Implementing PRISMA/DB in an OOPL *

Annita N. Wilschut Paul W.P.J. Grefen Peter M.G. Apers
University of Twente

Martin L. Kersten
Centre for Mathematics and Computer Science

Abstract

PRISMA/DB is implemented in a parallel object-oriented language to gain insight in the usage of parallelism. This environment allows us to experiment with parallelism by simply changing the allocation of objects to the processors of the PRISMA machine. These objects are obtained by a strictly modular design of PRISMA/DB. Communication between the objects is required to cooperatively handle the various tasks, but it limits the potential for parallelism. From this approach, we hope to gain a better understanding of parallelism, which can be used to enhance the performance of PRISMA/DB.

1 Introduction

The PRISMA project is a large scale research effort in which the development of a multi-computer and the implementation of non-trivial applications on top of this multi-computer are research issues. The project comprises the development of parallel hardware, the implementation of an operating system, the implementation of a parallel object-oriented language, and the implementation of applications, such as a database management system and an expert system shell, in this language. In this paper, we discuss our experiences with the implementation of a main memory DBMS (PRISMA/DB) in the parallel object-oriented language, called POOL [Amer88].

Because the hardware, the operating system, the language implementation, and the DBMS are developed in parallel, many design decisions in PRISMA/DB were taken without knowledge of the performance behavior of the underlying system. The potential for parallel execution, a phenomenon that is not yet well-understood, makes it even harder to obtain a DBMS architecture that fully exploits the capacity of the multi-computer system. To alleviate the problems, we have heavily used the facilities in POOL to set up PRISMA/DB in a *modular* way as a set of rather independent objects (processes), which each perform certain tasks, such as query processing, concurrency control, etc. This

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

modular design leads to potential *parallelism* between those objects. On the other hand, *communication* between the objects is needed to synchronize the global query handling process. This communication may reduce the potential parallelism. Also, communication between different processors may cause a large amount of network traffic with obvious consequences for the performance of the system. Hence, both modularization and communication influence the potential for parallelism.

When splitting up PRISMA/DB into functional components, it is not clear what grain size to choose and which tasks to execute in parallel. Our approach is to design PRISMA/DB in a strictly modular way into gradually smaller objects. The language POOL allows us to dynamically control the allocation of objects to processors. In this way, we can decide which objects can execute in parallel and which messages really have to go over the network. As a first approach, objects that form a functional component of PRISMA/DB, such as a parser or a query optimizer, are allocated to the same processor. As soon as the multi-computer system is available, we intend to experiment with the allocation of objects to study parallelism and to enhance the performance.

The paper is organized as follows. The remainder of this section gives an overview of the architecture of PRISMA/DB and a short introduction to the object-oriented language POOL. Sections 2, 3, and 4 deal with modularity, communication, and parallelism and their interaction. Section 5 is about the possibilities we have to experiment with the architecture of PRISMA/DB when the multi-computer system is available.

1.1 An overview of PRISMA/DB

PRISMA/DB is a database mangement system with the following features:

- *Main memory* storage of the entire database. Main memory storage improves performance, because no disk I/O is needed for retrieval. Furthermore the design of the data manager becomes simpler, because there is only one level in the storage hierarchy. Secondary storage is used only for logging and back-up to enable recovery.

- *Distribution* of relations allows parallelism in the execution of simple ralational operations like a selection or a join. By dynamically allocating fragments of relations to different processors, performance is gained.

- Support of two user interface languages: *standard SQL* as data definition and data manipulation language, and a logic retrieval language called *PRISMAlog* [Hout88], which is comparable with languages like Datalog and LDL [Morr86] [Tsur86]. An implementation of the SQL standard has been chosen to accommodate existing application environments relatively easy. A logic programming interface is chosen to have the possibility to integrate data and knowledge processing on the PRISMA machine.

The system has a modular structure with well-defined interfaces. An important role in these interfaces is played by the internal language XRA (eXtended Relational Algebra), which is a relational algebra extended with recursion, iteration, and transaction and session management. This language provides a uniform way to express queries throughout the whole DBMS. The architecture of PRISMA/DB is sketched in figure 1. For a

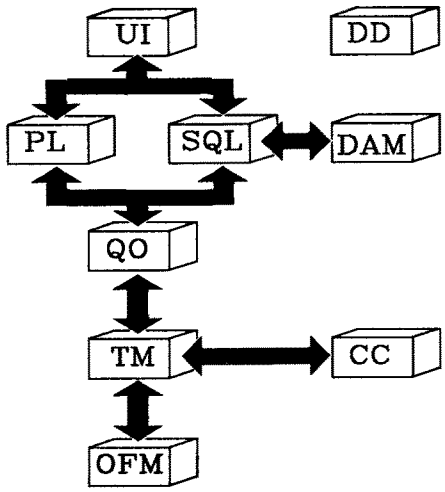

Figure 1: Architecture of PRISMA/DB

comparison with other systems we refer to [Kers87]. The components of PRISMA/DB are briefly described below.

The *Data Dictionary* (DD) is the central storage of all system information. The *Data Allocation Manager* (DAM) constructs fragmentation and allocation schemes for relations. The *Concurrency Controller* (CC) is responsible for the serializability of concurrent transactions; currently a standard two-phase locking protocol with exclusive and shared locks is used, because such a protocol has proven to perform well [Care88]. The locking granularity is that of relation fragments.

The *User Interface* (UI) is the terminal server between user and one of the parsers. The *SQL Parser* (SQL) translates SQL programs into XRA. The *PRISMAlog Parser* (PL) translates PRISMAlog programs into XRA. In particular, the parser translates the prolog-like recursion of PRISMAlog into μ-calculus expressions in XRA [Aper86b] [Aper86c].

The *Query Optimizer* (QO) deals with fragmentation transparency, removal of views, translation of μ-calculus expressions into iterations and transitive closure expressions, and, of course, the traditional optimization of queries.

The *Transaction Manager* (TM) manages the execution of schedules produced by the query optimizer. This involves requesting locks from the CC, creation of new OFMs to store intermediate results and controlling the OFMs involved in the execution of a transaction.

A *One-Fragment Manager* (OFM) controls a single fragment in the database. The OFMs form the smallest granularity for parallelism. There is a separate manager for each fragment, because this approach allows us to generate specilized OFMs to control a specific schema or to execute a specific XRA operation.

The components listed above can be divided into two groups: permanent and temporary components. The permanent components are the Data Dictionary, the Data Allocation Manager, the Concurrency Controller, and the One-Fragment Managers that

manage the fragments of the permanent relations in the database; they are created when the DBMS is initialized and they remain accessible during the lifetime of the DBMS. The temporary components are created dynamically to support user sessions and transactions during a session. For example, for each user session a new User Interface is created. For each language session within a user session, either a SQL or a PRISMAlog parser and a Query Optimizer are created. Finally, for each transaction within a language session, a new Transaction Manager is created; this TM controls the creation of temporary One-Fragment Managers for the management of intermediate results of queries. These temporary objects are garbage collected, when not needed anymore.

1.2 The implementation language POOL-X

POOL stands for Parallel Object-Oriented Language [Amer87] [Amer88]. Our DBMS has been implemented in POOL-X, an extended version of POOL which has specially been designed for the implementation of a DBMS. POOL-X has many features in common with other object-oriented languages; inheritance, however, is not supported, because the semantics of the concept inheritance is not yet fully understood. Below we will highlight the most important features of the language.

POOL-X is a *strongly typed* object-oriented language. *Objects* are the basic building blocks in the language. They are instances of *Classes.* All objects in one Class have the same data and routines acting on these data. Objects can be *created dynamically.* Conceptually, objects are independent, active entities; objects behave as processes. Hence, parallelism can be modeled simply by creating objects.

Class definitions are grouped into *Units.* A Unit is a functionally coherent part of a system. A Unit consists of two parts: The *Specification Unit* describes the features of the Classes in the Unit that are visible to other Units. The *Implementation Unit* contains the implementation of the Classes in the Unit; the implementation details are not visible to other Units.

Objects *communicate* by sending each other messages. A receiver will only accept a message if it has explicitly stated that it is ready for it. While performing the action requested in the message, the receiver may send messages to other objects or even to the sender of the original message.

POOL-X messages come in two sorts: synchronous and asynchronous. When a *synchronous* message is sent, the sender is blocked, until an answer arrives. When an *asynchronous* message is sent, the sender just goes on with its own activities after sending the message.

The language POOL-X is designed for programming a multi-computer system. Objects are, by default, allocated to some processor by the operating system. POOL-X provides *allocation pragma's* that overrule the default allocation. Using these pragma's the DBMS programmmer can control the allocation of objects to processors, and thus influence the parallelism within the DBMS.

2 Modularity

When building a large and complex system, one should split up the system into a number of functionally independent modules with precisely defined interfaces. There are two aspects to this: firstly, the architecture of the system should have a structure that makes

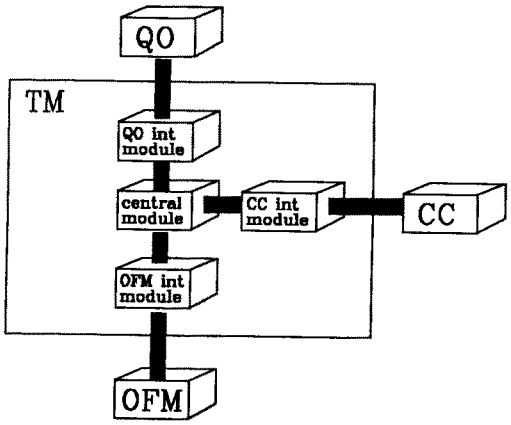

Figure 2: Transaction Manager Architecture

a modular decomposition feasible; secondly, the implementation language should provide thorough support for constructing a modular system. These two aspects are discussed in the following two subsections. Furthermore, the modular structure of a system has a great impact on the communication structure within the system and, consequently, on the possibilities for exploiting parallelism. These issues are addressed in sections 3 and 4.

2.1 Designing a modular DBMS

PRISMA/DB is designed according to a strictly modular approach. Applying this approach to all design levels resulted in the following hierarchical structure of the system.

At the highest level we identify the *components* of PRISMA/DB as described in section 1. These components make up the architecture of the DBMS at the highest level of abstraction.

One level lower, we use a modular structure to describe the internal architecture of each component. The component is further decomposed into a number of *modules* each having a well-defined functionality. As an example, the Transaction Manager is split up into the following four modules (figure 2):

- the Query Optimizer Interface Module, which takes care of some preprocessing tasks on incoming queries;
- the Central Module, which does all central administration and synchronisation of (sub)queries;
- the Concurrency Control Unit Interface Module, which does the local lock administration and handles the two-phase locking protocol;
- the One-Fragment Manager Interface Module, which takes care of the actual execution of queries, the transport of data between One-Fragment Managers, the two-phase commit protocol, and the integrity constraint handling.

At the lowest level we have the internal structure of modules; here we find local data structures for instance.

2.2 Implementing a modular system in POOL

Modularity can be viewed at a number of abstraction levels. POOL provides three abstraction levels to do so.

- At the highest level is the notion of System, consisting of an entire application.
- At the second level the language provides the notion of Units. As described before, a Unit is a functionaly coherent piece of software with a strictly defined interface to the outside world. As such it is a good tool to model modularity at the highest abstraction level of the system being implemented.
- Below the unit level, POOL provides the notion of Classes. A Class can be seen as an active abstract data type. As such, it can be used to model modularity on lower abstraction levels and to provide data abstraction.

It can be concluded here, that POOL provides the means for a well-structured architecture of a system into three layers, supported by the notions of System, Unit, and Class. If one wants to add more layers to the system hierarchy, POOL does not offer language support in the sense of scoping (nesting of Units or Classes is not allowed). The notion of Classes however, is a good aid for well-structured program design. So a multi-level hierarchical structure of the application must both rely on the language facilities as on good programming discipline. To conclude, we should keep in mind that one of the goals of modularization is to make explicit where potential parallelism is available.

3 Communication in PRISMA/DB

In the preceding section, we have seen that PRISMA/DB can be viewed as a collection of fairly independent components (objects). On the one hand, these objects all have their internal activity; on the other hand, several components have to cooperatively handle the global processes such as query processing. The latter requires communication between the components.

To keep the design of the global communication structure manageable, the communication between the components (and also on the lower levels of the system) should comply with the following restrictions: The *communication protocol* must be *well-defined*, otherwise it is hard to agree on it. The communication must be *correct* in the sense that it may not result in deadlock or livelock problems. The communication may not have more negative impact on the amount of possible parallelism than necessary; this means that no unnecessary serialization of processes may be caused by communication protocol. This last aspect of communication is dealt with in the next section.

In this section, we first make the distinction between interfaces and protocols. Secondly, we discuss our strategy to handle the complexity of the communication protocols. Thirdly, we give an example of a communication protocol. Finally, some protocol problems are discussed.

3.1 Interfaces and Protocols

As stated in the introduction on POOL, objects are grouped into Classes. All objects belonging to one Class have the same external interface, which is specified in the speci-

fication of the Class. The *interface* between an object of Class A and an object of Class B consists of the set of messages A can send to B and vice versa.

It is possible that the interface between two objects contains synchronous messages in two directions. If no agreement exists on the protocol that should be used, the two objects could simultaneously start sending a synchronous message to each other, which would leave them waiting for answer for ever. We use the word *protocol* for an agreement on how the interface should be used. It is clear that a good protocol is needed to let the components of PRISMA/DB work cooperatively and to prevent the problems mentioned above. POOL supplies a formal way to specify interfaces; there is, however, no way to formalize protocols on a higher abstraction level than the code in the Implementation Units, which makes reasoning about their correctness (e.g. deadlock freedom) difficult.

3.2 Communication driven software engineering

For PRISMA/DB we decided to implement the communication protocols before actually implementing the components. The primary reason for this is, as stated above, that getting the protocols right is extremely important. This is even more so in our situation, where PRISMA/DB is implemented by a group of people working at 2 different locations. Integrating the work of several people may cause many problems, due to all sorts of misunderstanding. By integrating the "empty shells" of the components before adding internal functionality to them, we expected to have fewer problems at integration time. This approach has already proven to work. Assembling our very first working DBMS from its functional components only took less than a week.

The second reason for our strategy is a modeling one. The typical activity of a component can be described by the following pseudo-code:

```
initialize;
WHILE still_work_to_do
DO wait_for_request;
   receive_request;
   handle_request;       %% may include sending answer
                         %% or sending messages
                         %% to other objects
   internal_administration;
OD
```

As shown by code, the communication layer is the top layer of each component. Therefore, it seems natural to start here.

In PRISMA/DB we can distinguish various types of communication. Firstly, queries have to be transported through the system. A User Interface, a Parser, a Query Optimizer, a Transaction Manager, and some One-Fragment Managers each work on query execution; so, queries have to be sent down this chain of components. Secondly, if an error occurs during query execution, an error message has to be sent in opposite direction through the chain of components. Finally, all sorts of control messages have to be sent through the system (e.g. information from the data dictionary may be asked, the transaction manager has to request locks on fragments from the concurrency controller, transactions need a two-phase commit protocol to guarantee atomic execution of transactions).

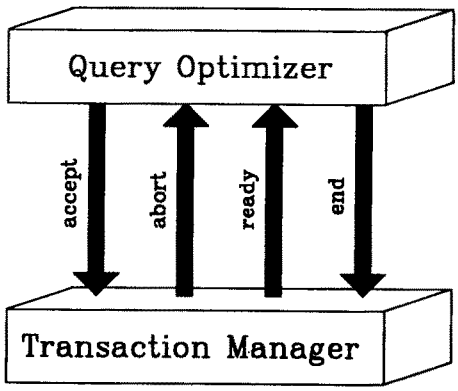

Figure 3: Interface between Query Optimizer and Transaction Manager

We use a uniform protocol for query and error handling, between all components involved, which reduces the number of different interfaces in the system. As an example, the protocol between the query optimizer and the transaction manager is described in the next subsection. Each type of control information needs its own protocol. The same issues are important in all these protocols, however, so one example will give a good idea of the communication in PRISMA/DB.

3.3 An example protocol

The interface between the query optimizer and the transaction manager consists of 4 methods (see figure 3).

- The query optimizer (QO) can send an optimized query to the transaction manager (TM), using the method *accept*.
- The TM can tell the QO that the transaction is aborted (due to some error or a concurrency control problem) using the method *abort*.
- The TM can tell the QO that the execution of a transaction is ready using the method *ready*.
- The QO can tell the TM that the transaction is ready using the method *end*.

These methods are asynchronous.

The protocol over (and rationale behind) this interface is as follows. In the normal flow of control (without errors or concurrency problems) the QO will (after creating a TM) start sending optimized parts of a transaction to the TM. The last statement of every transaction is "commit" or "abort", so the TM knows when a transaction ends. The QO may never use a method *accept* of the TM anymore after sending a "commit" or "abort" (also using *accept*).

If nothing ever went wrong, an interface consisting of one method would do here. But unfortunately errors do occur. In case of an error, the TM sends an *abort* message to the QO, which may still be busy optimizing the rest of the query. Some parts of the query may be on their way from the QO to the TM, while the *abort* message is sent in the opposite direction. The TM will have to wait for these messages before it can terminate.

For this reason the method *end* is added to the interface. *End* has to be the last message from the QO to the TM.

On the other hand, the QO must know when the execution of a transaction is ready, because it could get an *abort* message from the TM while the transaction is still executing. The TM sends a *ready* message to the QO after succesful completion of the transaction, so the QO knows no *abort* can come from this transaction anymore.

3.4 A protocol problem

The various protocols between objects have been designed in the way stated above. We are confident that all individual protocols are fine. But, if all protocols between the components are depicted in one picture, a complex structure emerges. Research was done in modeling the communication protocols in the DBMS as a Petri-net. This approach appeared infeasible though, because of the size of the resulting Petri-net, and because of the fact that if-then-else statements cannot properly be modeled in a Petri-net [Voor89]. Though this is considered a serious problem, it turns out that system deadlocks do not seem to occur anymore after an initial phase.

4 Parallelism

4.1 A simple taxonomy of parallelism in a DBMS

Before giving a discussion on parallelism, we start with a simple taxonomy of the forms of parallelism. Three different forms can be identified (figure 4):

multi-tasking By multi-tasking we mean the ability of a system to execute several independent transactions at the same time. This concept is closely related to the multi-tasking notion in operating systems.

pipe-lining The execution of a transaction is performed in a number of consecutive stages: parsing, query optimization, interpretation of the schedules, and execution at the lowest level of the DBMS. If a transaction is broken into pieces that are sent one after another through the consecutive stages of processing in such a way that several components are working simultaneously, we speak of pipe-lining.

task spreading If the processing of a (sub)transaction at a certain level is decomposed into a number of sub-processes that are executed at the same time, we speak of task spreading. Task spreading involves three steps: firstly, splitting the task into subtasks, creating processes for the execution of the subtasks, and sending the subtasks to the appropriate objects; next, parallel execution of the subtasks; finaly, sending the results of the subtasks to a central process and creating the final result.

We can identify two types of task-spreading (see for example [Kahn87]):

and-parallelism The result of the task is made up out of the results of all subtasks. Hence the task can be completed after all subtasks are completed.

or-parallelism The result of the task is made up out of the result of one or more subtasks. Hence the task may be completed before completion of all subtasks.

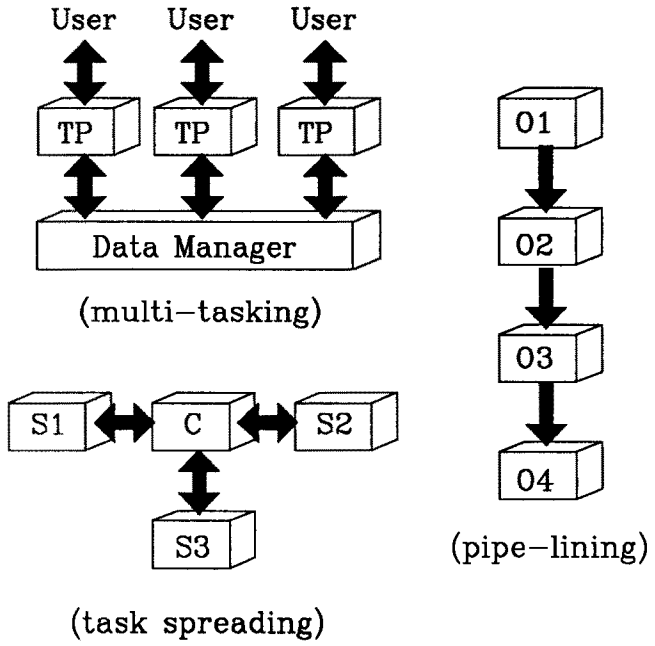

Figure 4: Forms of parallelism

Note that these concepts can be applied in a sort of recurrent way: within a subtask at certain level of the DBMS we can have pipe-lining again. A related description of parallelism in a DBMS can be found in [Bora85].

4.2 Modeling parallelism in POOL

The language POOL gives much freedom in exploiting parallelism. New parallel processes can be created by simply creating new objects each having their own local process. To make use of parallelism, however, we have to synchronize all these processes is some way. This can be achieved in two ways: the creation of new objects and the communication between objects. We briefly discuss this for the various forms of parallelism.

multi-tasking Each task is executed by a collection of POOL objects. The objects belonging to one task communicate with each other; there is no communication between objects that belong to different tasks. Note that all tasks use the central resources of the system, which are objects too. Therefore, the communication between an object O_1 belonging to a task and an object O_2 belonging to a central resource, should be designed in such a way, that O_1 cannot block or delay O_2. For this reason, synchronous method calls from O_2 to O_1 should not be used.

pipe-lining In pipe-lining the objects (processes) are arranged into a sequence $O_1 \cdots O_n$. Each object O_i in principle only needs to send messages to its successor in the pipe-line O_{i+1}. Because we want the objects to execute independent of each other in time, we need both non-blocking communication and some means of message

buffering between O_i and O_{i+1}; both concepts are offered by POOL asynchronous methods. The most natural way of creating a pipe-line is by letting an object O_i create its successor O_{i+1}.

task spreading Task spreading needs a coordinating object, say C. Defining n as the number of subtasks, this objects creates objects $S_1 \cdots S_n$ for carrying out the subtasks. After creation, each S_i gets its task via an asynchronous method from C. Upon completion of its task, each S_i sends its result to C. Note that synchronous communication between C and S_i is not suitable since this would not allow C to go on after having activated the first subtask.

4.3 Parallelism in PRISMA/DB

Looking at the architecture of PRISMA/DB at the highest abstraction level (the component level), we can easily identify all three forms of parallelism.

The *multi-tasking* occurs where several autonomous global query processes are running at the same time. A global query process consists of a User Interface, a Parser, a Query Optimizer, a Transaction Manager, and a set of One-Fragment Managers to hold intermediate results. All global query processes are independent of each other apart from sharing the same central resources of PRISMA/DB.

Pipe-lining is found in the sequence of components in the global query process as decribed above. Parts of transactions are transported and processed in a pipe-line manner through the various levels of the transaction processor. Further, pipe-lining is used in processing query trees at the One-Fragment Manager level. Here a Transaction Manager builds an execution structure out of OFM's to process intermediate results and channels for tuple transport. The layout of this structure is the same as that of the query tree. Processing is done in a data flow manner, resulting in a high degree of pipe-lining between the levels of the query tree.

Task spreading is possible at all levels of PRISMA/DB, so also within every component. It should only be used for computation-intensive work, however. The most obvious use here is at the lowest level, where the data storage is handled. Instead of having one manager for all the data in the entire database, we chose to have separate managers for fragments of relations. Each One-Fragment Manager process can be thought of as a subtask of the data manager task. Task spreading may also occur in a single component; an example could be the Query Optimizer, where we could split up the optimization process into a number of sub-processes each doing part of the optimization.

When looking at task spreading in a DBMS we notice that or-parallelism can hardly be ever used. An example of possible or-parallelism can be found in the standard two-phase commit protocol (see [Date83]). When we execute the local pre-commit decisions in parallel at the One Fragment Manager level, we can use or-parallelism in deciding about global commit; this is possible because a single negative pre-commit decision always results in a global abort. Note however, that we gain only in an abort situation here.

4.4 Logical versus operational parallelism

When talking about parallelism it is important to distinguish between two concepts:

logical parallelism By logical parallelism we mean the parallelism that we model into the software system; this is closely related to the modular structure chosen for the

system. The amount of potential parallelism here is only limited by the number of processes (objects) existing at the same time.

operational parallelism As soon as we execute a system on real hardware, it is obvious that there cannot be more processes running in parallel than there are processors in the system. The number of active processes running concurrently is called operational parallelism. It should be clear that the possible amount of parallelism here is limited both by the available number of processors and by the amount of logical parallelism.

Developing a large system in POOL gives the freedom to look at these two appearances of parallelism in truly independent steps. When designing system, attention is paid to logical parallelism only; concern is focused on a good software structure, the underlying hardware is not important. In a second stage of the development process, the allocation pragma's in POOL can be used to bind the various processes (objects) to processors in the system. By grouping objects on the various processors in the system, we determine which processes will be running in operational parallelism (on different processors) and which not (on the same processor).

Determining a good allocation of the objects of PRISMA/DB is difficult. Therefore, we need a great deal of experimenting with the allocation to obtain a high degree of parallelism and thus a good performance. This problem is discussed in the next section.

5 Experiments

As stated before, we plan to do experiments to gain understanding of the behavior of parallel systems. If the ratio between transmission costs and the costs for local processing is unknown, it is hard to decide whether sending some amount of work to another node for parallel execution will speed up some process as a whole.

The section on modularity showed the possibilities for a modular design of the database system in POOL. This modular design makes it possible to experiment with the allocation of components and to move certain subtasks from one component to another. The section on parallelism explains that logical and operational parallelism are independent in POOL. So, the design for the PRISMA database can be made with all sorts of possible parallelism in mind, from very coarse (e.g. multi-tasking) to very fine (e.g. task spreading in the optimization of one query). The possibility to move subtasks and to change the parallelism gives us a nice opportunity to experiment with the database system, to gain knowledge on parallel execution of programs and to get the best possible performance from our system. Below some examples of possible experiments are given.

5.1 Sequential versus Parallel execution of queries

Consider the following Relational Algebra expression:

$$(\sigma_{3="PRISMA"} EMPLOYEE) \bowtie_{5=1} (\pi_{1,6} CITY)$$

Calculating the result of this expression requires the calculation of a selection and a projection, and after those of a join. The selection, the projection and the join are each executed by a separate OFM. There are several schemes to organize this calculation.

- The calculation can be done sequentially on one processor. In this case there are no transmission costs, but the processor has to be shared among three OFMs. This scheme can only be chosen if the OFMs for base relations *EMPLOYEE* and *CITY* reside on the same processor.
- The calculation of the selection and the projection can be done in parallel on two nodes followed by the join on one of the two nodes that are already involved. In this case, the result of either the projection or the selection has to be sent to the other node.
- The calculation of the selection and the projection can be done in parallel on two nodes followed by the join on a third node. This gives an extra possibility for parallelism: the result of the selection and the projection can be pipe-lined to the join. In this case both the result of the selection and the result of the projection have to be sent to another node.

It is not at all clear which of the three schemes is most efficient. If the transmission overhead is high, the first one is probably best. The implementation of PRISMA/DB in POOL gives the possibilty to experiment. Going from one scheme to another only involves changing some allocation pragmas. Nothing else in the program has to be changed; the potential for parallel execution is available, but it does not need to be exploited. Once the behavior of such executions is understood, the query optimizer can be programmed to dynamically generate the right allocation pragma's.

5.2 An experiment with the boundary between components

As shown in figure 1, there is an interface between the Transaction Manager and the Concurrency Controller. In general, a TM is located on another node than the CC. If a TM wants to use the same fragment several times, it may repeatedly request the same lock. This strategy is simple for the TM: it does not have to do its own lock administration. Every use of a fragment requires transmission of a request over the network though. On the other hand, the TM may have its local lock administration. Before requesting a lock, it finds out whether it already holds a lock on the fragment. If so, no request needs to be sent over the network; if not, the lock has to be requested from the CC. This strategy saves transmission costs at the expense of some local overhead in the TM.

The implementation of the system in POOL gives us the possibilty to postpone the decision on such issues until after implementing the system. In the modular design it is easy to switch between the two ways of handling the problem. Our current implementation has a local lock administration in the TM. The central module of the TM has an interface with its lock administrator. It does not take much time to eliminate this object from the TM and adjust the interface of the CC to what the central TM expects from its local lock administration. Nothing has to be changed to the the central TM or to the main part of the CC.

These two examples show that the object-oriented approach allows us to postpone some of the design decisions or to changed them quite easily.

6 Conclusions and a Look into the Future

Currently we have a first prototype of PRISMA/DB running on an interpreter on a sequential machine. The status of this prototype shows that it is well possible to implement a complex DBMS in a modern (and even experimental) object-oriented language. POOL-X offers a programming environment that makes implementation of complex parallel processes possible in a reasonably short time. The parallel object-oriented programming environment has shown to be a good basis for modeling modularity, communication, and parallelism into a complex system. One of the strong points, is the flexibility of the system, which allows us to experiment with the architecture to study parallelism and to enhance the performance.

Our major concern at this time is the performance of the system on the target machine, which is still being developed. The high level features of POOL-X may cause some loss of performance when a straight-forward implementation of the language is used. Therefore, we expect that the performance of PRISMA/DB on the multi-processor machine will highly depend on the optimization qualities of a POOL-X compiler. However, we are confident that with the modular structure of PRISMA/DB, it is relatively easy to tune or adjust the system at a later step in this project.

Acknowledgements

We wish to thank the project members for providing a challenging environment and productive cooperation with the teams from Philips Research Laboratories Eindhoven, the University of Amsterdam and the Centre for Mathematics and Computer Science Amsterdam in the development of our DBMS. In particular, we wish to thank dr. A.J.Nijman for bringing academia and industry together, dr. H.H.Eggenhuisen for providing good project management and for stimulating the interaction between the various subprojects, P.America for his work on the definition of POOL-X, M.Beemster and J.v.d.Spek for their work on the realization of the POOL-X interpreter and finally the other members of the database group for their cooperation.

References

[Amer87] P. America, *An introduction to object-oriented programming*, Doc.nr. 364, Esprit Project 415A, Philips Research Laboratories, Eindhoven, The Netherlands, 1987.

[Amer88] P. America, *Language definition of POOL-X*, Doc.nr. 350, PRISMA Project, Philips Research Laboratories, Eindhoven, The Netherlands, 1988.

[Aper86a] P.M.G. Apers, J.A. Bergstra, H.H. Eggenhuisen, L.O. Hertzberger, M.L. Kersten, P.J.F. Lucas, A.J. Nijman, G. Rozenberg, *A Highly Parallel Machine for Data and Knowledge Base Management: PRISMA*, Doc.nr. 1, PRISMA Project, Philips Research Laboratories, Eindhoven, The Netherlands, 1988.

[Aper86b] P.M.G. Apers, M.A.W. Houtsma, F. Brandse, *Extending a Relational Interface with Recursion*, Proceedings of the 6th Advanced Database Symposium, Tokyo, Japan, 1986.

[Aper86c] P.M.G. Apers, M.A.W. Houtsma, F. Brandse, *Processing Recursive Queries in Relational Algebra*, in Data and Knowledge (DS-2), Ed. R.A.Meersman, A.C.Sernadas, Elsevier Science Publishers, IFIP, 1988.

[Aper88] P.M.G. Apers, M.L. Kersten, H.C.M. Oerlemans, *PRISMA Database Machine: A Distributed, Main-Memory Approach*, Proceedings of the International Conference on Extending Database Technology, Venice, Italy, 1988.

[Bora85] H. Boral, S. Redfield, *Database Machine Morphology*, Proceedings of the 11th International Conference On Very Large Data Bases, Stockholm, Sweden, 1985.

[Care88] M.J. Carey, M. Livny, *Distributed Concurrency Control Performance: A Study of Algorithms, Distribution and Replication*, Proceedings of the 14th International Conference on Very Large Data Bases, Los Angeles, USA, 1988.

[Date83] C.J. Date, *An Introduction to Data Base Systems Part II*, Addison Wesley, 1983.

[Hout88] M.A.W. Houtsma, H.J.A. van Kuyk, J. Flokstra, P.M.G. Apers, M.L. Kersten, *A Logic Query Language and its Algebraic Optimization for a Multiprocessor Database Machine*, Memorandum INF-88-52, University of Twente, 1988.

[Kahn87] K. Kahn, E.D. Tribble, M.S. Miller, D.G. Bobrow, *Vulcan: Logical Concurrent Objects*, Research Directions in Object-Oriented Programming, MIT Press, Cambridge, Massachusetts, 1987.

[Kers87] M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, H.J.A. van Kuijk, R.L.W. van de Weg, *A Distributed, Main Memory Database Machine*, Proceedings of the 5th International Workshop on Database Machines, Karuizawa, Japan, 1987.

[Morr86] K. Morris, J.D. Ullman, A.V. Gelder, *Design overview of the NAIL! system*, Stanford University, STAN-CS-86-1108 Stanford, CA, 1986.

[Tsur86] S. Tsur, C. Zaniolo, *LDL: A Logic-Based Data-Language*, Proceedings of the 12th International Conference on Very Large Databases, Kyoto, Japan, 1986.

[Voor89] L.v.d.Voort, *A qualitative analysis of the Prisma/DB Empty Shell*, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989.