Dynamic Load Balancing in Hierarchical Parallel Database Systems

Luc Bouganim, Daniela Florescu, Patrick Valduriez

Dynamic Load Balancing in Hierarchical Parallel Database Systems

Luc Bouganim, Daniela Florescu, Patrick Valduriez

N° 2815

March 1996

PROGRAMME 1

ISSN 0249-6399

Dynamic Load Balancing in Hierarchical Parallel Database Systems

Luc Bouganim[*][**], Daniela Florescu[**], Patrick Valduriez[**]

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Rodin

Rapport de recherche n°2815 — March 1996 — 21 pages

Abstract: We consider the execution of multi-join queries in a hierarchical parallel system, i.e., a shared-nothing system whose nodes are shared-memory multiprocessors. In this context, load balancing must be addressed at two levels, locally among the processors of each shared-memory node and globally among all nodes. In this paper, we propose a dynamic execution model that maximizes local load balancing within shared-memory nodes and minimizes the need for load sharing across nodes. This is obtained by allowing each processor to execute any operator that can be processed locally, thereby taking full advantage of inter- and intra-operator parallelism. We conducted a performance evaluation using an implementation on a 72-processor KSR1 computer. The experiments with many queries and large relations show very good speedup results, even with highly skewed data. We show that, in shared-memory, our execution model performs as well as a dedicated model and can scale up very well to deal with multiple nodes.

Key-words: Databases

(Résumé : tsvp)

This work has been done in the Groupement d'Intérêt Economique Dyade (joint R&D venture between Bull and Inria) and has been partially supported by the Commission of European Communities under Esprit project IDEA.

[*]Bull Grenoble, France
[**]E-mail: {`Firstname.Lastname`}`@inria.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

Dynamic Load Balancing in Hierarchical Parallel Database Systems

Résumé : Nous considérons l'exécution de requêtes complexes dans une architecture parallèle hiérarchique, consistant d'un ensemble de noeuds multiprocesseurs à mémoire partagée, reliés par un réseau rapide. Dans ce contexte, l'équilibrage de charge doit être effectué à deux niveaux, localement entre les processeurs de chaque noeud à mémoire partagée puis globalement entre les differents noeuds. Dans ce papier, nous proposons un modèle d'exécution dynamique qui maximise l'équilibrage locale de charge et minimise ainsi le besoin de répartitionnement du travail entre les noeuds. Cela est obtenu en permettant à chaque processeur d'un noeud d'exécuter n'importe quelle opération pouvant être traitée localement, tirant ainsi parti du parallélisme inter et intra-opération présent dans les requêtes complexes. Nous effectuons une évaluation de performances en utilisant une implémentation de notre modèle sur une machine KSR1 comprenant 72 processeurs. Les mesures effectuées sur un ensemble de requêtes mettant en jeu de grandes relations montrent de très bon gains (speed-up), même dans le cas de mauvaises répartitions des données. Nous montrons que notre modèle se comporte, sur un noeud à mémoire partagée, aussi bien que des modèles dédiées et prend en compte efficacement plusieurs noeuds.

Mots-clé : Bases de données

1 Introduction

Parallel system designers have long opposed shared-memory versus shared-nothing architectures. Shared-memory provides flexibility and performance for a restricted number of processors while shared-nothing can scale up to high-end configurations [DeWitt92, Valduriez93]. To combine their respective benefits, hierarchical parallel systems consisting of shared-memory multiprocessors interconnected by a high-speed network [Graefe93] are gaining much interest. As an evidence, symmetric multiprocessors (SMP), e.g., Sequent, are moving to scalable cluster architectures, while massively parallel processors (MPP), e.g., NCR's Teradata, are evolving to use shared-memory nodes. Another example is Bull's PowerCluster which is a cluster of PowerPC-based SMP nodes jointly developed with IBM.

In this paper, we consider the execution of multi-join queries in hierarchical parallel database systems. Such queries are getting increasingly important as parallel database systems are gaining wider use for decision support (e.g., data warehouse applications). The objective of parallel query processing is to reduce query response time as much as possible by distributing the query load among multiple processors. The major barrier to this objective is poor load balancing, i.e., some processors are overloaded while some others remain idle. As the response time of a set of parallel activities is that of the longest one, this can severely degrade performance.

There are two dimensions for parallelizing multi-join queries: horizontally (i.e., intra-operator parallelism) by distributing each operator among several processors, and vertically (i.e., inter-operator pipelined or independent parallelism) by distributing all operators of the query among several processors. Solutions for load balancing typically focus on one dimension on a given architecture (shared-memory, shared-disk, or shared-nothing).

In shared-nothing, intra-operator parallelism is based on relation partitioning [Boral90, Apers92, DeWitt90]. Skewed data distributions which are quite frequent in practice (see [Walton91] for a taxonomy) can yield poor intra-operator load balancing. This problem has been addressed by developing specific join algorithms that handle different kinds of skew [Kitsuregawa90, DeWitt92, Shatdal93, Berg92] based on dynamic data redistribution.

With inter-operator parallelism, distributing the query's operators among all processors can also yield poor load balancing. Much research has been dedicated to inter-operator load balancing in shared-nothing [Garofalakis96, Mehta95, Rahm95] which is done statically during optimization or dynamically just before execution.

The potential reasons for poor load balancing in shared-nothing and different solutions are studied in [Wilshut95]. First, the degree of parallelism and the allocation of processors to operators, decided in the parallel optimization phase, are based on a possibly inaccurate cost model. Second, the choice of the degree of parallelism is subject to discretization errors because both processors and operators are discrete entities. Finally, the processors associated with the latest operators in a pipeline chain may remain idle a significant time. This is called the pipeline delay problem. These problems stem from the fixed association between data, operators and processors which is inherent to distributed architectures.

In shared-disk[Pirahesh90], there is more flexibility since all processors have equal access to disks. Thus, intra-operator parallelism does not require static relation partitioning which can be performed dynamically [Davis92, Lu91]. Inter-operator parallelism is also less constrained since any processor can be allocated to any operator. Load balancing for independent parallelism is addressed in [Hsiao94] while only pipeline parallelism is considered in [Lo93].

Shared-memory offers even more flexibility since all processors have equal access to memory and disks. The solutions to load balancing [Shekita93, Hong92, Murphy91] can be much more dynamic (i.e., at run-time) since redistributing the load incurs low cost. With the self-balancing process model proposed in [Shekita93], each processor reads tuples from I/O buffers and performs the joins along the pipeline chain using synchronous pipelining (procedure calls) as in [Hong92].

DBS3 [Bergsten91, Dageville94] has pioneered the use of an execution model based on relation partitioning (as in shared-nothing) for shared-memory. This model reduces processor interference and shows excellent load balancing for intra-operator parallelism [Bouganim96]. However, inter-operator load balancing was not addressed.

To our knowledge, no work has addressed the problem of load balancing in hierarchical systems. In this context, load balancing is more difficult because it must be addressed at two levels, locally among the processors of each shared-memory node and globally among all nodes. None of the previous approaches can be easily extended to deal with this problem. Load balancing strategies for shared-nothing would have their inherent problems worsening (e.g., complexity and inaccuracy of the cost model) and would not take advantage of the flexibility of shared-memory. On the other hand, adapting of the solutions for shared-memory would incur high communication overhead.

In this paper, we propose an execution model for hierarchical systems which dynamically performs intra and inter-operator load balancing. The basic, new idea is that the query work is decomposed in self-contained units of sequential processing, each of which can be processed by any processor. Intuitively, a processor can ‘migrate horizontally and vertically along the query work’. The main advantage is to reduce to the minimum the communication overhead inherent to inter-node load balancing by maximizing intra and inter-operator load balancing within shared-memory nodes. To validate the model and study its performance, we did an implementation on a 72-processor KSR1 computer[1].

The paper is organized as follows. Section 2 gives a number of assumptions regarding the execution system and states the problem more precisely. Section 3 presents the basic concepts underlying our model and its load balancing strategy. Section 4 completes the description of our execution model with its main implementation techniques. Section 5 gives a performance evaluation of our model, with comparison with two other load balancing strategies, using our implementation on the KSR1 computer. Section 6 concludes.

2 Problem Formulation

A parallel execution model relies on assumptions regarding the target execution system and parallel query optimization decisions. Assuming a hierarchical architecture changes the parallelization decisions made by the query optimizer [Srivastava93]. In this section, we make precise our assumptions regarding the execution system and the parallel execution plans. These assumptions will also help making the problem statement and presenting the execution model.

2.1 Execution System

We consider a shared-nothing parallel database system with several shared-memory multiprocessor nodes, or SM-nodes for short (see Figure 1). Each SM-node has several processors, several disk units and a memory shared by all processors. Inter-node communication is done via message-passing while inter-processor communication within a node is done more efficiently via shared-memory. This architecture is fairly general and can include either limited numbers of powerful SM-nodes (e.g., 4 nodes, each having 16 processors) or higher numbers of less powerful SM-nodes (e.g., 16

[1]Although Kendal Square Research Inc. is gone, our KSR1 computer is still up and running thanks to Inria’s ackers.

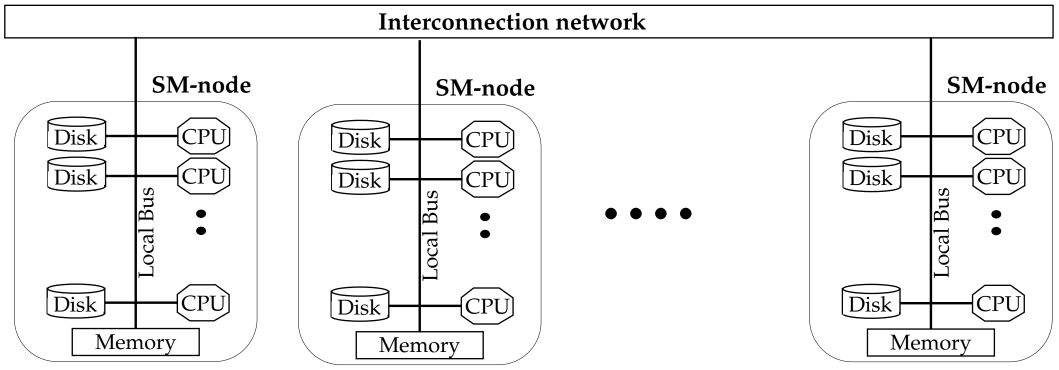


Figure 1: Hierarchical Architecture

nodes, each having 4 processors) for an equivalent CPU power and disk capacity. We are primarily interested in the first alternative which is exemplified by the recent SMP cluster architectures.

Relations are horizontally partitioned across nodes, and within each node across disks. The *degree of partitioning* of a relation is a function of the size and heat of the relation [Copeland88]. Relation partitioning is based on a hash function applied to some attribute. The home of a relation is simply the set of SM-nodes which store its partitions.

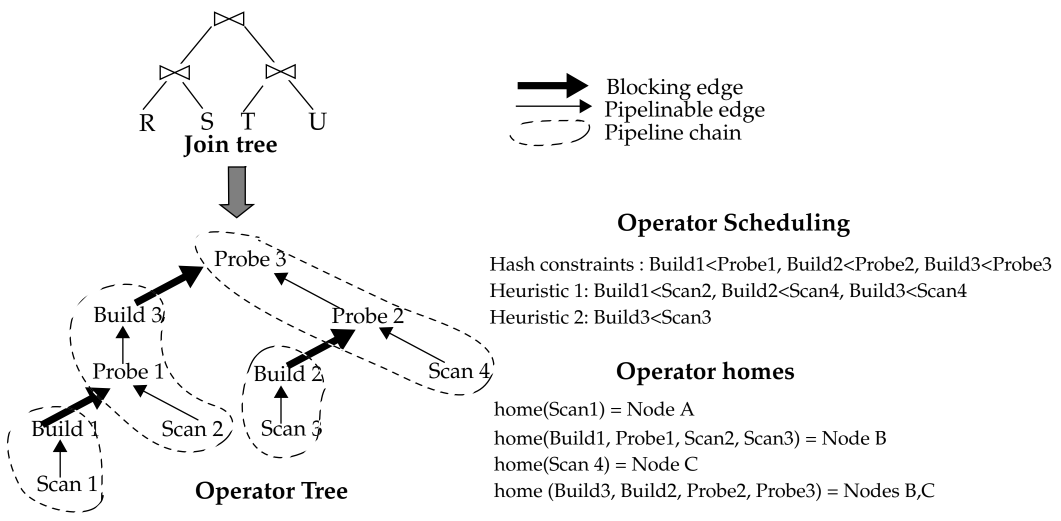
As in many other papers, we consider only parallel hash join methods since they generally offer superior performance [Valduriez84, Schneider89]. Hash joins provide two opportunities for parallelism: (i) several joins can be pipelined (inter-operator parallelism); and (ii) each join can be done in parallel on partitioned relations (intra-operator parallelism). Both relations involved in the join are fragmented in the same number of buckets, according to the same hash function applied to the join attribute. Then, the parallel hash join proceeds in two phases: build and probe. First, the buckets of the building relation are scanned in parallel and a hash table is built for each bucket. Second, the buckets of the probing relation are scanned in parallel, probing the corresponding hash table and producing result tuples.

2.2 Parallel Execution Plans

The result of parallel query optimization is a *parallel execution plan* that consists of an operator tree with operator scheduling and allocation of computing resources to operators. Different shapes of join tree can be considered: left-deep, right-deep, segmented right-deep, zigzag [Ziane93] or bushy. Bushy trees are the most appealing because they offer the best opportunities to minimize the size of intermediate results [Shekita93] and to exploit all kinds of parallelism [Lanzelotte93]. Thus, we concentrate on the execution of bushy trees in this paper.

The *operator tree* results from the "macro-expansion" of the join tree [Hassan94]. Nodes represent atomics operators that implement relational algebra and edges represent dataflow. In order to exhibit pipelined parallelism, two kinds of edges are distinguished: blocking and pipelinable. A blocking edge indicates that the data is entirely produced before it can be consumed. Thus, an operator with a blocking input must wait for the entire operand to be materialized before it can start. A pipelinable edge indicates that data can be consumed "one-tuple-at-a-time". So the consumer can start as soon as one input tuple has been produced.

Let us consider an operator tree that uses hash join. Three operators are needed: scan to read each base relation, build and probe. The build operator produces a blocking output, i.e., the hash table, while probe produces a pipelinable output, i.e., the result tuples. Thus, for a hash join operator, there is always a blocking edge between build and probe, i.e., a blocking constraint between operators. An operator tree does not enforce constraints between two operators that have no data dependency.



Operator Scheduling

Hash constraints : Build1<Probe1, Build2<Probe2, Build3<Probe3
Heuristic 1: Build1<Scan2, Build2<Scan4, Build3<Scan4
Heuristic 2: Build3<Scan3

Operator homes

home(Scan1) = Node A
home(Build1, Probe1, Scan2, Scan3) = Node B
home(Scan 4) = Node C
home (Build3, Build2, Probe2, Probe3) = Nodes B,C

Figure 2: A join tree and a parallel execution plan

But the concurrent execution of two independent operators can yield poor sharing of the available resources (memory, disk, processor). To optimize resource sharing among operators, the optimizer may decide to add new blocking constraints in order to make their execution sequential. Thus, *operator scheduling* as decided by the optimizer reflects the optimization constraints as well as the constraints implied by the hash join method. It is expressed by a partial order on the set of operators of the tree where $O_1 < O_2$ states that operator O_2 cannot be started before the end of O_1.

An operator tree can be decomposed as a set of maximum pipeline chains, i.e., chains with highest numbers of pipelined operators, also called fragments [Shekita93] or tasks [Hong92]. For simplicity, we assume that each pipeline chain can be entirely executed in memory. Otherwise the optimizer should add new blocking edges between the operators of the pipeline chain, using new operators (e.g. store). Considering this case would make the discussion unnecessarily longer.

A typical parallel optimization decision is the allocation of processors to operators. In shared-memory, this decision reduces to the optimal number of processors since each processor has equal access to disk and memory. In shared-nothing, processor allocation is highly dependent on data location and is typically done by a fixed association between processors and operators. A hierarchical architecture topology introduces a new dimension to the processor allocation problem. In this case, it is more important to decide the set of SM-nodes where an operator is executed, which we call *operator home*, rather than the set of participating processors. Thus, the parallel execution plan provides operator homes that respect the following obvious constraints: (i) the home of a scan operator is that of the scanned relation; and (ii) the build and probe operators of the same join have necessarily the same home.

Figure 2 shows a bushy tree involving 4 relations and a possible parallel execution plan which consists of an operator tree adorned with operator scheduling and operator home information. In addition to the blocking constraints implied by the hash join algorithm, the given scheduling specifies constraints corresponding to two possible heuristics: (i) the execution of a pipeline chain is started only when all the hash tables are ready; and (ii) pipeline chains are executed one-at-a-time. Such a parallel execution plan is the input to our execution model and the optimization and scheduling decisions, supposed to be good, are strictly followed.

2.3 Problem Statement

Based on the above definitions and assumptions, we can now simply state the problem. Given a parallel execution plan which consists of an operator tree, operator scheduling and operator homes, the problem is to produce an execution on a hierarchical architecture which minimizes response time. A necessary condition to minimize response time is to avoid processor idle time using dynamic load balancing. This must be done at two levels: (i) within an SM-node, load balancing is achieved via fast interprocess communication; (ii) between SM-nodes, more expensive message-passing communication is needed. Thus, the problem is to come up with an execution model that provides a two-level dynamic load balancing strategie so that the use of local load balancing is maximized while the use of global load balancing is minimized.

3 Parallel Execution Model

Intuitively, parallelizing a query amounts to partition the total work along two dimensions. First, each operator is horizontally partitioned to yield intra-operator parallelism. Second, the query is vertically partitioned into dependent or independent operators to yield inter-operator parallelism. We call *activation* the finest unit of sequential processing, i.e., which cannot be further partitioned. The main property of our model is to allow any thread to process any activation of its SM-node. Thus, there is no static association between threads and operators. This should yield perfect load-balancing for both intra-operator and inter-operator parallelism within an SM-node, and thus, reduce to the minimum the need for global load balancing, i.e., when there is no more work to do in an SM-node.

In the rest of this section, we present the basic concepts underlying our model and its load balancing strategy which we illustrate with an example.

3.1 Basic Concepts

Our execution model is based on a few concepts: activations, activation queues, fragmentation, and threads. These concepts are simple and their combination provide much flexibility and generality.

Activations. An activation represents a sequential unit of work. Since any activation can be executed by any thread, activations must be self-contained and reference all information necessary for their execution: the code to execute and the data to process. Two kinds of activations can be distinguished: trigger activations and data activations. A *trigger activation* is used to start the execution of a leaf operator, i.e., scan. It is represented by an $(Operator, Bucket)$ pair which references the scan operator and the base relation bucket to scan. A *data activation* describes a tuple produced in pipeline mode. It is represented by an $(Operator, Tuple, Bucket)$ triple which references the operator to process (build or probe), the tuple to process, and the corresponding bucket. For a build operator, the data activation specifies that the tuple must be inserted in the hash table of the bucket. For a probe operator, it specifies that the tuple must be probed with the bucket's hash table. Although activations are self-contained, they can only be executed on the SM-node where the associated data (hash tables or base relations) is located.

The quality of load balancing depends on the granularity of parallelism [Bouganim96]. Fine-grain parallelism (i.e., parallel execution of data activations) achieves perfect load balancing but may yield high overhead. Conversely, coarse-grain parallelism (i.e., parallel execution of trigger activations) has limited overhead but may yield poor load balancing. To obtain good load balancing with little overhead, we reduce the granularity of trigger activations by replacing a bucket by one or more pages of a bucket, and increase the granularity of data activations by buffering.

Activation Queues. Moving data activations along pipeline chains is done using activation queues, called table queues in [Pirahesh90], associated with operators. If the producer and consumer of an activation are on the same SM-node, then the move is done via shared-memory. Otherwise, it requires

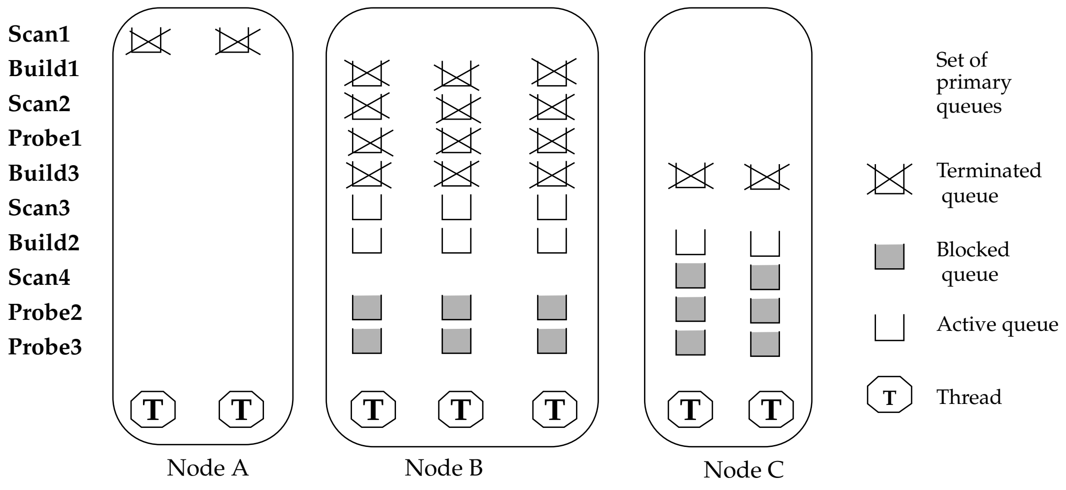


Figure 3: Snapshot of an execution

message-passing. To unify the execution model, queues are used for trigger activations (inputs for scan operators) as well as tuple activations (inputs for build or probe operators).

Each operator needs a queue to receive input activations. Since all threads have unrestricted access to all queues located on their SM-node, managing a small number of queues (e.g., one for each operator) may lead to interference. To reduce interference, we associate one queue per thread working on an operator. Note that a higher number of queues would likely trade interference for queue management overhead. To further reduce interference without increasing the number of queues, we give each thread priority access to a distinct set of queues, called its primary queues. Thus, a thread always tries to first consume activations in its *primary queues*.

During execution, operator scheduling constraints may imply an operator to be blocked until the end of some other operators (the blocking operators). Therefore, a queue for a blocked operator is also blocked, i.e., its activations cannot be consumed but they can still be produced if the producing operator is not blocked. When all its blocking operators terminate, the blocked queue becomes consumable, i.e., threads can consume its activations. This is illustrated in Figure 3 with an execution snapshot for the operator tree of Figure 2.

Threads can freely consume activations in any of the unblocked queues. Without any restriction on the way activations are selected, memory consumption may well increase. For instance, consider the concurrent execution of two pipelined operators. If the selection strategy favors the producer operator, then it may well end up materializing the entire intermediate result at the expense of memory consumption. To avoid this situation, we simply limit the size of the queues and use a flow control mechanism similar to [Graefe93, Pirahesh90] to synchronize producers and consumer in a pipeline chain.

Fragmentation. Let us call *degree of fragmentation* the number of buckets of the building and probing relations. To reduce the negative effects of data skew, the typical solution is to have a degree of fragmentation much higher than the degree of parallelism (i.e., the number of processors allocated to the build or probe operators) [Kitsuregawa90, DeWitt92]. However, one problem is the potential overhead of managing many buckets for each processor. Since our activations are self-contained and reference their own bucket, we can mix activations of different buckets in the same queue and thus reduce the overhead of queue management. More generally, by using a very high degree of fragmentation, our model eases load balancing.

Threads. A simple strategy for obtaining good load balancing inside an SM-node is to allocate a number of threads much higher than the number of processors and let the operating system do thread scheduling. However, this strategy incurs high numbers of system calls due to thread scheduling, interference and convoy problems [Blasgen79, Pirahesh90, Hong92].

Instead on relying on the operating system for load balancing, we choose to allocate only one thread per processor per query. This is made possible by the fact that any thread can execute any operator assigned to its SM-node. The advantage of this one-thread-per-processor allocation strategy is to significantly reduce the overhead of interference and synchronization. At each SM-node, we then create one queue per operator and thread so that each operator has the same potential degree of parallelism. Furthermore, since there is only one thread per processor for the entire query, we do not have the traditional start-up overhead.

This one-thread-per-processor strategy can yield good load balancing provided that a thread is never blocked, i.e., waiting for some event, which would cause processor idle time. During the processing of an activation, a thread can be blocked in the following situations:

- the thread cannot insert an activation in a pipeline queue because the queue is full (flow control);
- the use of asynchronous I/O (for multiplexing disk accesses with data processing) can create waiting situations;
- with multiple transactions, processing an activation referencing a shared data item may cause the thread to be blocked on a lock request.

Waiting situations are typically solved using operating system synchronization mechanisms (e.g., signals). This solution is not optimal because of the overhead of synchronization and context switching. However, it is acceptable with many more threads than processors since the operating system can switch threads for better processor utilization. In our model, with a single thread per processor, these waiting situations may lead to processor idle time. We solve this problem as follows. A thread in a waiting situation suspends its current execution by making a procedure call to find another local activation to process. The advantage is that context saving is done by procedure call, which is much less expensive than operating system based synchronization.

3.2 Load Balancing Strategy

Load balancing within an SM-node is obtained by allocating all activation queues in a segment of shared-memory and by allowing all threads to consume activations in any queue. To limit thread interference, a thread will consume as much as possible in its set of primary queues before considering the other queues of the SM-node. Therefore, a thread gets idle only when there is no more activation of any operator, which, in turn means that there is no more work to do on that SM- node which is starving. Thus, local load balancing can be obtained at low cost.

When an SM-node gets starving, i.e., there are no more activations in all unblocked queues, we can apply load sharing with another SM-node by acquiring some of its workload [Shatdal93]. However, acquiring activations (through message-passing) incurs communication overhead. Furthermore, activation acquisition is not enough since associated data, i.e., hash tables must also be acquired. Thus, we need a mechanism that can dynamically estimate the benefit of acquiring activations and data.

Let us call “requester” the SM-node which acquires work and “provider” the SM-node which gets off-loaded by providing work to the requester. The problem is to select a queue to acquire activations and decide how much work to acquire. This is a dynamic optimization problem since there is a trade-off between the potential gain of off-loading the provider and the overhead of acquiring activations and data. This trade-off can be expressed by the following conditions: (i) the requester must be able to store in memory the activations and corresponding data; (ii) enough work must be acquired in order to amortize the overhead of acquisition; (iii) not too much work must be acquired in order to avoid overloading the requester; (iv) only probe activations can be acquired since triggered activations require disk accesses and build activations require building hash tables locally; (v) there is no gain to

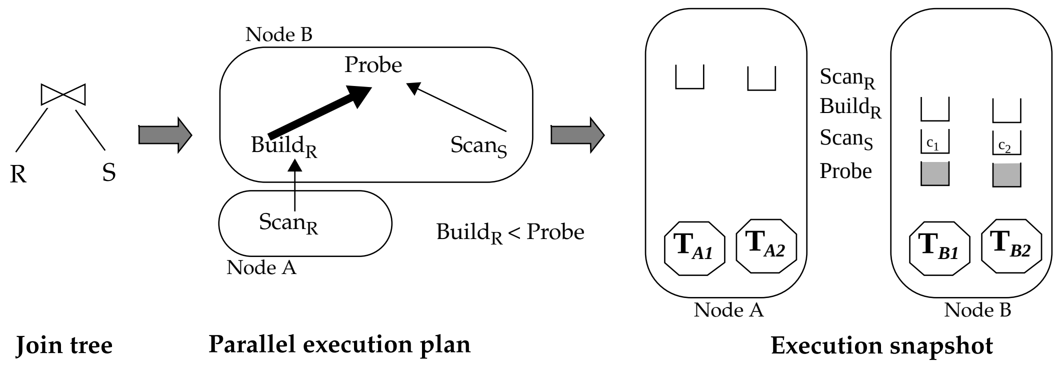


Figure 4: A simple example of query execution

move activations associated with blocked operators which could not be processed anyway. Finally, to respect the decisions of the optimizer, an SM-node cannot execute activations of an operator that it does not own, i.e., the SM-node is not in the operator home. More details about the global load balancing policy are given in section 4.

The quality of the load balancing obtained depends on the number of operators that are concurrently executed which provides opportunities of finding some work to share in case of idle times. Increasing the number of concurrent operators can be done by allowing concurrent execution of several pipeline chains or by using non-blocking hash-join algorithms [Wilshut95]. As an extreme case, it is possible to execute all the operators of the bushy tree concurrently with the *full parallel strategy* described in [Wilshut95]. On the other hand, executing more operators concurrently can increase memory consumption. Static operator scheduling as provided by the optimizer should avoid memory overflow and solve this tradeoff.

3.3 Example

We now illustrate these main concepts on a simple example. Although the model is designed for multi-join queries, a complex example would be unnecessarily lengthy. We consider the join of relations R and S executed on two SM-nodes A and B, each having two processors and thus two threads. Relation R is stored at node A and relation S at node B. Figure 4 gives the parallel execution plan and its execution snapshot at the beginning.

Execution at node A proceeds as follows. Threads T_{A1} and T_{A2} consume trigger activations for the scan$_R$ operator in their associated queue. For each activation, they execute the scan operator on a partition of R, reading R tuples from disk and sending selected tuples in pipeline mode to the build operator at node B. If T_{A1} and T_{A2} get blocked because the build queues are full, and since there is no other operator to process at A, they must wait for the build queues to free. If one thread, say T_{A1}, terminates processing all activations in its queue, it may consume activations in T_{A2} 's queue in order to balance the remaining load. Operator scan$_R$ terminates when there is no more activation to process.

Execution at node B proceeds as follows. Threads T_{B1} and T_{B2} start consuming trigger activations for the scan$_S$ operator in their associated queue. For each activation, they execute the scan operator on a partition of S, reading S tuples from disk and sending selected tuples in pipeline mode to the probe operator. To illustrate execution switching by a thread, we assume that activations C_1 and C_2 produce more selected tuples (data activations) than the probe queues can store. Thus, during execution of C_1, T_{B1} fills the probe queues. To avoid waiting, T_{B1} suspends the execution of C_1 by calling the procedure that selects activations, thus saving C_1's execution context. Since executing other scan$_S$ activations would result in blocking T_{B1} again and since the probe operator is blocked,

T_{B1} selects build activations produced by T_{A1} and T_{A2}. The same execution happens for T_{B2} with C_2. T_{B1} and T_{C2} execute all build activations until termination (i.e., the hash table has been entirely built), which unblocks the probe operator. Thus, they can execute probe activations, which frees the probe queues, and resume execution of C_1 and C_2. Processing of new scan$_S$ activations may again fill the probe queues, in which case T_{B1} and T_{B2} would switch to process probe activations, and so on until termination of scan$_S$.

This simple example shows the value of using activation queues and procedure calls. Threads T_{B1} and T_{B2} are always busy during query execution and the load of node B perfectly balanced. Furthermore, threads T_{A1} and T_{A2} are fully busy during the first phase of execution according to the optimizer decisions.

4 Basic Techniques

In this section, we present the basic techniques to support our execution model. We do so by following the various steps of query execution in a hierarchical system.

Initialization. Let s be the number of SM-nodes, each having p processors. The execution is initialized by creating, at each SM-node participating in the query, p execution threads, and, for each unblocked operator located on this node, p queues. Furthermore, an additional thread, called *scheduler*, is created at each SM-node to deal with message-passing. During execution, the scheduler receives messages from the remote SM-nodes and directs them to the queues of its SM-node. The scheduler also manages inter-node communication as needed for global load balancing and detection of operator end. Locally, the scheduler communicates with other threads using operating system signals.

Query execution. It starts by sending trigger activations to all unblocked scan queues. Then, each execution thread processes activations by consuming queues in priority order. First, it consumes its primary queues, then the other queues at the same SM-node and finally, other queues at another SM-node using load sharing. Conceptually, each thread performs a simple loop which ends when the last operator of the query terminates. At each iteration, an activation is processed. If there is no local activation to process and no global activation to acquire through load sharing (see section 3.2), the thread falls asleep. If new activations come in or some of the local queues become unblocked, the scheduler wakes up the thread which can resume the loop.

Local activation selection. To maximize SM-node load balancing, activation selection must minimize thread interference when accessing queues and avoid useless access to queues associated with operators that are either terminated or blocked. Our solution is based on a circular list of references to all active queues, i.e., neither terminated nor blocked, which is accessed by all threads during activation selection. To avoid interference, each thread starts accessing the list at a different position corresponding to its first primary queue (see Figure 5). This list is created at the beginning of query execution, and updated at the end of each operator to delete all queues associated with it. Furthermore, if the end of the operator causes unblocking of some other operators, the unblocked queues are inserted in the circular list. If no activation is found in any queue of the circular list, it means that there is no more work to do at that SM-node and global activations must be selected by the global load balancing strategy.

Global Activation Selection. Global activation selection, i.e., for load sharing with another SM-node, contributes to global load balancing. When a thread does not find local activations, it sends a signal to its local scheduler which, in turn, sends a starving message to the other SM-nodes. This message indicates the available memory of the requester node. The scheduler at each SM-node, when receiving a starving message, looks up its candidate queues. Section 3.2 enumerates the necessary conditions for a queue to be candidate, *i.e.* the conditions which insure that this stealing is possible and beneficial. The scheduler selects between the candidate queues the one with the best benefit/overhead ratio. The benefit obtained by stealing a candidate queue is proportional to the number of activations in the queue, *i.e.* the work removed from the overloaded SM-node. The overhead is proportional to

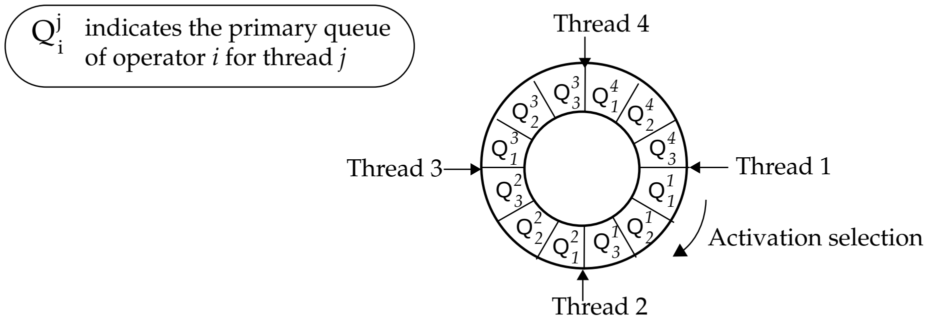


Figure 5: Circular list of activation queues.

the size of the data to be transmitted (hash table and activations). Then, the scheduler back to the requester information on the selected queue (if any) as well as information on its actual load. After receiving answers from all SM-nodes, the requester selects the most loaded SM-node and requests for activations and corresponding data, i.e., hash tables to be sent. Upon receipt of activations, the scheduler of the requester SM-node wakes up the executions threads to process them.

To optimize in case of repeated starving of the same SM-node, a list of stolen queues can be maintained at the requester SM-node. At the next starving situation, it can try to steal activations from theses queues whose associated data have been already copied.

Activation Execution. An activation has a reference to the code to be executed (scan, probe or build). Thus, activation is simply performed by calling the corresponding operator code. However, when executing that code, a thread can perform a *blocking action*, e.g., by reading from disk or writing in a queue that is full. Thus, in the code of each operator, the potential blocking actions are modified as follows:

```
if (cannot perform (BlockingAction) then
    while cannot perform (BlockingAction) do
        ProcessAnotherActivation
    end
```

The procedure ProcessAnotherActivation will not consume activations of the same operator in order to avoid new blocking situations. For instance, reading NbPages from Disk can be expressed by the following code:

```
IoRequest = IO_InitAsync(Disk, NbPages)
NbP = 0
while (NbP != NbPages)
    while (IO_Read(IoRequest) == 0)
        ProcessAnotherActivation
    end
    Process the current page
    NbP = NbP + 1
end
```

With this technique, we avoid blocking actions and maximize processor utilization without system calls.

Detection of Operator End To avoid useless resource consumption, it is important to detect as soon as possible the global end of an operator. In particular, a delay between actual end and detection can create a starving situation at an SM-node because queues are blocked. This leads to the use of global load balancing which could have been avoided by faster detection of operator end.

In our case, the problem is complicated by the fact that activations can be processed by any SM-node through load sharing. Thus, we need coordination of SM-nodes which is done by one SM-node, called *coordinator*, and a protocol similar to two-phase commit.

Detecting the end of operator O_i proceeds as follows. During activation consumption, each thread that empties a queue Q of O_i checks whether the producing operator O_{i-1} has terminated. If it has, Q will no longer receive activations and becomes inactive. In this case, the thread sends a EndOfQueue signal to its scheduler. After receiving p EndOfQueue signals (p queues are created for each operator on an SM-node), the scheduler sends a EndofQueuesAtNode message to the coordinator scheduler. The receipt of s such messages (one per SM-node) indicates the coordinator that all queues of O_i are inactive. This means that O_i is almost terminated but there may still be threads that are processing O_i activations. Therefore, a second synchronization phase between the coordinator and the threads (via their scheduler) is needed to make sure that they have terminated processing O_i activations. Then, the coordinator can tell all schedulers to update their list of consumable queues. This protocol is cheap ($4s$ inter-node messages) and minimizes the delay between end of operator and detection.

5 Performance Evaluation

Performance evaluation of a parallel execution model for multi-join queries is made difficult by the need to experiment with many different queries and large relations. The typical solution is to use simulation which eases the generation of queries and data, and allows testing with various configurations. However, simulation would not allow us to take into account some important performance aspects such as the overhead of thread interference within an SM-node. On the other hand, using implementation and benchmarking would restrict the number of queries and make data generation very hard. Therefore, we decided to fully implement our execution model on a multiprocessor and simulate the execution of operators. Thus, query execution does not depend on relation content and can be simply studied by generating queries and setting relation parameters (cardinality, selectivity, skew factor, etc.).

In the rest of this section, we describe our experimentation platform and report on performance results on load balancing at two levels: locally within an SM-node and globally among SM-nodes.

5.1 Experimentation Platform

We now introduce the multiprocessor configuration we have used for our experiments and the way we have generated parallel execution plans and relations. We also present the methodology that was applied in all experiments.

5.1.1 Multiprocessor Configuration

We have implemented our execution model on a 72-processor KSR1 computer [Frank93] for two reasons. First, it is freely available to us at Inria. Second, its shared virtual memory architecture and high number of processors make it possible to organize as a hierarchical parallel system. Each processor is 40 MIPS fast and has its own 32 Megabytes memory, called local cache. KSR1's Allcache system uses hardware to provide a shared virtual memory space which includes all local caches.

To experiment with various hierarchical system configurations, we cluster processors as SM-nodes[2] and simulate inter-node communication using the following typical network parameters:

[2]In order to simulate a real SM-node and avoid the influence of the NUMA architecture of the KSR1, we force all read and write data accesses to be local, *i.e.* in the local cache of the processor.

Network Parameters	*Values*
Bandwidth (based on [Mehta95])	Infinite
End to end transmission delay	0.5 ms
CPU cost for sending 8K byte	10000 instr.
CPU cost for receiving 8K byte	10000 instr.

Furthermore, to experiment with multiple disks (only one disk of the KSR1 was available to us), we simulate disk accesses to base relations with the following parameters:

Disk Parameters	*Values*
Nb. of disks	1 per processor
Disk latency [Mehta95]	17 ms
Seek Time	5 ms
Transfer Rate	6 MB/s
CPU cost for asynchronous I/O init.	5000 instr.
I/O Cache Size	8 pages

5.1.2 Parallel Execution Plans

The input to our execution model is a parallel execution plan obtained after compilation and optimization of a user query. To generate queries, we use the algorithm given in [Shekita93] with three kinds of relations: small (10K-20K tuples), medium (100K-200K tuples) and large (1M-2M tuples). First, the predicate connection graph of the query is randomly generated. Since most multi-join queries in practice tend to have simple join predicates, we consider only acyclic connected graphs. Second, for each relation involved in the query, a cardinality is randomly chosen in one of the small, medium or large ranges. Third, the join selectivity factor of each edge (R,S) in the predicate connection graph is randomly chosen in the range $[0.5 * min(\mid R \mid, \mid S \mid) / \mid R \times S \mid, 1.5 * max(\mid R \mid, \mid S \mid) / \mid R \times S \mid]$.

The result of query generation is an acyclic connected graph adorned with relation cardinalities and edge selectivities. We have generated 20 queries, each involving 12 relations. Each query is then run through our DBS3 query optimizer [Lanzelotte93] which gives us full control over optimization. For each query, the two best bushy operator trees are retained. To automatically produce parallel execution plans from operator trees, we make a number of assumptions. First, relations are fully partitioned across all SM-nodes. Second, all SM-nodes are allocated to all operators of the plan. Third, pipeline chains are executed one-at-a-time. Although these assumptions cannot yield the best parallel execution plan, they are reasonable. Furthermore, our goal here is to produce the best execution for a given parallel execution plan.

Without any constraint on query generation, we would obtain very different executions which would make it difficult to give meaningful conclusions. Therefore, we constrain the generation of operator trees so that the sequential response time is between 30 mn and one hour. Thus, we have produced 40 parallel execution plans involving about 1.3 Gigabytes of base relations and about 4 Gigabytes of intermediate results. Since we ignore the content of relations, we could generate automatically large relations with given cardinalities.

5.1.3 Experimentation Methodology

In the following experiments, each point in a graph will be obtained from a computation based on the response times of 40 parallel execution plans. Since the different parallel execution plans correspond to 20 different queries, computing the average response time does not make sense. Therefore, the results will always be in terms of comparable execution times. For instance, in a speedup experiment, let the speedup be the ratio of response time with p processors over the response time with one processor, each point will be computed as the average of the speedups of all plans.

More generally, each point of a graph is obtained with n measurements, each on a different plan, using the following formula:

$$\frac{1}{n}\sum_{i=1}^{i=n}\frac{response\ time\ of\ experiment}{reference\ response\ time}$$

where the reference response time will be indicated for each experiment. To obtain precise measurements, each response time is computed as the average of five successive measurements.

5.2 Local Load Balancing

To study the performance of our model within an SM-node, we compare it with two other execution models. Then we study the impact of data skew.

5.2.1 Performance Comparisons

To compare with our model in the shared-memory case, we have chosen and implemented two well-known load balancing strategies. The first strategy is synchronous pipelining (SP) [Shekita93] and is designed for shared-memory. Each processor is multiplexed between I/O and CPU threads and participates in every operator of a pipeline chain. I/O threads are used to read the base relations into buffers. Each CPU thread reads tuples from the buffers and probes all the hash tables along the pipeline chain. Unless there is severe data skew (which yields high variations in tuple processing time), this model will achieve perfect load balancing. However, SP cannot be implemented in shared-nothing because data redistribution between two successive operators would imply costly remote procedure synchronization. The second strategy has been designed for shared-nothing [DeWitt90, Boral90]. For each pipeline chain, processors are statically allocated to operators based on a ratio of the estimated complexity, including CPU and I/O costs, of each operator versus the global complexity of the pipeline chain. This strategy yields good load balancing as long as the cost model is accurate. We adapt this strategy for shared-memory, allowing intra-operator load balancing and call it *fixed processing* (FP). This was implemented by using our execution model, restricting each thread to process activations associated with only one operator. To compare with SP and FP, we call our model *dynamic processing* (DP) to reflect the fact that processors are dynamically allocated to operators of a pipeline chain.

Figure 6 compares the relative performance of the three strategies for different numbers of processors with no data skew. The reference response time is that of SP which is always best. FP is always worse because of discretization errors which worsen as the number of processors decreases. The performance of our strategy is very close to that of SP from 8 and 32 processors and remain close for higher numbers. The small performance difference is due to thread interference and queue management in DP.

The performance of FP strongly depends on the accuracy of its cost model and we were interested in studying the impact of errors in cost estimates. Figure 7 shows the relative performance of FP versus the error rate, using several degrees of parallelism, with SP's response time used as reference.

To obtain a measurement with an error rate r, the cardinalities of base and intermediate relations are distorted by a value chosen in [-r,+r], which propagates errors in estimating the cost of operators and the number of allocated processors. The measurements have been performed with a realistic error rate between 0 and 30%. Given the random nature of the measurements, we have chosen to restrict the number of execution plans tested. However, for each error rate, three distortions are randomly picked for each plan.

The results show that response time degrades significantly as the error rate increases and that it depends much on the degree of parallelism. With few processors (e.g., 8), performance degradation is small with a small error rate but increases significantly as the error rate increases. This is because badly allocated processors get idle which is more significant with few processors (e.g., 1/8 is worse

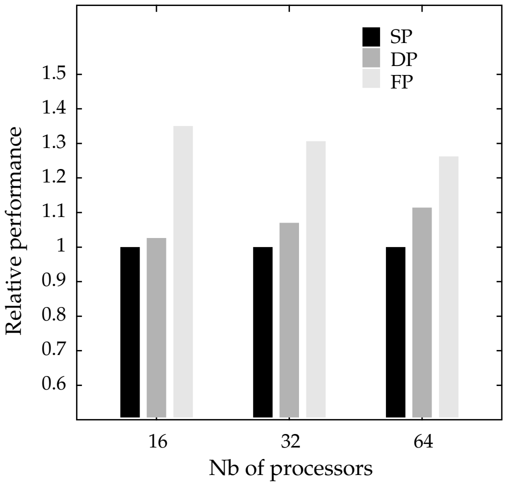


Figure 6: Relative performances of SP, FP and DP

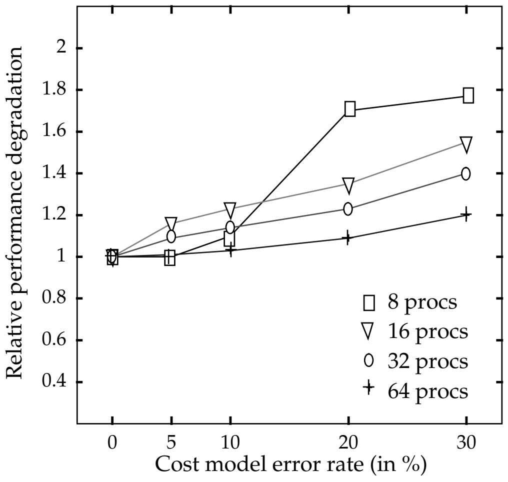


Figure 7: Impact of cost model errors on FP

than 1/64). So with more processors (e.g., 64), a small error rate changes the effectiveness of processor allocation, but the impact on performance is proportionally less. The worst load balancing is obtained when a few processors are allocated to costly operators while all others are allocated to cheap operators. This is reached sooner or later depending on the degree of parallelism. For instance, with 5 operators and 8 processors, the worst load balancing is reached with only 3 processors badly allocated, whereas it is reached with 59 with 64 processors. This explains the threshold around 20% with 8 processors and the steadier and smaller degradation with more processors. These experiments confirm the limitations of static load balancing, thus motivating the need for dynamic load balancing.

Figure 8 shows the average speed-up of all query executions for each strategy, with r=0 for FP. Again SP is always slightly better than DP, and FP is always worse. Up to 32 processors, SP and DP yield near-linear speedup. Beyond, the overhead of data access across KSR1's memory hierarchy reduces a bit the performance improvement. However, a hierarchical system would typically include SM-nodes with less than 32 processors thereby making DP an excellent strategy.

5.2.2 Impact of Data Skew

In our model, all threads have access to all local activation queues and thus can interfere with each other. The interference overhead increases with bad distributions of activations in queues which stem from various forms of data skew [Walton91]. Attribute value skew or tuple placement skew lead to unbalanced relation partitions thereby causing bad distribution of trigger activations in scan queues. Redistribution skew leads to bad distribution of data activations in pipeline queues.

In this experiment, we study the overhead of interference in our model in case of skew. To do so, we have introduced redistribution skew in the production of trigger activations and in all operators producing pipelined tuples. For simplicity, the skew factor of a producer operator does not impact that of the consumer operator. All operators have the same skew factor based on a Zipf function [Zipf49] that yields a factor between 0 (no skew) and 1 (high skew).

Figure 9 shows the relative performance of DP versus the skew factor with 64 processors, the reference response time being that with no skew. The important conclusion is that the impact of skew on our model is insignificant. This is due to several design decisions. First, our model allows a high degree of operator partitioning which reduces the negative effect of skew [Kitsuregawa90]. Second,

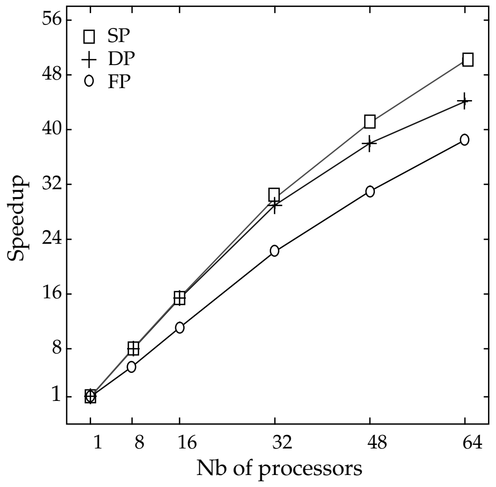


Figure 8: Speedup of SP, FP, DP

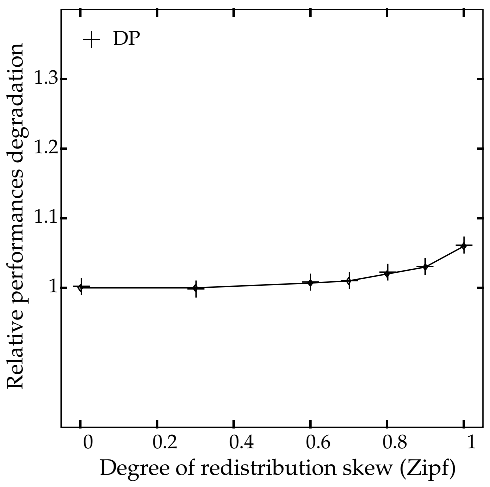


Figure 9: Impact of data skew on DP

the priority-based association of queues to threads reduces interference. Finally, interferences are further reduced by caching read and write activations.

This experiment could not consider all consequences of data skew. In particular, data skew can also yield different processing times for activations. This happens in case of attribute value skew, selectivity skew and join product skew. In effect, these kinds of skew can overload some queues because of higher processing times, and not because of higher numbers of activations. And this yields the same good behavior of the model.

5.3 Global Load Balancing

In our model, we minimize the use of global load balancing (which incurs communication overhead) by favoring more efficient local inter- and intra-operator load balancing. To assess the performance gain of such strategy in a hierarchical system, we compare it with FP which performs well in shared-nothing. In our experiment, we adapt FP as follows. As each operator is present on all SM-nodes, the distribution of the processors of each SM-node over the operators of the pipeline chain is done independently, according to the strategy described in Section 5.2.1.

Since FP imposes that processors process activations of only one operator, a processor that becomes idle triggers the use of global load balancing. Therefore, an uneven operator load distribution on the nodes may lead to global load balancing at the end of each operator.

In order to create poor load balancing within SM-nodes, we simply introduce skew as before. Without skew, we have experimentally observed that global load balancing is almost never used, and, the response time of each plan with or without global load balancing is similar.

We first compare the behavior of FP and DP for a simple execution plan, i.e., a pipeline chain of 5 operators, each having a redistribution skew factor of 0.8. The hierarchical system is configured as 4 SM-nodes, each having 8 processors. We measured the amount of data exchanged between nodes with FP and DP. For this experiment, FP requires 9 Megabytes data to be transferred versus only 2.5 Megabytes for DP. The difference observed is explained by the following.

With FP, all processors can become idle independently of each other. Since there is no dynamic inter-operator load balancing, a processor allocated to an operator can be idle whereas another processor allocated to another operator is overloaded. The idle processor will then invoke global load balancing to steal work from a remote processor allocated to the same operator. Thus, several starving

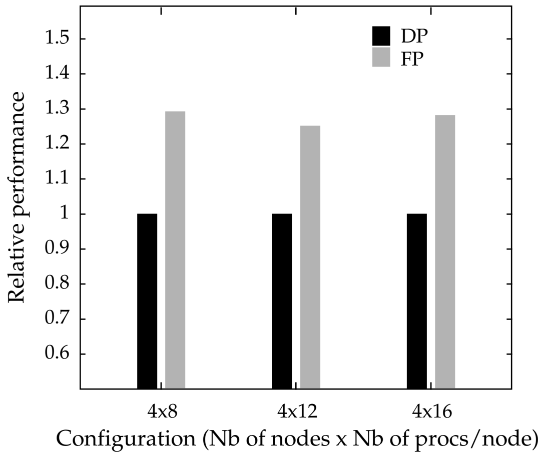


Figure 10: Relative performance of FP and DP

situations can appear at the same SM-node. Furthermore, there can be mutual stealing between two SM-nodes.

With DP, these problems are avoided. When a processor becomes idle, this is because the entire SM-node is starving. Since load sharing is applied at the level of the SM-node (rather than the processor), there cannot be repeated or mutual starving situations.

As in the previous experiments, we have also done other measurements with 40 execution plans (bushy trees involving 12 relations), with three configurations and a skew factor of 0.6. Figure 10 shows the performance gain of DP over FP with 4 nodes of 8, 12, respectively 16 processors. We observed, among all executions, performance gains between 14 and 39%. This is due to less utilization of global load balancing for DP as well as better performance of DP on SM-nodes. The communication overhead due to global load balancing is 2 to 4 times smaller for DP. Also, processor idle time with DP is almost null whereas it is quite significant with FP. We did not observe any relationship between the number of processors on each node and the performance difference between FP and DP.

6 Conclusion

In this paper, we have addressed the problem of dynamic load balancing for multi-join queries in a hierarchical parallel system, i.e., a shared-nothing system whose nodes are shared-memory multiprocessors (SM-nodes). Given a parallel execution plan resulting from the parallel optimization of a query, the goal of dynamic load balancing is to minimize query response time by avoiding processor idle time. We have proposed a new, dynamic solution that maximizes load balancing locally within shared-memory nodes and reduces as much as possible the need for load sharing across nodes. This is obtained by decomposing the work in self-contained activations that represent the finest units of sequential processing and allowing any thread to process any activation of its SM-node. Thus, there is no static association between threads and operators. This yields much flexibility in exploiting intra-operator and inter-operator parallelism within an SM-node, and thus, reduces to the minimum the need for global load balancing, i.e., when there is no more work to do in an SM-node.

Furthermore, our execution model eases static optimization, which is tipically complex in a hierarchical architecture, by avoiding to statically decide the operator scheduling and the association between operators and processors. However, if static distribution is decided by the optimizer, our execution model can exploit it and would still minimize the overhead of dynamic load balancing. The cost estimate errors, unavoidable in such architecture will have small impact on response time due to dynamic load balancing.

To evaluate the performance of our model, we did an implementation on a 72-processor KSR1 computer. KSR1's shared virtual memory architecture and high number of processors have made it easy to organize as a hierarchical parallel system. To experiment with many different queries, large relations and different relation parameters (cardinality, selectivity, skew factor, etc.), we have simulated the execution of atomic operators. We have performed various experiments at two levels: locally within an SM-node and globally among SM-nodes.

In the shared-memory case, we have compared our load balancing strategy called dynamic processing (DP) with synchronous pipelining (SP) and fixed processing (FP). SP is best for shared-memory but does not work in shared-nothing whereas FP is designed for shared-nothing and also works in shared-memory. FP is always worse because of discretization errors which worsen as the number of processors decreases. The performance of our strategy is very close to that of SP from 8 to 32 processors and remain close for higher numbers. Both SP and DP strategies show very good speedup, even with highly skewed data.

To assess the performance of our global load balancing strategy in a hierarchical system, we have compared it with FP which performs well in shared-nothing. Our strategy outperforms FP by a factor between 14 and 39% and the communication overhead due to global load balancing is 2 to 4 times smaller. Finally, processor idle time is almost null with DP whereas it is quite significant with FP.

To summarize, in shared-memory, our execution model performs as well as a dedicated model and can scale up very well to deal with multiple nodes. Considering the current multiprocessor towards hierarchical architectures with database as the main target application, such a model provides two strong advantages: predictable performance across many different configurations and portability of DBMS software.

Acknowledgments
The authors wish to thank Benoit Dageville for many fruitful discussions on parallel execution model and Jean-Paul Chieze for helping us with the KSR1 at Inria.

References

[Apers92] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, A. N. Wilschut, "PRISMA/DB: A Parallel Main Memory Relational DBMS". *IEEE Trans. Knowledge and Data Engineering,* 4(6), December 1992.

[Berg92] C. A. van den Berg, M, L, Kersten, "Analysis of a Dynamic Query Optimization Technique for Multi-join Queries". *Int. Conf. on Information and Knowledge Engineering*, Washington, 1992.

[Bergsten91] B. Bergsten, M. Couprie, P. Valduriez, "Prototyping DBS3, a Shared Memory Parallel System". *Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, December 1991.

[Blasgen79] M. Blasgen, J. Gray, M. Mitoma, T. Price, "The Convoy Phenomenon". *Operating Systems Review* 13(2), Avril 1979.

[Boral90] H. Boral, W. Alexander, L. Clay, G.Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez, "Prototyping Bubba, A Highly Parallel Database System". *IEEE Trans. Knowledge and Data Engineering.*, 2(1), March 1990.

[Bouganim96] L. Bouganim, B. Dageville, P. Valduriez, "Adaptative Parallel Query Execution in DBS3". *Int. Conf. on EDBT, to appear, 1996.*

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

[Dageville94] B.Dageville, P.Casadessus, P.Borla-Salamet, "The Impact of the KSR1 AllCache Architecture on the Behavior of the DBS3 Parallel DBMS". *Int. Conf. on Parallel Architectures and Language*, Athens, 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 Trans. on Knowledge and Data Engineering*, 2(1), March 1990.

[DeWitt92] 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, February 1993.

[Garofalakis96] M. N. Garofalakis, Y. E. Yoannidis, "Multi-dimensional Resource Scheduling for Paralle Queries". *ACM-SIGMOD Int. Conf., to appear, 1996*.

[Graefe93] G. Graefe, "Query Evaluation Techniques for Large Databases". *ACM Computing Surveys* 25(2), June 1993.

[Hassan94] W. Hassan, R. Motwani, "Optimization Algorithms for Exploiting the Parallel-Communication Tradeoff in Pipelined Parallelism". *Int. Conf on VLDB*, Santiago, Chile, 1994.

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

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

[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, 1990.

[Lanzelotte93] R. Lanzelotte, P. Valduriez, M. Zait, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces". *Int. Conf. on VLDB*, Dublin, August 1993.

[Lenoski92] D. Lenoski, J. Laudon, K. Gharachorloo, W. D. Weber, A. Gupta, J. Henessy, M. Horowitz, M. S. Lam, "The Stanford Dash Multiprocessor". *IEEE Computer*, 25(3), March 1992.

[Lo93] M-L. Lo, M-S. Chen, C. V. Ravishankar, P. S. Yu, "On Optimal Processor Allocation to Support Pipelined Hash Joins". *ACM-SIGMOD Int. Conf.*, Washington DC, May 1993.

[Lu91] H. Lu, M.-C. Shan, K.-L. Tan, "Optimization of Multi-Way Join Queries for Parallel Execution". *Int. Conf. on VLDB*, Barcelona, September 1991.

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

[Murphy91] M. C. Murphy, M-C. Shan, "Execution Plan Balancing". *IEEE Int. Conf. on Data Engineering*, Kobe, April 1991.

[Pirahesh90] H. Pirahesh, C. Mohan, J. Cheng, T. S. Liu, P. Selinger, "Parallelism in relational database systems: Architectural issues and design approaches". *Int. Symp. on Databases in Parallel and Distributed Systems*, Dublin, July 1990.

[Rahm95] E. Rahm, R. Marek, "Dynamic Multi-Resource Load Balancing in Parallel Database Systems". *Int. Conf. on VLDB*, Zurich, Switzerland, September 1993.

[Shatdal93] A. Shatdal, J. F. Naughton, "Using Shared Virtual Memory for Parallel Join Processing". *ACM-SIGMOD Int. Conf.*, Washington, May 1993.

[Shekita93] E. J. Shekita, H. C. Young, "Multi-Join Optimization for Symmetric Multiprocessor". *Int. Conf. on VLDB*, Dublin, August 1993.

[Schneider89] D. Schneider, D. DeWitt, "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment". *ACM-SIGMOD Int. Conf.*, Portland, May-June 1989.

[Srivastava93] J. Srivastava, G. Elsesser, "Optimizing Multi-Join Queries in Parallel Relational Databases". *Int. Conf. on Parallel and Distributed Information Systems*, San Diego, January 1993.

[Valduriez84] P. Valduriez, G. Gardarin, "Join and Semi-join Algorithms for a Multiprocessor Database Machine". *ACM Trans. on Database Systems*, 9(1), March 1984.

[Valduriez93] P. Valduriez, "Parallel Database Systems: open problems and new issues". *Int. Journal on Distributed and Parallel Databases*, 1(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, September 1991.

[Wilshut95] A. N. Wilshut, J. Flokstra, P.G Apers, "Parallel Evaluation of multi-join queries". *ACM-SIGMOD Int. Conf.*, San Jose, CA, 1995.

[Ziane93] M. Ziane, M. Zait, P. Borla-Salamet, "Parallel Query Processing With Zig-Zag Trees". *VLDB Journal*, Vol. 2, No 3, December 1993.

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

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399