# Parallel Evaluation of Multi-join Queries *

Annita N. Wilschut and Jan Flokstra and Peter M.G. Apers

University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
Phone: +3153-894190
{annita, flokstra, apers}@cs.utwente.nl

**Abstract.** A number of execution strategies for parallel evaluation of multi-join queries have been proposed in the literature. In this paper we give a comparative performance evaluation of four execution strategies by implementing all of them on the same parallel database system, PRISMA/DB. Experiments have been done up to 80 processors. These strategies, coming from the literature, are named: *Sequential Parallel, Synchronous Execution, Segmented Right-Deep,* and *Full Parallel.* Based on the experiments clear guidelines are given when to use which strategy.

## 1 Introduction

For years now, research has been done on the design, implementation, and performance of parallel DBMSs. Teradata [CaK92], Bubba [BAC90], HC186-16 [BrG89], GAMMA [DGS90], and XPRS [SKP88] are examples of systems that actually were implemented. The performance evaluation of these systems is mainly limited to simple queries.

Recent developments in the direction of support of non-standard applications, the use of complex data models, and the availability of high-level interfaces tend to generate complex queries that may contain larger numbers of joins between relations. A number of parallelization strategies was proposed [CLY92,CYW92,HoS91,HCY94,ScD90] and their performance was evaluated via simulation. However, no comparative experimental performance evaluation is available. This paper describes the proposed strategies in a common framework. Four strategies are implemented on PRISMA/DB and a comparative performance evaluation is done. The results yield clear guidelines for the choice of a strategy.

### 1.1 Implementation Platform

PRISMA/DB was used to do the experiments. PRISMA/DB is full-fledged parallel, relational DBMS [ABF92]. A fully functional prototype is running on a 100-node multiprocessor machine. PRISMA/DB is used for research in various directions [Gre92, HWF93,Wil93,WiA91,WFA92]. PRISMA/DB is a main-memory DBMS and therefore the experiments described in this paper refer to a main-memory context.

* This is an extended abstract; the full paper appeared in Proc. ACM SIGMOD'94, Minneapolis, Minnesota, May 24–27, 1994

### 1.2 Optimization and Parallelization of Multi-join Queries

In System R [SAC79], join trees are restricted to linear trees, so that available access structures for the inner join operand can optimally be exploited. System R chooses the cheapest (in the sense of minimal total costs) linear tree that does not contain cartesian products.

Subsequently, it is remarked in [KBZ86] that the restriction to linear trees may not be a good choice for parallel systems. However, the space of possible join trees is very large if restriction to linear trees is dropped [LVZ93].

Obviously, when optimizing the response time of a complex query, it is not sufficient to optimize towards minimal total costs. Rather, the exploitation of parallelism has to be taken into account as well. However, the search space that results if all possible trees and all possible parallelizations for these trees are taken into account is gigantic. To overcome these problems, [HoS91] proposes a two-phase optimization strategy for multi-join queries. The first phase chooses the tree that has the lowest total execution costs and the second phase finds a suitable parallelization for this tree. The same strategy is used here.

### 1.3 Organization of Paper

This paper is organized as follows. Section 2 shortly introduces PRISMA/DB, it shows how the different parallelization strategies for the execution of multijoins can be implemented on PRISMA/DB, and it discusses some results from earlier research in the context of PRISMA/DB that are used to explain the results of this paper. Section 3 describes four execution strategies for multi-join queries and their trade-offs in detail. Section 4 describes a comparative performance evaluation and Section 5 summarizes and discusses the results of this paper.

## 2 PRISMA/ DB

PRISMA/DB has extensively been used for research in the area of parallel query processing [Wil93,ApW94]. Our previous research followed two lines. First, the system was used to experiment with large-scale intra-operation parallelism for single operation queries [WFA92]. Second, a theoretical study of the behavior of pure inter-operation parallelism in multi-join queries was done [WiA93]. The work presented in this paper combines those two lines of research: we study the use of both *inter-* and *intra-*join parallelism for the execution of multi-join queries via experimentation.

### 2.1 The System

PRISMA/ DB is a full-fledged parallel, main-memory relational DBMS, designed and implemented in the Netherlands. A goal of the PRISMA project was to provide flexibility in architecture and query execution strategy, to enable experiments with the functionality and performance of the system. This flexibility is used here to implement various strategies for the parallel evaluation of multi-join queries and to evaluate their performance. PRISMA/ DB currently run on a 100-node shared-nothing multi-processor.

Each node consists of a 68020 processor with 16 Mbytes of memory, a disk, and a communication processor. A full description of design, architecture, and implementation of PRISMA/ DB can be found in [Ame91,ABF92].

## 2.2 Results from Previous Research

**Parallel Execution of Single Operator Queries** [WFA92] studies the use of intra-operator parallelism for main-memory database systems. In that study, it is concluded that observed linear speedup for small numbers of processors cannot always be extrapolated to larger numbers of processors. This is caused by the fact that the overhead from starting on operations on processors —this overhead increases with increasing degree of parallelism— dominates the actual processing time —which decreases with increasing degree of parallelism— for a large degree of parallelism. The optimal number of processors to be used appears to be proportional to the square root of the size of the operands. As a consequence, larger problems allow a larger degree of parallelism. Also, it is concluded that the optimal number of processors for the parallel execution of an operation is smaller for a main-memory system than for a disk-based system.

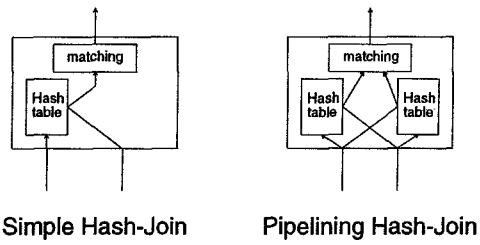


**Fig. 1.** Simple hash-join and Pipelining hash-join algorithm in a main-memory system

**The Pipelining Hash-join Algorithm** In [WiA91,WiA93] it is shown how special main-memory algorithms can be used that enhance the effective parallelism from pipelining. These pipelining algorithms aim at producing output as early as possible, so that a consumer of the result can start its operation. In particular, [WiA91,WiA90] proposes a pipelining Hash-Join algorithm. As opposed to the well-known two-phase, build-probe hash-join [ScD89,WiA91] (this algorithm is called simple hash-join in this paper), this symmetric algorithm builds a hash-table for both operands (See Figure 1). The join process consists of only one phase. As a tuple comes in, it is first hashed and used to probe that part of the hash table of the other operand that has already been constructed. If a match is found, a result tuple is formed and sent to the consumer operation. Finally, the tuple is inserted in the hash table of its own operand. Compared to the simple hash-join, the pipelining algorithm can produce result tuples earlier during the join process at the cost of using more memory to store a second hash-table. Using this algorithm, pipelining along both operands of the join is possible.

# 3 Parallel Execution Strategies for Multi-joins

The parallel execution strategies for multi-join queries that are dealt with in this paper all use known parallel algorithms to evaluate the constituent binary join operations. The difference between the various strategies lies in the way in which binary joins are allocated to processors. A lot of work was done on the use of intra-operator parallelism for the evaluation of binary join operations. It is generally agreed on that the parallel hash-join is the algorithm of choice [ScD89]. Two version of this algorithm are considered here: the simple hash-join and the pipelining hash-join (see Section 2.2).

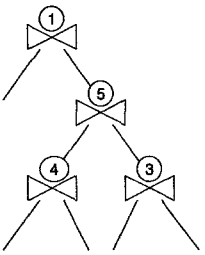


**Fig. 2.** The 5-way join tree that is used to explain the parallel multi-join strategies in this paper.

A parallel execution strategy for a multi-join query uses a parallel hash-join algorithm for the constituent binary joins. Apart from this *intra-operator parallelism*, also inter-operator pipelining or parallelism may or may not be used. The four strategies that are regarded here differ in the way in which *inter-operator parallelism* and *intra-operator parallelism* are used. Note, that we concentrate on adding inter-operator parallelism. This means that the available processors may have to be distributed over the operations in the join-tree. We do not allow a single processor to work concurrently on different join operations.

In the following, each of the strategies is described in detail. The 5-way join tree in Figure 2 is used as an example. The constituent joins in this tree are labeled with a number, which indicates the relative amounts of work in the join operations. So, the second join operation from the top needs five times the computation time of the top join operation.

## 3.1 Sequential Parallel Execution (SP)

The sequential parallel execution strategy does not use any inter-operator parallelism. The constituent joins are executed sequentially in parallel, using all available processors for each join operation. This strategy does not require pipelining between join operations, so the simple hash-join algorithm can be used.

The processors first work together on the join labeled with 4, then they work on the join labeled with 3 etc. This strategy does not need a cost function to estimate the costs of the join operation. Also, the idealized load balancing is perfect.

### 3.2 Synchronous Execution (SE)

This strategy uses inter-operator parallelism apart from intra-operator parallelism. The strategy was proposed in [CYW92]. The idea is to execute independent subtrees in the join tree independently in parallel. A join operation is started only after its operands are ready. The only inter-operation parallelism that is used in a join tree is the parallelism between independent subtrees of a bushy tree. An algorithm is proposed in [CYW92] that aims at equal processing time for both operands to be ready for joining. This is done by allocating a number of processors to a subtree that produces an operand, that is proportional to the total amount of work in the subtree. In this way, operands are supposed to be available at the same time so that no processors have to wait. This strategy does not require pipelining between join operations, so the simple hash-join algorithm is used.

The allocation algorithm needs a cost-function to estimate the processing costs for subtrees in the join tree. No perfect load balancing is achieved due to discretization errors (see Section 3.5) in the allocation of differently sized loads to a small number of processors.

### 3.3 Segmented Right-Deep Execution (RD)

In contrary to SE, segmented right-deep execution uses inter-operator pipelining in addition to intra-operator parallelism. This strategy is proposed in [CLY92], a paper which was inspired by [ScD90].

Schneider [Sch90,ScD90] describes the differences in possible parallelism between left-deep and right-deep linear join trees**, when the simple hash-join is used for the individual join operations. In a right-deep tree the build-phases of all join operations can be executed in parallel and after that probe-phases can be executed using extensive pipelining. Left-deep trees on the other hand only allow parallel execution of the probe phase of one join-operation and the build-phase of the next. It is concluded in this study that, due to the possibilities of extensive exploitation of pipelining right-deep trees perform better than left-deep trees.

The results of Schneider are extended in [CLY92] to bushy trees. That paper proposes to see a bushy tree as a segmented right-deep tree, which is a bushy tree that consists of right-deep segments. The right-deep segments can be evaluated using inter-operation parallelism as proposed in [Sch90,ScD90]. Each operation in a segment is assigned a number of processors that is proportional to the estimated amount of work in the join operation. Segments that have a producer-consumer relationship are evaluated sequentially. Independent segments, however, may be evaluated in parallel, using disjoint subsets of the available processors. In this approach, a left-deep tree is a bushy tree consisting of many small right-deep segments.

This strategy first uses all available processors to process the right-deep subtree that consists just of the join labeled with 4. Subsequently, the available processors are distributed over the other join operations, which also form a right-deep subtree. This last

** In this terminology the inner join-operand, which is used to build a hash-table, is called the left operand, and the outer join-operand, which is used to probe the hash-table is called the right operand.

subtree is executed in a pipelined fashion. Each of the join operations starts immediately hashing its left operand. However, during the probe-phase the join labeled with 3 (which has relatively few processors) cannot saturate the joins that are higher up in the pipeline so those operations cannot fully utilize their processor during the probe phase.

Again, this strategy needs a cost function to estimate the amount of work in each join operation. This strategy also does not yield perfect load balancing due to discretization errors in the allocation of work to a small number of processors and due to delays over the pipeline.

### 3.4 Full Parallel Execution (FP)

This strategy adds both inter-operator pipelining and inter-operator parallelism to intra-operator parallelism in the individual join-operations. The strategy was proposed in [WiA91,WAF91]. The idea behind this strategy is to allocate each join-operation to a private (set of) processors, so that all join-operations in the schedule are executed in parallel. Depending on the shape of the query tree, pipelining and independent parallelism are exploited. The strategy uses the pipelining hash-join algorithm (see Section 2.2). Because, this algorithm can exploit pipelining along both the right and the left operand, all individual join-operations can be executed in parallel. The available processors are distributed over all join-operations proportionally to the amount of work in each operation. Each join-operation starts working as soon as input is available.

The bottom two join operations start immediately on the processors allocated to them, as their operands are available as base-relations. The join operation labeled with 5 has to wait some time until its operands start producing output (see Section 3.5). The top join operation may start immediately hashing its left-operand. However, it has to wait for its right operand to become available, and therefore its processor is not fully utilized later during the join operation.

Again, this strategy depends on a cost function to estimate the amount of work in each join operation. It is clear that this strategy does not offer perfect load balancing either.

### 3.5 Tradeoffs

There are a number of barriers that prevent performance gain from parallelism. A general discussion of this issue can be found in [DeG92]. These barriers affect the execution strategies introduced above in a different way, resulting in a number of tradeoffs: startup, coordination, descretization error, and delay over pipelines.

Obviously, each of these four factors affects the execution strategies studied in a different way. Also, it is expected that the extent to which a strategy is affected by each of the factors depends on the shape of the query tree that is parallelized. For example, RD is expected to work fine for right-oriented trees, but not so well for e.g. a left-linear tree. Similarly, SE is expected to work better for bushy trees than for trees that are (almost) linear. SP, on the other hand, is not expected to be very sensitive to the shape of the query tree. Experiments are used to find out how these tradeoffs work out in reality.

# 4 Performance Evaluation

As stated in the introduction of this paper, we study the second phase of a two-phase optimization/parallelization strategy. The first phase, finds the join tree with minimal total costs for a given multi-join and the second phase generates a parallel execution strategy for this plan. To keep the problem manageable we decided to study one multi-join query. For this join query, we vary the *parallelization strategy*, the *number of processors used*, the *shape of the query tree*, and the *size of the problem.*

## 4.1 Test Data and Query

The join query studied in this performance evaluation consists of ten relations that contain equal numbers of Wisconsin tuples [BDT83]. These tuples consist of two unique integer attributes and a number of other attributes up to a total size of 208 bytes per tuple. The ten relations are joined one-by-one on their first integer attributes, and after each join they are projected to the second integer attributes and the remaining attributes of one of the operands, so that the result of each operation again is a Wisconsin relation equal in size to the operands. This test problem is similar to the problem used in [Sch90], in [ZZS93], and in [WiA93]. All possible join trees for this query have the same total execution costs. Also, the individual join operations are equal in costs and sizes of its operands. So, any differences in response time are caused by differences in the shape of the tree and the parallelization used. Therefore, such a regular tree is very suitable to study the effectiveness of the various parallelization strategies.

## 4.2 Experimental Setup

As said before, in our experiments, we vary the *parallelization strategy*, the *number of processors used*, the *shape* of the query tree, and the *size of the problem.* Each of three parameters is varied in the experiments. The following parameter values are chosen.

Four parallelization strategies used: SP, SE, RD, and FP. These strategies have been described above. Two problem sizes are used: the small experiment uses relations consisting of 5000 tuples each, so a total of 50000 tuples were involved in this query. The large experiment uses relations consisting of 40000 tuples each amounting to a total of 400.000 tuples in the query. These sizes will be referred to as the 5K and 40K experiments. For the 5K experiment, the number of processors used is varied from 20 to 80; for the 40K experiment we use 30 to 80 processors. The total size of the 40K query was too large to run on fewer than 30 processors. Finally, as explained in Section 3.5, we expect the strategies to perform differently for different query shapes. We are especially interested in the difference between (almost) linear and bushy trees, and in the difference between left and right-oriented trees. Therefore, the following 5 query shapes are used for this query: a right-linear, a right-oriented long bushy, a wide bushy, a left-oriented long bushy, and a left-linear tree.

## 4.3 Results

**Left Linear Join Tree** As a linear tree does not have any independent subtrees, SE allocates all available processors sequentially to each join. In this way, SE degenerates

to SP for linear trees. Also, a left linear tree does not show any right-deep segments, and therefore RD allocates all available processors sequentially to each join operation. So, RD also degenerates to SP. The experiments indeed show coinciding performance for SP, SE, and RD, both for the 5K and for the 40K experiment.

Also, it is clear that SP (and for this case also SE and RD) works reasonable for small numbers of processors, but its performance degenerates for larger numbers of processors. The 5K experiment shows this effect stronger than the 40K experiment. This performance degradation is explained by the startup costs and coordination overhead. SP needs to start one operation process for each join on each processor. So, for the 80 processor case, 800 operation processes need to be initialized. Also, the coordination overhead for redistribution of operands may be large. The 5K experiment shows a more extensive performance degradation than the 40K experiment. This result corresponds to performance results for single operation queries (see Section 2.2).

FP execution of this query tree does show performance gain from parallelism. However, for the 40K experiment, its performance for a low degree of parallelism is not as good as SP.

**Left-oriented Bushy Join Tree** The results for SP are similar to the results for the left linear tree. This fits with the expectation that SP is not very sensitive to the shape of the query tree.

The results show that SE and RD work much better than for the left linear case, but not as well FP (at least not for higher numbers of processors). The shape of this query is not very suitable for either RD or SE. RD profits from independent right-deep segments, which are very short for this tree. SE profits from independent subtrees, and those are very small. As a result, there is not much room for inter-join parallelism for RD and SE. This explains why the performance of both RD and SE for this tree is in between SP and FP.

The behavior of FP is similar to its behavior for the linear tree, but a close inspection of the data shows that its performance for small numbers of processors is slightly worse than for the linear tree.

**Wide Bushy Join Tree** This query tree is very suitable for SE, because the tree is very wide resulting in nice independent subtrees. The results indeed show a good performance for SE. For the large experiment SE wins; for the small experiment SE is almost as good as FP.

FP performs well for the small experiment. This is caused by the fact that the operands are small, so FP does not suffer too much from delay over the pipeline. For a large number of operands, SE uses more operation processes than FP, so that the startup and coordination overhead dominates.

Like in the previous case FP suffers from pipeline delay for a small number of processors. This results in bad performance for a small number of processors and large operands, as explained for the previous case. Its speedup characteristics, however, outperform those of the other strategies and the performance for a large number of processors is good.