Parallelism in a Main-Memory DBMS: The performance of PRISMA/DB[1]

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

University of Twente
P.O.Box 217, 7500 AE Enschede, the Netherlands
telephone +3153 893705, fax +3153 339605
annita@cs.utwente.nl

Abstract

This paper evaluates the performance of the parallel, main-memory DBMS, PRISMA/DB. First, an abstract architecture for parallel query execution is presented. A performance model for the execution of simple relational operations on this architecture is developed. The parameters in the model are set using experiments on PRISMA/DB and the performance of PRISMA/DB is analized in the context of the model. Several conclusions can be drawn from the model combined with the results of the performance experiments. Firstly, the performance of PRISMA/DB appears to be competitive with respect to other systems. Secondly, the developed model can explain the results from the performance experiments to a large extent. Also, it is concluded that observed linear speedup for small numbers of processors cannot always be extrapolated to larger numbers of processors. Finally, 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. The results of this study can be used to design data fragmentation strategies for large parallel machines.

1 Introduction

In the last decade, research has been done on the design, implementation, and performance analysis of parallel DBMSs. Teradata [Ter83], Bubba [BAC90], HC186-16 [BrG89], GAMMA [DGS90] are examples of systems that actually were implemented, and many papers were written on their performance. The systems mentioned above are disk-based DBMSs, and obviously their performance and parallel behavior is influenced by the use of disk as main data storage.

On the other hand, the last years yield an increasing number of papers on main-memory DBMSs [DKO84, Eic89,GLH83,LeC86,LeR87]. A main-memory DBMS stores the entire database, or a considerable part of it in its primary memory. Of course, main-memory systems have their own characteristics, for example with respect to recovery [Eic87] and query processing [DKO84]

PRISMA/DB combines the two features mentioned above: it is a parallel, main-memory system. The system offers full relational DBMS facilities, and a fully functional prototype running on a 100-node multiprocessor is currently available. PRISMA/DB is used for research in various directions like parallel integrity constraint enforcement [GrA90], recursive query processing [AHB88,HAC90,HoA92], and parallel evaluation of multi-join queries [WiA90,WiA91,WiA92]. [ABF92] describes the system in general. This paper addresses the specific aspects of the performance of a parallel main-memory system in relationship to data fragmentation strategies with PRISMA/DB as an example.

Teradata, GAMMA and HC186-16 use full declustering as data fragmentation strategy. This means that the tuples belonging to one relation of the database are distributed over all nodes in the system. Also, the degree of parallelism for the execution of one

[1] The work reported in this paper was conducted as part of the PRISMA project, a joint effort with Philips Research Laboratories Eindhoven, partially supported by the Dutch "Stimuleringsprojectteam Informaticaonderzoek (SPIN)".

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

relational operation (which is the number of processors used to evaluate the operation) is taken equal to the size of the system. Using these data fragmentation and parallel query execution strategies, linear speedup is achieved for up to 30 processors.

Bubba uses a more flexible data fragmentation strategy [CAB88], which allows relations to be distributed over fewer than the available nodes. In simulation experiments that optimize the transaction throughput for some workload, they find that full declustering, indeed, is not the right default fragmentation strategy. However, the optimal degree of declustering they find is relatively high: 736 is reported to be the right degree of declustering in an experiment described in [CAB88].

PRISMA/DB uses a flexible data fragmentation and query execution strategy that allows various degrees of declustering for base relations and various degrees of parallelism for the execution of one operation. Some queries of the Wisconsin Benchmark [BDT83] are used to evaluate the performance of the system and the goal is minimization of the response time of these queries. The important questions in this context are: "What are the speedup characteristics of the parallel execution of simple operations on a main-memory system?" and "What is the optimal degree of parallelism to be used for the parallel execution of these queries?". To facilitate the analysis, first an abstract architecture is introduced with a model for the parallel execution of operations on this architecture. The model developed in this paper is similar to the models presented in [Cve87]. The performance of PRISMA/DB is evaluated in the context of this model.

This paper is organized as follows: Section 2 introduces the abstract architecture with a query execution model, Section 3 describes PRISMA/DB, and it highlights some aspects of the system that are important in the context of this paper, Section 4 evaluates the performance of PRISMA/DB using queries from the Wisconsin Benchmark [BDT83], and finally, Section 5 summarizes and concludes the paper.

2 An abstract architecture for parallel query execution

To evaluate the behavior of the parallel execution of an operation, we present a simple abstract architecture of the query execution layer of a parallel DBMS, that can serve as an abstraction of many known parallel systems. Subsequently, a model for query execution on this architecture is presented. After that, the difference between disk-based and main-memory systems is described in the context of this model.

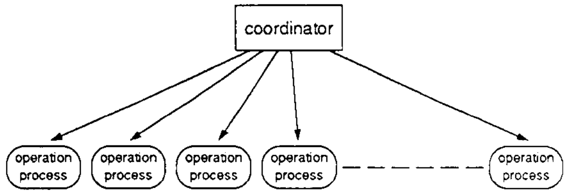


Figure 1: An abstract architecture for parallel query execution

Abstract architecture

The proposed abstract architecture is illustrated in Figure 1. The hardware support assumed consists of a shared-nothing multi-processor, with a high bandwidth communication network and sufficient nodes. The software architecture consists of one coordinator process, and a number of operation processes. Each process is assumed run on a private node. The coordinator initializes the operation processes in a *sequential* process. As a consequence, the operation processes cannot all start at exactly the same time. Rather, every next operation process starts some time after the start of its predecessor.

Query Execution Model

Consider this architecture executing a relational operation. The operand data for this operation is distributed over (part of) the system into equally sized fragments. We assume that the costs of the operation are proportional to the number of tuples in its operand(s). This assumption holds for an important class of relational operations. For example, the costs of selections and projections are linear in the number of tuples in the operands, and, the costs of other operations with a hash-based implementation, like joins and unique operations, are also almost linear in the number of tuples in their operands. Parallel implementations of such operations aim at linear speedup. This means that the parallel execution on n processors is n times faster than the execution on one processor. In the remainder of this section, we will show that this goal is only achievable for relatively low numbers of processors, which number is *lower* for main-memory systems than for disk-based systems.

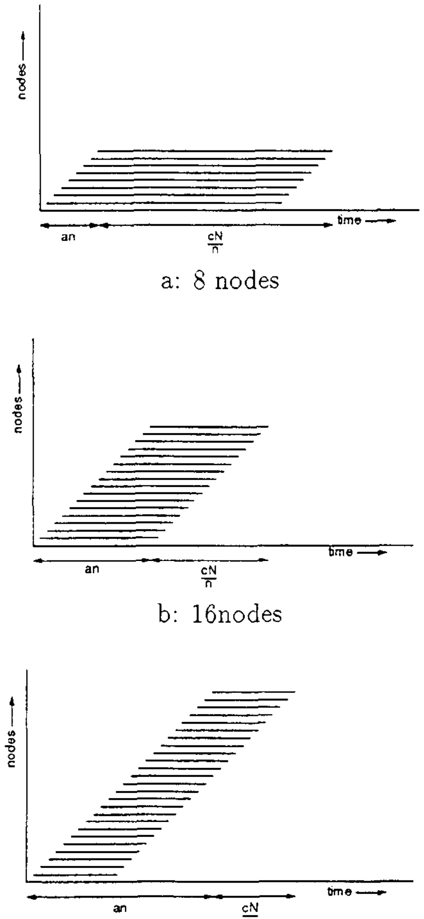
Let N be the number of operand tuples, and let c be the time needed to process one tuple. Then

$$cN$$

is the time needed for the entire operation, and, if n is the number of processors used than ideally,

$$\frac{cN}{n}$$

is the processing time of one operation process.



a: 8 nodes

b: 16nodes

c: 25 nodes

Figure 2: The behavior of the parallel execution of an operation on 8, 16, and 25 nodes

Figure 2 shows the theoretical characteristics of the parallel execution of an operation. In the diagrams in this figure, the horizontal axis is the time axis, and the node numbers are on the vertical axis. Each line in the diagrams represents one operation process; a line starts at the time at which the corresponding operation process starts, and its length is proportional the computation time of the process. The operation processes do equal amounts of work, and therefore, the lines have the same length, which is equal to cN/n. If a is the time the coordinator process needs to start one operation process, then the last operation process is initialized at time an, so it is obvious that the following equation holds for the response time (($\mathcal{R}$)) of an operation on N tuples executed on n nodes.

$$\mathcal{R} = an + \frac{cN}{n} \tag{1}$$

This is the central expression to analize the behavior of the parallel execution of an operation.

The *speedup* ($\mathcal{S}$) of the parallel execution of an operation is defined as the response time of the execution on one processor divided by the response time of the parallel execution. So, in this model, the speedup is equal to

$$\mathcal{S} = \frac{a + cN}{an + \frac{cN}{n}} = \frac{\left(\frac{a}{cN} + 1\right) n}{\frac{a}{cN} n^2 + 1}. \tag{2}$$

Note that the case when $a = 0$ corresponds to linear speedup. This means that all operation processes start at exactly the same time.

The *optimal number* of processors for an operation (n_o) is the number of processors for which the operation is executed with the smallest response time. This number can be found by setting the derivative of (1) to zero:

$$n_o = \sqrt{\frac{cN}{a}}. \tag{3}$$

Substitution of the optimal number of processors into Equation (2) yields the speedup of the optimal execution of an operation:

$$\mathcal{S}_o = \frac{1}{2}\sqrt{\frac{cN}{a}} = \frac{1}{2} n_o.$$

Note, that the optimal execution only yields half of the linear speedup aimed at. Also, the optimal number of processors depends on the parameters of the system, but given the optimal number of processors, the speedup is fixed to half that number.

Substitution of the value of n_o found in Equation (3) into both terms of the right-hand-side of equation (1) yields a relationship between the startup time and the computation time per fragment: As

$$an_o = \sqrt{acN}$$

and,

$$\frac{cN}{n_o} = \sqrt{acN}$$

it can be concluded that the optimal execution of an operation spends equal amounts of time on initialization and on the execution of one subtask. In other words, in the optimal execution, the last subtask starts at the time at which the first one is ready.

The results of this section are illustrated by Figures 2 and 3. Figure 2 shows the execution characteristics of the some abstract operation executed on 8, 16, and 25 nodes. From the diagrams it is obvious that 16 nodes is the optimal number for this operation, as the diagrams show that the 16th node starts processing at the time at which the first processor is ready. Note that the average utilization of the nodes participating in the optimal execution is 50 %. The execution on 8 nodes takes too long because the individual operation processes take too long, and 25 nodes is too many, because the coordinator takes too much time to initialize the operation processes.

From the calculations above it is clear that

$$\frac{cN}{a}$$

is the quantity that determines that behavior of the parallel execution of an operation. Equation (3) shows that the optimal number of processors is higher for higher values of cN/a. Equation 2 shows that the speedup deviates less from linear speedup for higher values of cN/a, because the contribution of the quadratic term in the denominator gets small for high values of cN/a. These results are intuitively right. Slow local evaluation (reflected in a high value for c) and large experiments (high value for N) allow a high degree of parallelism and almost linear speedup. Fast initialization of subtasks (small value for a) has the same effect.

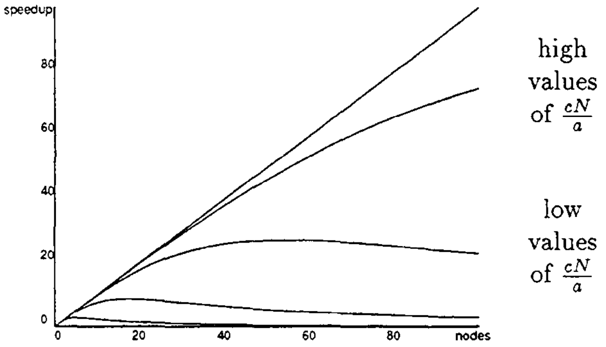


Figure 3: Theoretical speedup curves for different values of $\frac{cN}{a}$

Figure 3 shows the speedup characteristics for different values of cN/a. In this diagram, the number of nodes used for the execution is on the horizontal axis, and the speedup is on the vertical axis. The straight curve shows linear speedup; for the other curves the parameter setting is derived from the experiments described in Section 4. All speedup curves show almost linear behavior for a small number of processors. This is the linear speedup that has been reported for small systems. At a certain stage, however, they deviate from the linear speedup curve. This deviation starts being significant at a higher number of processors for larger values of cN/a.

Disk-based versus main-memory systems

An important question at this stage is: "Which are realistic values for c and a?". Obviously, if the value of c/a is relatively high, linear speedup is a reasonable assumption for the parallel execution of an operation on large operands. In that case, all processors of a relatively small system can be exploited to execute an operation. However, this model also shows that, linear speedup for a small number of processors does not guarantee scalability of this property to larger systems. Therefore, it is not clear to what extent all nodes of a system should be used to execute an operation, so that full declustering of base relations is the right fragmentation strategy.

The next sections of this paper describe the performance of a parallel, *main-memory* DBMS. A main-memory DBMS stores the entire database in its primary memory, and therefore, no disk-access is needed for retrieval queries. Obviously, this fact influences the costs of local processing: for a main-memory system, the costs of local processing are lower than for a comparable disk-based system, and therefore the value of c is also lower. The difference is estimated to be at least one order of magnitude [DKO84]. On the other hand, the value of a is determined by the costs of initialization of operation processes. This initialization mainly requires communication, and the main-memory character of a system does not affect the communication costs. Therefore, the value of a is equal for comparable main-memory and disk-based systems. As a consequence, the value of c/a, and so the optimal number of processors to be used for the execution of an operation on a main-memory system is lower than on a disk-based system. Based on the assumption that the difference in local processing costs is about one order of magnitude, the difference between the optimal number of processors for both systems is about 3 (see Equation (3)).

The remainder of this paper discusses the performance of the parallel, main-memory DBMS PRISMA/DB in the context of the model described above. PRISMA/DB is currently implemented on a 100-node multi-processor, and we will show that the optimal number of processors to be used for realistic operations may well be smaller than 100, so that full declustering should not be used as fragmentation strategy. As an introduction, the next section briefly introduces PRISMA/DB. After that, Section 4 describes the performance analysis.

3 PRISMA/DB

PRISMA/DB is a full-fledged parallel, main-memory relational DBMS that was designed and implemented from 1986 to 1991 in the Netherlands by several scientific and commercial research institutions. A full description of design, architecture, and implementation of PRISMA/DB can be found in [ABF92], here a brief introduction into the hardware used and the architecture of PRISMA/DB is given. After that two aspects of the system that are important in the context of this paper are described: data-fragmentation and parallel query execution, and the built-in benchmarker that allows detailed analysis of performance results.

The POOMA machine

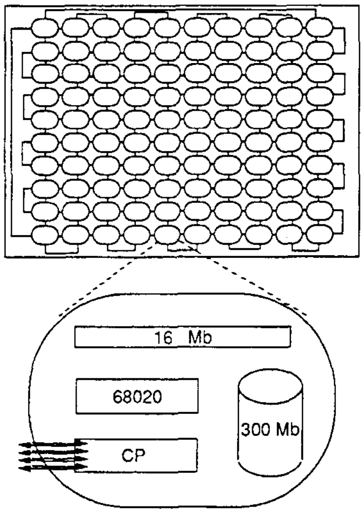


Figure 4: Hardware configuration of the POOMA machine

PRISMA/DB is implemented on a parallel multiprocessor, called the POOMA machine. The POOMA machine is a shared-nothing, parallel multi-processor, which consists of 100 nodes. Figure 4 shows the hardware configuration. Each node consists of a 68020 data processor with 16 Mbytes of memory, a disk, and a communication processor that links it to 4 other nodes using bidirectional links. Some nodes have an ethernet card that links the system to a Unix host. The entire system contains 1.6 Gbytes of memory.

On this hardware, the implementation language POOL [Ame89,Spe91] is implemented. POOL stands for Parallel Object-Oriented Language. POOL allows the definition of objects which are implemented as processes. *Parallelism* is supplied in a very natural way: conceptually, all objects that exist in the system execute concurrently. Allocation of objects to different processors makes them really run in parallel. Objects can dynamically be allocated to processors and objects can be created and deleted dynamically. These features turn a POOL program in execution into a flexible structure which allows run-time experimentation with various forms of parallelism.

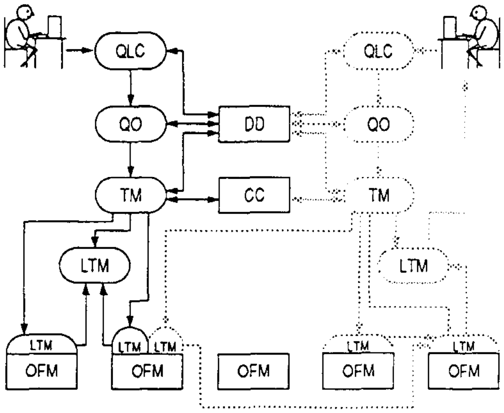


Figure 5: Global architecture of PRISMA/DB

It should be noted, that the language POOL and the supporting operating system were developed and implemented parallel to the design and implementation of PRISMA/DB. As a consequence, the currently used implementation of POOL is still experimental and not optimized in detail yet.

The architecture of PRISMA/DB

Figure 5 presents an overview of the architecture of PRISMA/DB. The architecture consists of a number of components that are implemented as POOL objects. Some components are instantiated several times in the system, others are central: they have one instantiation that serves the entire DBMS. The architecture is dynamic: components can be created and deleted dynamically, according to the use of the system.

The rectangles in Figure 5 represent permanent components, i.e. components that live as long as the system. The ovals represent transient components belonging to one user session; the life cycle of these components is related to user actions. The dotted ovals show transient components belonging to a second, concurrent user session.

Two central components of the system are the *data dictionary* (DD) and the *concurrency controller* (CC). The data dictionary is the central storage of all schema information of the system. The concurrency controller controls concurrent access to the database. It uses a standard two-phase locking protocol with shared and exclusive locks. Figure 5 shows that these central components are used by both user sessions.

The query preprocessing layer of the system is formed by the *query language compiler* (QLC) and *query optimizer* (QO) components. The query language compiler provides an interactive interface to the user and translates queries from a user language into the internal relational language of the system, called XRA. Translated queries are sent to the QO, which optimizes them into parallel execution plans.

The *transaction manager* (TM) forms the execution control layer of the system. The TM coordinates the execution of a transaction via an interface to the query execution layer of the system. Also, the TM enforces all ACID transaction properties.

The data storage and query execution layer consists of the *one-fragment managers* (OFMs) and the *local transactions managers* (LTMs). OFMs are permanent; they store and manage one fragment of a relation in the database. As OFMs serve as storage units of the database, these components can be accessed by all user sessions. LTMs are transient and private to the transaction they belong to; they are the relational engines in the system. An LTM can be attached to one OFM. In that case, the LTM is allocated to the processor hosting the OFM, and the LTM can directly access the fragment stored in the OFM. If different user sessions want to access an OFM concurrently, each user session attaches a private LTM to the OFM. LTMs that are not attached to an OFM process intermediate results. LTMs can exchange data by sending tuple streams.

An eXtended Relational Algebra (XRA) is used as internal representation of queries. This language consists of the normal relational operations extended with some primitives for grouping and for recursive query processing. Also, the language allows the expression of a wide range of parallel execution plans for a query. Each relational operation can be executed by an arbitrary number of processors, and the result of an operation can efficiently be redistributed over an arbitrary number of destinations using a split operation. Such a split operation explicitly lists the addresses of the destinations. The language is described in detail in [GWF91].

Data fragmentation and Parallel query execution

In PRISMA/DB, performance is gained through parallel execution of queries. Fragmentation of the data belonging to one relation over (part of) the available nodes is known to be a prerequisite for performance gain from parallel query execution.

PRISMA/DB uses *hash-based fragmentation* of base relations and intermediate results. The number of fragments used for a relation, the fragmentation attribute, and the allocation of fragments to processors can be set by the user. Tuples that are inserted into a relation are automatically inserted into the right fragment of the relation. To decide into which fragment a tuple is to be inserted, the system applies a system-wide used hash-function to the fragmentation attribute, and the resulting value modulo the number of fragments used for the relation indicates the fragment where the tuple belongs. The number of fragments that a relation is fragmented over, is called the fragmentation *degree* of the relation. An arbitrary number of fragments can be used for a relation, however, fragmentation degrees over 100 are probably not useful on a 100-node machine.

PRISMA/DB allows a wide variety of *parallel execution strategies* for simple and complex queries. [ABF92] describes the execution model for multi-operation queries, and [WiA91] presents the results of our research on the parallel execution of multi-join queries. In this paper, the performance and speedup characteristics of some queries from the Wisconsin Benchmark are described. To study the behavior of these queries, only the parallel execution of simple operations has to be described. This description starts at the transaction manager level, as we do not want to take the query preprocessing phase into account here.

A query enters the transaction manager as a parallel execution plan, that specifies the operations that have to be executed on the fragment level. For example, a range selection from a relation is expressed as a number of range selections from the fragments that belong to the relation. The TM starts an LTM for each fragment, and initializes it with the fragment selection operation that it has to execute. As one TM serves the entire transaction, this initialization phase is a sequential process, and, as a consequence, the fragment selection operations cannot all start at exactly the same time.

The execution of binary operations is a bit more complex. A join operation is described as an example. The parallel execution plan of a join in which both operands are fragmented on the join attribute consists of a set of fragment joins. The operands of one fragment join are stored in two OFMs. Because LTMs are private to OFMs, the execution of this join requires the initialization of two LTMs: one that sends its base-fragment to the other, and the other one that joins the incoming stream of tuples to its base fragment. When the join operation needs redistribution of one or both operands, the sending LTMs redistribute their data. Note, that a join operation that requires redistribution of both operands needs an additional set of LTMs for the redistribution of the second operand.

This architecture closely follows the abstract architecture described in Section 2. The next section will show that its implementation mainly follows the model, however it deviates from it in some cases.

The benchmarker

PRISMA/DB has a built-in benchmarker that allows detailed analysis of the execution characteristics of a query. The benchmarker allows writing cheap log messages during the execution of the query. The expensive collection of the benchmark data is postponed until after the query execution. The benchmarker consists of a datastructure, called the benchmark collector, on each processor, that is shared by all processes that run on that processor. Processes can write simple log messages to their local benchmark collector. Writes to a benchmark collector are atomic. A log message consists of the local time on the processor (the local clocks are synchronized), the process identity, and an indication of what the process is doing, like "start" or "ready". After the execution of the query the data from all benchmark collectors are collected into one file, which can be analized. In the current version of PRISMA/DB, all LTMs log the time at which they initialize, and the time at which they are ready. This requires two log messages per LTM, and therefore the execution of the query is not influenced much by this benchmarking. From the benchmark data, the costs of initializing one operation process can retrieved using the initialization time of subsequent LTMs, and the costs of local processing can be found from the difference between the "init" and "ready" mark of one LTM.

4 Performance

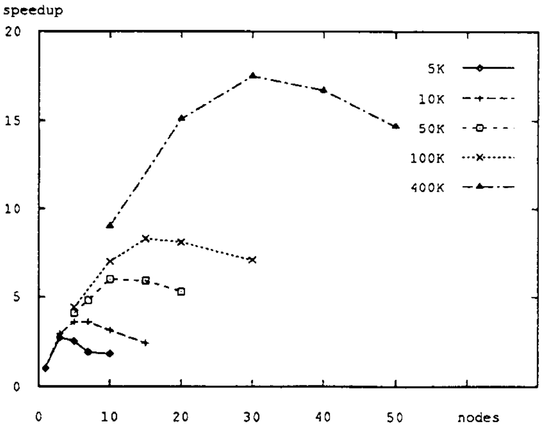
Some queries from the Wisconsin Benchmark [BDT83] are used to evaluate the performance of PRISMA/DB.

Selection queries

A query that selects 1% of its input is used to evaluate the performance of selection queries. The source relation is fragmented over a number processors and the selection criterion is not on the partitioning attribute, so all fragments have equal probability to find qualifying tuples. The result is stored fragmented without redistribution on the processors generating result tuples, but obviously, as PRISMA/DB is a main-memory system, the results are not written on disk. Different sizes for the source relation are used, ranging from 5 000 (5K) tuples to 400 000 (400K) tuples. For each source relation size, a speedup experiment is done. The numbers of processors used are adjusted to the size of the source relation, following the theory developed in Section 2 that shows that larger source relations have a higher optimal number of processors.

processors	5K	10K	50K	100K	400K
1	480	912			
3	**176**	306			
5	188	**248**	775	1416	
7	208	252	656		
10	262	292	**524**	876	2796
15		384	530	**735**	
20			596	760	1646
30				860	**1426**
40					1486
50					1692

response times in ms



speedup diagram

Figure 6: Performance of selection queries

Figure 6 shows the response times resulting from the selection queries, and the speedup diagrams that can be calculated from them. All response times are given in ms. The **best** response time for each source relation size is printed in bold font.

The response times are a measure for the absolute performance of the system. The absolute performance figures are reasonable compared to other systems. Comparison of the absolute performance of systems is hard, because there are too many differences between systems in hardware, functionality etc. However, to give an indication, Figure 7 lists the response times of some other systems, with the number of processors used for a 1% selection from 100K tuples. The absolute performance of PRISMA/DB seems reasonable from these data.

name	#proc	response time	
Teradata	20	28 220	[DGS87]
GAMMA(VAX)	8	13 830	[DGS87]
Silicon DBM	3	10 900	[LeR87]
PRISMA/DB	15	735	
GAMMA(Intel)	30	450	[DGS90]

Figure 7: Response times of some parallel DBMSs to a 1% selection from 100 tuples in ms

The speedup characteristics illustrate the relative performance of the system. This aspect of the system can be analized in the context of the model presented in Section 2.

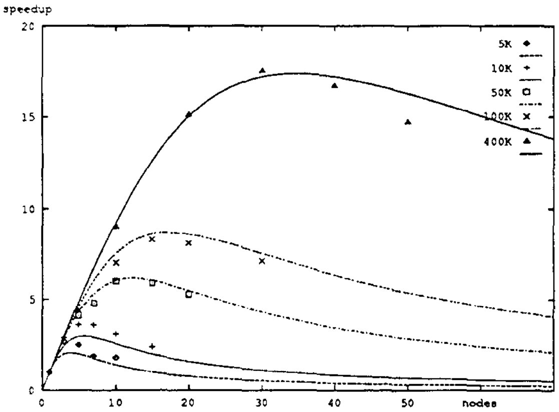


Figure 8: Theoretical speedup curves for selection queries

From the benchmark data on the selection queries, the value of a (the costs of initializing one selection LTM), and c (the costs of evaluating the selection for one tuple) can be derived. The results from each query yield approximately the same value for both quantities: a is equal to 19 ms and c is equal to 0.06 ms. Substitution of these values into Equation (2) yields expressions for the theoretical speedup curves. Figure 8 plots the measured speedups in points and also the theoretical speedup curves. The measured points nicely coincide with the theoretical curves. From these experiments, we can conclude that the optimal number of processors to be used for a parallel selection is lower than 100 on PRISMA/DB.

Two additional remarks can be made about these results. The first one is about the consequences of these results for data fragmentation. Well-known query optimization strategies [CeP84] push selections down to the leaves of a query tree. Therefore, many queries execute selection operations on the base relations before executing other, more complex operations. Such selection operations execute in the shortest time if the relation is fragmented over the number of processors that is optimal with respect to the cardinality of the relation. Therefore, in a system like PRISMA/DB it is not a good idea to fragment relations over all available nodes. Rather, the degree of fragmentation for a relation should be equal to the optimal degree of parallelism for a selection on that relation.

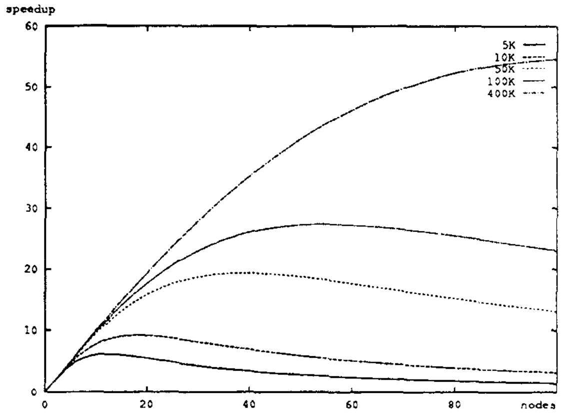


Figure 9: Theoretical speedup curves for selection queries using low initialization costs

The second remark is about the costs of initializing one operation. The value of a is high compared this value to other prototype research DBMSs. The fact that a has a high value in PRISMA/DB is caused by the fact that the implementation of POOL is experimental and by the fact that a new LTM object is started for each initialization of an operation. Reusage of LTMs and optimization of this aspect of the POOL implementation should improve this value by an order of magnitude [DeW91]. However, reducing a to 2 ms only shifts the the execution characteristics somewhat, but the optimum behavior is not shifted out of the relevant range of processors (1 - 100 for PRISMA). Figure 9 shows the theoretical speedup characteristics with a set to 2 ms. Although, the optimal numbers of processors are larger (by a factor of $\sqrt{10}$) than the real ones for PRISMA/DB in its current implementation, it should still be concluded that full declustering is not the appropriate default data distribution strategy.

4.1 Join queries

The join query used in the performance experiments is a query joining a 10K tuple relation to a 100K tuple relation in which every tuple of the 10K relation matches to one tuple in the 100K relation, so the result consists of 10K tuples. This query is called the joinABprime query in [BDT83]; A is the 100K relation and Bprime is the 10K relation. Three different execution strategies were tested, which are called join1 through join3. In all cases, both operands were fragmented into equal numbers of fragments, and each fragment was stored on a private processor.

join1 Both relations are fragmented on the join attribute. The Bprime fragments are sent to the A fragments for joining.

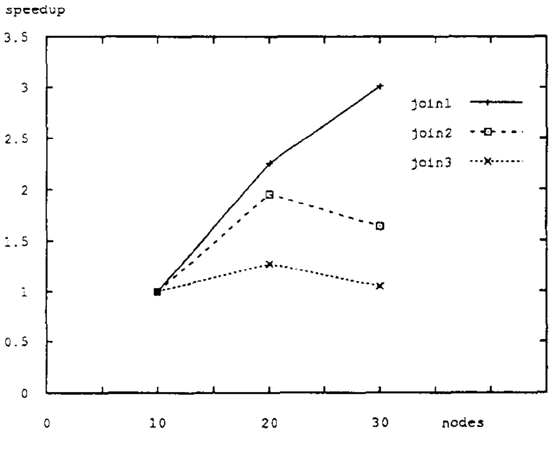
join2 Relation A is fragmented on the join attribute and relation Bprime is fragmented on another attribute into equal numbers of fragments. Relation Bprime is redistributed and sent to relation A for joining.

join3 Both relation are fragmented on another attribute than the join attribute into equal numbers of processors. Both relations are redistributed and sent to the join processors for joining.

These three strategies were tested using 10, 20, and 30 processors for the joins combined with a fragmentation degree of 10, 20, or 30 for the initial fragmentation of the relations.

#	join1	join2	join3
10	6132	6324	9036
20	2718	3240	7100
30	2034	3838	8566

response times in ms



speedup characteristics

Figure 10: Performance of join queries

Figure 10 shows the response times measured in this experiment, and the speedup with respect to the response time of the 10-processor queries. Note, that in this case linear speedup yields a speedup factor 3 for the 30-processor queries.

name	#proc	response time	
Teradata	20	131300	[DGS87]
GAMMA(VAX)	8	45600	[DGS87]
Silicon DBM	3	23900	[LeR87]
HC16-186	16	10000	[BrG89]
GAMMA(Intel)	30	3340	[DGS90]
PRISMA/DB	30	2034	

Figure 11: Response times of some parallel DBMSs to a 100K × 10K join, fragmented on the join attribute

The achieved absolute performance for "join1" is good compared to other systems. Figure 11 lists the response times for the same query reported by other projects. Again, it is hard to compare systems, as they differ in many ways. Yet, we like to report that the response time measured on PRISMA/DB outperforms all other reported performance figures on this query.

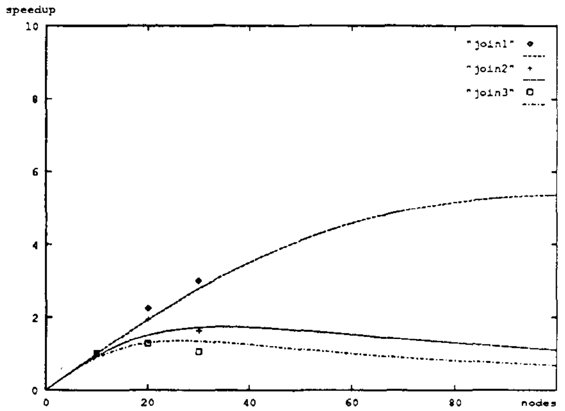


Figure 12: Theoretical speedup curves for join queries

From the benchmark data, again the values for a, and c can be derived. For "join1", the value of a is 32 ms. This value is larger than for selection, because each fragment join requires the initialization of two LTMs, one for the join and one to send the data to the join-LTM. The value of c (calculated as the costs per result tuple) is 4.6 ms. Substitution of these values into equation (2) yields the theoretical speedup curve for the join query, which is shown in Figure 12. Although, the measured points for this query show linear speedup, the theory shows that the speedup characteristics do flatten when more processors are added. This observation again illustrates that linear speedup behavior cannot be extrapolated to larger systems.

The measured speedup for "join2" is disappointing, and "join3" is even worse. Analysis of the benchmark data yields a good explanation for the bad parallel behavior of these queries. The values for c that can be derived from the benchmark data are consistent and reasonable: 5.8 ms for "join2", and 8.8 ms for "join3". These values are higher than the value for join1, due to the redistribution overhead, however, they are constant for each speedup experiment. The value of a, however, increases with the number of processors used. This means that the initialization overhead per LTM gets higher when more LTMs are used, and therefore the theory developed in Section 2 cannot be used to model the parallel behavior of these queries. For "join2" the value of a increases from 39 ms for 10 processors to 85 for 30 processors. For "join3", the value of a increases from 77 for 10 processor to 172 for 30 processors. This increase in caused by the fact that each redistribution LTM is initialized with a large XRA-expression that tells the LTM how to distribute the data over the join LTMs. This expression gets larger if more join-LTMs are involved. Shipping these large XRA-expressions is too expensive in the current POOL implementation. Here, we are faced with the limitations of our flexible parallel execution model. However, we are currently studying how the POOL implementation can be improved on this point. The model developed in Section 2 assumes a to be constant. Therefore, this model cannot be applied to "join2", and "join3". The curves in Figure 12 use average values for c. As expected the measured data deviate from the theoretical curve.

5 Summary and conclusions

This paper analizes the performance of the parallel, main-memory DBMS, PRISMA/DB. This DBMS tries to combine the performance advantages from parallelism and from main-memory implementation of relation operations. Here, the results of this experiment are reported.

In this paper, the performance analysis is described against the background of a simple analytical model. This model can explain the obtained results to a large extent. Deviations from the model can easily be accounted for.

The absolute performance of the system is measured as response time to some queries from the Wisconsin Benchmark. The absolute performance appears to be competitive with respect to other research prototypes.

Linear speedup is the ultimate goal of parallel processing. PRISMA/DB does not achieve linear speedup to up the size of the system (which consists of 100 processing elements). Rather, speedup experiments for selection and join operations show linear speedup for small numbers of processors and optimum behavior for larger numbers with the optimal number of processors to execute an operation on below 100. The nonlinearity of the speedup is caused by the relatively fast local processing of a main-memory system. Also, the optimal number of processors to execute a (cheap) selection is lower than the optimal number for the (more expensive) join operation.

Disk-based DBMSs differ in the following way from a main-memory DBMS: the local processing is slower on a disk-based system than on a main-memory system, but the communication costs are similar on both DBMS types. From the developed theory it follows that consequently disk-based systems show linear speedup up to higher numbers of processors than main-memory systems. However, the theory also shows that the speedup of a disk-based system is expected to flatten when a considerable number of processors is used.

The analysis of the experimental results in this paper was greatly simplified by the use of a benchmarker that records what the system is doing at a certain moment in time. The processing costs of this benchmarker are very low and therefore, the measured phenomena are not influenced by the use of the benchmarker. The benchmark results allow accurate determination of the problem parameters, and, in case of the redistribution joins, it revealed that the developed theory could not account for observed phenomena. Also, analysis of the benchmark data gives clear insight in the nature of the problems for these queries, so that it is clear what sort of optimizations will alleviate the performance problems of redistribution joins.

The results of this study can be used to design a data fragmentation strategy for main-memory DBMSs. Because many queries execute selection operations on their base data before executing the more complex operations, it seems a good idea to use a fragmentation degree for a relation that is equal to the optimal number of processors to execute a selection operation on that relation. This issue deserves further research.

The results of our study also have consequences for the hardware architecture of a parallel main-memory DBMS. It is obvious that a main-memory DBMS needs a large amount of main memory to store the entire database. However, our study shows that also the amount of memory for one processor should exceed a certain threshold: on the one hand, the size of the memory limits the size of the subtasks by putting a limit to number of operand tuples that can be stored on one processors; on the other hand, the subtasks need to be relatively large to allow performance gain from parallelism. Therefore, the only way achieve satisfactory parallel behavior of main-memory query execution is having so much memory per processor that subtasks can be large enough to allow considerable performance gain form parallel query execution. The exact sizes are to be derived from the system parameters and the application domain. Yet, it is perfectly feasible that a 50 node system with 32 Mbytes per processor has better performance characteristics than a 100 node system with 16 Mbytes per node, because it allows larger subtasks in parallel query execution.

In the future, PRISMA/DB will be used for performance studies in various direction. Firstly, we try to resolve some of the performance problems identified in this paper, especially those that relate to redistribution joins. Also, the performance analysis of multi-join queries [WiA91] will be continued, and work on the performance analysis of the parallel execution of transitive closures [HAC90] was recently started.

Acknowledgement

The authors wish to thank all the members of the PRISMA group for their cooperation, and especially Paul Grefen for the fruitful discussions, and Ben Hulshof from Philips Research Laboratories for his help with the execution of experiments on the 100-node POOMA.

References

[Ame89] P. America, "Issues in the design of a parallel object-oriented language," *Formal Aspects of Computing* 1 (1989), 366–411.

[ABF92] P. M. G. Apers, C. A. vanden Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten & A. N. Wilschut, "PRISMA/DB: A Parallel Main-Memory Relational DBMS," Memorandum INF92-12, Universiteit Twente, Enschede, The Netherlands, 1992, Submitted to the special issue on Main-Memory databases of the IEEE transacions on Knowledge and Data Engineering.

[AHB88] P. M. G. Apers, M. A. W. Houtsma & F. Brandse, "Processing Recursive Queries in Relational Algebra," in *Data and Knowledge (DS-2)*, R. A. Meersman & A. C. Sernadas,eds., Elsevier Science Publishers, IFIP, 1988.

[BDT83] D. Bitton, D. J. DeWitt & C. Turbyfill, "Benchmarking Database Systems - A Systematic Approach," in *Proceedings of Ninth International Conference on Very Large Data Bases, Florence, Italy, October 31–November 2, 1983.*

[BAC90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith & P. Valduriez, "Prototyping Bubba, A Highly Parallel Database System," *IEEE Transactions on Knowledge and Data Engineering* 2 (1990), 4–24.

[BrG89] K. Bratbergsengen & T. Gjelsvik, "The Development of the CROSS8 and HC16-186 (Database) Computers.," in *Proceedings of the Sixth International Workshop on Database Machines, Deauville, France, June 1989*, 359 –372.

[CeP84] S. Ceri & G. Pelagatti, *Distributed Databases, Principles and Systems*, McGraw-Hill, New York, NY, 1984.

[CAB88] G. Copeland, W. Alexander, E. Boughter & T. Keller, "Data Placement in Bubba," in *Proceedings of ACM-SIGMOD 1988 International Conference on Management of Data, Chicago, IL, June 1–3, 1988.*

[Cve87] Z. Cvetanovic, "The Effects of Problem Partitioning, Allocation, and Granularity on the Performance of Multi-Processor Systems," *IEEE Transactions on Computers* 36 (1987).

[DGS90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao & R. Rasmussen, "The GAMMA Database Machine Project," *IEEE Transactions on Knowledge and Data Engineering* 2 (March 1990), 44–62.

[DeW91] D. J. DeWitt, "Personal communication.

[DGS87] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, R. Jauhari, M. Muralikrishna & A. Sharma, "A single user evaluation of the GAMMA Database Machine," in *Proceedings of the Fifth International Workshop on Database Machines, Karuizawa, Japan, October 1987.*

[DKO84] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebreaker & D.Wood, "Implementation techniques for main memory database systems.," in *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data, Boston, MA, June 18–21, 1984*, 1–8.

[Eic87] M. Eich, "A classification and comparison of main memory database recovery techniques," in *Proc. of the 1987 Database Enginering Conference, 1987*, 332–339.

[Eic89] M. Eich, "Main Memory Database Research Directions," in *Proceedings of the Sixth International Workshop on Database Machines, Deauvil le, France, June 1989*, 251–268.

[GLH83] H. Garcia-Molina, R. J. Lipton & P. Honeyman, "A Massive Memory Database System," Technical Report 314, Department of Comp Science, Princeton University, September 1983.

[GrA90] P. W. P. J. Grefen & P. M. G. Apers, "Parallel Handling of Integrity Constraints on Fragmented Relations," in *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems, Dublin, Ireland, July 2-4 1990*, 138 - 145.

[GWF91] P. W. P. J. Grefen, A. N. Wilschut & J. Flokstra, "PRISMA/DB1 User Manual," Memorandum INF91-06, Universiteit Twente, Enschede, The Netherlands, 1991.

[HAC90] M. A. W. Houtsma, P. M. G. Apers & S. Ceri, "Distributed Transitive Closure Computations: The Disconnection Set Approach.," in *Proceedings of Sixteenth International Conference on Very Large Data Bases, Brisbane, Australia, August 13–16, 1990*, 335–346.

[HoA92] M. A. W. Houtsma & P. M. G. Apers, "Algebraic optimization of recursive queries," *Data and Knowledge Engineering* 7 (March 1992).

[LeC86] T. J. Lehman & M. J. Carey, "Query processing in main memory database management systems.," in *Proceedings of ACM-SIGMOD 1986 International Conference on Management of Data, Washington, DC, May 28–30, 1986*, 239–250.

[LeR87] M. D. P. Leland & W. D. Roome, "The Silicon Database Machine: Rational, Design, and Results," in *Proceedings of the Fifth International Workshop on Database Machines, Karuizawa, Japan, October 1987*.

[Spe91] J. vander Spek, "POOL-X and its implementation," in *Proceedings of the PRISMA Workshop on Parallel Database Systems, Noordwijk, The Netherlands, 1990*, P. America,ed., Springer-Verlag, New York–Heidelberg–Berlin, 1991, 309–344.

[Ter83] Teradata Corporation, "Teradata, "DBC/1012 Database Computer Concepts and Facilities," C02-0001-00, 1983.

[WiA91] A. N. Wilschut & P. M. G. Apers, "Dataflow Query Execution in a Parallel Main-Memory Environment," in *Proceedings of the First International Conference on Parallel and Distributed Information Systems, Miami Beach, Florida, USA, December 1991*.

[WiA92] A. N. Wilschut & P. M. G. Apers, "Dataflow Query Execution in a Parallel Main-Memory Environment," in *To appear in Journal of Distributed and Parallel Databases.*.

[WiA90] A. N. Wilschut & P. M. G. Apers, "Pipelining in Query Execution," in *Proceedings of the International Conference on Databases, Parallel Architectures and their Applications, Miami, USA, March 1990*.