Distributed and Parallel Databases 7, 99–121 (1999)

Load Balancing for Parallel Query Execution on NUMA Multiprocessors*

LUC BOUGANIM luc.bouganim@inria.fr
DANIELA FLORESCU daniela.florescu@inria.fr
PATRICK VALDURIEZ patrick.valduriez@inria.fr
INRIA Rocquencourt, France

Received ; Revised June 9, 1997; Accepted August 1, 1997

Recommended by: Tamer Ozsu

Abstract. To scale up to high-end configurations, shared-memory multiprocessors are evolving towards Non Uniform Memory Access (NUMA) architectures. In this paper, we address the central problem of load balancing during parallel query execution in NUMA multiprocessors. We first show that an execution model for NUMA should not use data partitioning (as shared-nothing systems do) but should strive to exploit efficient shared-memory strategies like Synchronous Pipelining (SP). However, SP has problems in NUMA, especially with skewed data. Thus, we propose a new execution strategy which solves these problems. The basic idea is to allow partial materialization of intermediate results and to make them progressivly public, i.e., able to be processed by any processor, as needed to avoid processor idle times. Hence, we call this strategy Progressive Sharing (PS). We conducted a performance comparison using an implementation of SP and PS on a 72-processor KSR1 computer, with many queries and large relations. With no skew, SP and PS have both linear speed-up. However, the impact of skew is very severe on SP performance while it is insignificant on PS. Finally, we show that, in NUMA, PS can also be beneficial in executing several pipeline chains concurrently.

Keywords: parallel databases, query execution, load balancing, NUMA, synchronous pipeline, execution engines

1. Introduction

Commercial database systems implemented on shared-memory multiprocessors, such as Sequent, Sun, Bull's Escala, are enjoying a fast growing success. There are several reasons for this [45]. First, like almost all parallel database systems, they are used primarily for decision support applications, (e.g., data warehouse) a strong market which is now doubling every year. Second, shared-memory provides a uniform programming model which eases porting of database systems and simplifies database tuning. Third, it provides the best performance/price ratio for a restricted number of processors (e.g., up to 20) [3].

Unlike shared-nothing [12], shared-memory does not scale up to high-end configurations (with hundreds of processors and disks). To overcome this limitation, shared-memory multiprocessors are evolving towards Non Uniform Memory Access (NUMA) architectures.

*This work has been done in the context of Dyade, a joint R&D venture between Bull and INRIA.

The objective is to provide a shared-memory programming model and all its benefits, in a scalable parallel architecture.

Two classes of NUMA architecture have emerged: Cache Coherent NUMA machines (CC-NUMA) [1, 16, 26, 28, 29], which statically divide the main memory among the nodes of the system, and Cache Only Memory Architectures (COMA) [14, 18], which convert the per node memory into a large cache of the shared address space. When a processor accesses a data item which is not locally cached, the item is shipped transparently from the remote memory to the local memory. Because shared-memory and cache coherency are supported by hardware, remote memory access is very efficient, only several times (typically four times) the cost of local access.

NUMA is now based on international standards and off-the-shelf components. For instance, the Data General nuSMP machine and the Sequent NUMA-Q 2000 [31] using the ANSI/IEEE Standard Scalable Coherent Interface (SCI) [23] to interconnect multiple Intel Standard High Volume (SHV) server nodes. Each SHV node consists of 4 Pentium Pro processors, up to 4 GB of memory and dual peer PCI/IO subsystems [10, 24]. Other examples of NUMA computers are Kendal Square Research's KSR1 and Convex's SPP1200 which can scale up to hundreds of processors.

The "strong" argument for NUMA is that it does not require any rewriting of application software. However, some rewriting is necessary in the operating system and in the database engine. In response to the nuSMP announcement from Data General, SCO has provided a NUMA version of Unix called Gemini [9], Oracle has modified its kernel [8] in order to optimize the use of 64 GB of main memory allowed by NUMA multiprocessors.

In this paper, we consider the parallel execution of complex queries in NUMA Multiprocessors. We are interested in finding an execution model well suited for NUMA which provides a balanced parallel execution. There are two dimensions for parallelizing complex queries: intraoperator parallelism (by executing each operator in parallel) and interoperator pipelined or independent parallelism (by executing several operators of the query in parallel). The objective of parallel query execution is to reduce query response time by balancing the query load among multiple processors. Poor load balancing occurs when some processors are overloaded while some others remain idle. Load balancing strategies have been proposed on either shared-nothing [2, 5, 11, 15, 25, 33, 38, 40, 47], shared-disk [22, 30, 32] or shared-memory [6, 21, 35, 41].

In shared-nothing, the use of a nonpartitioned execution model, i.e., where each processor potentially accesses all the data, is inefficient because of intensive remote data access. Thus, parallelism is obtained through data partitioning, i.e., relations are physically partitioned during query processing using a partitioning function like hashing. Then, each processor accesses only a subset of the data, thus reducing interference and increasing the locality of reference. However, a partitioned execution model has three main drawbacks: (i) overhead of data redistribution for complex queries, (ii) processor synchronization, (iii) difficult load balancing which requires much tuning.

One of the key advantages of shared-memory over shared-nothing is that execution models based on partitioning [17] or not [21, 32, 36] can be used. The use of nonpartitioned techniques eases load balancing and tuning. Moreover, there is no redistribution. However, it can lead to high interference and poor locality of reference. Shekita and Young [41]

analytically compared the two approaches in a shared-memory context. It is concluded that, if cache effects are ignored, the two approaches have similar behavior.

A very efficient execution strategy for shared-memory is *Synchronous Pipelining* (SP) [21, 37, 41]. It has three main advantages. First, it requires very little processor synchronization during execution. Second, it does not materialize intermediate results, which incurs little memory consumption. Third, unlike shared-nothing approaches, it does not suffer from load balancing problems such as cost model errors, pipeline delay and discretization errors [49].

NUMA also allows using either model. However, the heterogeneous structure of the memory may impact the choice of one model. In this paper, we first show that an execution model for NUMA should not use data partitioning but should strive to exploit efficient shared-memory strategies like Synchronous Pipelining (SP). However, considering NUMA has two consequences which can worsen the impact of skewed data distributions [48] on load balancing. First, the global memory is potentially very large, and makes it possible for the optimizer to consider execution plans with long pipeline chains. Second, the number of processors is potentially large. We show that these two factors have a negative impact on SP in case of skew.

We propose a new execution strategy which solves the inherent problems of SP, which are magnified in NUMA. The basic idea is to allow partial materialization of intermediate results and make them progressivly *public*, i.e., able to be processed by any processor, as needed to avoid processor idle times. Hence, we call this strategy *Progressive Sharing* (PS). It yields excellent load balancing, even with high skew. In addition, PS can be beneficial in executing several pipeline chains concurrently. To validate PS and study its performance, we have implemented SP and PS on a 72-processor KSR1 computer which is a COMA multiprocessor.[1]

The paper is organized as follows. Section 2 presents the parallel execution plans, which are the input for the execution model. Section 3 discusses the relevance of shared-memory and shared-nothing execution models for NUMA. Section 4 details the SP strategy and discusses its problems in NUMA, in particular, with data skew. Section 5 describes our parallel execution model for NUMA. Section 6 describes our implementation on the KSR1 computer and gives a performance comparison of SP and PS. Section 7 concludes.

2. Definitions

Parallel execution plans are the input for the parallel execution model. A *parallel execution plan* consists of an operator tree with operator scheduling. Different shapes can be considered: left-deep, right-deep, segmented right-deep, or bushy. Bushy trees are the most appealing because they offer the best opportunities to minimize the size of intermediate results [41] and to exploit all kinds of parallelism [27].

The *operator tree* results from the "macroexpansion" of the query tree [19]. Nodes represent atomic operators that implement relational algebra and edges represent dataflow. In order to exhibit pipelined parallelism, two kinds of edges are distinguished: blocking

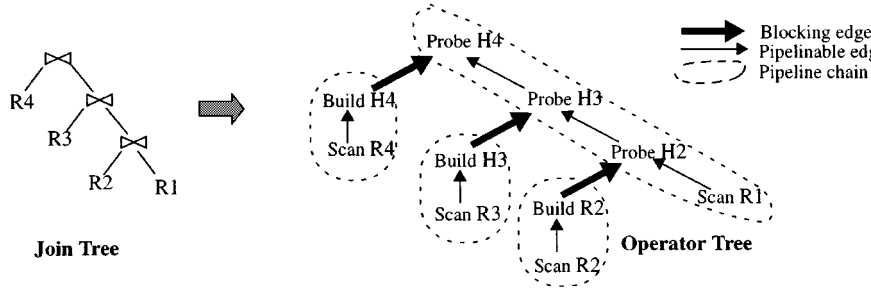

Figure 1. A query tree and the corresponding operator tree.

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". To simplify the presentation, we consider an operator tree that uses only hash join[2] [39, 46]. In this case, three atomic operators are needed: scan to read each base relation, build and probe. The build operator is used to produce a hash table with tuples from the inner relation, thus producing a blocking output. Then, tuples from the outer relation are probed with the hash table. Probe produces a pipelinable output, i.e., the result tuples.

An operator tree can be decomposed as a set of maximum pipeline chains, i.e., chains with highest numbers of pipelined operators, which can be entirely executed in memory. These pipeline chains are called fragments [41] or tasks [21]. Figure 1 shows a right-deep tree involving four relations and the corresponding parallel execution plan. Reading of relations R2, R3 and R4 as well as creation of the corresponding hash tables H2, H3 and H4 can be done in parallel. The execution of the last pipeline chain probing the hash tables is started only when H2, H3 and H4 have been built.

Assuming NUMA impacts the parallelization decisions made by the optimizer [43] since the available global memory is supposed to be large. Thus, with complex queries, we can expect that the best execution plans produced by the optimizer will often be bushy trees with long pipeline chains (at the extreme, only one pipeline chain) because they avoid materialization of intermediate results and provide good opportunities for inter- and intraoperator parallelism.

3. Execution models relevant for NUMA

In this section, we expose the two basic ways to process a multi-join query on a multiprocessor: (i) partitioned execution model with real pipeline, well suited for shared-nothing; and (ii) nonpartitioned execution model with implicit pipeline (synchronous pipeline), well suited for shared-memory. Then we present some experimental results from a previous study which help us selecting the best execution model for NUMA multiprocessors.

3.1. Partitioned execution model

In a shared-nothing architecture, a partitioned execution model is used in order to avoid intensive communication between processors. In such model, the execution of an algebra operator is split in suboperators, each one applied to a subset of the data. For example, the result of R join S can be computed as the union of the join of R_i and S_i, if R_i and S_i were obtained by a partitioning function like hashing on the join attribute.

The problem of such approach is that each join operand must be redistributed on the join attribute, using the same partitioning function, before processing the join. This has three main drawbacks:

- The communication implied by the redistribution phase itself is very costly (however unavoidable in shared-nothing architecture).
- The redistribution phase implies a lot of synchronization between producers and consumers.
- Even with a perfect partitioning function, skewed data distributions [48] may induce uneven partitions. These uneven partitions may destroy the *locality of reference* obtained by the partitioning (to avoid unbalanced execution, the data will be dynamically redistributed).

3.2. Synchronous pipeline strategy

SP has proven to yield excellent load balancing in shared-memory [21, 37, 41]. 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 applies successively each atomic operator of the pipeline chain using procedure calls. SP is, in fact, a simple parallelization of the monoprocessor synchronous pipeline strategy.

To illustrate this strategy, we describe below the algorithm of operator ProbeH2 (see figure 1):

1. **ProbeH2(t_{R1}: Tuple(R1), H2: HashTable(R2))**
2. foreach tuple t_{H2} in H2 matching with t_{R1}
3. if pred(t_{R1},t_{H2}) then ProbeH3([t_{R1},t_{H2}], H3)

A procedure call to ProbeH2 is made for each tuple t_{R1} that satisfies the Scan predicate. For each tuple produced by probing tuple t_{R1} with the hash table H2, a procedure call is made to ProbeH3, and so on until producing the last result tuple of the pipeline chain.

Therefore, with synchronous pipeline, there is no synchronization between processors. In fact, the producer of a tuple is also its consumer (the tuple is the argument of the procedure call). However, during the join, each processor potentially accesses all the data involved in the join (and not a subset of the data as in a partitioned execution model).

3.3. Previous study

In [7], we have proposed an execution model, based on partitioning, called *Dynamic Processing* (DP), for a hierarchical architecture (i.e., a shared-nothing system whose nodes are shared-memory multiprocessors). In DP, the query work is decomposed in self-contained units of sequential processing, each of which can be carried out by any processor. Intuitively, a processor can migrate horizontally and vertically along the query work. The main advantage is to minimize the communication overhead of internode load balancing by maximizing intra- and interoperator load balancing within shared-memory nodes. Experimental comparison between DP and a classic static load balancing technique have shown performance gains between 14 and 39%.

In this study, we also compare the performance of DP and SP on a shared-memory machine. For this purpose, we had to simulate shared-memory on the KSR1. All data accesses were artificially made in the local memory, in order to avoid the effect of NUMA (remote memory access). The experiments have shown that, in a shared-memory multiprocessor, the performance of DP is close to that of SP (see figure 2(a)), which confirmed the analysis in [41].

As a starting point of the work herein described, we have modified our implementation on the KSR1 computer in order to take into account the NUMA properties of the KSR1, by making remote memory access for reading or writing tuples. To assess the relevance of a partitioned execution model for NUMA, we made measurements with the NUMA version of SP and DP. The same measurements as in [7] have shown a performance difference of the order of 35% in favor of SP (see figure 2(b)).[3]

The performance degradation of DP stems from intensive data redistribution (all relations), which implies interference between processors and remote data writing (much more costly than remote data reading during the probe phase of the synchronous pipeline strategy).

To summarize, a partitioned execution model is well suited for shared-nothing or hierarchical architectures. However, the use of a nonpartitioned execution model like SP seems more appropriate for NUMA.

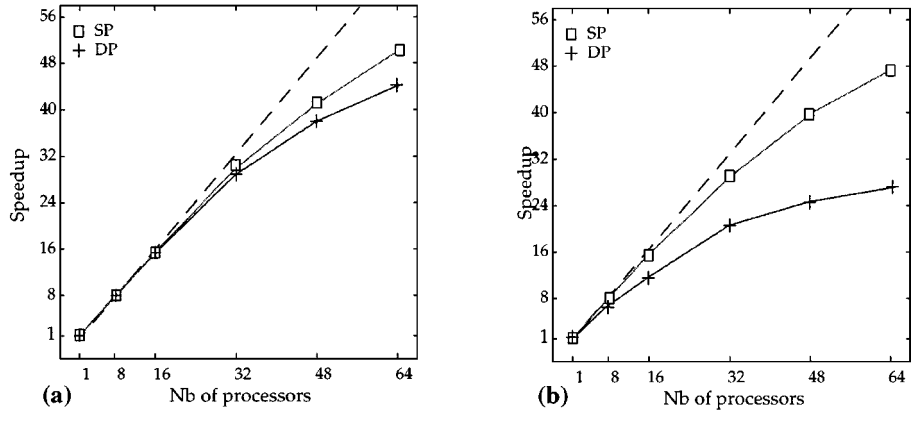

Figure 2. Speed-up of SP and DP: (a) on shared-memory, (b) on NUMA.

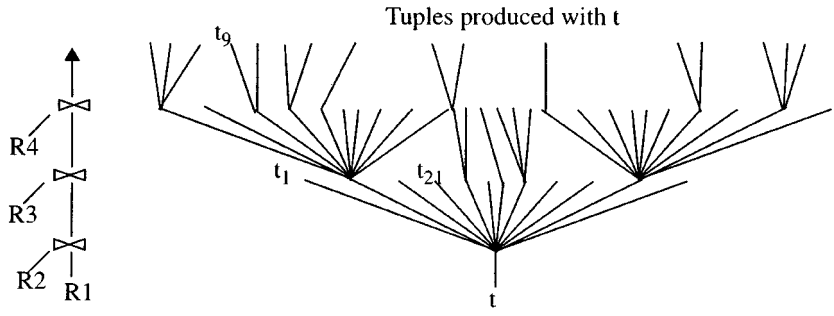

Figure 3. Processing of one tuple of R1.

4. Synchronous pipelining and data skew

Skewed (i.e., nonuniform) data distributions are quite frequent in practice and can hurt load balancing. The effects of skew on parallel execution are discussed in [48] in a shared-nothing context. The taxonomy of [48] does not directly apply to execution models which do not use data partitioning, as is the case for SP. As an evidence, *redistribution skew* does not hurt SP.

Figure 3 illustrates the processing of one tuple along the probe chain. The tuples produced by a tuple t of the base relation R1 form a virtual tree. Two observations can be made. First, SP does depth-first processing of this tree (for instance, t9 is produced before t21). At any time, the tuples being processed are referenced in the stacks of procedure calls. Second, all the tuples in the tree rooted at t are produced and processed by a single thread. On this simple example, we can indentify two kinds of skew that do hurt SP.

- **High selectivity skew** appears on a select operator (if any) at the beginning of the pipeline chain, if the result reduces to a small number of tuples, less than the number of threads. Since each tuple produced by the select operator is entirely processed (across the entire pipeline) by a single thread, this can lead to poor load balancing. An extreme case is when a single tuple is initially selected and the entire load generated is done by a single thread (while all the other threads are idle).
- **Product skew** appears on a join operator, when there are high variations in the numbers of matching tuples of the inner relation for the tuples of the outer relation. This is rare for joins on a key attribute, but quite frequent for inequi-joins or joins on non-key attributes.

These two kinds of skew lead to high differences in tuple processing time. Moreover, consuming base relations one-tuple-at-a-time can yield high interference overhead [6, 20] and this is generally avoided by having each thread consuming several tuples in batch mode. Given the fact that each batch is entirely processed by one thread, this can worsen the impact of skew.

To evaluate the negative effects of skew on a parallel execution in NUMA, we use a simple analytical model based on the following assumptions. (i) Only one operator of the pipeline chain, at the rth rank among n operators, has skew (we consider that the operators

are numbered from 1 to n following their order in the pipeline chain). (ii) The global cost of each operator is the same, i.e., assuming T to be the total CPU time of the sequential execution, T/n is the execution time of each operator. (iii) One tuple, the "skewed" tuple produces k% of that operator's result, and there are p threads allocated to the query.

The thread processing the skewed tuple does k% of the skewed operator and of all the subsequent operators, thus taking, $k * (n - r + 1) * \mathrm{T}/n$ to process this tuple. For instance, if the skewed operator is the first one, it will process k% of the entire pipeline. Thus, the maximum speed-up can be computed by:

$$\min\left(p, \frac{n}{k * (n - r + 1)}\right). \tag{1}$$

Figure 4 shows the speed-ups obtained versus k for different ranks r from one graph to the other. The results illustrate that even a small (but realistic) skew factor can have a very negative impact on the performance of SP.

Based on this simple model, we can observe that the speed-up degradation in case of skew depends on: (i) the skew factor k, (ii) the position r of the skewed operator in the pipeline chain and (iii) the number of threads p. The speed-up gets worse as the skewed operator reaches the beginning of the chain. Furthermore, the maximum speed-up is bound by a factor independent on the number of processors. Therefore, the loss of potential gain increases with the number of processors.

Although it does not appear on our analytical model, the length of the pipeline chain also impacts load balancing in case of skew. When several operators are skewed, the negative effects can be, in the worst case, exponential in the length of the pipeline chain. A final consideration is the impact of batch size (the granule of parallelism) on load balancing. It seems obvious that load unbalancing, i.e., variations in batch processing time, increases with the batch size [20].

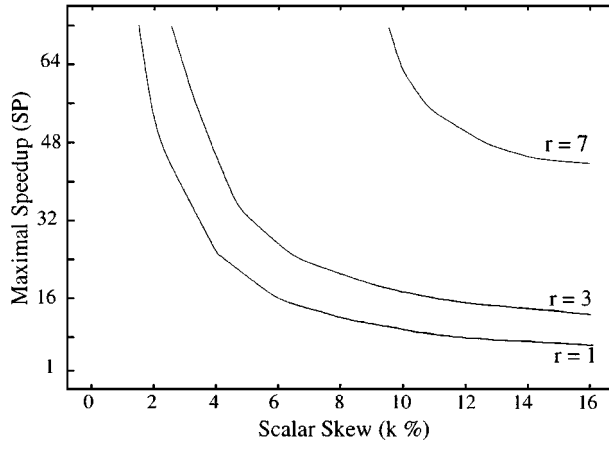

Figure 4. Maximal theoretical speed-up.

In NUMA, because the number of processors is higher and the pipeline chains tend to be longer,[4] all these problems are magnified.

5. Parallel execution model for NUMA

In this section, we propose a parallel execution model for NUMA which strives to keep the advantages of SP without load unbalancing in case of skew. Since SP does not materialize intermediate tuples, there is no easy way to do load balancing. Our PS strategy is based on partial materialization of intermediate results, using activations [7], in order to enable load redistribution. Load balancing is then achieved by allowing any thread to process any materialized tuple (activation), thus ensuring that no thread will be idle during the query execution.

A naive implementation of this idea could yield high overhead. Public access (by all processors) to intermediate results induces much interference for synchronization. Furthermore, tuple materialization incurs copying and memory consumption. One priority in designing our execution model was to minimize those overheads. Synchronization overhead is reduced by using two execution modes. In *normal mode*, when no processor is idle, partially materialized results can be accessed only by the processors that produced them, thereby avoiding processor interference.[5] In *degraded mode*, i.e., as soon as one processor is idle, intermediate results are progressively made public to allow load redistribution. Furthermore, our implementation avoids data copying and limits memory consumption.

In the rest of this section, we present the basic concepts underlying our model. Then, we present in more details PS in normal and degraded modes. Finally, we illustrate PS with an example.

5.1. Basic concepts

A simple strategy for obtaining good load balancing 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 a lot of system calls which are due to thread scheduling, interference and convoy problems [4, 21, 37]. Instead of relying on the operating system for load balancing, we choose to allocate only one thread per processor per query. The advantage of this one-thread-per-processor allocation strategy is to significantly reduce the overhead of interference and synchronization.

An *activation* represents the finest unit of sequential work [7]. Two kinds of activations can be distinguished: *trigger activation* used to start the execution of a scan on one page of a relation and *tuple activation* describing a tuple produced in pipeline mode. Since, in our model, any activation may 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. A trigger activation is represented by an (*Operator, Page*) pair which references the scan operator and the page to scan. A tuple activation is represented by an (*Operator, Tuple, HashTable*) triple which references the operator to process (build or probe), the tuple

to process, and the corresponding hash table. For a build operator, the tuple activation specifies that the tuple must be inserted in the hash table. For a probe operator, it specifies that the tuple must be probed with the hash table.

The processing of one activation by one operator implies the creation of one or more activations corresponding to result tuples, for the next operator. These activations are stored in *activation buffers* whose size is a parameter of our model. At any time, a thread has an *input activation buffer* to consume activations and an *output activation buffer* to store result tuples.

Activation buffers associated with an operator are grouped in *activations queues*. To avoid activation copies (from activation buffers to activation queues), only references to buffers are stored in queues. 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 has a queue to receive input activations, i.e., the pipelined operand. We create one queue per operator and thread. Assuming p processors executing a pipeline chain with n operators numbered from 1 to n (operator n produces the final result), the set of queues can be represented by a matrix Q. $Q_{i,j}$ ($i \in [1, n]$, $j \in [1, p]$) represents the queue of thread j associated with operator i. A column represents the set of queues associated with a thread while a row represents all the queues associated with an operator. The sizes of the buffers and queues impose a limit to memory consumption during query execution.

Each queue is locked by a semaphore (mutex) to allow concurrent read and write operations. Each thread holds a *queue lock* on all its associated queues at the beginning of the execution. We call *private queue* a queue associated with a thread. During normal execution mode, access to private queues, i.e., *private access*, is restricted to the owner only, thus without synchronization. Degraded execution mode implies releasing some queue locks. When a thread releases a lock on a private queue, the queue becomes *public*. Access to public queues, i.e., *public access*, is allowed to every thread, but with synchronization.

Figure 5 illustrates all these concepts with an initial configuration of our model while executing the last pipeline chain presented in figure 1. There are four operators executed by three threads. Therefore, we have 12 queues, each one protected by a lock. Thread T_1 is

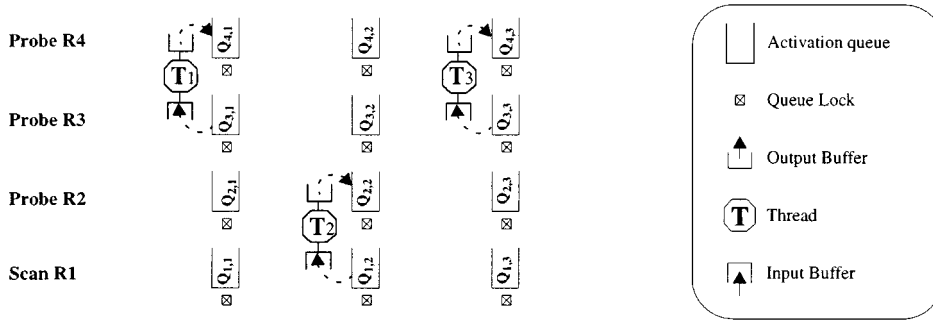

Figure 5. Initial configuration of PS.

executing operator *Probe R3*, so its input buffer contains activations from queue $Q_{3,1}$ and writes results in the output buffer, which is flushed in Queue $Q_{4,1}$.

5.2. Execution in normal mode

During execution in normal mode, a thread consumes and produces activations in its set of private queues (each thread has one private queue for each operator of the pipeline chain), without requiring synchronization.

Processing activations of the pipeline chain proceeds as follows. For each activation (trigger or tuple) of the input buffer, the thread executes the code of the operator referenced by the activation on the corresponding tuples, thus producing activations for the next operator in the output buffer. When an output buffer gets full, its reference is inserted in the queue of the next operator and an empty buffer is allocated. When the input buffer has been entirely processed, it becomes the output buffer of the next operator which becomes current. This process is repeated until the end of the pipeline chain. If the input buffer of the current operator is empty (there are no more activations to process), the thread tries to consume in the queue of the previous operator(s). Thus, when a thread executes an operator, all the queues of the following operators are necessarily empty. In other words, operators at the end of the pipeline chain get consumed in priority with respect to the operators at the beginning. This minimizes the size of intermediate results.

Processing activations of the first operator (scan) is slightly different. At the beginning of execution, trigger activations are inserted in the queues of the scan operator. This is done in a way that maximizes I/O parallelism. For instance, assuming that each processor has its own disk, the trigger activations inserted in $Q_{1,j}$ must yield accesses to disk j. Thus, when p threads read in parallel p trigger activations of the scan operator, they also do parallel access to p disks. Trigger activations are processed as follows. The thread first performs a synchronous I/O to read the first page and initiates asynchronous I/Os for the subsequent pages. After reading the first page, the thread selects (and projects) the tuples in the output buffer. When the trigger activation (i.e., one page) has been consumed, the next operator is started and execution proceeds as described above. When the thread comes back to the scan operator (to consume other trigger activations), asynchronous I/Os are likely to be completed and there is no waiting.

A problem occurs when processing an activation produces more tuples than what can be stored in the queue of the next operator. There are two main reasons for this. Either the current activation produces too many results tuples because of data skew, or the memory space allocated to activation buffers and queues is not enough. In either case, the thread suspends its current execution by making a procedure call to process the output activation buffer which cannot be flushed into the full activation queue. Thus, context saving is done efficiently by procedure call, which is much less expensive than operating system-based synchronization (e.g., signals).

To summarize, each activation is processed entirely and its result is materialized. This is different from SP where an activation is always partially consumed. If we consider the tree of tuples produced by a trigger activation (see figure 3), activation consumption is partially breadth-first in PS versus strictly depth-first in SP.

5.3. Execution in degraded mode

Execution in normal mode continues until one thread gets idle, because there is no more activation to process in any of its private queues. Then, a form of load sharing is needed whereby the other threads make public some of their activations which can then be consumed by the idle thread.

Any thread can make a queue public, simply by releasing the lock it holds since the beginning of execution. After releasing the lock, subsequent access to the queue by all threads (including the initial thread) requires synchronization. To further reduce interference in degraded mode, only a subset of the queues is made public at a time. We use a simple heuristic for the choice and the number of queues to be made public:[6] (i) all queues $Q_{k,j}$ corresponding to operator k are made public together; (ii) operators are chosen in increasing order of the pipeline chain (from 1 to n).

At any time, two global indicators are maintained: FirstActiveOp and SharingLevel. *FirstActiveOp* indicates the first active operator in the pipeline chain. Thus, at the beginning of execution, we have $FirstActiveOp = 1$. *SharingLevel* indicates the operator of highest rank whose queues have been made public. At the beginning of execution, we have $SharingLevel = 0$. These two indicators can be updated by any thread and can only increase during execution, i.e., a public queue cannot become private again. Query execution ends when *FirstActiveOp* is $n + 1$.

Query execution can now be summarized as follows. During normal execution, each thread t consumes activations in its private queues $Q_{i,t}$ ($i \in [SharingLevel + 1, n]$), without any synchronization. When it has no more activations in its private queues, a thread attempts to consume activations in the set of public queues ($Q(i, j), i \in [FirstActiveOp, SharingLevel], j \in [1, p]$), by locking them. When no more activations are available in the public queues, *SharingLevel* is increased by 1 which notifies all threads that they must release their lock, i.e., make public the queues associated with operator *SharingLevel*. Notice that a thread may consume activations from public queues during degraded mode but will always produce activations for its private queues.

5.4. Simple example of PS execution

We now illustrate the main concepts of our model on the example given in figure 1, executing the join of R1, R2, R3 and R4, with three processors and thus three threads. We concentrate on the degraded mode, with one idle thread, which is more interesting. We assume that the join with R2 of tuple t, resulting from the select of R1, produces a high number of tuples for operator *Probe R3* because of product skew. Thread T_2 processes this tuple.

Figure 6 gives an execution snapshot when T_2 is still producing tuples resulting from joining t with R2. T_3 ends processing its activation buffer, coming from queue $Q_{4,3}$. When T_3 finishes its last activation, it tries to find new activations in its private queues $Q_{4,3}$ and $Q_{3,3}$. Since it fails, it looks up the public queues $Q_{2,3}$, $Q_{2,2}$ and $Q_{2,1}$, using synchronized accesses. Again, it fails finding activations. So it increments *SharingLevel* and tries to consume in queue $Q_{3,2}$ (which contains activations), however, it gets blocked because T_2 is currently holding the lock. When T_2 detects the change of *SharingLevel*, it makes public the

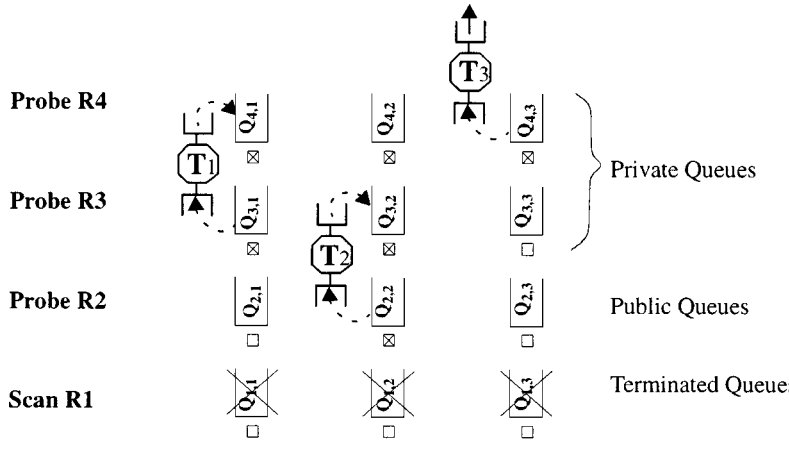

Figure 6. A simple example of PS execution.

queue $Q_{3,2}$ by releasing its lock (for further access to this queue, T_2 will use synchronized accesses). T_3, having requested the lock of $Q_{3,2}$ gets it, and consumes its activations. This produces activations for queue $Q_{4,3}$ (simply because T_3 is processing these activations) which can then be stored and consumed by T_3 without locking (because it is still private).

Two observations can be made. If *SharingLevel* is rarely tested, T_3 can remain blocked on the lock of queue $Q_{3,2}$ for a while. Thus, it is important that *SharingLevel* be often tested. However, since it is potentially accessed by all threads, *SharingLevel* could become a bottleneck. In fact, this does not happen for two reasons. First, each thread frequently accesses *SharingLevel* which gets locally cached. Second, even if writing *SharingLevel* implies invalidating it in each cache, this happens only n times (n is the length of the pipeline chain) which has a negligible effect.[7]

This simple example shows the benefits of using partial materialization of intermediate results, and partial access by all threads to intermediate results.

5.5. Execution of bushy trees

For simplicity, we have considered the execution of only one pipeline chain. However, our execution model can be easily generalized. We now show how this model can deal with more general bushy trees.

The best parallel plan produced by the optimizer is not necessarily a pure right-deep tree and can well be a bushy tree connecting long pipeline chains. The pros and cons of executing several pipeline chains concurrently in shared-memory are discussed in [21, 41]. The main conclusion is that it would almost never make sense to execute several pipeline chains concurrently since it increases resource consumption (memory and disk). In [41], it is shown that, assuming ideal speed-up, this strategy may be used in only two cases: (i) one pipeline chain is I/O-bound and the other is CPU-bound; (ii) the two pipeline chains access relations which are partitioned over disjoint sets of disks.

In NUMA, this conclusion must be reconsidered since the assumption of ideal speed-up with a high number of processors [41] is no longer valid. Also, the memory limitation constraint is relaxed. Thus, the execution of several CPU-bound[8] pipeline chains is beneficial only if each one is executed on a disjoint subset of processors. This strategy would have the following advantages. First, as the locality of reference decreases with the number of processors, fewer remote data accesses will be performed, specially if the execution occurs on a subset of the processors at the same node (shared-memory node in the Convex'SPP, NUMA-Q and nuSMP). Second, less interference will occur. Finally, executing each pipeline chain with a restricted number of processors will yield better speed-up on each pipeline chain. However, we cannot conclude that independent parallelism is always better. The choice must be made by the optimizer based on several parameters: estimated response time, degree of partitioning, available memory, length of pipeline chains, etc.

Executing several pipeline chains can be done easily in our model. The allocation of pipeline chains to the processors is decided based on their relative estimated work. Normal execution proceeds as before. However, in degraded mode, *SharingLevel* is global to all concurrent pipeline chains. Thus, a public queue can be accessed by any processor, even if it was initially working on a different pipeline chain. This is possible simply because each activation is self-contained. The only necessary modification is to create a number of queues for each thread that is equal to the length of the longest pipeline chain. In this way, load balancing is globally achieved on the set of pipeline chains.

6. Performance evaluation

Performance evaluation of a parallel execution model for complex 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 the effect of NUMA as well as important performance aspects such as the overhead of thread interference. On the other hand, using full 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 NUMA multiprocessor and simulate the execution of operators. To exercise the effect of NUMA, real tuples are actually exchanged, i.e., read and written, but their content is ignored. 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.

6.1. Experimentation platform

We now introduce the multiprocessor configuration we have used for our experiments and discuss the influence of NUMA on query execution. We also explain how we have

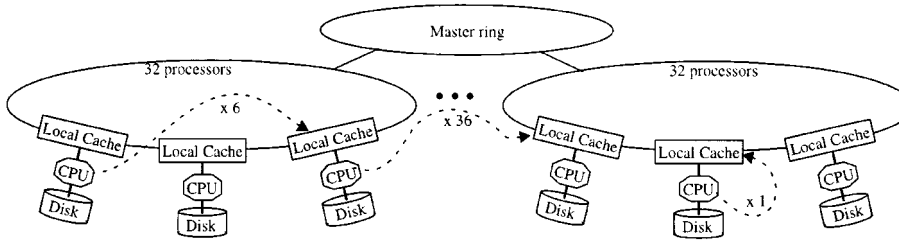

Figure 7. KSR1 architecture.

generated parallel execution plans, and present the methodology that was applied in all experiments.

6.1.1. KSR1 multiprocessor. We have implemented SP and PS on a 72-processor KSR1 computer at Inria. Each processor is 40 MIPS and has its own 32 Mbytes memory, called local cache. The KSR1 computer implements the concept of NUMA with a hardware-based shared virtual memory system, called Allcache. The time to access a data item depends on the kind of operation (read, write) and on its location (processor's subcache, local cache, remote cache).

To better understand the effect of virtual shared memory on query execution, we have performed an experiment comparing two basic operators (build and probe) implemented in two different ways: one which exercises NUMA and the other which simulates pure shared-memory. In the NUMA implementation, data accesses are done randomly in any cache (local or remote). In the shared-memory simulation, data accesses are made all local. Each processor accesses 30 MB of data.[9] For the build operator, only references to tuples are written (in the global hash table) whereas, for the probe operator, only a portion (2/3) of the tuple is accessed (to test the matching predicate). In this experiment, there is no I/O and no synchronization. The results are summarized below, as ratios of the NUMA response time versus the simulated shared-memory response time.

	16 proc.	32 proc.	64 proc.
Build operator	1.6	1.8	2.5
Probe operator	1.2	1.3	1.6

They suggest the following comments:

- Read operations, even intensive, have much less impact on performance than write operations. This is because writing implies broadcasting a cache line invalidation to all processors having a local copy.
- Remote operations have less impact than expected (the KSR technical documentation indicates a factor 6 between local and remote access). This is because memory access

represents a small portion of the operator's execution time. Taking disk accesses and synchronizations into account would further reduce this proportion.
- The relative overhead increases with the number of processors, especially for write operations. This is due to the increasing number of invalidations.

To summarize, this experiment illustrates the effectiveness of NUMA for database application and confirms the intuition that an execution model designed for shared-memory is a good choice.

As only one disk of the KSR1 was available to us, we simulated disk accesses to base relations. One disk per processor was simulated with the following typical parameters:

Parameter	Value
Rotation time [33]	17 ms
Seek time	5 ms
Transfer rate	6 MB/s
Asynchronous I/O init.	5000 inst.
Page size	8 KB
I/O cache size	8 pages

6.1.2. 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 [41] with three kinds of relations: small (10K–20K tuples), medium (100K–200K tuples) and large (1M–2M tuples). First, an acyclic predicate connection graph for the query is randomly generated. 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 $[0.5 * \min(|R|, |S|)/|R \times S|, 1.5 * \max(|R|, |S|)/|R \times S|]$.

The result of query generation is an acyclic connected graph adorned with relation cardinalities and edge selectivities. We have generated 20 queries, each involving 7 relations. Each query is then run through our DBS3 query optimizer [27] to select the best right-deep tree.

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 20 and 35 min. Thus, we have produced 20 parallel execution plans involving about 0.7 GB of base relations and about 1.3 GB of intermediate results. Note that executions with too small relations (in particular, internal ones which need to be materialized) would not allow to appreciate the effects of NUMA.

In the following experiments, each point in a graph will be obtained from a computation based on the response times of 20 parallel execution plans. 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 speed-up experiment, let the speed-up 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 speed-ups of all plans. To obtain precise measurements, each response time is computed as the average of five successive measurements.

6.2. Performance comparisons with no skew

This experiment was done to study the overhead of PS over SP when there is no skew. SP could be easily implemented by changing PS. The only modification we did was to replace the call to the function that store tuples in buffers (and then buffers in queues) by a procedure call to the next operator. Therefore, SP and PS are identical in terms of processing the trigger activations and performing (asynchronous) I/Os. The use of asynchronous I/O instead of multiplexing processors between I/O threads and CPU threads gives a small performance enhancement, because there are less synchronization and fewer system calls.

For PS, the size of the activation buffers is one page (8 KB) and the queue size is fixed to 20 buffer references. Also, a buffer size of 200 KB is allocated per processor. For instance, with 32 processors, a maximum of 6.4 MB of activations can be materialized. These values were experimentally defined and are directly related to the quality of load balancing which can be achieved.

Figure 8(a) shows the average speed-up of all query executions for SP and PS while figure 8(b) the relative performance of PS and SP, i.e., the average of the ratios of PS versus SP response times.[10] We can make two important observations.

First, SP and PS show near-linear speed-up. We may remark that the effects of the nonuniform memory access are also visible with one processor. For instance, with a single processor executing the entire query, the memory of other processors would also be used, because the outer relations do not fit in one processor's local cache. Furthermore, when more processors access the same large amount of data, the overhead implied by NUMA gets parallelized, i.e., the overhead of cache misses are shared between threads. This explains the very good speed-up which we obtain.

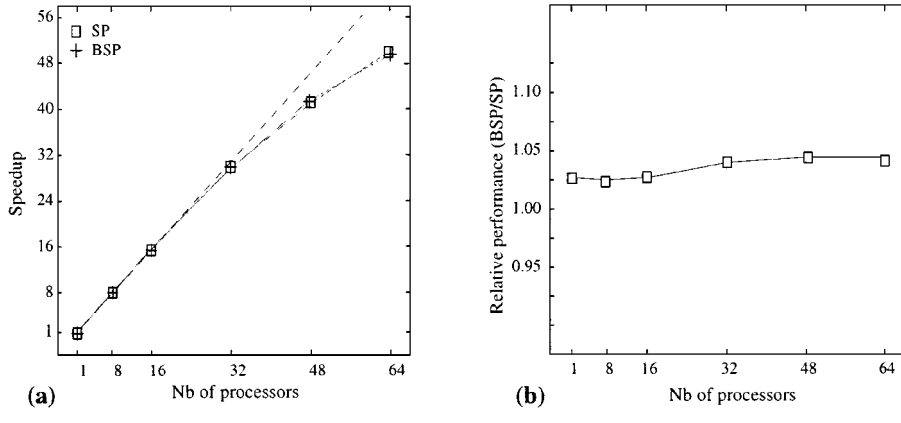

Figure 8. (a) Speed-up of SP and PS, (b) relative performance of PS versus SP.

Second, the performance of PS is very close to that of SP (the major difference does not exceed 4%) because the design of PS minimizes the overhead of materializing and sharing tuples. Without skew, execution proceeds in normal mode for almost all processing time (the degraded mode is activated at the end of the execution, but has no significant effect). Thus, there is no synchronization and no interference, even with a high number of threads. Furthermore, SP has no locality of reference [42] because it is always switching from one operator to another. As PS uses buffered input and outputs, it exhibits more locality of reference than SP.

6.3. Impact of data skew

In this experiment, we study the impact of data skew on the performance of SP and PS. We introduce skew in only one operator, at the rth rank of the pipeline chain. One tuple (or one batch for the first operator) of this operator produces k% of that operator's result. The other operators have no skew, so the production of result tuples is uniform. Such modeling of data skew is called *scalar skew* [13, 36, 48]. We chose to have k between 0 and 6%. This is reasonable and rather optimistic since we consider that only one operator of the pipeline chain has skew.

Figures 9(a) and (b) show the speed-up obtained for SP and PS for different skew factors. The skewed operator is placed at the beginning or in the middle (rank 4) of the pipeline chain. It is obvious that SP suffers much from data skew. With a skew factor of 4% on the first operator, the speed-up is only 22, which means a performance degradation of more than 200% compared with no skew. The theoretical speed-up (as computed in Section 2) is never reached. This is because the skewed tuple can be consumed at any time during the execution. A final remark on SP is that, when the number of processors is less than 10, the effect of skew is much reduced. This confirms the value of SP for small shared-memory multiprocessors.

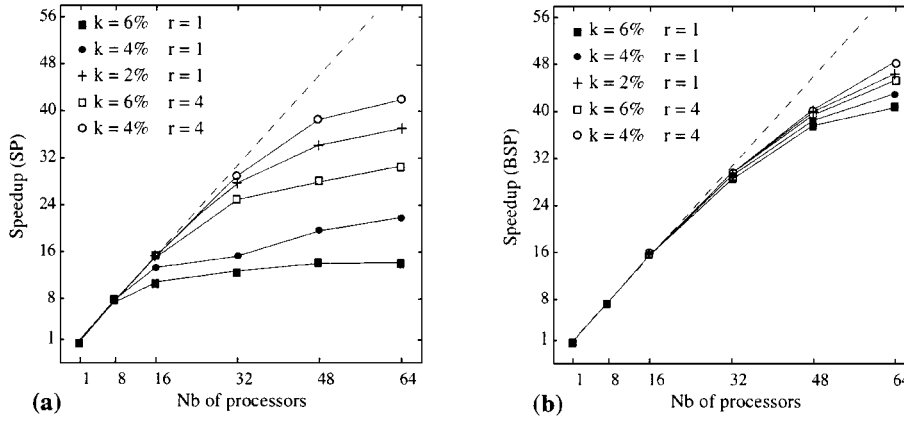

Figure 9. (a) Speed-up of SP with skew, (b) speed-up of PS with skew.

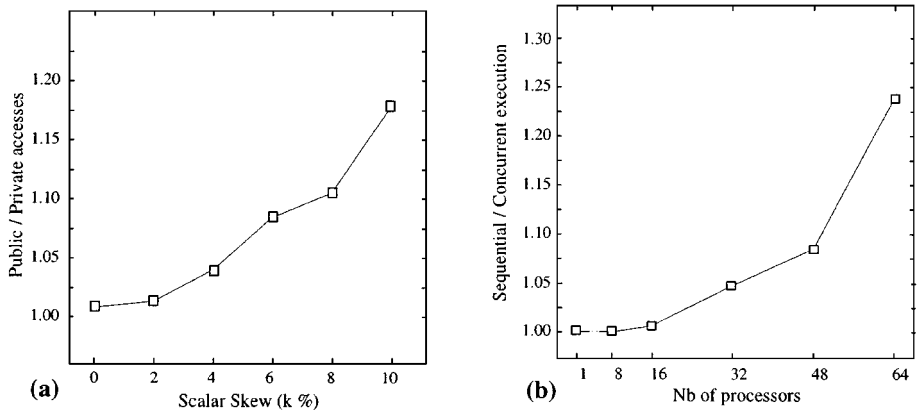

Figure 10. (a) Private versus public access, (b) gain of concurrent executions.

The impact of skew on PS is insignificant. Depending on the skew factor, the degraded mode starts sooner or later, producing thread interferences. However, thread interference is minimized by doing progressive load sharing. For instance, when the queues of operators 1 and 2 become public, the other operators (3–7) keep being processed without synchronization. Figure 10(a) shows the variation of the ratio of locked access to queues versus unlocked one with the skew (with 32 processors). We can observe that this ratio stays low (less than 10% with $k = 6\%$) because of our progressive load sharing mechanism.

6.4. Parallel execution of several pipeline chains

As discussed in Section 4, the concurrent execution of different CPU-bound pipeline chains, each by a disjoint subset of processors, seems better in NUMA because it reduces interference and remote data access, thereby yielding better speed-up.

In this experiment, we use a bushy tree containing three CPU-bound pipeline chains that can be executed concurrently, consisting of 3, 2 and 4 joins. We only consider the execution of these pipeline chains, ignoring the other operators of the bushy tree. We measured the response time of sequential versus concurrent execution of the three pipeline chains. For the concurrent execution, each pipeline chain is executed on a number of processors computed based on a simple estimate of their response time. This is achieved using a very simple cost model to evaluate the complexity of each pipeline chain and making a ratio of the number of processors (as in [22]).

Figure 10(b) shows the gain presented as the ratio of the response time of sequential versus concurrent execution, as a function of the number of processors. With less than three processors, some processors may execute several pipeline chains, thus in a sequential fashion. The gain is steadily increasing and gets significant (24%) with a high number of processors. As the number of processors increases, the speed-up obtained by a sequential execution worsens whereas it gets better for a concurrent execution. This confirms the benefit of executing several pipeline chains concurrently for large NUMA configurations.

7. Conclusion

In this paper, we have addressed the problem of parallel execution and load balancing of complex queries in NUMA. We started by arguing the relevance of execution models designed for shared-nothing or shared-memory. We first concluded that an execution model for NUMA multiprocessors should not use data partitioning (as shared-nothing systems do) because the overhead of data redistribution would be exacerbated when writing in remote memories. Instead, it should strive to exploit the efficient shared-memory model. Second, we have argued that the Synchronous Pipelining (SP) strategy which is excellent in shared-memory could be used in NUMA. Using an analytical model, we showed that, as a consequence of the scalability feature of NUMA (high number of processors and large memory), SP does not resist two forms of skew which we identified.

Thus, we have proposed a new execution model called PS, which is robust to data skew. The major difference with SP is to allow partial materialization of intermediate results, made progressively public in order to be processed by any processor which would otherwise be idle. This yields excellent load balancing, even with data skew. In addition, PS can exploit the concurrent execution of several pipeline chains.

To validate PS and study its performance, we have conducted a performance comparison using an implementation of SP and PS on a 72-processor KSR1 (NUMA) computer, with many queries and large relations. With no skew, SP and PS have both near-linear speed-up. However, the impact of skew is very severe on SP performance while it is insignificant on PS. For instance, with a skew factor of 4%, the performance degradation of SP is 200% while it is only 15% with PS. Finally, we have shown that executing several pipeline chains concurrently yields significant performance gains with a high number of processors.

To summarize, the experiments have shown that PS outperforms SP as soon as there is skew and can scale up very well. Although if the measurements have been conducted on the KSR1 (Cache Only Memory Architecture), we can expect the same behavior on Cache Coherent NUMA,[11] ho and pure shared-memory multiprocessors, since no specific feature of the KSR1 was used in our PS strategy. Thus, PS should be considered a strategy of choice for implementing database systems on both shared-memory and NUMA multiprocessors.

Acknowledgments

The authors wish to thank C. Mohan, B. Dageville and J.R. Gruser for many fruitful discussions on parallel execution models and Jean-Paul Chieze for helping us with the KSR1.

Notes

1. The goal of our analysis is not to compare different classes of NUMA (see [44, 34] for a discussion), but to propose an effective execution strategy for these architectures.
2. Any algorithm that allows pipeline execution, for example, index join, nested-loop join, pipelined hash join [49] could be used. However, some algorithms can be inefficient in NUMA because of data reorganizations which may incur intensive remote data reading or writing. Studying this issue is beyond the scope of this paper.

3. The performance of the classical partitioned execution model (called Fixed Processing (FP) in [7]) were always worse than DP.
4. Considering a large main memory.
5. Of course, the interferences inherent in SP (I/O, page management, etc.) cannot be avoided.
6. We have tried other appealing heuristics and we obtained similar performance.
7. We use a global variable instead of operating system signals, but the two approaches seem equivalent.
8. The optimization of I/O-bound tasks cannot be done at runtime because it essentially depends on data partitioning on disks.
9. With more than 32 MB of data, a local execution could not be obtained.
10. Speedup measurement is not sufficient since it does not show relative performance.
11. Considering CC-NUMA architecture, however, raises some new issues (e.g., initial configuration).

References

1. A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT alewife machine: Architecture and performance," Int. Symp. on Computer Architecture, June 1995.
2. P.M.G. Apers, C.A. van den Berg, J. Flokstra, P.W.P.J. Grefen, M.L. Kersten, and A.N. Wilschut, "PRISMA/DB: A parallel main memory relational DBMS," IEEE Trans. Knowledge and Data Engineering, vol. 4, no. 6, December 1992.
3. A. Bhide, "An analysis of three transaction processing architectures," Int. Conf on VLDB, Los Angeles, August 1988.
4. M. Blasgen, J. Gray, M. Mitoma, and T. Price, "The convoy phenomenon," Operating Systems Review, vol. 13, no. 2, April 1979.
5. 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, vol. 2, no. 1, March 1990.
6. L. Bouganim, B. Dageville, and P. Valduriez, "Adaptative parallel query execution in DBS3," Industrial Paper, Int. Conf. on EDBT Avignon, March 1996.
7. L. Bouganim, D. Florescu, and P. Valduriez, "Dynamic load balancing in hierarchical parallel database systems," Int. Conf. on VLDB, Bombay, September 1996. Can be retrieved at http://rodin.inria.fr/personnes/luc.bouganim/papers/VLDB.html
8. Data General Corporation, "Data general and oracle to optimize oracle universal server for ccNUMA system," can be retrieved at http://www.dg.com/news/press_releases/11_4_96.html
9. Data General Corporation, "The NUMA invasion," can be retrieved at http://www.dg.com/newdocs1/ccnuma/iw1_6_97.html
10. Data General Corporation, "Standard high volume servers: The new building block," can be retrieved at http://www.dg.com/newdocs1/ccnuma/index.html#a
11. D.J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen, "The gamma database machine project," IEEE Trans. on Knowledge and Data Engineering, vol. 2, no. 1, March 1990.
12. D.J. DeWitt and J. Gray, "Parallel database systems: The future of high performance database processing," Communications of the ACM, vol. 35, no. 6, June 1992.
13. D.J. DeWitt, J.F. Naughton, D.A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," Int. Conf. on VLDB, Vancouver, August 1992.
14. S. Frank, H. Burkhardt, and J. Rothnie, "The KSR1: Bridging the gap between shared-memory and MPPs," Compcon'93, San Francisco, February 1993.
15. M.N. Garofalakis and Y.E. Yoannidis, "Multi-dimensional resource scheduling for parallel queries," ACM-SIGMOD Int. Conf., Montreal, June 1996.
16. J.R. Goodman and P.J. Woest, "The Wisconsin multicube: A new large-scale cache-coherent multiprocessor," University of Wisconsin-Madison, TR 766, April 1988.
17. G. Graefe, "Volcano: An extensible and parallel dataflow query evaluation system," IEEE Trans. on Knowledge and Data Engineering, vol. 6, no. 1, February 1994.

18. E. Hagersten, E. Landin, and S. Haridi, "Ddm—A cache-only memory architecture," IEEE Computer, vol. 25, no. 9, September 1992.
19. W. Hasan and R. Motwani, "Optimization algorithms for exploiting the parallel communication tradeoff in pipelined parallelism," Int. Conf on VLDB, Santiago, 1994.
20. Y. Hirano, T. Satoh, A.U. Inoue, and K. Teranaka, "Load balancing algorithms for parallel database processing on shared memory multiprocessors," Int. Conf. on Parallel and Distributed Information Systems, Miami Beach, December 1991.
21. W. Hong, "Exploiting inter-operation parallelism in XPRS," ACM-SIGMOD Int. Conf., San Diego, June 1992.
22. H. Hsiao, M.S. Chen, and P.S. Yu, "On parallel execution of multiple pipelined hash joins," ACM-SIGMOD Int. Conf., Minneapolis, May 1994.
23. IEEE Computer Society, "IEEE standard for scalable coherent interface (SCI)," IEEE Std 1596, New York, August 1992.
24. Intel Corporation, "Standard high volume servers: Changing the rules for buiseness computing," can be retrieved at http://www.intel.com/procs/servers/feature/shv/
25. M. Kitsuregawa and 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.
26. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The Stanford FLASH multiprocessor," Int. Symp. on Computer Architecture, April 1994.
27. R. Lanzelotte, P. Valduriez, and M. Zait, "On the effectiveness of optimization search strategies for parallel execution spaces," Int. Conf. on VLDB, Dublin, August 1993.
28. D. Lenoski, J. Laudon, K. Gharachorloo, W.D. Weber, A. Gupta, J. Henessy, M. Horowitz, and M.S. Lam, "The Stanford dash multiprocessor," IEEE Computer, vol. 25, no. 3, March 1992.
29. D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The DASH prototype: Logic overhead and performance," IEEE Transactions of Parallel and Distributed Systems, vol. 4, no. 1, January 1993.
30. M.L. Lo, M-S. Chen, C.V. Ravishankar, and P.S. Yu, "On optimal processor allocation to support pipelined hash joins," ACM-SIGMOD Int. Conf., Washington, May 1993.
31. T. Lovett and R. Clapp, "STiNG: A CC-NUMA computer system for the commercial marketplace," Int. Symp. on Computer Architecture, May 1996.
32. H. Lu, M.-C. Shan, and K.-L. Tan, "Optimization of multi-way join queries for parallel execution," Int. Conf. on VLDB, Barcelona, September 1991.
33. M. Metha and D. DeWitt, "Managing intra-operator parallelism in parallel database systems," Int. Conf. on VLDB, Zurich, September 1995.
34. C. Morin, A. Gefflaut, M. Banâtre, and A.M. Kermarrec, "COMA: An opportunity for building fault-tolerant scalable shared memory multiprocessors," Int. Symp. on Computer Architectures, 1996.
35. M.C. Murphy and M.-C. Shan, "Execution plan balancing," IEEE Int. Conf. on Data Engineering, Kobe, April 1991.
36. E. Omiecinski, "Performance analysis of a load balancing hash-join algorithm for a shared-memory multiprocessor," Int. Conf on VLDB, Barcelona, September 1991.
37. H. Pirahesh, C. Mohan, J. Cheng, T.S. Liu, and P. Selinger, "Parallelism in relational database systems: Architectural issues and design approaches," Int. Symp. on Databases in Parallel and Distributed Systems, Dublin, July 1990.
38. E. Rahm and R. Marek, "Dynamic multi-resource load balancing in parallel database systems," Int. Conf. on VLDB, Zurich, Switzerland, September 1993.
39. D. Schneider and D. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," ACM-SIGMOD Int. Conf., Portland, May-June 1989.
40. A. Shatdal and J.F. Naughton, "Using shared virtual memory for parallel join processing," ACM-SIGMOD Int. Conf., Washington, May 1993.
41. E.J. Shekita and H.C. Young, "Multi-join optimization for symmetric multiprocessor," Int. Conf. on VLDB, Dublin, August 1993.
42. A.J. Smith, "Cache memories," ACM Computing Surveys, vol. 14, no. 3, September 1982.

43. J. Srivastava and G. Elsesser, “Optimizing multi-join queries in parallel relational databases,” Int. Conf. on Parallel and Distributed Information Systems, San Diego, January 1993.
44. P. Stenstrom, T. Joe, and A. Gupta, “Comparative performance evaluation of cache-coherent NUMA and COMA architectures,” Int. Symp. on Computer Architecture, May 1992.
45. P. Valduriez, “Parallel database systems: Open problems and new issues,” Int. Journal on Distributed and Parallel Databases, vol. 1, no. 2, 1993.
46. P. Valduriez and G. Gardarin, “Join and semi-join algorithms for a multiprocessor database machine,” ACM Trans. on Database Systems, vol. 9, no. 1, March 1984.
47. C.A. van den Berg and M.L. Kersten, “Analysis of a dynamic query optimization technique for multi-join queries,” Int. Conf. on Information and Knowledge Engineering, Washington, 1992.
48. C.B. Walton, A.G. Dale, and R.M. Jenevin, “A taxonomy and performance model of data skew effects in parallel joins,” Int. Conf. on VLDB, Barcelona, September 1991.
49. A.N. Wilshut, J. Flokstra, and P.G. Apers, “Parallel evaluation of multi-join queries,” ACM-SIGMOD Int. Conf., San Jose, 1995.