 :$

$f(t.A) = i\}$ and $r_{2i} = \{t \in r_2 : f(t.A) = i\}$, $1 \leq i \leq n$. In such a way, the join operation is decomposed into n subjoins; that is, $r_1 \bowtie r_2 = \bigcup_{i=1}^n (r_{1i} \bowtie r_{2i})$. In the second phase, the *local join phase*, these bucket pairs can be sent to different processors to execute concurrently.

To minimize the response time of parallel join computation, it is expected that each bucket pair carries the same amount of workload. In general, the join workload of $r_{1i} \bowtie r_{2i}$ can be defined as

$$JN(r_{1i}, r_{2i}) = g_1(|r_{1i}|, |r_{2i}|) + g_2(|r_{1i} \bowtie r_{2i}|) \quad (1)$$

where both g_1 and g_2 are non-decreasing functions with respect to each of their parameters [20]. Data skew, a phenomena of nonuniform data distribution, can result in fluctuation of workload among bucket pairs [17, 12]. Regarding to the parallel hash join computation, there are three major categories of data skew: *intrinsic skew (IS)* – some join attribute values appear more often than others in a relation; *re-distribution skew (RS)* – the hash function cannot distribute tuples evenly, due to a possible mismatch between the data distribution the hash function expected and the actual data distribution in the join operands; and *join product skew (JPS)* – the join selectivity factors of bucket pairs may be different.

It is a key issue to handle data skew problem in designing efficient parallel join algorithms. In the GRACE join algorithm [9], each tuple is sent to a processor determined by using a pre-defined hash function. To overcome possible IS and RS, more than one bucket pair is allocated to a processor so that it is possible for each processor to adjust bucket sizes locally. The hybrid hash join algorithm in the GAMMA system uses hashing for both bucket partitioning and bucket pair joining [3, 4]. The joining relations are split into a number of buckets by hashing such that each single bucket of the building relation can fit in

*The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

the aggregate memory of the participating processors. Then, all processors are used to join these bucket pairs, one by one, cooperatively by letting all processors read tuples of the probing relation and sending each of them to a processor determined by hashing in parallel. When a bucket of the building relation, which is to be held in memory during the join phase, overflows because of data skew, a new hash function is employed to split this bucket. This procedure may be repeated for further overflows. Both these algorithms may suffer from data skew. For the GRACE join algorithm, it is not always possible to even up bucket sizes locally; for the hybrid join algorithm, when bucket overflow occurs due to skewness, it loses the ground to claim its advantages of having a minimum I/O cost.

An alternative way to approach data skew is to spread one bucket over all the participating processors, instead of converging to a single node [9, 8, 7]. In [8], the tuples in these distributed buckets are redistributed through the Ω network, a multi-level network containing control logic to balance buckets, to obtain a balanced bucket-partition. As this approach requires to transfer tuples twice over the network, some scheduling methods are proposed to reduce the communication cost by doubling the I/O cost [18, 7]. These methods first apply a "fine" hash function to all the local tuples; then, all bucket sizes are reported to a coordinator where a coarser but more balanced bucket-to-processor distribution function is established; finally, all tuples are redistributed using this new function. Targeting to reduce both communication cost and I/O cost, the construction of a balanced partitioning function can rely on some sampling techniques so that only part of the relation needs to be scanned twice[5].

All these algorithms basically try to balance workload by making bucket sizes equal before the local join phase. We call them *the static scheduling algorithms*. They are explicitly or implicitly based on the assumption that equal bucket sizes lead to equal workload, i.e., $(|r_{1i}| = |r_{1j}| \wedge |r_{2i}| = |r_{2j}|) \Rightarrow JN(r_{1i}, r_{2i}) = JN(r_{1j}, r_{2j})$ where $\{(r_{1i}, r_{2i}) : i = 1..n\}$ is a bucket partition of r_1 and r_2 . That is, JPS has not been considered. This assumes that the cost to generate an output tuple is negligible or the join selectivity factors for all bucket pairs are uniform. In [16], it is pointed out that to generate an output tuple takes longer time than to evaluate a tuple to see if it satisfies one or more predicates. In the case of having a large join product, the output generation could even dominate the response time.

In spite of the importance of this problem, the design of load-balanced parallel join algorithms to deal with JPS has attracted little attention. The core

problem to be solved is the method to estimate join workload of bucket pairs in the presence of JPS. It is very difficult, if not impossible, to give a satisfactory JPS estimation without sufficient profile information about both operand relations, which cannot be assumed available in general [12]. In this paper, we propose a flexible two-phase scheduling parallel hash join algorithm for both RS and JPS problems. Two sets of scheduling heuristics are proposed, and performance of these heuristics are studied against various parameters. We show that many of the existing parallel join algorithms can be regarded as a special case of our dynamic algorithm. The cost of our algorithm is based on the nature of data skew. It can be as efficient as most existing algorithms for some types of data skew when there is certain semantic information about operand relation available. Moreover, it can also handle, at an extra cost, the data skew cases which other existing algorithms cannot approach.

The reminder of this paper is organised as follows. Section 2 presents a new data skew classification and some related properties. A typical static scheduling parallel join algorithm is studied in Section 3, with a new scheduling heuristic rule. As it is not always possible to achieve load-balancing by static scheduling, a two-phase scheduling approach is proposed for parallel hash join computation in Section 4. A detailed study on when the second scheduling phase is necessary is also included in this section. Further performance improvement are discussed in Section 5. We conclude this paper in Section 6.

2 Data Skew Classification

Data distribution is widely modelled by using the Zipf's law in parallel join computation research [14, 19, 18, 16]. Let a_i be the appearance number of i^{th} most frequently appeared value in relation r , Zipf's law states that [10]

$$a_i = \frac{|r|}{i^{1-\theta} \cdot \sum_{j=1}^{\ell(r)} \frac{1}{j^{1-\theta}}} \quad (2)$$

where $\ell(r)$ is the number of distinct values in r . When the Zipfian parameter $\theta = 1$, the Zipf distribution becomes a uniform distribution; $\theta = 0$ models highly skewed distributions; and $\theta = 0.5$ indicates mild-skewed distributions such as the distribution of personal income. When $\theta = \log .80 / \log .20 = 0.1386$, Equation (2) fulfils the commonly observed "80-20" rule of thumb that states that 80 percent of the transactions deal with the most active 20 percent of a file; and the same applies to this 20 percent.

Most of the existing data skew classifications (e.g., [12, 17]) consider data skew in a single relation, and are concentrated on the sources of data skew. To study the relationship between IS, RS and JPS and to deal with them, what we are interested in is the effects, rather than the sources, of various types of data skew, and the co-relationship of data skew in both operand relations of a join. In this paper, we classify IS and RS into the following four categories: 1) *uniform distribution (NS)*: both operand relations are uniform; 2) *single skew (SS)*: only one operand relation is uniform; 3) *double skew (DS)*: both relations are nonuniform, but the value appears more often in one operand also appears more often in another; and 4) *messy skew (MS)*: both relations are nonuniform, and value appearance frequency in one relation is not related with that in another.

This classification can lead to a better understanding of relationship among IS, RS and JPS. The following properties have been observed using this classification [20]: **Property 1**: after hash partitioning, all bucket pairs inherit the same data skew category from the unpartitioned operand relations; **Property 2**: for a NS or SS join, the assumption that equal bucket sizes lead to equal workload is correct (thus many of existing parallel join algorithms can achieve load-balancing) provided that there is an inclusion dependency relationship between the two operands. **Property 3**: a DS join always produces a larger output than a NS join, and a SS join produces the same amount of output as a NS join, if the tuple numbers and the value sets of the join attributes of the two operand relations of one join operation are the same as those of another join operation respectively.

As to be used in this paper, this classification also provides a systematic way to test performance of parallel join algorithms against various types of data skew.

3 Single-Phase Scheduling Algorithm

In this section, we study parallel hash join algorithms based on static scheduling. Different possible scheduling heuristics are tested. We show that a new static scheduling heuristic we proposed according to the new data skew classification performs better than other heuristics for all types of data skew. We also show that single-phase scheduling algorithm cannot always maintain load-balancing.

3.1 Algorithm

Algorithm 1 is a typical parallel hash join algorithm of static scheduling. We regard it as a representative of algorithms in this category (e.g., [8, 7, 5]). For simplicity, a tuple is represented by its join attribute hereafter. To concentrate on performance of parallel join computation, we assume that both operand relations are evenly distributed across all processors before join (i.e., r_{1i} and r_{2i} reside on processor p_i , $1 \leq i, j \leq n$, and $|r_{1i}| = |r_{1j}| \wedge |r_{2i}| = |r_{2j}|$), and the join result is not required to collect after join.

Algorithm 1 static parallel hash join algorithm

```

begin
1) for  $i = 0$  to  $n$  do in_parallel
2)   if ( $i > 0$ ) then /* for ordinary processor  $p_i$  */
3)     partitions by hashing  $r_{1i}$  and  $r_{2i}$  into
       bucket pairs  $\{(r_{1ik}, r_{2ik}) : k = 1..N\}$ ;
4)     reports  $|r_{1ik}|$  and  $|r_{2ik}|$  to  $p_0$ ,  $k = 1..N$ .
5)     waits for the task pool partitioning
       info from  $p_0$ ;
6)     does  $(\bigcup_{k=1}^n r_{1k, t.tid}) \bowtie (\bigcup_{k=1}^n r_{2k, t.tid})$  for
       all  $t \in T_i$ ;
7)   else /* for the coordinator  $p_0$  */
8)      $T = \emptyset$ ; /* initialise the task pool */
9)     for  $j = 1$  to  $N$  do
10)      adds  $\langle j, \sum_{k=1}^n |r_{1jk}|, \sum_{k=1}^n |r_{2jk}|, 0 \rangle$  to  $T$ ;
11)    end_for
12)    partitions  $T$  to  $\{T_k : k = 1..n\}$  by heuristic;
13)    sets  $t.loc = j$  if  $t$  is put into  $T_j$ ;
14)    broadcasts partitioning info to all processors;
15)  end_if
16)end_for
end .

```

In Algorithm 1, there is a scheduling phase between the hash partitioning phase and the parallel join phase. In the partitioning phase, a set of bucket pairs are generated. The number of bucket pairs, N , is larger than the number of the processors, n , such that these bucket pairs can be merged into larger but balanced tasks (the effects of bucket number will be discussed later). One processor, p_0 , is designated as the *coordinator*, and all other processors are *ordinary* processors. A *task pool* T is maintained by the coordinator. Each record of T is a task with the following attributes: the task identifier tid , sizes of operand buckets $size1$ and $size2$, and loc – the identifier of the processor where the operand buckets are stored.

Lines 3–5 of Algorithm 1 are for the ordinary processors to partition their local data and report such a

partition to the coordinator p_0 . Lines 8–14 are for the coordinator processor p_0 to obtain global partitioning information and determine a destination processor for each bucket pair. Then, each processor computes the joins assigned to it (line 6). Clearly, the performance of Algorithm 1, in terms of load-balancing, depends significantly on the heuristic rule used. We compare various heuristic rules in the next subsection.

3.2 Scheduling Heuristics

A fundamental scheduling problem, known as the *bin-packing problem*, is to assign N independent tasks of different execution costs to a collection of n identical processors in an effort to minimize the overall finish time. This problem is readily verified as NP-hard. Fortunately, there exists a number of near-optimal approximation algorithms with linear or sub-linear time complexities. The *longest-processing-time-first* (LPT) heuristic, which always assigns the currently largest task to the currently least loaded processor, can solve such a task scheduling problem efficiently [6].

The LPT scheduling strategy is used in some static scheduling algorithm to obtain equal-sized buckets by packing subbuckets to overcome the RS problem (e.g., [19]). To assign tasks according to their workload, the following heuristics can be used based on various applications of bucket sizes, which are probably the only available information before a join starts. We assume that r_1 is the more skewed relation if the skew degree of two relations are different.

- HP_1 : *packing by $|r_{1i}|$* . The workload of a bucket join task is expected to be indicated by the bucket size of the more skewed operand relation. This can be used when the local join is still a hash join and r_1 is the probing relation (e.g., [5]).
- HP_2 : *packing by $|r_{2i}|$* . This is opposite to HP_1 .
- HP_3 : *packing by $|r_{1i}| + |r_{2i}|$* . This heuristic rule assumes that the sum of tuples from both operand buckets determines the workload. This can be used when the local join algorithm is based on sort-merge method [18].
- HP_4 : *packing by $|r_{1i}| \cdot |r_{2i}|$* . This assumes that the product of tuples from both operand buckets determines the workload. As the nested-loops join algorithm is often used to implement the local bucket join (as in our simulation experiments), this rule uses the tuple comparison cost as the indicator of total local join cost;
- HP_5 : *packing by $|r_{1i}| \cdot |r_{2i}| \cdot (1 + \frac{|r_{1i}|}{|r_1|} \cdot N) \cdot (1 + \frac{|r_{2i}|}{|r_2|} \cdot N)$ where N is the number of partitions*. We suggest this formula to determine a relative order of

workload among bucket pairs because join selectivity is related to data skew and a double skewed join can potentially produce a larger output.

After bucket partitioning, the join task $T = r_1 \bowtie r_2$ is decomposed into N subtasks $t_i = r'_{1i} \bowtie r'_{2i}, 1 \leq i \leq N$. Let $T_i^h \subseteq T$ be all the tasks assigned to processor p_i based on scheduling heuristic rule $h, 1 \leq i \leq n$. Then the inaccuracy of a heuristic h is measured as:

$$Inaccuracy = \frac{\max_{i=1}^n \sum_{t \in T_i^h} JN(t)}{\max_{i=1}^n \sum_{t \in T_i^{RC}} JN(t)} \cdot n - 1 \quad (3)$$

where T_i^{RC} consists of the tasks assigned to p_i by the LPT heuristic using the real join costs.

HP	NS	SS	DS	MS	Average
HP_1	10.55	12.55	10.76	23.71	14.39
HP_2	8.12	32.33	27.27	38.91	26.66
HP_3	11.53	17.56	11.11	35.26	18.87
HP_4	10.43	12.54	10.76	23.71	14.36
HP_5	5.05	6.32	6.63	10.19	7.05

Table 1: Inaccuracy rate of HP heuristics (%)

Table 1 contains performance comparisons of various HP heuristics. These data are obtained from a range of random joins with join selectivity factor $0.0002 \leq \bar{\sigma} \leq 0.0013$ ($\bar{\sigma} = 0.0006$ by average). The cases of severe skew, in which there exist some tasks which are too large to be scheduled to maintain load-balancing, are excluded. It is clear that HP_5 performs better than others, and its accuracy is stable for different $\bar{\sigma}$ and skew types.

4 A Two-Phase Scheduling Approach

Due to the inaccuracy of scheduling heuristics and the existence of unexpected JPS, the workload carried with the bucket pair cannot be predicted according to only bucket sizes. Thus, the workload can be either underestimated or overestimated. We propose a two-phase scheduling approach in this section such that the estimation inaccuracy can be compensated at run time.

4.1 Algorithm

Instead of assigning multiple tasks to each processor before the local join phase starts in Algorithm 1, an ordinary processor in Algorithm 2 obtains one task each time from the coordinator when it becomes

idle. This on-the-fly task assignment is also heuristic-based.

Algorithm 2 *dynamic parallel hash join algorithm*

```

begin
1) for  $i = 0$  to  $n$  do in_parallel
2) /* partitioning and 1st-phase scheduling */
3) if ( $i > 0$ ) then /* for ordinary processor  $p_i$  */
4)   partitions  $r_{1i}, r_{2i}$  to  $\{(r_{1ik}, r_{2ik}) : k = 1..N\}$ ;
5)   reports  $|r_{1ik}|$  and  $|r_{2ik}|$ ,  $1 \leq k \leq N$ , to  $p_0$ ;
6)   waits for task pool partitioning info from  $p_0$ ;
7)   obtains  $r'_{i,t,tid} = \bigcup_{k=1}^N r_{i,k,t,tid}$ ,  $i = 1, 2$ 
      for all  $t \in T_i$ ;
8) else /* for the coordinator  $p_0$  */
9)    $T = \emptyset$ ;
10)  for  $j = 1$  to  $N$  do
11)    add  $\langle j, \sum_{k=1}^N |r_{1jk}|, \sum_{k=1}^N |r_{2jk}|, 0 \rangle$  to  $T$ ;
12)  end_for
13)  partitions  $T$  to  $\bigcup_{k=1}^N T_k$  using an HP;
14)  sets  $t.loc = j$  if  $t$  is put into  $T_j$ ;
15)  broadcasts partitioning info to all processors;
16) end_if
17)
18) /* 2nd-phase scheduling and local joining */
19) if ( $i > 0$ ) then /* for ordinary processor  $p_i$  */
20)   while ( $t \neq \emptyset$ ,  $t$  is task assigned by  $p_0$ ) do
21)     gets  $(r'_{1,t,tid}, r'_{2,t,tid})$  from  $p_{t.loc}$ ;
22)     executes  $r'_{1,t,tid} \bowtie r'_{2,t,tid}$ ;
23)   end_while
24) else /* for coordinator  $p_0$  */
25)   done = 0; /* num of finished processors */
26)   while (done <  $n$ ) do
27)     waits for a request from  $p_0$ ;
28)     if ( $T = \emptyset$ ) then  $t = \emptyset$ ; done = done + 1;
29)     else removes  $t \in T$  using an HS heuristic;
30)     end_if
31)     returns  $t$  to the requester;
32)   end_while
33) end_if
34) end_for
end .

```

The first phase of Algorithm 2 (lines 2–16) is similar to the scheduling phase of Algorithm 1. However, it is not necessary for a task to be processed at where it is allocated to by this phase when JPS occurs. The second scheduling phase (lines 18–33) is used to re-balance workload dynamically during the join phase to cope with JPS. This phase also contains two segments, one for ordinary processors (lines 20–23), and another for the coordinator (lines 25–32). In this phase, each ordinary processor obtains only one

bucket pair each time from p_0 (line 20). On receiving a task from the coordinator (line 27), a processor obtains the data of the task from the processor where these data are stored (line 21) and joins them locally (line 22). Once it finishes, another task is requested until all bucket pairs have been processed. The coordinator processor assigns tasks using some heuristic rules (line 29), to be discussed later, to compensate for the estimation inaccuracy in order to minimize the total response time of the join as well as the scheduling overhead.

Clearly, the second phase incurs extra cost in both scheduling and data exchange. The gains obtained from paying this price should be larger than the cost of load-imbancing. Therefore, the purpose of the second phase scheduling heuristic should be to avoid any unnecessary overhead if the first phase scheduling can already lead to a load-balanced execution.

4.2 Scheduling Heuristics

To apply the LPT heuristic (or any other heuristics to the bin-packing problem), the current workload of each processor and the workload for each remaining task should be determined. We have shown that HP_5 is a good estimation to the workload of subjoin tasks. In Algorithm 2, the solution to the estimation of current workload at each processor is simple: *the one that finishes earlier must have a smaller accumulated workload.*

Now, we give the heuristic rules of HS for the second phase scheduling.

- HS_1 : a processor chooses any of tasks assigned to it in the first phase. In this case, Algorithm 2 becomes the same as a static scheduling algorithm (i.e., Algorithm 1). HS_1 is designed to avoid the overhead of re-scheduling in the case that the estimate the workload in the first phase is already precise enough.
- HS_2 : the coordinator repeats the HP heuristic used in the first phase to choose a task. By re-applying the same HP rule, the processor with the smallest workload is then determined by the actual workload. Thus, the inaccuracy of the workload estimation in the first phase can be compensated. Note that without this scheduling, this inaccuracy of the estimation in the first phase may be accumulated on some processors.

Table 2 contains the results of HP rules coupled with HS_2 , obtained under the similar conditions Table 1 is obtained. Compared with Table 1, it is clear that HS_2 can make a more balanced workload distribution

across processors. Using HP_5 and HS_2 , the estimated workload on each processor is very close to the real workload. Note that some negative values in Table 2 (and also in Figure 1) are because that the response time of scheduling by the real cost is obtained by using the LPT heuristic, which is also a heuristic rule.

HP	NS	SS	DS	MS	Average
HP_1	3.46	1.65	0.78	9.49	3.85
HP_2	3.21	8.76	9.51	21.02	10.63
HP_3	3.46	4.98	0.78	9.46	4.67
HP_4	3.46	1.64	0.78	9.39	3.82
HP_5	-0.05	0.03	-0.03	0.19	0.04

Table 2: Inaccuracy rate of HP with HS_2 (%)

In the next subsection, we will present results of an empirical study in order to decide which HS heuristic to use for a given join.

4.3 Performance Study

We compare the performances between our two-phase scheduling algorithm and Algorithm 1, which is a representative of the static scheduling algorithms (e.g., [8, 7, 5]), against varying parameters such as join selectivity factor, skew degree, partition number, processor number and the local cost function. Readers are referred to the papers, as cited in Section 1, on static scheduling algorithms for a performance comparison between a static scheduling algorithm and other parallel join algorithms.

A simulation system is designed to compare these heuristics. In this system, each tuple is expressed as a 4-byte integer. The values of the distinct tuples are generated at random. The number of overlapped values (thus, the join selectivity factor) can be controlled. The Zipf's Law is used to determine the appearance number of repeated values, where the Zipfian parameters θ_1 and θ_2 for r_1 and r_2 can be set to simulate different data skew types and different data skew degrees.

The hash function for the initial bucket partitioning is $f(x) = x \bmod N$ where N is the number of buckets. This is regarded as a uniform partitioning function.

The cost function $JN(r_1 \bowtie r_2)$ is of the form

$$JN(r_1 \bowtie r_2) = \mu_1 \cdot |r_1| \cdot |r_2| + \mu_2 \cdot |r_1 \bowtie r_2| \quad (4)$$

where $\mu_1 = 1$ and $\mu_2 = 2$ unless otherwise stated. One can set $\mu_1 = 0$ for simulating the applications where the output generation dominates the processing

time (e.g., [15]), and for the study on the relationship between IS and JPS; or set $\mu_2 = 0$ for the applications where JPS is ignored.

The average workload for n processors to execute a join which is decomposed into $T = \{t_i : i = 1..N\}$ is defined as

$$JN_{avg} = \frac{\sum_{t \in T} JN(t)}{n} \quad (5)$$

Let $\{T_i^h : i = 1..n\}$ be a partition of T , where T_i^h is defined as before. The following performance measures are used:

$$JN_{opt} = \frac{\max_{i=1}^n \sum_{t \in T_i^{RC}} JN(t)}{JN_{avg}} - 1 \quad (6)$$

JN_{opt} is the cost obtained by using the LPT scheduling heuristic based the real cost of each task (note that it is possible for some other heuristics to lead to a workload distribution that is closer to JN_{avg} than JN_{opt} occasionally). We also use

$$JN_{HP_j+HS_j} = \frac{\max_{i=1}^n \sum_{t \in T_i^{HP_j+HS_j}} JN(t)}{JN_{avg}} - 1 \quad (7)$$

where HP_j and HS_j are heuristics for the first and second scheduling phases. If there exists any $t \in T$ such that $JN(t) > JN_{avg}$, we say this task set T is not *schedulable*. In our experiments we exclude any task decomposition which results in non-schedulable subtasks.

4.3.1 Effects of Join Selectivity

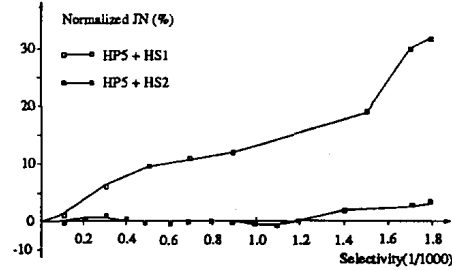


Figure 1: Normalised performance ($JN_{HP_i+HS_j} - JN_{opt}$) of HS heuristics with $\bar{\sigma}$

Figure 1 shows the effects of join selectivity factor. The performance results are normalised by the results obtained from scheduling by the real costs. These results are from a range of randomly generated joins with different data skew patterns and different skew degrees. The normalised results for a parallel hash join algorithm without any scheduling range from 30%

to 85% in our tests; and the results based on scheduling by real costs range from 1% to 10%. From Figure 1 one can see that when $\bar{\sigma} \leq 1/10000$, HS_2 cannot make much improvement with comparison to HS_1 . With the increase of $\bar{\sigma}$, the effect of JPS becomes apparent, and the gains of using HS_2 increases accordingly. Around the order of $1/1000$ are typical join selectivity factors in real database applications [1]. When $\bar{\sigma} > 1/1000$, the gain of Algorithm 2 with HP_3 and HS_2 over a static scheduling parallel join algorithm becomes more evident.

4.3.2 Effects of Partition Number

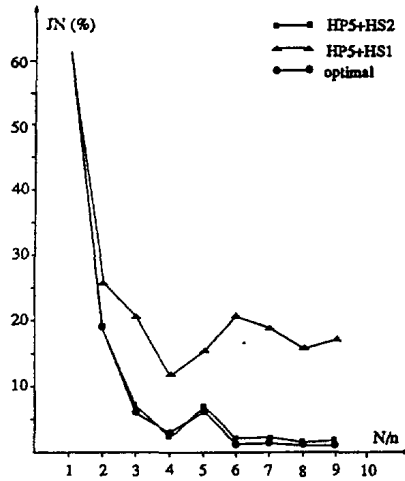


Figure 2: Performance with varying partition numbers for mild-skewed joins

The performance of a scheduling algorithm also depends on the number of partitions. It is obvious that the scheduling costs for both phases of Algorithm 2 increase with the increase of the number of partitions, while the resultant workload distribution can become more even at the same time. Intuitively, a smaller partition number is required for a less skewed join. From our simulations, we find a rule of thumb classification of data skew situations for *mild skew* and *severe skew*. The cases of the mild skew is listed in Table 3, where θ_1 and θ_2 are Zipfian parameters in r_1 and r_2 respectively. And the severe skew describes other cases. Figure 2 gives the relationship of the performance of Algorithm 2 and number of partitions for mild-skewed joins, where the x-axis is the ratio of partition number over processor number. For mild-skewed joins, the number of partitions must be larger than at least 4 times of that of processors (other sim-

NS	SS	DS	MS
always	$\theta > 0.3$	$\theta > 0.5$	$\theta > 0.3$

Table 3: Cases of mild skew ($\theta = \min\{\theta_1, \theta_2\}$)

ulation data in this paper is based on the ratio of 6).

4.3.3 Effects of Processor Number

Figure 3 shows the performance of Algorithm 2 using two HS heuristics over Algorithm 1 with varying number of processors (with fixed $N = 80$). With the increase of processor number, the effect of JPS on load-balancing also increases; thus, the gains of the dynamic scheduling over the static scheduling increases.

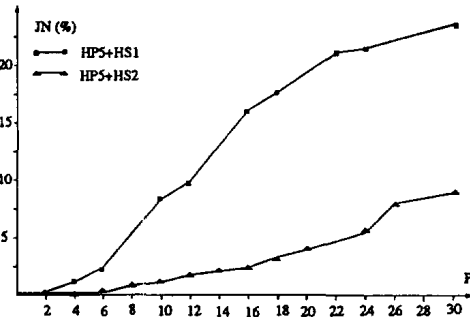


Figure 3: Performance with varying processor numbers

4.3.4 Effects of Cost Functions

Figure 4 shows the performance of two HS heuristics with varying local join functions. When μ_2 increases in the function $JN(t)$ (i.e., Equation (4)), the effects of JPS increase accordingly. Thus, as one can observe from Figure 4, the gains of HS_2 over HS_1 increase proportionally. In our experiments, there are some cases that an originally schedulable task partitioning become not schedulable when μ_2 increase.

4.4 Selection of HS Heuristics

From our discussions above, one can see that though HS_2 performs always better than HS_1 , the gains of using HS_2 over using HS_1 depends on some other factors such as join selectivity and skew types. Conservatively, the HS heuristic for Algorithm 2 to use is HS_2 . This may be the only choice when there

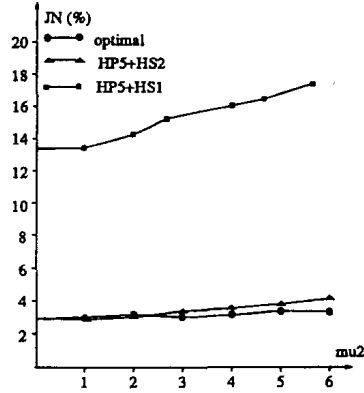


Figure 4: Performance with varying μ_2 in JN function

is no sufficient information about the operand relations is available. Now we discuss some cases where heuristic HS_1 can be used.

4.4.1 NS and SS Joins With Inclusion Dependency

For the case where there exists an inclusion dependency between the join attributes and the data distribution on the relation with the larger value set being uniform, we choose HS_1 . By the second property in Section 2, we know that in this case any tasks with both operands being the same size must have the same workload. That is, the load-balancing of parallel join computation can be obtained by assigning all processors with buckets which are equal in size. In this case, Algorithm 2 becomes the same as a static scheduling algorithm (e.g., [7]).

It is worthwhile to mention that this case covers a large percentage of joins in relational database systems. The following are some important applications: **Case 1: foreign key join** – in real database applications, most of the meaningful joins occur on the attributes which have a foreign key constraint between them. If $r_2.A$ is a foreign key referencing to $r_1.A$, then $r_1.A$ must be uniform (each tuple value appear only once) and $r_2.A$ is included by $r_1.A$. Therefore, it is sufficient for Algorithm 2 to use HS_1 to achieve load-balancing for any foreign key join; **Case 2: domain completeness** – if $r_1.A$ is *complete* on its domain D [11], i.e., every value in D appears in $r_1.A$, and $r_1.A$ is uniform, then HS_1 can be used for the joins between any relation defined on D and $r_1.A$, no matter what kind of data distribution another operand relation has; **Case 3: semi-join reduction** – semi-join is

often used in distributed database systems to reduce data transfer cost. If one operand relation, say r_1 , is uniform on its join attribute, and the other one, $r_2.A$, has been semi-join reduced by $r_1.A$, then HS_1 works for $r_1 \bowtie (r_2 \ltimes r_1)$.

As we mentioned before, most of parallel join algorithms only make efforts to even up bucket sizes for load-balancing. Though it is not sufficient for these algorithms to maintain load-balancing for all kinds of join applications, they do work well in many cases. For these joins, HS_1 is used; so Algorithm 2 incurs no extra cost than a static scheduling algorithm.

4.4.2 Join With Very Small $\bar{\sigma}$

For the case where the relational join selectivity is smaller than $1/10000$, from our experiments (see Figure 1) we find that HS_1 can also lead to a workload distribution close to the distribution based on real cost. When the join selectivity is small enough, the cost of a local join can be approximated as

$$JN(r_{1i}, r_{2i}) \approx g_1(|r_{1i}|, |r_{2i}|)$$

That is, JPS can be neglected. Table 4 gives a performance comparison with very small join selectivity factors. The raw performance is for a parallel hash join algorithm without using either static or dynamic scheduling phases.

5 Further Discussions

5.1 Handling Severely Skewed Data

Clearly, the best workload distribution for any scheduling algorithm on a join task set T with n processors to achieve is

$$\max_{t \in T} \{JN(t), \frac{\sum_{t \in T} JN(t)}{n}\} \quad (8)$$

Our discussion in this paper is based on the assumption that all the joins are mild-skewed, i.e.,

$$\max_{t \in T} JN(t) \leq \frac{\sum_{t \in T} JN(t)}{n} \quad (9)$$

For a severe-skewed join in which some tasks are larger than the average size, the response time could be dragged by those non-schedulable tasks when a “many-to-one” task-to-processor mapping is used. This problem can be solved by allowing more than one processor to process one non-schedulable task. Algorithm 2 can be modified to use *many-to-many* task-to-processing mapping in the following way: a

$\bar{\sigma}$ (1/1000)	0.01	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
Raw	11.90	43.43	16.90	39.41	27.18	20.00	34.08	22.63	31.60
Optimal	1.25	6.57	2.00	6.24	4.02	4.63	7.72	5.17	4.78
HP_5, HS_1	1.30	9.37	2.00	8.21	6.78	5.37	11.80	9.37	7.80
HP_5, HS_2	1.25	6.57	2.00	6.33	4.08	4.60	7.38	5.17	4.77
Gain	0.05	3.00	0.00	1.88	2.70	0.77	4.42	4.20	3.03

Table 4: Gains of HS_2 over HS_1 with small join selectivity factors (%)

timestamp is attached to each record in the task pool; and a field to record the processor to which this task is assigned (in case of being assigned to more than one processor, the first processor is recorded). Initially the timestamp for each task is set to “free”; it is set to the time when the task is assigned (this may be repeated as a task can be assigned more than once to several processors). When the first receiver of a task requests for another task, the timestamp of the task is set to “done”. When the coordinator p_0 finds a task exceeds the expected finish time, an idle processor can be assigned to this task. Then the first processor for a task redistributes workload among the later-coming processors. The parallel nested-loops join algorithm is used to process the same task, which cannot be further split by hashing, and the percentage of the remaining task needs to be moved to other processors can be determined using the heuristics in [13].

5.2 Reducing Scheduling Overhead

Both scheduling phases incur some scheduling overhead in Algorithm 2. For the first phase of scheduling, the sizes of bucket pairs need to be sorted in order to use the LPT heuristic at the coordinator processor; and some data need to be moved from one processor to another. For the second phase of scheduling, an extra cost for communicating between an ordinary processor and the coordinator to assign a task is attached to each task when HS_2 is in use. And some data need to be moved again from the processor where this task is stored to where this task is to be processed.

Compared with a static scheduling algorithm, the major extra cost for a dynamic scheduling algorithm is for moving data in the second phase. It is possible to modify Algorithm 2 such that a tuple is only moved to where the actual join is to be done directly from where it resides initially as in a static scheduling algorithm. In this case, a processor needs to request data from all processors (line 21), instead of from only one processor as in Algorithm 2, to obtain operand data before a local join. The advantage of using the first data movement (at line 7) in Algorithm 2 is that the run-time

interference among processors can be reduced. In the viewpoint of total communication cost, it is obvious that this modification can reduce communication cost. In terms of response time, which algorithm is better depends on the multiprocessor systems in use.

As the scheduling overhead is proportional to the number of tasks, it could be beneficial to merge small tasks in the first phase such that the scheduling cost of the second phase can be reduced.

6 Conclusions

Most existing parallel join algorithms cannot achieve load-balancing when JPS occurs. In this paper, a new parallel hash join algorithm based on dynamic task scheduling has been proposed to solve this problem. A new data skew classification scheme has been proposed to study the relationship among various types of data skew, and it is used to test the performance of parallel join algorithm systematically. The heuristic rule HP_5 is proposed to estimate the workload order of bucket pairs, and the actual workload on each processor at run time can be captured by dynamic nature of the second phase scheduling. Using HP_5 and HS_2 , our parallel join algorithm can lead to a workload distribution which is very close to the workload distribution determined by the real workload. Moreover, heuristic HS_1 is designed to avoid the scheduling overhead for the joins where the dynamic scheduling procedure is not necessary. Some of such cases, which cover a large percentage of joins in practical database applications, have been identified.

References

- [1] D. Bitton, H. Boral, D. J. DeWitt, and W. K. Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.

- [2] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *CACM*, 35(6):85–98, June 1992.
- [3] D. J. DeWitt and et. al. Implementation techniques for main memory database systems. In *ACM SIGMOD 84*, pages 1–8, 1984.
- [4] D. DeWitt et al. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.
- [5] D. DeWitt et al. Practical skew handling in parallel joins. In *Proceedings of 18th VLDB Conference*, pages 27–40, Vancouver, 1992.
- [6] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
- [7] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of 17th VLDB Conference*, 1991.
- [8] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc). In *Proceedings of 16th VLDB Conference*, Brisbane, 1990.
- [9] M. Kitsuregawa, H. Tanaka, and T. Motooka. Application of hash to database machine and its architecture. *New Generation Computing*, 1(1):66–74, 1983.
- [10] D. Knuth. *The Art of Computer Programming (Vol. 3)*. Addison-Wesley, 1973.
- [11] R. Kooi. *The Optimization of Queries in Relational Databases*. PhD thesis, Case Western Reserve University, Cleveland, Ohio, 1980.
- [12] M. S. Lakshmi and P. S. Yu. Limiting factors of join performance on parallel processors. In *Proceedings of 5th International Conference on Data Engineering*, pages 488–496, Los Angeles, 1989.
- [13] H. Lu, M.-C. Shan, and K.-L. Tan. Optimization of multi-way join queries for parallel execution. In *Proceedings of 17th VLDB*, pages 549–560, 1991.
- [14] C. A. Lynch. Selectivity estimation and query optimization in large database s with highly skewed distributions of column values. In *Proceedings of 14th International Conference on VLDB*, 1988.
- [15] P. Valdurize and S. Khoshafian. Parallel evaluation of the transitive closure of a database relation. *J. of Parallel Programming*, 17(1):19–42, 1988.
- [16] C. B. Walton. *Skew And Scalability of Parallel Join On Multicomputers*. PhD thesis, University of Texas at Austin, 1992.
- [17] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of 17th VLDB Conference*, 1991.
- [18] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings of 7th International Conference on Data Engineering*, pages 200–209, Kobe, Japan, 1991.
- [19] J.L Wolf, D. M. Dias, and P.S. Yu. An effective algorithm for parallelizing sort merge joins in the presence of data skew. In *The Second International Symposium on Databases in Parallel and Distributed Systems*, pages 103–115, Dublin, Ireland, 1990.
- [20] X. Zhou. *Parallel Processing in Relational Database Systems*. PhD thesis, The University of Queensland, 1994.
- [21] X. Zhou and M. E. Orłowska. A new dynamic approach for handling data skew problems in parallel hash join computation. In *Proceedings of the IEEE TENCON'93*, pages 133–137, Beijing, 1993. International Academic Press.