Adaptive Parallel Query Execution in DBS3

Luc Bouganim, Benont Dageville, Patrick Valduriez

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Adaptive Parallel Query Execution in DBS3

Luc Bouganim - Benoît Dageville - Patrick Valduriez

N° 2749

Décembre 1995

PROGRAMME 1

Architectures parallèles, bases de
données, réseaux et systèmes distribués

ISSN 0249-6399

Adaptive Parallel Query Execution in DBS3*

Luc Bouganim**, Benoît Dageville***,
Patrick Valduriez

Programme 1 : Architectures parallèles, bases de données,
réseaux et systèmes distribués

Projet Rodin

Rapport de recherche n°2749 - Décembre 1995

24 pages

Abstract: The gains of parallel query execution can be limited because of high start-up time, interference between execution entities, and poor load balancing. In this paper, we present a solution which reduces these limitations in DBS3, a shared-memory parallel database system. This solution combines static data partitioning (by hashing the relations across the disks) and dynamic processor allocation (using shared- memory) to adapt to the execution context. It makes DBS3 almost insensitive to data skew and allows decoupling the degree of parallelism from the degree of data partitioning. To address the problem of load balancing in the presence of data skew, we analyze three important factors that influence the behavior of our parallel execution model: skew factor, degree of parallelism and degree of partitioning. We report on experiments varying these three parameters with the DBS3 prototype on a 72-node KSR1 multiprocessor. The results demonstrate high performance gains, even with highly skewed data.

Key-words: Parallel databases, execution model, shared memory, data skew

(Résumé : tsvp)

A short version of this paper appears in the proceedings of the EDBT 96 International Conference - March 1996, Avignon, France.

* This work has been partially funded by the CEC under ESPRIT project IDEA.
** Email: {Luc.Bouganim}{Patrick.Valduriez}@inria.fr
*** Bull OSS, Echirolles, France. B.Dageville@bull.frec.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 - Télécopie : (33 1) 39 63 53 30

Exécution parallèle de requêtes dans DBS3

Résumé : Un ensemble de facteurs limitent les gains obtenus lors d'une exécution parallèle: temps d'initialisation, interférences entre les entités d'exécution et mauvaise répartition de la charge de travail. Dans ce rapport, nous présentons une solution qui réduit ces limitations dans DBS3, un SGBD parallèle pour architecture à mémoire partagée. Celle-ci combine un modèle de parallélisation basé sur la fragmentation statique des données avec un modèle d'allocation dynamique des processeurs afin de s'adapter à chaque contexte d'exécution. Cette flexibilité permet de fixer le degré de parallélisme indépendamment du degré de fragmentation, mais aussi de répartir finement la puissance de calcul ainsi allouée afin de supporter efficacement des charges de travail non uniformément réparties (Data Skew). Une analyse nous permet de dégager trois importants facteurs modifiant le comportement de notre modèle d'exécution: Degré de biaisage, de parallélisme et de fragmentation. Nous présentons plusieurs mesures de performances en variant ces trois paramètres effectuées avec le prototype DBS3 sur la machine KSR1 munie de 72 processeurs. Les mesures montrent des gains importants même en cas de données fortement biaisées.

Mots-clé : Bases de données parallèles, modèle d'exécution, mémoire partagée, répartition de charge.

1 Introduction

DBS3 (Database System for Shared Store) [Bergsten91] is a parallel database system for shared-memory multiprocessors [DeWitt92a][Valduriez93]. It has been implemented on an Encore Multimax (10 processors, 96 megabytes memory) and on a Kendal Square Research KSR1 (72 processors, 2 gigabytes memory). It supports ESQL [Gardarin92], an extension of SQL with objects and rules. Although DBS3's run-time is designed for large shared-memory systems, the ESQL compiler supports a more general parallel execution model. During the EDS ESPRIT project (1989-1993), the ESQL compiler has been used to generate parallel code for the EDS shared-nothing parallel computer, the now Goldrush product from ICL.

In a shared-memory architecture, each processor has uniform access to the entire database through a global main memory. Thus, the *parallel scheduler,* which allocates processors to the query's operations and controls their execution, has much freedom for balancing the query load onto processors. However, query response time can be hurt by several barriers [DeWitt92a] which must be overcome by the scheduler:

- **start-up time**: before the execution takes place, a sequential initialization step is necessary. The duration of this step is proportional to the degree of parallelism and can actually dominate the execution time of low complexity queries. Thus, the degree of parallelism should be fixed according to the query complexity.
- **interference**: parallel access to shared software and hardware resources (disks, data structures, etc.) can create hot spots which increase waiting time. Parallel operations must be isolated, i.e. working on separated data sets, to minimize interference.
- **poor load balancing**: the response time of a set of parallel operations is that of the longest one. Thus, load balancing must deal with skewed data distributions and operations complexity.

A parallel execution plan is a graph of operations (filter, join, etc..) on database relations. Inter-operation parallelism is obtained by executing different operations in parallel. Intra-operation parallelism is obtained by executing the same operation on different relation fragment. Relation partitioning can be static (on the disks) or dynamic (at run-time).

With static partitioning, relations are physically partitonned using a *parallel storage model* based on a partitioning function like hashing. Relation partitioning typically dictates the degree of intra-query parallelism [Mehta95]. This approach is very popular in research prototypes, e.g. Bubba [Boral90], Gamma [Dewitt90] and Volcano [Graefe94], and commercial products, e.g. DB2, Informix, Tandem and Teradata. Static partitioning reduces well interference between processors as they

work on distinct data sets. It can scale up to large numbers of nodes (hundreds of processors and disks), and works with either shared-memory or shared-nothing multiprocessors. However, static choices (degree of partitioning) have a strong impact on performance as they influence load balancing and the degree of parallelism. In particular, start-up time may hurt the response time of low complexity queries.

Dynamic partitioning is advocated in XPRS [Hong92] and Oracle [Davis92] to overcome the problems of static partitioning. Relations are not stored using a parallel storage model but split, page by page among all the disks. Intra-operation parallelism is then obtained dynamically depending of the number of processors allocated for the operation. Thus, the degree of parallelism can be adjusted to the query complexity and the availability of memory and processors, to yield good load balancing. However, the fact that several processors may access the same data set (i.e. the entire relation) can yield high interference. Furthermore, this approach only works with shared-memory or shared-disk architecture.

Our solution in DBS3 tries to combine the advantages of static and dynamic partitioning. We use static partitioning to reduce interference and for compile-time query parallelization. We use dynamic allocation of processors to operations, independent of the degree of static partitioning, in order to control start up time and load balancing. This hybrid approach also simplifies database tuning since the degree of partitioning is not directly related to the degree of parallelism*.

In this paper, we present the adaptative parallel query execution model of DBS3. To demonstrate the potential performance gains of our solution, we report on experiments varying the skew factor, the degree of parallelism and the degree of partitioning. The performance measurements were done on the KSR1 with 72 processors, with the relations cached in main memory. This main memory assumption is not a restriction of the model but a constraint of our KSR1 configuration which has a single disk.

The paper is organized as follows. Section 2 presents DBS3's parallel execution model which is based on the parallel algebraic language Lera-par to represent parallel execution plans. Section 3 describes the implementation of the parallel execution model on a shared-memory system. Section 4 address the problem of load balancing in the presence of skewed data distributions. A simple analysis outlines three important factors that influence the behavior of our model: skew factor, degree of parallelism and degree of partitioning. Section 5 reports on experiments varying these factors with the 72-node KSR1 version of DBS3.

* The degree of partitioning must be higher or equal than the degree of parallelism.

2 Parallel Execution Model

In DBS3, the compilation phase takes an ESQL query which is optimized [Lanzelotte94] and parallelized [Borla91]. The parallel execution plan produced by the compiler is expressed in Lera-par [Chachaty92] and captures the operations and their control.

Lera-par is a *dataflow* language whose expressive power is an extended relational algebra that supports ESQL. A Lera-par program is represented by a dataflow graph whose nodes are *operators* (like filter, join or map) and edges are *activators*. An activator denotes either a tuple (data activation) or a control message (control activation). In either case, when an operator receives an activation, the corresponding sequential operation is executed. Therefore, each activation acts as a sequential unit of work.

Lera-par's storage model is statically partitioned. Relations are partitioned by hashing on one or more attributes, and relation fragments are distributed onto disks in a round-robin fashion. Thus, the degree of partitioning can be independent of the number of disks. To obtain intra-operation parallelism, each node of the execution plan, whose input is a partitioned relation, gets as many instances as fragments. This yields an extended view of the Lera-par graph (see Figure 1).

Pipelined execution is an important aspect of Lera-par. It is expressed by using data activation between a producer node and a consumer node, which can then operate in parallel as soon as the consumer gets activated.

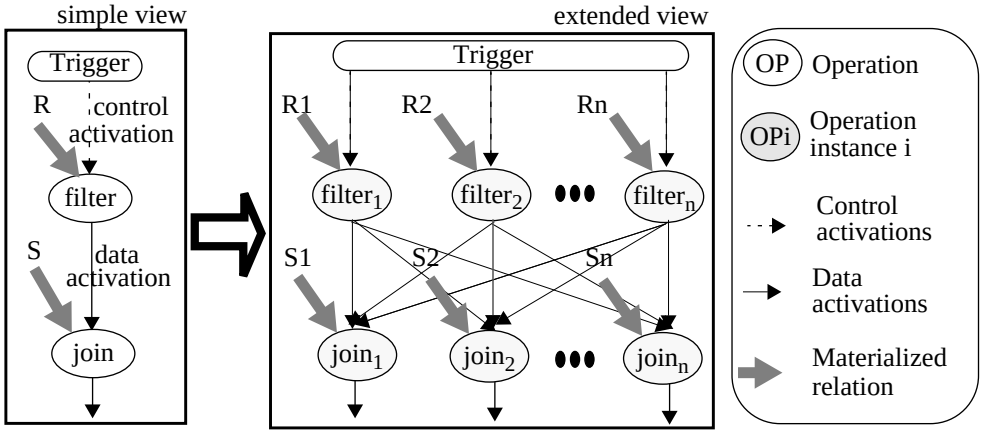


Figure 1: A parallel execution plan in Lera-par

Figure 1 illustrates a simple execution plan which performs a selection (filter) on relation R followed by a join with S. A triggering activation is sent to all filter operation instances, which can then process their associated fragment in parallel. The

result tuples from the filter operation are pipelined to the next join operation. Each result tuple is sent to one join instance which is automatically activated to perform the join with the associated S fragment.

To manage activations, a FIFO queue is associated to each operation instance. There are two kinds of queues, triggered or pipelined. A triggered queue is associated to an operation triggered by a control activation. It receives only one activation which starts the associated operation, e.g. the filter operation in our previous example (see Figure 2).

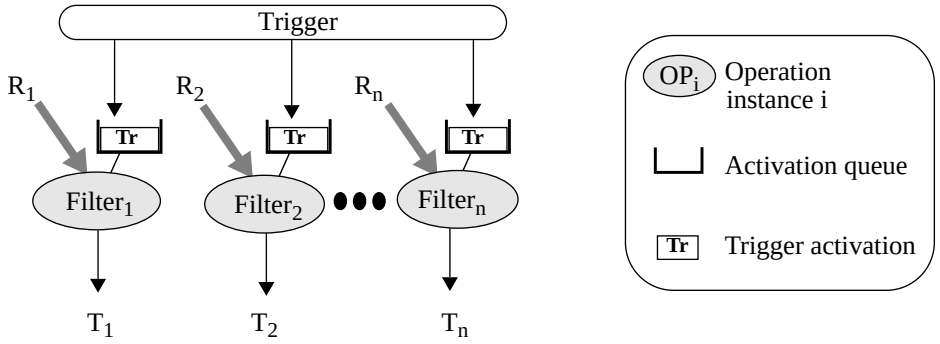


Figure 2: Triggered operation

A pipelined queue is associated to an operation which receives one operand in a pipeline fashion, e.g. the join operation in our example (see Figure 3). In this case, each activation conveys one tuple and the queue will receive as many activations as pipelined tuples.

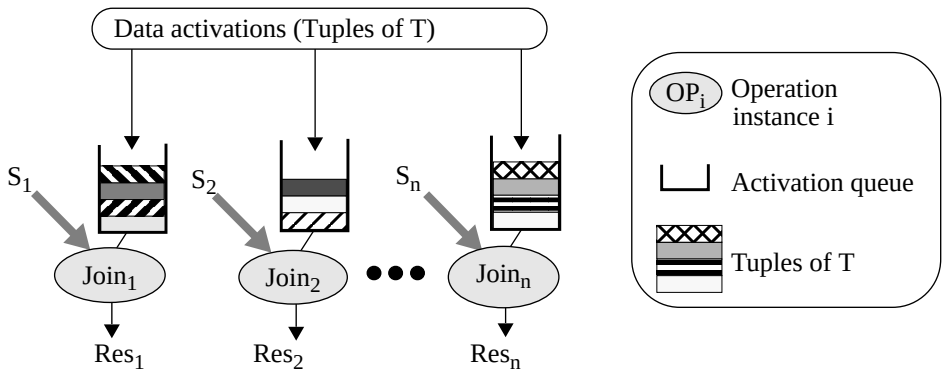


Figure 3: Pipelined operation

3 Shared-Memory Implementation

In a shared-memory architecture, it is possible to uncouple the implementation of the parallel execution model from thread allocation*. The typical thread allocation strategy would assign a single thread per operation instance. Instead, we allocate a pool of threads for the entire operation, independent of the operation instances (and of the degree of partitioning). This is done by allocating the queues of an operation's instances in a shared-memory segment so all the threads of a pool can access all queues associated with the operation. Therefore, the threads can execute code for any activation in order to increase load balancing.

Figure 4 shows the basic data structures used for the implementation of our parallel execution model. Each node of the execution plan is described by an **operation** structure, which uses a table of activation **queues** and a table of **threads** to consume those activations. .

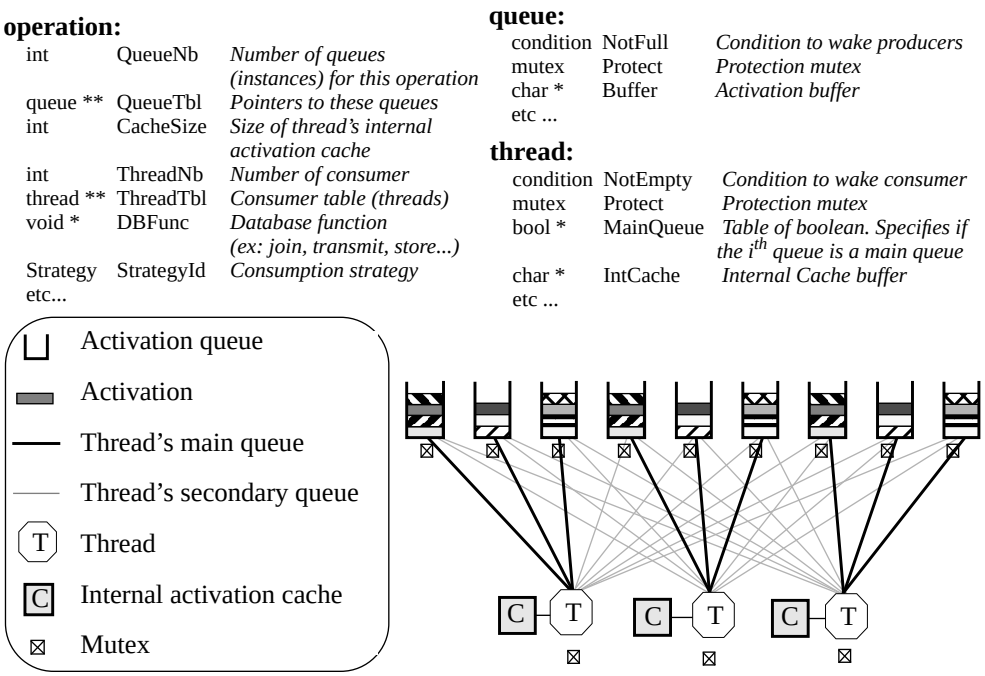


Figure 4: Basic data structures for the parallel execution model

Because of parallel access, each structure is protected by a *mutex* variable. *Condition* variables are used to synchronize consumers and producers. Furthermore, there is an internal cache mechanism for activations in order to reduce interference between activation producers and consumers, and to increase locality of access

* DBS3 performs processor allocation indirectly via threads.

Access conflicts to the activation queues are limited by defining, for each thread, two kinds of queues: main and secondary queues. For each operation, all activation queues are equally distributed among the associated threads and are marked as main queues. Therefore, each queue is the main queue of only one thread but each thread can have several main queues. A thread always tries to first consume the activations of the main queues. As there is a continuous activation flow, there is no interference for queue access. If all the main queues of a thread are empty, the thread would search in secondary queues. Thus, thread utilization is maximized as long as activations are available.

To increase run-time performance, the scheduler must devise the best configuration for the variables *ThreadNb*, *QueueNb*, *CacheSize* and *Strategy* of each operation. We use a top-down approach to fix the number of threads and distribute them on the query's operations. Figure 5 illustrates this approach with a dataflow graph made of pipelined operation chains (called *subqueries*) and result materializations between chains. There are four steps:

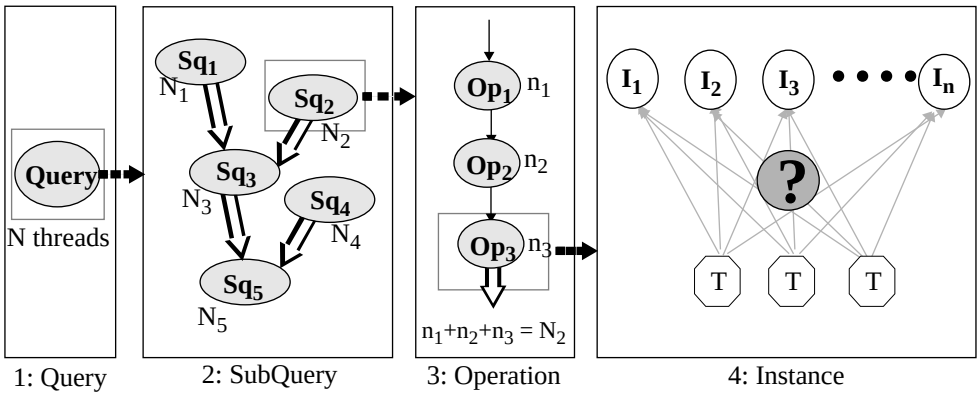


Figure 5: Steps for setting execution parameters

1 **Choosing the number of threads.** Based on the complexity of the query, as estimated by the compiler, we can choose the optimal number of threads using known techniques. For instance, we can compute the best number of threads which minimizes the parallel execution time [Wilschut92]. This number can then be reduced according to the average processor utilization in order to increase the multi-user throughput [Rahm93].

2 **Assigning the threads to subqueries.** The different subqueries of the query can be executed sequentially or in a parallel but dependent fashion. The assignment of the threads to the subqueries can then be done in a bottom up fashion, similar to [Hsiao94]. The execution graph (Figure 5, step 2) is considered as an inverted tree. First the total CPU power is allocated for the root (e.g. Sq5). This CPU power, is then distributed among root children (e.g. Sq3, Sq4) according to sequential complexity estimation of the work for each child and its subchildren. This is done

recursively and generates a set of n independent equations (n is the number of sub-queries). In our example, let T_i be the sequential complexity of Sq_i, N_i be the number of threads for Sq_i, N be the total number of threads for the query, we obtain the set of 5 equations which is easy to solve:

$$N_5 = N \qquad \frac{T_3 + T_1 + T_2}{N_3} = \frac{T_4}{N_4} \qquad N_1 + N_2 = N_3 \qquad \frac{T_1}{N_1} = \frac{T_2}{N_2}$$

$$N_3 + N_4 = N_5$$

3 **Assigning the threads to operations of pipelined chains.** The threads assigned to a simple pipelined chain, are distributed among its operations using the ratio of the estimated complexity of each operation by the estimated complexity of the chain:

$$NbThreads(Op_i) = NbThreads(Chain) \times \frac{Complexity(Op_i)}{Complexity(Chain)}$$

4 **Consumption strategy for operation instances.** For each operation, we must decide on the consumption strategy. Currently, DBS3 supports two strategies: *Random* and *LPT*. For all strategies, main queues are always considered first. Random is the default strategy. Each thread randomly chooses one queue among the non-empty ones, associated with the operation. The LPT (Longest Processing Time First) [Graham69] heuristic should be used in the presence of data skew (see Section 5.4); each thread chooses the activation queue which contains the most expensive activations. Others strategies can also be added for specific problems.

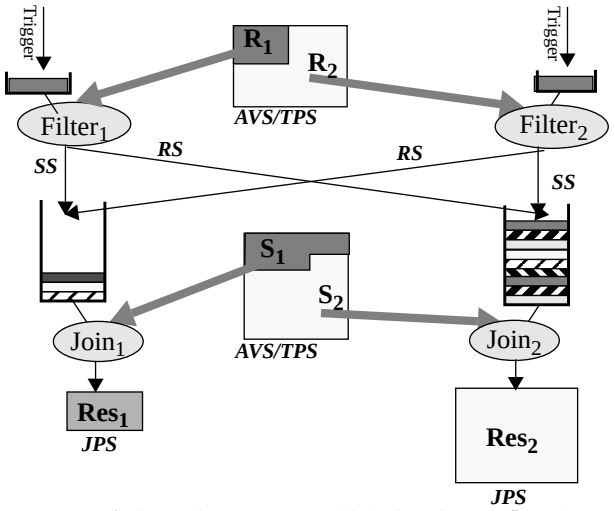
With this approach, steps 2-4 assign the CPU power allocated at step 1 so all the threads have approximately the same amount of work. To summarize, this thread allocation strategy reduces the major barriers of parallel query execution by offering several means to adapt to the execution context (query, data distribution, system load, etc.). First, we can define the degree of parallelism independent of the degree of partitioning. Second, by controlling the number of threads per pool, we can achieve better balancing of CPU power between operations. Finally, each thread can dynamically choose in which queue to consume activations which should yields good load balancing.

4 Load Balancing with Skewed Data Distributions

Several solutions have been proposed to reduce the negative impact from skew. [Kitsuregawa90] presents a robust hash-join algorithm for a specific parallel architecture based on shared-nothing. The idea is to partition each hash bucket in fragments and spread them among the processors (*bucket spreading*). Then a sophisticated network, the Omega network, is used to redistribute buckets onto the proces-

sors. The Omega network contains logic to balance the load during redistribution. [Omiecinski91] proposes a similar approach in a shared-memory parallel system, using the *first fit decreasing* heuristic to assign buckets to processors. Finally, [DeWitt92b] suggests the use of multiple algorithms, each specialized for a different degree of skew, and the use of a small sample of the relations to determine which algorithm is appropriate.

The effects of non uniform data distribution (i.e. skew) on parallel execution [Walton91] are summarized in Figure 6. The example shows a filter-join query applied to two relations *R* and *S* which are poorly partitioned. Such poor partitioning stems from either the data (AVS) or the partitioning function (TPS). Thus, the processing times of the two activations for triggering the operation instances *filter1* and *filter2* are not equal. The case of the join operation is worse. The uneven size of *S* fragments yields different processing times for the activations from the filter operation (AVS/TPS). Furthermore, the number of activations received is different from one instance to another because of poor redistribution of the fragments of *R* (RS) or variable selectivity according to the fragment of *R* processed (SS).



Taxonomy of data skew in parallel databases [Walton91]

- **AVS** (Attribute Value Skew): Skew inherent to the dataset. For example, the 'Paris' value will be more frequent than 'Cannes' in France's resident relation.
- **TPS** (Tuple Placement Skew): Initial unbalanced repartition of tuples. With the previous example, partitioning on city attribute will lead to TPS.
- **SS** (Selectivity Skew): Selection selectivity varies from one instance to another.
- **RS** (Redistribution Skew): Redistribution leads to unbalanced temporary partitions.
- **JPS** (Join Product Skew): The join selectivity varies from one instance to another.

Figure 6: Taxonomy of data skew on Filter-Join example

Assuming all the data to be processed are main-memory resident, the problem of skewed data distribution reduces to that of optimizing CPU utilization. Thus, to obtain a query response time that is insensitive to skew, we must equally balance the load of each operation onto all the allocated threads. In the rest of this section, we consider the effect of skew on a single operation.

4.1 Analysis

We now analyze the effect of skew in our model on an operation execution. The objective is to derive an analytical formula that gives the overhead on the operation response time induced by skew. This is important in order to understand the major parameters which can be tuned to reduce the effect of skew.

In DBS3, each thread can access all the activation queues of the operation. The default mode of queue consumption is random, i.e. the thread randomly chooses one queue among the non empty ones associated with its operation. Thus, thread utilization is maximum as long as activations are available. However, at operation end, when there is no more activation, threads become idle as they terminate until the last thread completes processing its activation.

Let us now consider an operation execution with a activations and n threads. P indicates the average processing time for an activation. To maximize thread utilization, we must have $n \leq a$, otherwise, n-a threads would be idle. In the worst case, one thread will consume the last activation when all other threads have terminated. During the processing of this last activation, only one thread is active and thread utilization is minimum.

Let T_{ideal} be the ideal execution time for the operation, when all threads complete simultaneously, and T_{worst} be the worst time. To compute v, the overhead of the worst time, we have the following equation for T_{worst}:

$$T_{worst} = (1 + v) \times T_{ideal} = (1 + v) \times \left(\frac{a \times P}{n} \right) \qquad (1)$$

The worst case scenario can be seen with two phases. In the first phase, all activations but the most expensive one are processed. Let P_{max} be the time to process the last activation (i.e. the most expensive one), the execution time for the first phase is: $((a \times P) - P_{max}) / n$:

The second phase corresponds to the processing of the last activation whose time is P_{max}. Thus, we have:

$$T_{worst} \leq \frac{(a \times P) - P_{max}}{n} + P_{max} \qquad (2)$$

By substituting T_{worst} with the previous formula from equation (1), we can compute v as follows:

$$v \leq \frac{P_{max}}{P} \times \frac{(n-1)}{a} \qquad (3)$$

Equation (3) exhibits that the overhead depends on three factors: skew factor (P_{max}/P), degree of parallelism (n) and number of activations (a). For the latter, we have two interesting cases, depending on whether a is high or low:

- **The number of activation is high.** This case corresponds to a pipelined operation with lots of tuples. a is then equal to the cardinality of the pipelined relation. Thus, v is quite small and the execution time of the operation is close to T_{ideal}. This good result is independent of thread consumption strategy and of skew.
- **The number of activation is low.** This case corresponds to a pipelined operation with few tuples or to a triggered operation. The overhead due to skew can then be quite serious. A solution is to use a consumption strategy that reduces this overhead, like LPT (Longest Processing Time First) [Graham69] which processes the most expensive activations with highest priority. To implement this heuristic in DBS3, we do not need to estimate the execution time of each activation. Instead, we can arrange the operation instance in decreasing order of estimated execution time, for instance, based on static information on fragment sizes.

5 Experiments

In this section, we show though experimentation how our adaptative parallel execution model can be exploited to deal with a varying execution context. We first present the environment for the experiments, in particular, the KSR1 machine and the queries. Then we address the problem of load balancing in the presence of skewed data distributions. We report experiments by varying three important factors that influence the behavior of our model: skew factor, degree of parallelism and degree of partitioning.

5.1 The KSR1 Machine

DBS3 runs on a KSR1 multiprocessor at Inria*. The KSR1 machine provides a shared-memory programing model in a scalable highly parallel architecture [Frank93]. It has a hybrid architecture in the sense that the memory is physically distributed and virtually shared using hardware mechanisms. Each processor has its own 32 Megabytes memory, called local cache.

* Although KSR1 production has been stopped, our machine will be up and running for another two years since many researchers like it for experimentation.

The Allcache memory system provides the user with a virtual shared-memory space which corresponds to the collection of all the local caches. When a processor accesses a data item which is not already cached, this item is shipped transparently to the local memory of this processor by the Allcache memory system. Compared to conventional shared-memory machines (like the ENCORE Multimax machine) this memory organization may have some negative behavior which will be studied in the next section.

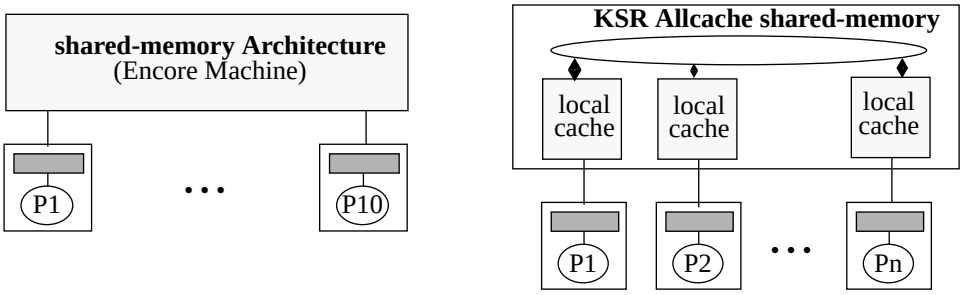


Figure 7: Two implementations of the shared-memory model

Figure 7 shows the difference between the two memory organizations according to the physical realization of the shared-memory space. The configuration used for the experiments includes 72 * 40 MIPS processors for a total main memory of 2.3 Gigabytes.

5.2 Impact of the Allcache model on DBS3

We now quantify the performance penalty that cannot be avoided on the KSR1 because of the heterogeneous structure of the virtual shared-memory. In the KSR1, data may move from one local cache to another; it is this feature which gives the global shared-memory view. Therefore, it is difficult to benefit from data locality. Because the access to a remote cache line is 6 times that of the access to a local cache line, cache misses may dramatically slow down the query execution time. To measure the overhead of cache misses, we have run a parallel selection over a 200k tuples relation (the DewittA relation of the Wisconsin benchmark). The objective is to compare the execution time with local or remote data access. *Tl* is the execution time in the local case, while *Tr* is that of the remote case. The number of threads allocated to perform the selection is varying from 5 to 30. We have measured (*Tr* - *Tl*) in order to obtain a result independent from the global execution time. Note that under 5 threads, *Tr* is equal to *Tl*. In fact, as fewer threads are allocated for the query, each thread gets more work. Under 5 threads, the local cache size is too small to contain all the data which are selected and a local execution cannot be obtained. The results (Figure 8) show that the difference *Tr* - *Tl* represents approximately 4% of the total execution time, which is not a high overhead. This experiment also shows (Figure 9) that the

difference *Tr - Tl* decreases with the number of threads. This is exactly what was expected as remote accesses are (roughly speaking) parallelized, i.e. the overhead of cache misses are shared between the threads.

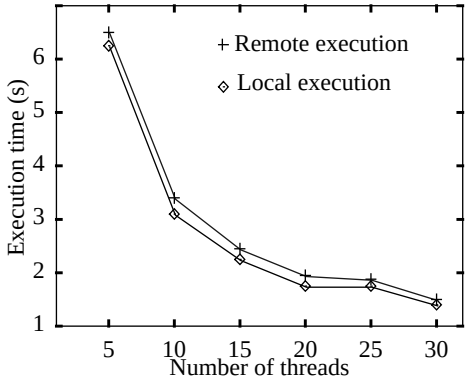


Figure 8: Impact of the remote access for a 200K tuples selection

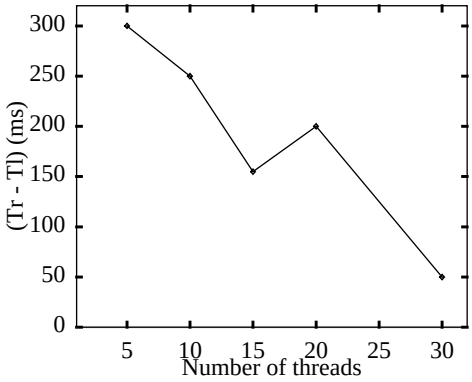


Figure 9: Difference of remote and local execution time

For more complex queries (e.g. join), this overhead would become even smaller. In DBS3, a fragment of code manipulates only a small fragment of data which means that once caches are filled with relevant data, all accesses get local. Of course, each bucket of a relation must be relatively small compared to the size of a local cache in order to benefit from caching.

In summary the hardware management of cache misses by the Allcache system associated with the parallelization model adopted in DBS3 should reduce the overhead of cache coherency management in the virtual shared-memory of the KSR1 machine. A comparison of DBS3 on the KSR1 and the Encore Multimax can be found in [Dageville94]. It shows attractive performance on the KSR1 and similar speed-up for the two implementations.

5.3 Databases and Queries

In all the experiments, we use the relations of the Wisconsin benchmark [Bitton83]. These relations are partitioned based on hashing. Performance measurements are done with the relations cached in main memory for the simple reason that only one disk was available to us, which mean sequential disk accesses.

In our experiments, we use two Lera-par execution plan: IdealJoin which indicates a parallel join where both operands (A and B') are partitioned on the join attribute in the same number of buckets, and AssocJoin where one operand (B') must be dynamically repartitioned before the parallel join (the other one (A) is partitioned

on the join attribute). Figure 10 and 11 shows the parallel execution plans for these operations:

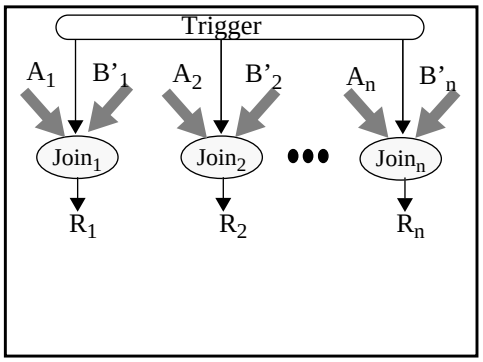


Figure 10: IdealJoin execution plan

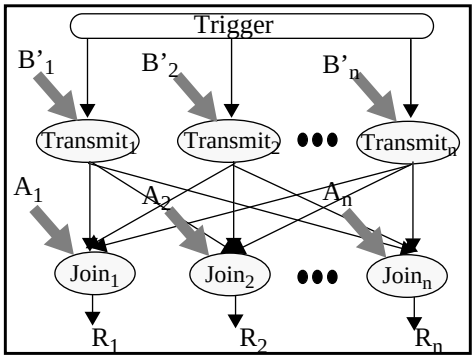


Figure 11: AssocJoin execution plan

To do a thorough study of skewed data partitioning and high degree of partitioning, we had to generate a large number of databases (more than 50). The disk size available to us (the KSR1 is shared by many people) was relatively small (one Gigabyte). Therefore, measurements on large databases were avoided whenever possible. With small databases (i.e. 100 or 200 KTuples), index-based joins run too fast and make the result analysis difficult as the response time is of the same order of magnitude than measurement errors. Thus, when the join algorithm has no impact, we use a nested-loop join in order to slow down the execution time. In other cases, we use larger databases (500 Ktuples) and build indexes on the fly. We repeated each measurement six times and took the average result.

5.4 Expt 1: Varying the Skew

To get more practical insights on the previous analytical results, we performed several experiments with IdealJoin and AssocJoin for varying skews. We have created many databases for which we have varied the tuple distribution within fragments. To determine fragment cardinality, we use a Zipf function [Zipf49] which yields a factor between 0 (no skew) and 1 (high skew). In practice, many skewed data distributions can be modelled by this kind of function [Lynch88].

Each database has two relations A and B' of 100K and 10K tuples, respectively. Each relation is statically partitioned in 200 fragments (see Section 5.6). It is enough to have only one skewed relation, in our case A, since we have experimentally verified that the skew of the two relations could made equivalent by increasing the skew of one relation and leaving the other one unskewed.

For each database (obtained by changing the skew of A), we have run the two queries AssocJoin and IdealJoin with 10 threads to obtain their response time. In the case of AssocJoin, B' is redistributed, so 10K tuples move through the pipeline. Fig-

ure 12 confirms the analytical results. The execution time measured is constant whatever the skew. It also shows the graph for the worst time (T_{worst}) suing the analytical formula. Even in the worst case, the maximum deviation is small (3%). Thus, this experiment shows that we obtain an ideal execution time, independent of the skew factor.

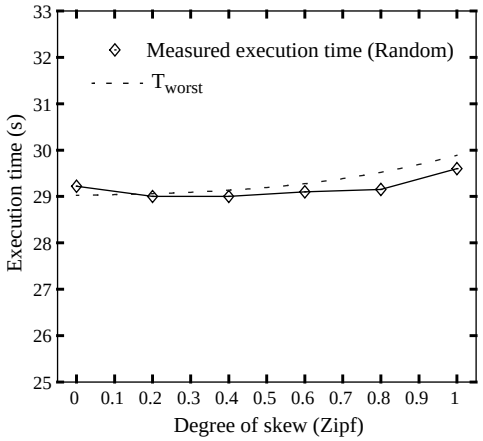


Figure 12: AssocJoin execution

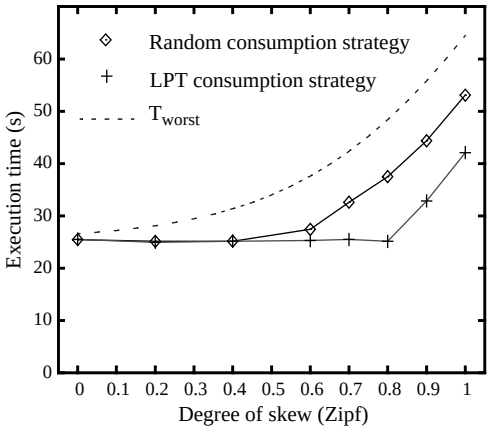


Figure 13: Ideal Join execution time

IdealJoin is a triggered operation. Thus, the number of activations is equal to the number of operation instances. In Figure 13, we show the results of running this operation by varying the skew and changing the thread consumption strategy (Random or LPT). We also show the T_{worst} curve. For low skew factors (less than 0.4), we obtain good results independent of the consumption strategy. Since the relation has 200 fragments, each fragment is relatively small which yields good load balancing, even with Random. However, with a higher skew, LPT becomes better than Random, and remains insensitive to skew up to a skew factor of 0.8 (less than 2% overhead with respect to the ideal time). The inflection after 0.8 is due to the execution time of the longest activation. This is because after 0.8, the execution time of this activation is higher than the ideal time of the whole operation, that is $P_{max} > (a \times P) / n$. With LPT, even if this activation is processed first, the operation response time is equal to the execution time of this first activation.

5.5 Expt 2: Varying the Degree of Parallelism

The previous experiments were done with relatively small numbers of threads, which was enough for studying the effect of skew. We now turn to the impact of increasing the degree of intra-operation parallelism on load balancing. We use similar databases, but with larger relations (200K and 20K tuples) in order to minimize error propagation in measurements.

We ran the queries AssocJoin and IdealJoin (using nested loop) on a set of skewed databases. For each database, we vary the number of threads from 1 (sequential) to 100 using 70 processors of the KSR1 (which we could reserve). Figure 14 and 15 show the speed-up results. With non skewed relations, the results are very good with a speed-up greater than 60 with 70 processors for both queries. For such simple queries, there is no benefit in allocating more threads than available processors since speed-up is decreasing after 70.

With skewed relations, the results are different depending on whether the operation is pipelined or triggered. In the case of pipelined operation (AssocJoin), the high number of activations (20,000) can well absorb bad distributions, even with a high number of threads. Using Equation (3), we can analyze the behavior of AssocJoin. With 70 threads and the worst skew (Zipf=1), the execution time is only 12% worse than the ideal time*. Our measurements indicate that this worst case is overestimated since it never exceeds 5%.

In the case of triggered operation (IdealJoin), the results are not as good. With skewed relations, the speed-up reaches a ceiling of number of threads depending on the skew. Again, this is because of the longest activation P_{max}. When $P_{max} > (a \times P) / n$, the operation execution time is that of this single activation, independent of the number of threads. Thus, there is no gain in using a degree of parallelism greater than $n_{max} = (a \times P) / P_{max}$, i.e. the sequential execution time of the operation over the sequential time of the longest activation. From this formula, we can compute n_{max} for each skew factor. We obtain $n_{max} = 6$ with Zipf = 1, 19 with 0.6 and 40 with 0.4. These theoretical values are confirmed in Figure 15

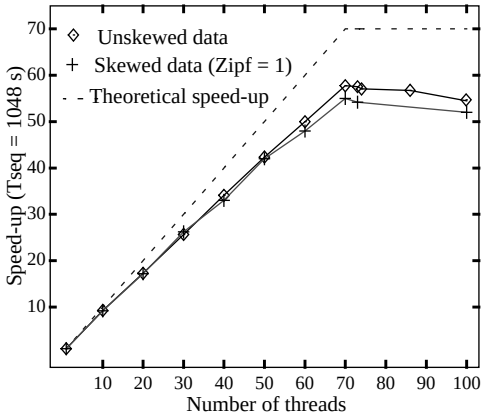


Figure 14: AssocJoin speed-up

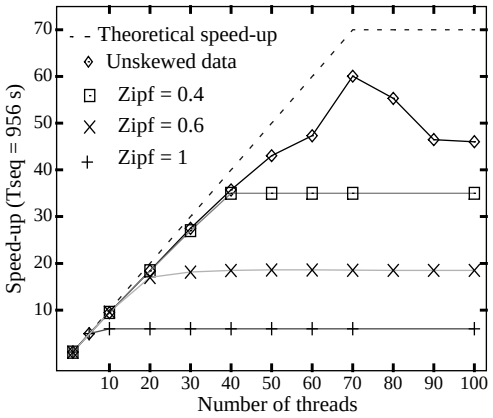


Figure 15: IdealJoin speed-up

* With Zipf = 1 and a = 200 buckets, we have P_{max} = 34 P. With 70 threads, we have v = 34 x 69 / 20000 = 0.117

To summarize, the horizontal fragmentation of an operation into a high number of sequential units of work (pipelined operations) can absorb skewed data distributions and get much better performance than with small numbers of fragments (triggered operations). In DBS3, the initial degree of partitioning can be dynamically raised to increase the number of activations and reduce their execution time. However, a high degree of partitioning can generate an overhead (for queue creation and management) which offsets the gains obtained from a better load balancing. We address this problem in the next section.

5.6 Expt 3: Varying the degree of partitioning

The degree of relation partitioning on disks typically determines the degree of parallelism, hence the choice of full declustering [Mehta95] or partial declustering [Copeland88]. In DBS3, the degree of partitioning can be higher than the number of disks which is useful to reduce the effect of skewed data distribution. However, having more fragments than disks can induce some overhead since there are more queues to be created and accessed. In the rest of this section, we evaluate this overhead through experimentation and measure the improvement of a high degree of partitioning on skew.

5.6.1. Overhead of a High Degree of Partitioning

To study this overhead, we created several databases by varying the degree of partitioning of the two unskewed relations (of 100K and 10K tuples). We use the query IdealJoin for the overhead of a triggered join and AssocJoin for the overhead of a pipelined join. The measurements use 20 threads and a degree of partitioning varying from 20 to 1500.

Figure 16 shows the overhead for IdealJoin and AssocJoin without indexes. This overhead is computed as the difference of the measured time and the theoretical time*. Note that this difference does not depend on the join algorithm but on the kind of operation (triggered/pipelined).

The overhead can be approximated with a straight line (dotted line in the Figure), using the ratio 0.45 ms/degree for IdealJoin and 4 ms/degree for AssocJoin. The difference between these ratios has a simple explanation. In the case of IdealJoin, there is only one activation per fragment and as many queues to create. In the case of AssocJoin, there are two groups of queues (one for the redistribution and one for the join) and 10K activations.

* Let T^{20} be the time measured with a degree of partitioning 20, the theoretical time for degree d is obtained by $T^d = T^{20} \times (20/d)$ (Nested loop algorithm).

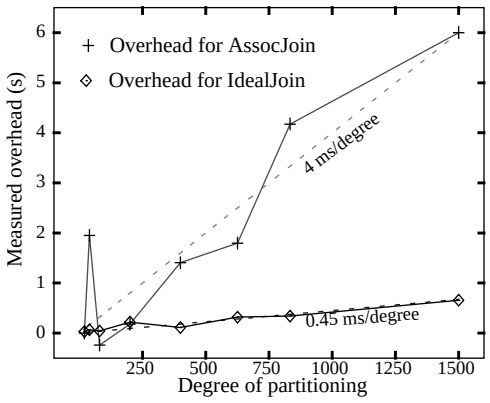


Figure 16: Partitioning overhead for IdealJoin and AssocJoin (No temp. index)

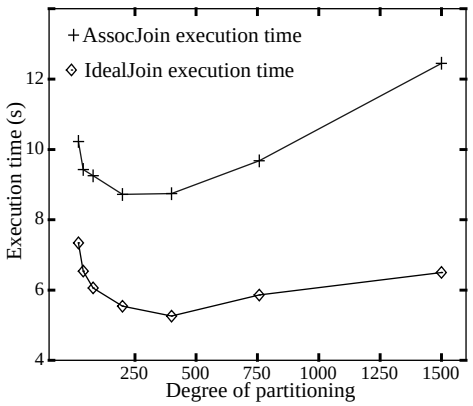


Figure 17: Execution Time for IdealJoin and AssocJoin (with temporary index)

The overhead for a pipelined execution may be significant. However, it enables very high degree of partitioning (in the order of 1K) with little global overhead. The reason is that the gains obtained from algorithmic simplification can compensate the overhead due to partitioning. This is obvious for a nested loop join. It is less obvious when using indices. Thus, Figure 17 shows the results for the same queries using a temporary index with relations of 500K and 50K tuples. In this Figure, the overhead dominates the gain when d > 1000 for AssocJoin and d > 1400 for IdealJoin.

These experiments show the limited impact of the overhead incurred by a high degree of partitioning on unskewed relations.

5.6.2. Using a High Degree of Partitioning for Skewed Data

We now use this property of low impact overhead incurred by a high degree of partitioning to deal with skewed data distributions in the case of triggered operations. We run the IdealJoin query with 20 threads and relations of 500K and 50K tuples, and 100K and 10K tuples. The degree of partitioning varies from 20 to 1500. We ran the queries with a uniform distribution (Zipf = 0) and a skewed distribution (Zipf = 0.6). The consumption strategy for the threads is LPT. We used the measurements to compute the overhead v of execution time ($T_{0.6}$) with respect to execution time without skew (T_0) obtained as follows: (see equation 1)

$$v_{0.6} = \frac{T_{0.6}}{T} - 1$$

Figure 18 shows the values of $v_{0.6}$ for IdealJoin with and without index. The two curves are almost identical. It confirms that the behavior of our model with data skew is independent of the join algorithm. The other graph shows the worst value of v (see equation 3). Figure 19 shows the time saved by increasing partitioning, which is to be compared with the value for T_0 for IdealJoin with index.

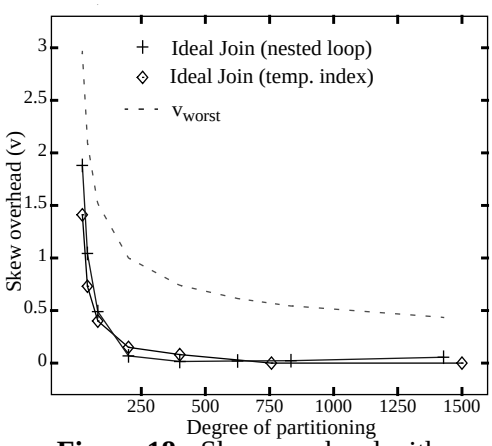


Figure 18: Skew overhead with IdealJoin

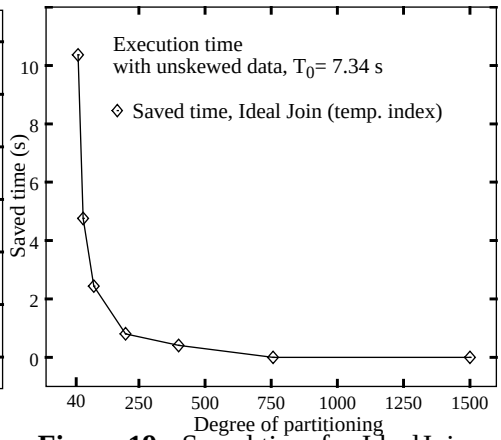


Figure 19: Saved time for IdealJoin with index.

In the case of a triggered operation applied to skewed data, the advantage of a high degree of partitioning is clear. By increasing the degree of partitioning, the granule of the sequential unit of work gets smaller (one activation = one fragment) and the LPT strategy can better balance the load on the threads.

In the case of a pipelined operation, the granule of sequential processing is very small (one activation = one tuple) and a higher degree of partitioning does not modify the overhead since the number of activations remains the same. We also verified this observation by experimenting with AssocJoin, and we obtained $T_{0.6} = T_0$ for any degree of partitioning ($v_{0.6} < 0.03$).

In general, complex queries will include both triggered and pipelined operations. A high degree of partitioning allows more efficient processing of skewed data distributions for triggered operations. However, it yields some overhead for pipelined operations which is well compensated by the gains obtained on triggered operations.

6 Conclusion

The barriers to parallel query execution are start-up time of parallel operations, interference and poor load balancing among the processors due to skewed data distribution. In this paper, we have described how these problems are addressed in DBS3, a shared-memory database system implemented on a 72-node KSR1 multiprocessor.

Our solution combines the advantages of static and dynamic partitioning. We use static partitioning of relations to reduce interference and dynamic allocation of processors to operations to reduce start-up time and improve load balancing. This ad-

aptative approach also simplifies database tuning since the degree of partitioning does not dictate the degree of parallelism.

A major advantage of our solution is to be able to deal efficiently with skew by allowing each execution entity (thread) to dynamically choose which operation's instance it will execute and by increasing the degree of partitioning. To quantify the potential gains, we did a performance analysis and ran experiments on our prototype with different databases of the Wisconsin benchmark.

The behavior of our parallel execution model in front of skew depends heavily on the nature of the operation. In DBS3, pipelined operation are *naturally* insensitive to skew. This is because the high number of units of work (activations) produced by pipelined execution yields good load balancing even in difficult situations (high skew, high degree of parallelism).

For a triggered operation, the number of activations depends on the degree of partitioning of the operand relations. Heuristics can be used to reduce the overhead of skew. However, with high skew, the execution time is bounded by the time of the longest activation.Thus, there is no gain in increasing the degree of parallelism. An effective solution is to use a high degree of partitioning, since our models allow it with an insignificant overhead.

With a high degree of partitioning, our model is almost insensitive to skew and yields excellent performance. We obtained good speed-up using the 72 processors of the KSR1, even in presence of skew. These results are due to both our DBS3 hybrid model (static partitioning, dynamic processor allocation) and the hybrid architecture of the KSR1 (physically distributed, virtually shared-memory).

Since pipelined operations are insensitive to skew, a simple execution strategy would be to run as many operations as possible in pipelined mode. This strategy resists to bad data distributions but may yield high overhead, especially with limited memory size. The problem reduces to that of grain of parallelism. Coarse-grain parallelism (triggered operation) is bad with data skew but has limited overhead. Conversely, fine-grain parallelism (pipelined operation) makes the operation insensitive to skew but may yield high overhead. Future work in DBS3 will address this problem by allowing the choice of the grain of parallelism independent of the operation semantics.

Acknowledgments:

The authors thank Michael Franklin for careful reading of the paper. They also want to thank J.P. Chieze, A. Clo who have allowed efficient use of the KSR1 at Inria, and all the members of the DBS3 team for their cooperation.

7 References

[Bergsten91] B. Bergsten, M. Couprie, P. Valduriez, “Prototyping DBS3, a shared-memory parallel database system”. *Int. Conf. on Parallel and Distributed Information Systems*, Florida, USA, December 1991.

[Bitton83] D. Bitton, D. J. DeWitt & C. Turbyfill, “Benchmarking database systems - A systematic approach”, *Int. Conf. on VLDB*, Firenze, Italy, October 1983.

[Boral90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, P. Valduriez, “Prototyping Bubba, A highly parallel database system”. *IEEE Knowledge and Data Engineering*, Vol. 2, 1990.

[Borla91] P. Borla-Salamet, C. Chachaty, B. Dageville, “Compiling Control into Database Queries for Parallel Execution Management”. *Int. Conf. on Parallel and Distributed Information Systems*, Florida, USA, December 1991.

[Chachaty92] C. Chachaty, P. Borla-Salamet, M. Ward, “A Compositional Approach for the Design of a Parallel Query Processing Language”, *Int. Conf. on Parallel Architectures and Language Europe*, Paris, France, June 1992.

[Copeland88] G. Copeland, W. Alexander, E. Boughter & T. Keller, “Data Placement in bubba”, *Int. Conf. ACM-SIGMOD*, Chicago, June 1988.

[Dageville94] B. Dageville, P. Casadessus, P. Borla-Salamet, “The Impact of the KSR1 AllCache Architecture on the Behaviour of the DBS3 Parallel DBMS”, *Int. Conf. on Parallel Architectures and Language Europe*, Athens, Greece, July 1994.

[Davis92] D. D. Davis, “Oracle’s Parallel Punch for OLTP”, *Datamation*, August 1992.

[Dewitt90] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao & R. Rasmussen, “The Gamma Database Machine Project”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, March 1990.

[DeWitt92a] D.J. DeWitt, J. Gray, “Parallel Database Systems: the Future of High Performance Database Systems", *Comm. of the ACM*, Vol. 35, No. 6, June 1992.

[DeWitt92b] D.J. DeWitt, J.F. Naughton, D.A. Schneider, S. Seshadri, “Practical Skew Handling in Parallel Joins”, *Int. Conf. on VLDB*, Vancouver, Canada, August 1992.

[Frank93] S. Frank, H. Burkhardt, J. Rothnie, “The KSR1: Bridging the Gap Between Shared-Memory and MPPs”, *Compcon’93*, San Francisco,

USA, February 1993.

[Gardarin92] G. Gardarin, P. Valduriez, “ESQL2, an Extended SQL2 with F-logic Semantics.”, *IEEE Int. Conf. on Data Engineering*, Phoenix, Arizona, February 1992.

[Graefe92] G. Graefe, S. Thakkar, “Tuning a Parallel Database Algorithm on a Shared-Memory Multiprocessor”, *Software - Practice and Experience*, Vol. 22, No.7, July 1992.

[Graefe93] G. Graefe, D. L. Davison, “Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Processing”, *IEEE Transactions on Software Engineering*, Vol. 19, August 1993.

[Graefe94] G. Graefe, “Volcano, An Extensible and Parallel Dataflow Query Processing System”, *IEEE Transaction on Knowledge and Data Engineering* Vol. 6, February 1994.

[Graham69] R. L. Graham, “Bounds on Multiprocessing Timing Anomalies”, *SIAM Journal of Applied Mathematics*, Vol. 17, March 1969.

[Hong92] W. Hong, “Exploiting Inter-Operation Parallelism in XPRS”, *Int. Conf. ACM-SIGMOD*, San Diego, CA, June 1992.

[Hsiao94] H. Hsiao, M. S. Chen, P. S. Yu, “On Parallel Execution of Multiple Pipelined Hash Joins”, *Int. Conf. ACM-SIGMOD*, Minneapolis, May 1994, 185-196.

[Kitsuregawa90] M. Kitsuregawa, Y. Ogawa, “Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer”, *Int. Conf on VLDB*, Brisbane, Australia, 1990.

[Lanzelotte94] R. Lanzelotte, P. Valduriez, M. Zait, M. Ziane, “Industrial-Strength Parallel Query Optimization: issues and lessons”, *Information Systems*, Vol. 19, No. 4, 1994.

[Lynch88] C. Lynch, “Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values”, *Int. Conf. on VLDB*, Los Angeles, CA, August 1988.

[Mehta95] M. Metha, D. DeWitt, “Managing Intra-operator Parallelism in Parallel Database Systems” *Int. Conf. on VLDB*, Zurich, Switzerland, September 1995.

[Omiecinski91] E. Omiecinski, “Performance Analysis of a Load Balancing Hash-Join Algorithm for a Shared-Memory Multiprocessor”, *Int. Conf on VLDB*, Barcelona, Spain, September 1991.

[Rahm93] E. Rahm, R. Marek, “Analysis of Dynamic Load Balancing Strategies for Parallel Shared-Nothing Database Systems”, *Int. Conf. on VLDB*, Dublin, Ireland, August 1993.

[Valduriez93] P. Valduriez, “Parallel Database Systems: open problems and new issues.”, *Int. Journal on Distributed and Parallel Databases*, Vol. 1, No. 2, 1993.

[Walton91] C.B. Walton, A.G. Dale, R.M. Jenevin, “A taxonomy and Performance Model of Data Skew Effects in Parallel Joins” *Int. Conf. on VLDB*, Barcelona, Spain, September 1991.

[Wilschut92] A. N. Wilschut, J. Flokstra, P. M. G. Apers, “Parallelism in a main-memory system: The performance of PRISMA/DB”, *Int. Conf. on VLDB*, Vancouver, Canada, August 1992.

[Wolf91] J. L. Wolf, D. M. Dias, P. S. Yu, J. Turek, “An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew”, *IEEE Int. Conf. on Data Engineering*, Kobe, Japan, April 1991.

[Zipf49] G. K. Zipf, *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*, Reading, MA, Addison-Wesley, 1949.

Unité de recherche INRIA Lorraine, technopôle de Nancy-Brabois, 615 rue du jardin botanique, BP 101, 54600 VILLERS-LÈS-NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, domaine de Voluceau, Rocquencourt, BP 105, LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur

Inria, Domaine de Voluceau, Rocquencourt, BP 105 LE CHESNAY Cedex (France)

ISSN 0249-6399