1991

C.A. van den Berg, M.L. Kersten, S. Shair-Ali

Dynamic parallel query processing

Computer Science/Department of Algorithmics and Architecture Report CS-R9112 February

Dynamic Parallel Query Processing

C.A. van den Berg
M.L. Kersten
S. Shair-Ali

CWI
P.O. Box 4079, 1009 AB Amsterdam,
The Netherlands

Abstract

Traditional query optimizers for multiprocessor database systems produce a mostly fixed query evaluation plan based on assumptions about data distribution and processor workloads. However, these assumptions may not hold at query execution time due to contention caused by concurrent use of the system or lack of precision in the derivation of the query plan. In this paper, we propose a dynamic query processing scheme based on subdividing the query into subtasks and scheduling these adequately at run-time. We present the performance results obtained by simulation of a queueing network model of the proposed software architecture. Furthermore, we present the results from a validation of the simulations on the PRISMA 100-node multiprocessor.

Key Words & Phrases: database machines, dynamic query processing, performance analysis.
1985 Mathematics Subject Classification: 69C40, 69C24, 69H24, 69H26, 69H33
1990 CR Categories: C.4, C.2.4, H.2.4, H.2.6, H.3.3, I.6.3

1 Introduction

Query processing in current database machines is generally based on the combination of two ideas: operation pipelining and static scheduling. Operation pipelining is the process of mapping the operations of the operator tree for a query onto virtual processors. In this model the data flows from the leaf processors to the processor associated with the top of the operator tree. Before query execution starts, the virtual processors are allocated to physical processors using a (heuristic) scheduling algorithm. This algorithm uses assumptions based on the datadistribution, communication delays, and the selectivity of the operations to come up with an optimal allocation, where optimal means with a minimal response time. This

Report CS-R9112
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

approach is widely used in database machine architectures like Bubba [Bea90], Gamma [DGS+90], and PRISMA [KAH+88].

Although operation pipelined query processing results in a natural and elegant processing model, it also has some disadvantages. It is difficult with pipelined processing to obtain a fair load distribution among the processors involved. The prime reason being the data reduction taking place within the tree. The processors at the top of the tree will therefore remain underutilized for most of the time.

Furthermore the assumptions about the data distribution for which an optimal allocation is determined may not hold at the query execution time. The order in which the operations are performed may thus be badly choosen, and lead to inefficient use of the available processors.

These disadvantages related to static scheduling have already been identified in the literature [Ape88, GW89, Mur89]. Apers [Ape88] compared the influence of a static scheduling and dynamic scheduling method on a distributed database system. Graefe [GW89] focussed at improving the performance on a uniprocessor by choosing the access method or join algorithm at run-time. For this purpose the query optimizer prepares a query evaluation plan, which contains enough information to make the choice dynamically. Like Murphy [Mur89], we concentrate on parallel query execution. Similarly, we have decomposed the join processing in a join operation on pages (segments), and we are interested in finding an optimal schedule for the page join operations. The main difference is that we determine the scheduling at run-time, taking the actual execution times for the join subtasks into account.

In this paper we propose a query processing model, which is based on dynamic scheduling of query subtasks. Instead of decomposing the query into a tree of communicating processes, a special purpose program is created, which solves the query efficiently in main memory. For instance, for the three-way join operation on relations R, S, and T, a query program is created, which can solve the query for a small database. The whole query is evaluated by partitioning the relations into segments R_i, S_i, T_i, and then executing the query program for each (R_i, S_j, T_k) combination. The resulting tasks can be executed in parallel using a central scheduler to dispatch the tasks over the available processors at run-time. This scheduler can easily detect idle processors. Therefore, it can adjust the number of query processors to the amount actually required to effectively use parallelism. The order in which the join operations are performed within a QP is not fixed. The query program contains code to perform the operation in either order, depending on the arrival time of relation segments. The net result of this architecture is that it improves the ratio communication/processing, it results in a better load distribution over the available processors, and it is adaptive in the amount of processors used for the each query.

In the rest of this paper we will discuss the new architecture in detail. Section two presents the architecture. In section three we give a short overview of the queueing model used for the performance evaluation. Section four, discusses the results obtained from the simulation. A validation of the queueing model on a real 100-node multi-processor system can be found in section five. In section six, finally, we give a summary of our results and present our future research goals.

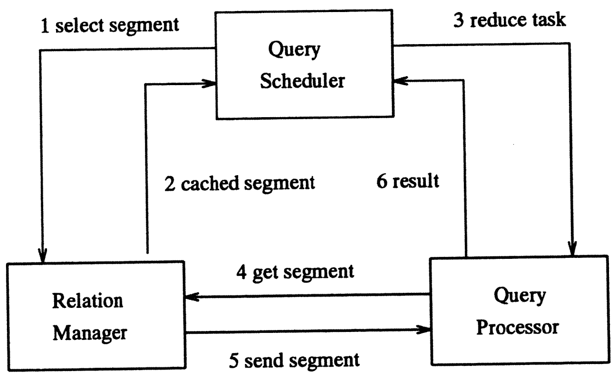

Figure 1: The basic system communication pattern

2 Dynamic Query Processing Architecture

Our architecture is based on two different kinds of processes: query dependent and schema dependent processes. The query dependent processes are called the Query Scheduler (QS) and the Query Processor (QP). Typically, during query processing, there will be a single QS and several QP processes available. The schema dependent process is the Relation Manager (RM). The query dependent processes are generated for a specific query. The schema dependent processes are generated for a specific relation or view. In the following we will briefly discuss their functionality as it pertains to our query processing scheme. Figure 1 illustrates the communication pattern between the components.

2.1 The Relation Manager

The Relation Managers provide a persistent storage and retrieval facility for a single relation. The relation storage is composed of a number of segments. Upon receiving a *select* request from the Query Scheduler, the RM retrieves the segments from disk, stores it in its cache and acknowledges the QS with a *cached* message that it has successfully read a segment. The RM can not only retrieve the raw segments from disk, but it can also apply selection, projection and partitioning operations on the data.

When the segment data has been cached, a Query Processor can subsequently retrieve this data by sending the *get* message to the RM. The RM then transports the data over the network to the QP.

2.2 The Query Processor

The Query Processors form the engine of the query evaluation process. The Query Scheduler translates the query over the relations into a batch of queries over the relation segments. Each Query Processor solves a query for a given combination of relation segments. The final query result is obtained by taking the union of the partial results produced at the QS.

In contrast with a pipelined execution model the execution order of the individual operations in the query is not fixed at compile time. It depends on the availability of the segments. For instance, if the QP is requested by the Query Scheduler to calculate $R_1 \bowtie S_2 \bowtie T_1$, and segments S_2 and T_1 arrive first, it will first calculate $S_2 \bowtie T_1$, and store the intermediate result in its cache for further use. When segment R_1 arrives, it completes the join operation, and informs the QS that it has reduced the (R_1, S_2, T_1) segment combination.

2.3 The Query Scheduler

The Query Scheduler, finally, coordinates the distribution of work over the QPs'. It consists of two major components: a task filter and a task allocator. The QS maintains a task table containing all the segment combinations to be reduced by the QPs'.

The task filter removes those tasks from the task table, which will not contribute to the final query result. This will occur, if for instance, the relations R and S are hash partitioned on their join attributes. Another possibility is exploit the feedback information available from the QPs. For example the QP can detect which segment combinations produce empty partial join results, say $S_3 \bowtie T_4$. The task filter can then remove the remaining (R_i, S_3, T_4) tasks from its task table.

The task allocator coordinates the execution of the query by selecting an idle QP for each task and removing it from the task table. Currently we consider three different schedule algorithms: *random* scheduling, *cache* scheduling, and *pipeline* scheduling. Clearly, the algorithm to select a task has a great influence on the efficiency. The *random* scheduling method selects randomly a new task from the task table. The consequence is that for practically each task, the QP has to obtain the required segments from the RMs. For the *cache* scheduling method, the QS keeps track of the segments being cached by the QPs. The QS minimizes the IO by selecting a task for which (almost) all the segments are already available. In the *pipelined* scheduling method, the QS also keeps track of the intermediate results available in the QP. It will send those tasks to the QP, which require the same intermediate results, thereby reducing both communication and processing time for a task. In the performance study described here, we have focussed on the least complex scheduling method, namely the random scheduling method. If this technique is already profitable, the other approaches can only further improve the performance.

We expect from this architecture that the performance of the system does not degrade as the number of QP is increased. We rather expect that the throughput converges towards a maximum value determined by the cost for moving segments around, but not because of scheduling overhead. The primary target for the simulation is to asses whether this assumption holds for this architecture.

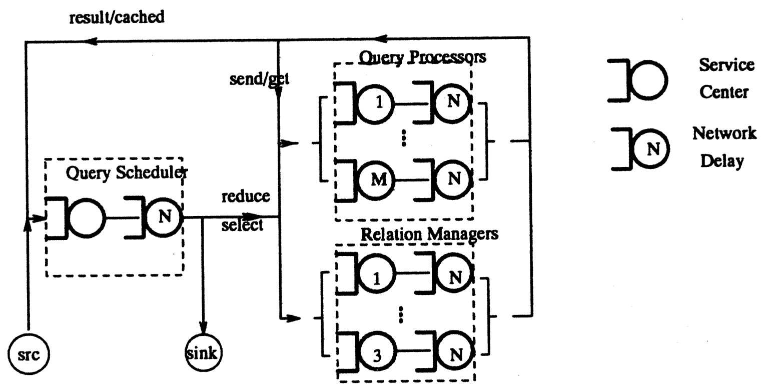

Figure 2: The three-way join queueing model

3 The Simulation Model

To experiment with the design and to gain insight into the key factors affecting the performance, we developed a queueing model of the system architecture. Clearly the RM, QS and QP can be associated with process classes. With each process class in the system we associate an arrival time distribution, a service time distribution, the number of services and the service discipline. Each service handles different kind of requests, each with its own service time distribution and arrival time distribution. For instance, the QP receives *reduce* requests from the QS and *send* requests from the RM. Right from the start we decided to use simulation of the queueing network as a tool for experimentation, because we aimed for a realistic simulation of the algorithms. Striving for an analytical solution would require too many simplifications in our model.

The queueing network modelling package QNAP2[1]version 5.0 has been used [CII84]. This package contains algorithms for discrete event simulation and algorithms for analytic solutions. The analytic solvers yield exact and approximate steady-state solutions provided the simulation model satisfies some severe constraints, such as, mutual independence between stations.

The simulation was focussed on acquiring insight in the system utilization and the speedup factor as a function of the number of QPs'. We also wanted to see if our assumption that the QS is not a bottleneck in the system is really true. The consequence of this desirable property is that the addition of new QPs' to the system does not degrade the system throughput.

We use the three-way join operation as the example query. We choose this operation for our simulation, because the join operation is a commonly used and communication expensive operation. It can easily be parallelized and it is the smallest query, allowing two different execution orders.

[1]QNAP2 (Queueing Network Analysis Package 2) is copyrighted by CII HONEYWELL BULL and INRIA 1981,1982

We have made the simplifying assumption that the network will not be the bottleneck of the system. Thus we have only modelled the network delay in the system by routing each message leaving a process through a network center. The simulation results show that the maximum network throughput is not reached, so that this assumption is valid [KSAvdB90]. The simulation model then consists of three Relation Managers, one for each relation, M Query Processors, a single Query Scheduler and several Network centers. (Figure 2).

Two different system architectures have been investigated. The basic architecture is described in the next section. In the extended model, the QPs' act as alternative segment caches. A segment can then be retrieved from a Relation Manager or from another Query Processor. We expect from this extension that the load is better distributed among the Relation Managers and Query Processors.

3.1 The basic model

In the basic model new tasks are inserted into the system by the **src** service center. This center thus mimicks a query request from a user. The QS then sends for each request a *select* message to the three RMs'. The RM reads the requested segments from disk and reports this back to the QS using the *cached* message.
When the QS has received the three *cached* messages for a task, it selects an idle QP and asks it to process the task. This QP requests the cached segments from the RM using the *get* message and receives the segments from the RM by the *send* message.
When the QP has received all the segments, it calculates the result of the three-way join operation and sends the result back to the QS using the *result* message. The result finally leaves the system through the sink service center.
This cycle is repeated endlessly in the simulation to find the steady state behaviour.

The queueing discipline is FCFS for all the centers. Furthermore we assume that the system consists of independent Poisson processes. The interarrival time distribution for the processes is therefore an exponential function [Kle75]. We also assume that the service time is exponentially distributed.

3.2 The Parameter Setting

To come up with a realistic parameter setting, we have measured the service time for each process, and the delay of the network on an existing multiprocessor database machine, named PRISMA [KAH+88]. This machine is also used to validate the simulation model. The parameters can be found in Table 1. The PRISMA machine is a loosely coupled multiprocessor machine consisting of 100 MC68020 processors connected through a proprietary point to point network.

We have run the simulation for the basic model and extended model for an extended period of time so as to obtain statistical measures with a 95% confidence interval of 10% around the mean. The main parameter, the number of QPs', covered the range of 1 to 50 processors. The results will be discussed furtheron.

Parameter	Unit	Description
average result size	30 tuples	
segment size	160 tuples	160 tuples is about 32 kByte
prepare segment	244 msec	Time spent in RM to read a 32k segment from disk.
scheduler processing	0.1 msec	Time spent in QS for *result*, and *cached* message.
join processing	296 msec	Time spent in the QP for joining three segments.
network delay (1)	38 msec	Delay for messages *get*, *cached*, *select*, and *reduce*
network delay (2)	64 msec	Delay for sending the join result to the Query Scheduler.
network delay (3)	388 msec	Delay for copying a segment to another node

Table 1: The parameter setting for the simulation.

Message type	size
RPC	0.5 Kb
Data	4.8 Kb
Segment	32.0 Kb

Table 2: Messages sizes

4 Evaluation

The simulation results for the basic model (see Section 4.1) show that the load of the different processes is not equally distributed. The basic model is improved by introducing a segment exchange mechanism between QPs. This extended model shows a better load distribution. These results can be found in Section 4.2.

4.1 Basic model

Figure 3 shows the utilization levels for the three system components. In this architecture, the load on the RMs is much higher than on the QS and QP. In particular, its reaches 90% utilization with 20 processors already. The underlying cause is that the ergodic constraint at the RMs reaches equality at this point ($\frac{\lambda}{\mu} \approx 1$). That is, the expected number of arrivals (λ) at a center reaches the serving capacity (μ).

An indication of the total throughput for each process is given in Figure 4. The central role of the QS is highlighted by the throughput of messages through this center. Furthermore, the bottleneck is the RM which can not adequately handle an increasing number of QPs. Therefore the throughput for QPs ultimately decreases.

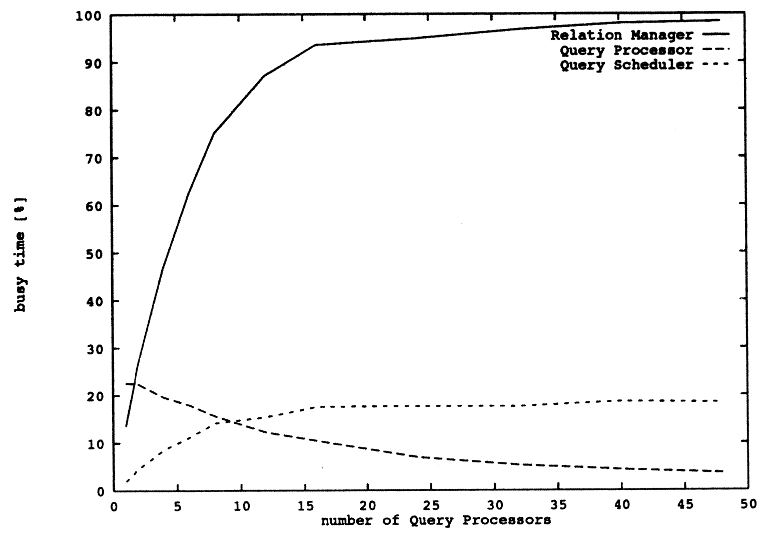

Figure 3: Utilization level for the basic model

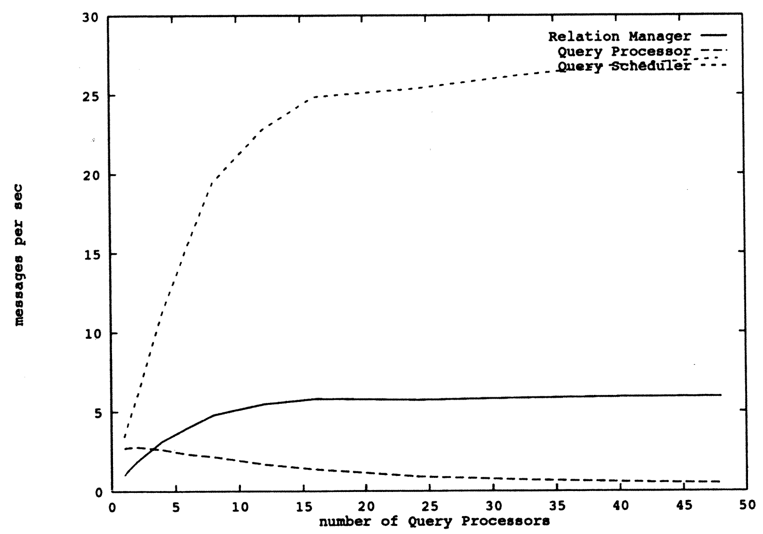

Figure 4: Message Throughput in the basic model

Process	I/O	Description	Type	Messages/sec	Kb/sec
QS	IN	*segment cached* from RMs	RPC	82.5	
		result from QPs	Data	27.0	
					170.9
	OUT	*reduce vector* to QPs	RPC	27.0	
		select segment to RMs	RPC	82.5	
		partial result to user	Data	27.0	
					184.4
					355.3
QP	IN	*reduce vector* from QS	RPC	0.7	
		send segment from RMs	Segment	2.1	
					67.6
	OUT	*result* to QS	Data	0.7	
		get segment to RMs	RPC	2.1	
					4.4
					72.0
RM	IN	*select segment* from QS	RPC	27.5	
		get segment from QPs	RPC	27.0	
					27.3
	OUT	*segment cached* to QS	RPC	27.5	
		send segment to QPs	Segment	27.0	
					877.8
					905.1

Table 3: Network requirements for the basic model

The assumption that the network will not limit the system still holds, as can be seen in Table 3. Using the message type distribution, obtained from the simulation, and the size of each message type (Table 2), we have calculated in this table the total data throughput for each process. It turns out that for all process types the required bandwidth is much lower than the network limit of 2 Mbytes/sec. Thus the capacity of a single network link is not exceeded. Under the assumption that a fully interconnected network is used, we may conclude that the global network bandwidth is not exceeded either.

4.2 Extended model

In the previous section we observed that the RM forms the potential bottleneck in the basic model. The congestion of the RMs can be avoided by also using the QPs as a cache for the tuple segments, thereby spreading the load for accessing tuple segments over both the QPs and RMs. To simplify the model, we assume unbounded caching resources at the QPs.

This model was simulated under the same conditions as the previous model (i.e. accuracy, simulation time). The results from these runs are presented in Figure 5 - 6.

Figure 5 shows the utilization level of the three system components. If the utilization of the QS is extrapolated, we see that the QS reaches saturation with around 60 QPs. However, the throughput peak of the QS is reached for a much lower number of QPs (43,

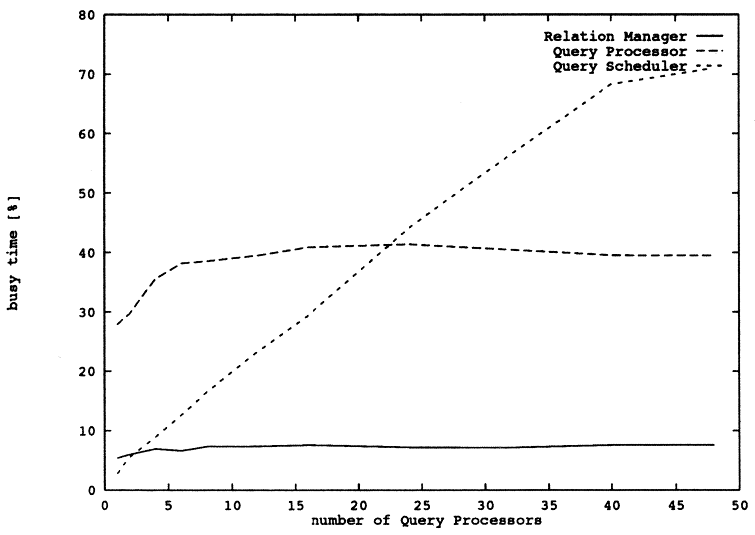

Figure 5: Utilization level for the extended model

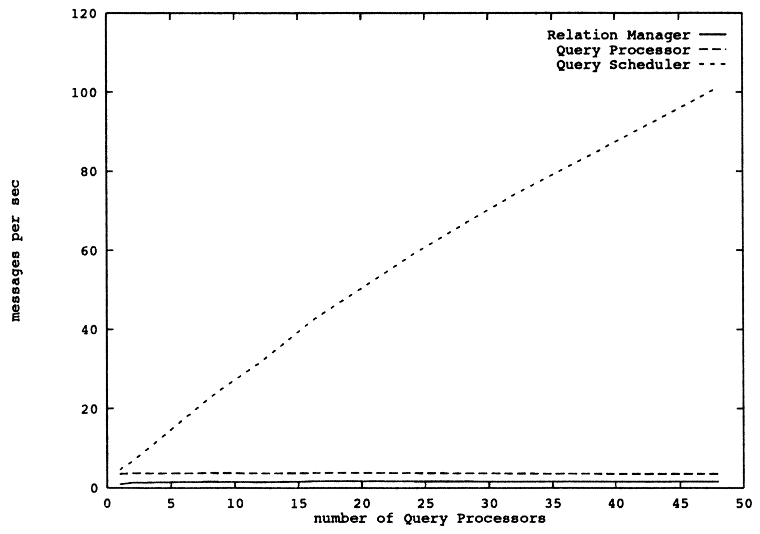

Figure 6: Message throughput in the extended model

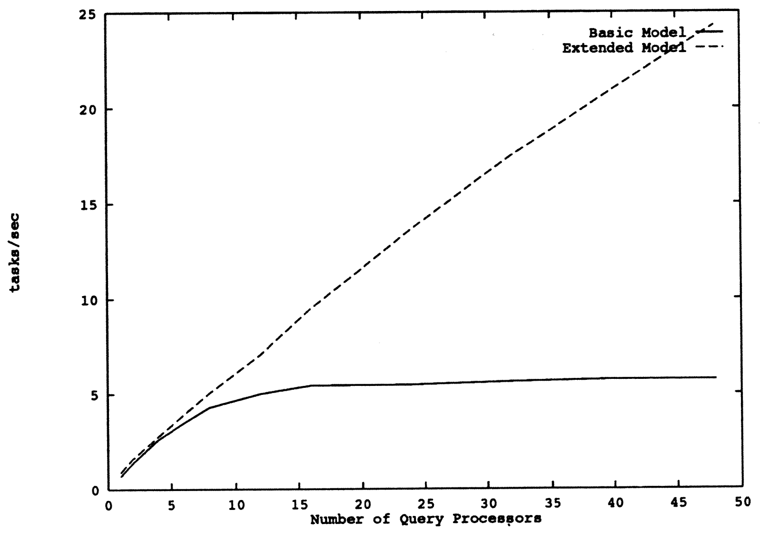

Figure 7: Task throughput in the basic and extended model

see Figure 7), which means that between 43 and 60 one already faces a reduced payoff of parallel execution.

Compared with the basic model we have doubled the effective number of active QPs and we obtained a 5 times higher throughput (48 QPs, Figure 7) by better utilization. Furthermore, the model displays linear speedup in processing up to 25 QPs. The RM is no longer the bottleneck, allowing parallel query execution without causing a significant performance degradation.

As with the simulation of the basic model we have verified the network assumption using the simulation results. As these calculations are similar to the previous one we simply refer to [KSAvdB90] for the details.

5 The Validation

For a simulation experiment there is a tradeoff between the level of detail modelled and the implementation effort. To assess whether the architecture has been modelled in enough detail, we have also performed a validation study for the three-way join operation.

We have made a prototype implementation of the architecture on the POOMA machine in POOL-X. The POOMA architecture and operating system were specially designed to support the language POOL-X. At the moment the POOMA machine can only run single POOL-X programs. The program is compiled, linked with the operating system code and downloaded to the machine. The result is that the program runs in complete isolation and that there are no other activities affecting the performance.

A program run displays, in contrast with the simulation, the transient behaviour of the

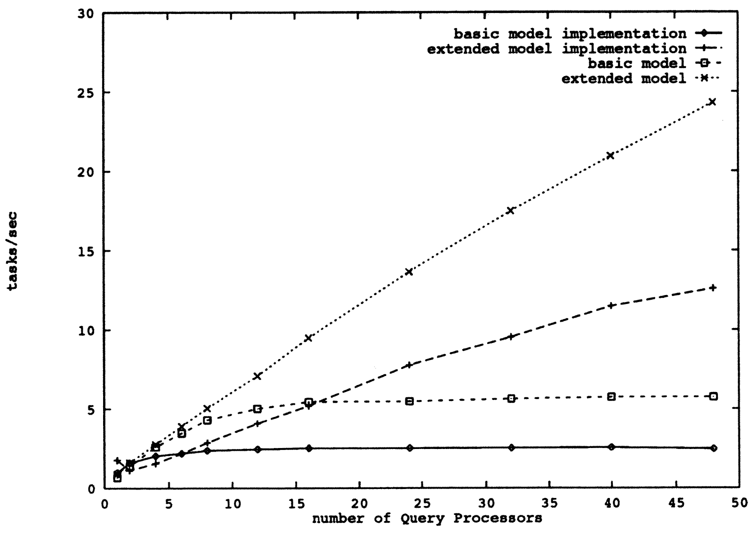

Figure 8: Task throughput for the simulation and the prototype

system. However, the transient behaviour converges towards the steady state behaviour, if the total number of tasks to be reduced is choosen sufficiently large. In total 1000 tasks are processed during a single run.

In the validation experiment we ran the versions with and without segment exchange, changing the amount of Query Processors from one to fifty. The processes were all allocated on different physical nodes. We used the hardware timers provided by the processing nodes, to measure the total time spent in the query evaluation.

The validation experiment only measures the total time for a query. It was not possible to measure the idle time for a processor directly. Figure 8 shows the measured task throughput for the runs, and the task throughput obtained through simulation.

The model and the prototype display the same behaviour. In the range of 1 to 50 Query Processors the extended model and its prototype imlementation show a lineair speedup upto 25 QPs. The basic model and its implementation reaches its maximum throughput at 15 Query Processors.

On the whole is the performance for the simulated model better than for the implementation. For the extended model this effect is caused by the simplifying assumption that the cache of the QP is unbounded. In the implementation cache misses do occur, which result in additional segment requests for the RM.

A close examination of the implementation showed that there is a relation between processing speed and communication activity. Each communication action requires a packetize and depacketize operation. We attribute the constant discrepancy between the model and its implementation to this relation.

6 Conclusions and Future Research

In this paper, we have presented an alternative approach for query processing on large multiprocessors. Our approach is based on breaking the query into two smaller problems, namely, how to solve the query for a small portion of the database and, how to schedule a large number of tasks, which together form the query program. We think that this combination leads to better system utilization and smooths the fluctuations normally encountered in parallel query processing.

A queueing network model has been constructed that captures the processing aspects of our architecture. It has been used to drive a discrete event simulation to experiment in a time efficient way with two processing strategies, i.e. a central and decentralized caching of tuple segments.

The two simulations show that the central scheduler does not lead to an immediate bottleneck. The linear speed-up curve flattens before the scheduler becomes overloaded. That is, the speedup from parallelism becomes neglectable before the Query Scheduler becomes saturated. Furthermore, the decentralized caching of segments proved effective.

The system utilization in both cases is still limited, mainly due to the network activities, which are modelled as independent processes. Thus, once a Query Processor has issued a request for a tuple segment it has to wait for delivery; it does not take part in handling the communication protocols.

The validation experiment has strengthened our confidence in our simulation model. Although there is a slight discrepancy in the measured figures and the figures from the simulation, the prototype implementation displays the same behaviour as the simulation model. As a result we will use the simulation model as the basis for some further experiments. We are especially interested in the behaviour of the system in situations where the system load is not equally distributed. Currently we are also studying the effect of the *cached* and *pipelined* scheduling algorithms on the performance.

Designing the filter algorithm as well as query specific scheduling is an open-ended track. The filter can do a better job once more feedback information is passed to the QPM about the contents of the segments being cached. For example, as part of the message *cached* one could also return the min/max over the join attributes. This would enable the filter to precompute the overlap of a proposed segment pairing (and drop it when no such overlap exists). Furthermore, one can easily configure a more static evaluation plan within the Query Scheduler to enforce a specific order of vector evaluation.

The results of these studies will be reported in a forthcoming paper.

References

[Ape88] Peter M.G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems*, 13(3):263–304, September 1988.

[Bea90] H. Boral and et al. Prototyping bubba, a highly parallel database system. *IEEE Transactions On Knowledge and Data Engineering*, 2(1), March 1990.

[CII84] CII Honeywell Bull and INRIA. *QNAP2*, 1984. Introduction to QNAP2 and Reference Manual.

[DGS+90] D. J. DeWitt, S. Ghadeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions On Knowledge and Data Engineering*, 2(1), March 1990.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *Proc. SIGMOD*, 1989.

[KAH+88] M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, E.J.A. van Kuyk, and R.L.W. van de Weg. A distributed, main-memory database machine; research issues and a preliminary architecture. In M. Kitsuregawa, editor, *Database Machines and knowledge Base Machines*, pages 353–369, 1988.

[Kle75] Leonard Kleinrock. *Queueing Systems, Theory*, volume 1. John Wiley & Sons, 1975.

[KSAvdB90] M.L. Kersten, S. Shair-Ali, and C.A. van den Berg. Performance analysis of a dynamic query processing scheme. Internal Report CS-R9050, CWI Amsterdam, The Netherlands, 1990.

[Mur89] M.C. Murphy. Effective resource utilization for multiprocessor join execution. In *Proc. of the fifteenth VLDB Conference*, pages 67–75, August 1989.