TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

Giannis Bozas, Michael Jaedicke,
Andreas Listl, Bernhard Mitschang,
Angelika Reiser, Stephan Zimmermann

TUM-I9625
SFB-Bericht Nr.342/14/96 A
Mai 1996

TUM–INFO–05-96-I25-75/1.–FI

Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©1996 SFB 342 Methoden und Werkzeuge für
die Nutzung paralleler Architekturen

Anforderungen an: Prof. Dr. A. Bode
Sprecher SFB 342
Institut für Informatik
Technische Universität München
D-80290 München, Germany

Druck: Fakultät für Informatik der
Technischen Universität München

On Transforming a Sequential SQL-DBMS into a Parallel One: First Results and Experiences of the MIDAS Project

Giannis Bozas, Michael Jaedicke, Andreas Listl, Bernhard Mitschang, Angelika Reiser, Stephan Zimmermann
Department of
Computer Science
Technische Universität München
Arcisstr. 21, 80290 München, Germany
e-mail: {mitsch|reiser}@informatik.tu-muenchen.de

Abstract

One way to satisfy the increasing demand for processing power and I/O bandwidth is to have parallel database management systems (PDBMS) that employ a number of processors, loosely or tightly coupled, serving database requests concurrently.

In this paper we want to show an evolution path from an existing and commercially available sequential SQL database system to a parallel SQL database system. This transformation process is described from a software engineering and software reuse point of view emphasizing the system architecture. We report on first results and experiences gained while transforming the existing sequential system and constructing the new PDBMS. In order to show the viability of our PDBMS, a number of specific investigations that exploit this PDBMS testbed are presented as well.

1. Introduction

To fit the needs of nowadays complex database applications it is necessary to dramatically increase the performance of relational database systems. One of the enabling technologies perceived is parallelism. However, in the context of relational database systems, parallelism comes in several flavours with different impact on the database system architecture.

In this paper we report on the MIDAS project. MIDAS, the **M**un**I**ch Parallel **DA**tabase **S**ystem, is never meant to become a full-fledged PDBMS, but can be viewed as a testbed PDBMS that is well suited to serve as a platform for the exploration of various parallel database technology and its integration into database system architecture. The starting point of the MIDAS project was a commercially available sequential relational DBMS, the TransBase[1] SQL-DBMS. We reused the TransBase source code as the code basis for MIDAS and step by step integrated parallel database technology either by component (code fragment) exchange or by system extension.

Since both the sequential TransBase DBMS as well as the parallel MIDAS DBMS are supposed to run on general purpose workstations, symmetric multi-processor workstations, as well as on workstation clusters, we restrict our considerations to software-based parallelism only. That is, parallelism based on special hardware components (e.g. hardware issues in database machines [Bo89, Hs83]) does not belong to our current scope. As a result, we focus our studies and discussions on the following two types of parallelism:

- *Inter-transaction parallelism* that allows for a parallel processing of concurrent transactions, thus increasing transaction throughput.
- *Intra-transaction parallelism* that focuses on parallel query processing within a single transaction, thus reducing the response time of especially complex transactions.

Both types of parallelism are addressed in MIDAS in order to satisfy the requirements set by modern database applications. In order to achieve full compatibility with existing SQL applications, we decided to keep the application/database interface, i.e. the SQL-API, unchanged. Consequently our current investigations w.r.t. intra-transaction parallelism refer to intra-query parallelism only. Please note that inter-query parallelism (the other kind of intra-transaction parallelism) requires changes to the SQL-API (for example asynchronicity) in order to issue requests in parallel.

1.1 Our Approach Compared to Other Work

Nowadays parallel database management systems are becoming available on the DBMS market. Current system overviews can be found in the latest proceedings from the major database conferences [ACM95, VLDB95], database tutorials [Gr95], as well as in some tutorial-like journal articles [DG92, MMK90, Va93], and already in modern textbooks [Ra94].

There are basically two alternatives for PDBMS, the so-called revolutionary approach and the evolutionary one. Referring to the first approach, the PDBMS has to be developed from scratch. The other approach is to integrate parallelism into an existing sequential DBMS by means of system redesign and system extensions. Obviously, the latter alternative is preferred as far as the transformation costs and the performance of the resulting PDBMS fits the requirements and is not behind the development costs and the performance of a PDBMS that resulted from the from-scratch approach. The main benefit from the transformation approach is obvious, i.e., extensive software reuse and, if staged the right way, early availability because of the time savings compared to the time investment necessary for a from-scratch development.

1. TransBase is a trademark of [TB95].

The general objective of the MIDAS project is not to present yet another PDBMS approach, but to describe and discuss the methodology, system architecture, and implementation strategy for parallelizing an existing sequential relational DBMS. In that, our approach goes along the same lines of thought as is done by many DBMS vendors like e.g. IBM [BFG95, MPTW94], Oracle [Li93], Informix [Ge95], whilst many university endeavors towards PDBMS perform the from-scratch approach like DB3 [ZZB93], Prisma [Ap92], and especially like all approaches that have a strong relationship to database machines as for example Bubba [Bo90], Gamma [De90], or the Super Database Computer [Hi90].

Currently, MIDAS is designed and optimized to run efficiently on a shared-disk architecture consisting of a workstation cluster. Thus, MIDAS refers to the same class of architecture as Digital's Rdb [LARS92] on the original Digital VAXcluster, the IBM Parallel Sysplex [MN92], and the Oracle Parallel Server [Li93]. However, the majority of approaches to PDBMS (as for example Teradata [WCK93], Gamma, Bubba, Arbre [LDHSY89], EDS [Sk92], Prisma, Tandem's NonStop SQL [Ze90], and IBM's DB2 Parallel Edition [BFG95]) belong to the class of shared-nothing architectures and only few approaches (XPRS [HS93], DBS3, and Volcano [Gr90]) adhere to the class of shared-memory architectures.

MIDAS represents a shared-disk approach, since this type of architecture is perceived to provide for an easy migration from a uniprocessor system to a parallel architecture [MPTW94, Va93]. Furthermore, as a shared-disk PDBMS, MIDAS can (in accordance to [Va93]) be seen on one hand as a compromise between the limited extensibility/scalability of shared-memory systems and the load balancing problem of shared-nothing architectures. On the other hand, a shared-disk architecture can be viewed as the starting point for a new type of system architecture that combines the shared-disk approach with the shared-memory approach if the nodes are shared-memory multiprocessors. This kind of 'intermediate' architecture [Gr90, Va93] combines the advantages of both architectural concepts, whilst still being conceptually as well as implementationally simpler than the shared-nothing approach. A more detailed discussion on these various types of architectures and their pros and cons can be found in [MPTW94, Va93].

Being a database testbed targeted for research in PDBMS, we carefully crafted the MIDAS components in order to be able, in the long run, to adapt also to shared-memory, shared-nothing as well as to the above mentioned 'intermediate' architecture. Furthermore, we embedded the whole system into the **P**arallel **V**irtual **M**achine environment (PVM [Ge94]). This resulted into a high degree of portability, that is absolute necessary for the usefulness of a testbed system. The most important design decisions w.r.t. MIDAS are reported and discussed in the next sections. They further help in showing the differences between MIDAS and the other approaches mentioned over here.

1.2 Overview and Contribution

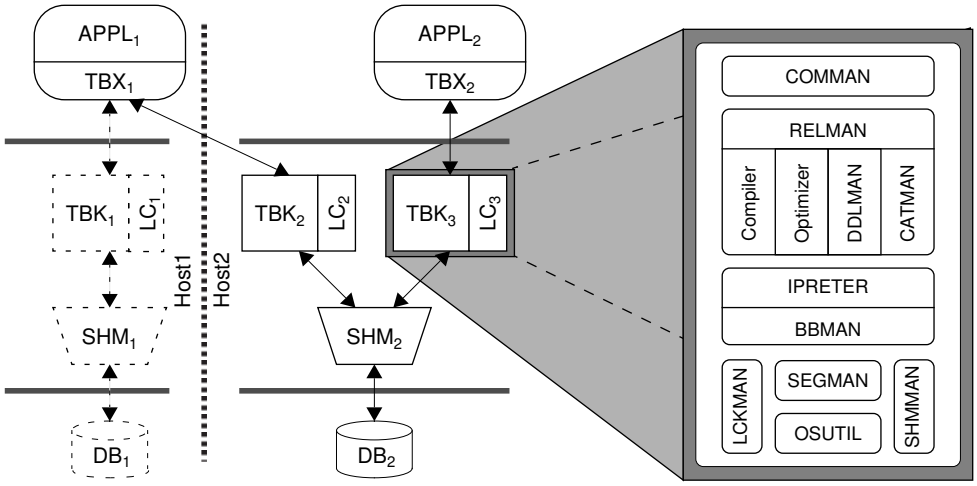
The focus of the MIDAS project is on investigating and understanding parallel database technology from a system point of view. In order to achieve this goal, we decided for a system approach starting with a sequential DBMS that got transformed to a PDBMS step by step. This evolutionary process is grounded on a deep understanding of database system modularization and system extensibility that will be described in Section 2. Section 3 focuses on one specific investigation of parallel database technology that was the first conducted in the MIDAS testbed, whilst other and remaining investigations, given in Section 4, will be just reported due to space limitations. Finally, Section 5 sums up giving some first results and experiences in transforming a sequential SQL-DBMS into a Parallel SQL-DBMS and gives a brief outlook to future work.

2. The Transformation Phase

In this section we report on the most significant system development steps that are encountered during the transformation of the underlying sequential SQL-DBMS TransBase to the PDBMS testbed MIDAS. Firstly, we give a system overview of TransBase emphasizing its modular architecture. Secondly, we introduce the MIDAS PDBMS detailing its process and component architecture. Additionally, the current state of the implementation is given and the transformation process from TransBase to MIDAS is described from a software engineering and software reuse point of view.

2.1 The Sequential SQL-DBMS TransBase

TransBase is a the well-structured, modular designed, sequential SQL-DBMS, which is built according to commonly accepted database architecture principles [St94]. It realizes a relational multiuser, multidatabase, remote access database system, whose overall architecture is depicted in Figure 1.



a: Database Process Structure (Multiuser, Multidatabase, Remote Access Aspects)

b: Database Kernel

Figure 1: TransBase Architecture

As can be seen in Figure 1a, the database (DB) itself is placed on the external memory (UNIX file system), with each relation stored in a file. A resident shared memory region (SHM) is installed in main memory for every database under user access. This shared memory not only serves as a database cache for buffering data pages but also contains information about synchronization of concurrent transactions and catalog data. For every application (APPL) accessing local or remote databases, a corresponding TransBase kernel (TBK) will be started on that particular host managing and controlling the database access. The communication interface between the application and the kernel is provided by the TransBase exchange (TBX) module. A local cache (LC) is related to each kernel buffering intermediate results, meta data, and hot spot data.

The TransBase kernel (TBK) consists of several hierarchically layered modules as shown in Figure 1b. On top of the hierarchy is the communication manager COMMAN, which controls the communication with the application program. A user request is directed to the relation manager RELMAN, which passes it on to its submodules depending on the type of request. System information about tables, columns, views and so on are hot spot data and therefore are managed in an efficient manner by the catalog manager CATMAN in the local cache LC. The data-definition-language manager DDLMAN is responsible for the creation/deletion of database objects (rela-

tions, views, indexes). A data-manipulation request (retrieval-/update-query) is compiled at first into a procedural expression of an extended relational algebra by the compiler module *Compiler*. This resulting operator tree is transformed by the *Optimizer* applying various rule-based optimization techniques. At last the optimized operator tree will be processed by the interpreter IPRETER using the well-known access mechanisms of its submodule BBMAN. Records are managed by the B-tree manager BBMAN storing a relation in a prefix-B*-tree and making available efficient sorting methods both for prefix-B*-trees and sequentially organized files. The segment manager SEGMAN controls segments being page-structured, pseudo-linear address spaces as containers for relations, which are mapped to files; additional tasks are physical I/O-control, cache management and recovery. The lowest module OSUTIL simply serves as interface to the underlying operating system and especially imitates UNIX-specific system calls when porting the system to other operating systems. Further on there are two other modules, the shared memory manager SHMMAN and the lock manager LCKMAN. The former manages and controls its part of the shared memory, the latter guarantees the synchronization of concurrent transactions.

2.2 Transformation to a Parallel DBMS

In a first, very early step, we transformed the sequential TransBase to run on a parallel computer. We used the iPSC/2 [INTEL89] which had 32 processors with local memory all connected via a high bandwidth hypercube interconnect. The main component which had to be redesigned when processing queries on several processors with distributed memory was the database cache, i.e. the shared memory region (SHM, see Figure 1a). In order to implement this distributed cache, we used the MMK (Multiprocessor Multitasking Kernel) [MMK90] which was developed by a neighboring research project at the same university. MMK is a programming model based on communication via messages and has been implemented on top of Intel's operating system NX/2. The experiences in using MMK on iPSC/2 were very discouraging, because of poor performance and system instability mainly due to iPSC/2-specific hard- and software failures.

To overcome system instability, we resorted to a cluster of workstations which share disks via NFS. MMK was ported to this environment, now called MMK/X. Before implementing the distributed database cache, we made a case study [LSF94], examining different programming models on distributed memory multiprocessors. It turned out that for our purposes in PDBMS the message passing library PVM (Parallel Virtual Machine, [Ge94]) was most suitable. Among the main advantages of PVM are the retention of the TransBase process-structure (UNIX-processes) and its support for UNIX shared memory. Additional decisive characteristics in favour of PVM are its efficiency and flexibility combined with high portability and the available system management and administration tools. Further on, the possibility to dynamically start new PVM tasks helps to implement a scalability feature.

The negative experiences with both a proprietary system environment (MMK and MMK/X) and a (kind of) proprietary parallel computer (iPSC/2 which, by the way, vanished from the market) forced us to rethink our endeavor towards a PDBMS. As a result from that we decided to develop a PDBMS testbed, i.e. MIDAS, that is supposed to run on general purpose workstations, symmetric multi-processor workstations, as well as on workstation clusters. We further restricted our considerations to software-based parallelism only.

MIDAS Prototype and Architecture

MIDAS realizes a client/server architecture[LPRBL95] as depicted in Figure 2. The MIDAS clients are the database applications. They are sequential programs performing transactions on the MIDAS server. A MIDAS client can run on any computer having access to the MIDAS server.

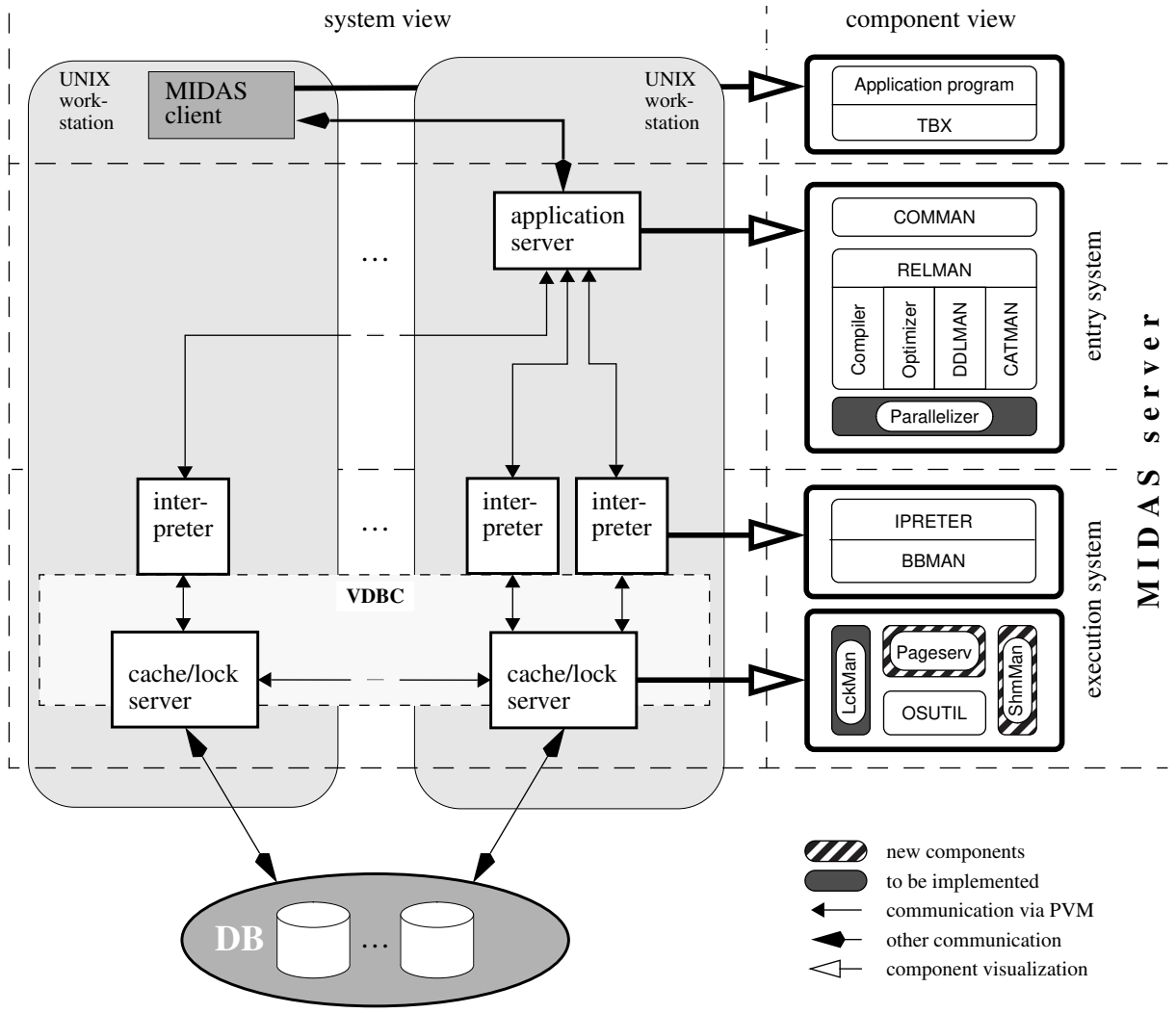


Figure 2: MIDAS Prototype

Parallel processing in MIDAS comes as real inter-transaction parallelism and not as the kind of pseudo-parallelism that comes from interleaved processing of concurrent transactions. Inter-transaction parallelism is simply achieved by assigning database kernels (formerly in TransBase the TBK) to multiple nodes in the workstation cluster. However, the main parallel processing feature in MIDAS is intra-query parallelism. Here, a query is processed exploiting multiple interpreters being independent processes that access the database via the shared distributed database cache (**V**irtual **D**ata**B**ase **C**ache VDBC, see Figure 2) of MIDAS. The interpreters processing portions of the same query communicate via the VDBC, i.e. via the exchange of buffer frames. Hence, one of the main problems to be solved when designing the VDBC is coherency. A detailed discussion of our solution to this can be found in Section 3.

The MIDAS server is composed of two layers, the MIDAS *entry system* that corresponds to the logical database processor and the MIDAS *execution system* that corresponds to the physical database processor as introduced in [Fr87]. All components of the MIDAS server are implemented as a set of PVM tasks. All MIDAS clients are conventional UNIX processes. We decided not to implement the clients as PVM tasks, because we wanted to protect the database against unauthorized access.

MIDAS entry system The MIDAS *entry system* supports inter-transaction parallelism, i.e. parallelism between different MIDAS client applications. For that purpose, the entry system consists of a varying number of *application servers* that provide a mechanism such that an arbitrary and varying number of clients can issue

their queries to the MIDAS server in parallel. Whenever a new MIDAS client arrives, the MIDAS server can increase its potential for parallelism by starting new PVM tasks within the MIDAS server. These new PVM tasks are dedicated to the additional work caused by new MIDAS clients. In particular, any client has a MIDAS application server directly associated, which is exclusively at the MIDAS client's disposal.

An *application server* receives queries from its client. The queries are purely descriptive (SQL) and do not contain any parallel constructs. A query gets compiled, optimized, parallelized, and finally transformed into a **P**arallel **Q**uery **E**xecution **P**lan (PQEP) which can be performed by the execution system in parallel. The application server initiates, schedules, and controls the parallel execution of these PQEPs. For that purpose, it (dynamically) starts and removes *interpreters* belonging to the MIDAS execution system. Finally, the *application server* sends back the results produced by the execution of the PQEPs.

MIDAS execution system The MIDAS *execution system* is responsible for intra-query parallelism. It is achieved by the capability of the execution system to work on different parts of one PQEP simultaneously. Interim results can be exchanged between the concurrently running *interpreters* involved in the parallel processing of a PQEP. Furthermore, the MIDAS execution system comprises the VDBC that provides to all concurrently running *interpreters* efficient access to the physical database stored on nonvolatile memory like magnetic disks.

The MIDAS execution system is designed as a set of *cache/lock servers* and a set of *interpreters*. The *cache/lock servers* constitute the VDBC. They are embedded into PVM tasks and belong to the static part of MIDAS, since their number is constant. Exactly one *cache/lock server* is running on each node of the virtual machine, i.e. the workstation cluster. The *interpreters* are PVM tasks as well and belong to the dynamic part of MIDAS. Their actual number depends on the number of PQEPs concurrently performed by the MIDAS execution system and the chosen degree of parallelism for each PQEP. They are started and removed by the *application servers* as needed. Their job is to execute one PQEP in parallel. Each *interpreter* works on one part of a PQEP which is assigned and sent to it by the *application server*. A PVM process-pool can be used to reduce the overhead for dynamically starting and terminating processes.

The *cache/lock server* implements a database cache and provides to its local *interpreters* efficient and transparent access to all database pages. Each *cache/lock server* installs a UNIX shared memory on its local node. This UNIX shared memory makes up what we call the local database cache (LDBC). The sum of all local database caches forms the virtual database cache VDBC. If an *interpreter* requires access to a database page during the execution of its portion of the PQEP, it sends a request to its local *cache/lock server*. The *cache/lock server* puts the desired page into the local database cache and the requesting *interpreter* is informed. Using an LDBC on each node of the virtual machine causes the well-known cache coherency problem. It is solved by using a variation of the invalidation approach combined with dynamic page ownership to guarantee weak cache coherency. The communication between the *cache/lock servers* which is necessary to realize cache coherency and concurrency control as well is done by means of PVM messages. A detailed discussion of our coherency algorithm and its realization into the MIDAS testbed is given in Section 3.

2.3 Current State of Implementation and Software Reuse

This section details the current state of the MIDAS implementation and the transformation process from TransBase to MIDAS is described from a software engineering and software reuse point of view as exemplified in Table 1.

The MIDAS system provides an application programming interface (API) to the MIDAS clients. This API consists of a set of routines included in the MIDAS library, which is to be linked by the application programs. The API implements the connection from the client to the *application server* and is completely taken from original TransBase offering unchanged functionality. With this we could accomplish an important legacy system (or migration)

module	lines of code	original	changed	new
application server	46000	82 %	8 %	10 %
interpreter	26000	77 %	6 %	17 %
cache/lock server	16000	12 %	3 %	85 %
all components (incl. administration tools)	93000	63 %	7 %	30 %

Table 1: Code Size and Code Reuse in MIDAS

feature, i.e. every application written for TransBase and the TransBase API now runs unchanged in the MIDAS environment as well; old TransBase applications only have to be relinked, but need not any recompilation. Some diagnostic services were added to the API to check and test the new interfaces between processes in the MIDAS system.

At the moment the *application server* is able to connect to the lower layers of the MIDAS system and to receive queries from an application program. These queries are compiled using the original TransBase compiler, TransBase optimizer, DDL manager DDLMAN as well as the catalog manager CATMAN from TransBase. In case of a retrieval or DML operation the result of this transformation is an operator tree which is passed to the next component, the *interpreter*. To enable this only a few changes had to be done adapting the TransBase source code to MIDAS. Next step of implementation is the development of a parallelization component (the *parallelizer*) as part of the *application server*, which takes a non-parallelized operator tree as input, splits it into parallel executable subtrees and passes these subtrees to multiple *interpreters* for parallel execution.

The task of the *interpreter* is to execute the operator tree received from the *parallelizer* of the *application server*. Since the general implementation of these trees hasn't changed, all routines to act on nodes of the operator tree were directly taken over from TransBase. They only had to be completed by some new routines. Due to the newly created process border, new routines for receiving operator trees from the *application server* and sending results back had to be implemented. Similar communication routines were built between *interpreter*s as well as between an interpreter and its local *cache/lock server*. The *cache/lock server* had to be built almost from scratch, since it is responsible for the realization of the virtual shared memory concept of the VDBC that, of course, didn't exist in TransBase. More detailed information on the *cache/lock server* as well as on query processing and evaluation is given in Section 3 and Section 4, respectively.

Currently, MIDAS supports inter-transaction parallelism. This is achieved through the possibility of running several *application servers*, each equipped with one *interpreter* on different nodes in parallel. Since application programs are able to connect to these *application servers*, their requests can be executed in parallel, as the *application servers* run in parallel.

Since code reuse is a major point of interest in this project, we documented the amount of software reuse comparing the old TransBase code with the MIDAS components. Table 1 shows the current size of the MIDAS modules. In addition to the amount of lines of C code (LOC) in each module, the columns three to five give the percentage of code that is original and unchanged TransBase code, TransBase code that had to be adapted to MIDAS, or newly implemented code.

Remarkable is the low percentage of newly implemented code for both the *application server* and the *interpreter*. Almost every code fragment was taken over from TransBase. Mainly communication mechanisms had to be added, since all modules are embedded into PVM tasks and communicate via PVM as well. Due to the necessity of new components, the percentage of reused code in the *cache/lock server* is quite low. Only the file-management component OSUTIL of TransBase could be reused.

3. A First Testcase for the MIDAS Testbed

In this section we describe some of the work done for the development of MIDAS's *cache/lock server* in detail and show some interesting results. Because of the importance of this component in a shared-disk architecture, we decided not to directly implement new concepts into the prototype, but to determine their performance first, using a simulation system. For this purpose we built an extensive simulation system, called DBSIM. In the following, we first motivate and introduce the new concepts we developed for the MIDAS *cache/lock server* in Subsection 3.1, then report on the simulation results performed in Subsection 3.2 and finally present the resulting low-level MIDAS architecture in Subsection 3.3. Subsection 3.4 concludes this section.

3.1 First Steps Towards Making a Scalable, Efficient and Flexible Cache Manager

When we started designing the MIDAS *cache/lock server* it was clear that the reuse of existing TransBase code for these components would be minimal. Building MIDAS's *cache/lock server*, we had to replicate the simple TransBase cache manager (SEGMAN and SHMMAN) in each node of the parallel system, rewrite and extend most of its routines and add a whole new bunch of routines to take care of the problems that come up due to the shared-disk approach (invalidation/update propagation). In order to achieve the scalability goal, we had to distribute the lock manager too and take care of the distribution issues like communication, deadlock detection, etc. Furtheron logging and recovery concepts had to be adapted as well. All these issues and their details are beyond the scope of this paper. Here we restrict our discussions on the design of the MIDAS cache coherency algorithms as well as their performance.

Rewriting the biggest part of the Transbase cache/lock component, we decided that more efficient algorithms could be implemented at no extra cost. But, should we use algorithms proposed in the literature or should we design our own? In this area, work has been done the last ten years by numerous researchers with very interesting results. A good survey for cache/lock management algorithms for shared-disk database systems can be found in [Ra91]. Performance evaluation for some of the algorithms described there has been done using simulation systems. In all those studies very small systems were considered (1-4 nodes). Since for our project, scalable performance is one of the major goals, our studies refer to larger system sizes. Another important goal for us is support for intra-query parallelism and, again, there are no studies known (to the authors) that investigate cache/lock management under intra-query parallelism. That's why we decided to test some of the proposed algorithms before implementing them into the prototype. For this reason we developed a simulation system called DBSIM. Using DBSIM we could find out that the performance of many of the proposed algorithms suffered when scaling the system size. The limit of acceptable performance was often as low as 16 nodes, which is definitely not enough for a modern parallel database system. That's why we developed a new set of algorithms that show better scalable performance and support for intra-query parallelism.

The main idea of the new algorithms is to divide cache pages into smaller units, which we call subpages, and use them instead of pages to maintain cache coherency and concurrency control. However, we still use the relatively large I/O exchange unit (I/O page), which has the same size as a cache frame, thus keeping I/O efficiency. Cache coherency is maintained at the subpage level, exchanging the smaller subpages between the nodes, thus using the network more efficiently. Locking is done at the subpage-level as well, giving us a much smaller lock granularity and thus less conflicts. The algorithms are very flexible and adaptive as they allow different subpage sizes for different relations. In setting the number of subpages appropriately, we can easily adapt to different access profiles for different relations, It was clear that we had to develop a new locking scheme for these caching protocols and that is exactly where one part of our interest is concentrated now.

3.2 Simulation Results

In this section we present some results of simulation studies we conducted and show, that our subpage-based algorithms perform and scale better than conventional full-page algorithms. We give a brief description of the simulation model used and present some interesting, representative results.

3.2.1 Simulation System

Our simulation system DBSIM is a trace-driven and event-based simulation system which simulates the hardware (CPUs, memory, disks, network) as well as the transaction processing algorithms of a shared-disk database system. At the moment only the low-level layers of the system (cache/lock server, transaction manager and some scheduler functions) are simulated in detail. More details about the DBSIM internals can be found in [LB95].

DBSIM reads its input from trace files, which can be produced either from TransBase, running the multiuser Wisconsin Benchmark, or from a tool we developed, called LoadGenerator, which generates synthetical traces. LoadGenerator is a very flexible tool which allows the user to define classes of transactions, read/write ratios of their accesses, probability of hot-spot data access, etc. Using LoadGenerator we were able to produce various kinds of workloads capturing the characteristics of different (run-time) environments. The results presented in the next subsection refer to an OLTP workload created by LoadGenerator. This workload consists of different kinds of short transactions (they access 13 database pages on the average). 58% of them are update transactions which together change 35% of all referenced pages. 5.5% of the transaction accesses are done to hot-spot pages (catalog data etc.). A very small number of transactions (3%) change hot-spot pages. We had a 200 MB large database, 4KB pages, an 8MB database cache per node and let 2048 transactions run through DBSIM. Table 2 shows the parameters we used for the simulations.The maximum number of transactions which run on one node concurrently is determined by the multiprogramming level (MPL) parameter. The meaning of the parameters should be self-explanatory. The network we used had a capacity which scales logarithmically with the number of nodes in the system. This is a common property of network usage. With capacity we mean the maximum number of messages which can be sent across the network simultaneously without blocking. Each node had a CPU and two disks: one for the database and one for the log. Since we wanted to get valuable simulation results, we considered a simple logging scheme based on subpages.

3.2.2 Performance Results

Our main interest was to test the efficiency and scalability of our algorithms and to compare them with their page-based counterparts. We were able to do that by varying the number of subpages per page. Of course, one subpage per page mimics the paged-based case. We did various tests, varying the number of subpages per page, the system size (number of nodes), the problem size (number of transactions in the system), transaction workloads, etc.

The results we present in this subsection refer to the OLTP workload described in the previous subsection. We had 1, 4, and 16 subpages, scaled the system size from 1 to 64 nodes and kept the problem size constant (2048 transactions). That means, we did a speedup test.

Figure 3 shows the speedup results for the OLTP workload by displaying the throughput measured in number of transactions per second (tps) versus the number of processing nodes. Thereby the graph consists of three curves, showing the throughput using 1, 4 and 16 subpages respectively.

Looking at Figure 3, one can easily see that the throughput obtained with 16 subpages is significantly higher than the throughput using 4 or 1 subpages. 16 subpages lead to the best results and 4 subpages still show an improvement over the 1 subpage case. To give an explanation for this encouraging behavior, let's take a look

parameter	value
number of nodes N	1, 2, 4, 8, 16, 32, 64
multiprogramming level MPL	4
CPU performance	30 MIPS
number of instructions for: begin of transaction end of transaction access to a record locking operation send/receive a message I/O process a message local communication	 25000 25000 10000 3000 5000 3000 1000 3000

parameter	value
disk parameters: bandwidth average access time	 2 MBytes/sec. 15 ms (5 ms for logging)
network parameters: bandwidth message length latency	 20 MBytes/sec. 256 Bytes 40 µs per message
local cache size	4% of DB size
page size	4 Kilobytes
number of subpages per page	1, 4, 16

Table 2: Simulation Parameters

at Table 3, which displays additional results for the 32 node case. There we can see that the number of lock conflicts during transaction processing decreases as the number of subpages increases. In other words, dividing a page into 4 resp. 16 subpages leads to a smaller number of lock conflicts per transaction than using only 1 subpage. Thus, when using 1 subpage a transaction has to wait in the average longer for a requested lock to be granted (locking delay) than in the 4 resp. 16 subpages case. Short locking delays lead to a better utilization of the processing nodes; that is why 4 resp. 16 subpages show higher transaction rates. Secondly, dividing a page into subpages decreases the average size of messages over the network. Short messages have the advantage of a high throughput on the network, which positively effects the transaction rate. When the number of messages increases beyond a point where the network bandwidth becomes the bottleneck then the positive effect of short messages on the transaction rate is weakend.

Furtheron, Table 3 shows that we pay something for these gains. We can see that dividing a page into 4 (16) subpages leads to 1.52 (1.83) times more concurrency control messages compared to not dividing a page which equals to the 1 subpage case. There is a simple explanation why this happens: at the moment DBSIM locks every subpage explicitly using the RAX protocol [BHR80]. As locking scheme we use the primary copy algorithm [Ra86]. That means, we need two messages each time we want to lock a subpage, if the corresponding lock authority is not at the local node. Therefore, having many subpages per page leads to many lock requests that can't be satisfied locally and thus to many concurrency control messages. An increasing number of concurrency control messages decreases the throughput and sustains the network. So what we see is a trade-off of concurrency control messages and lock-conflicts. Increasing the number of subpages per page, lowers the lock-con-

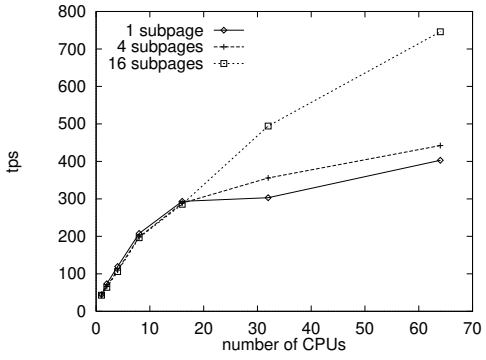


Figure 3: Speedup Results for a an OLTP Workload

average number of concurrency control messages per transaction, in case of	
1 subpage:	18.51
4 subpages:	28.25
16 subpages:	34.01
average number of lock conflicts per transaction, in case of	
1 subpage:	1.03
4 subpages:	0.78
16 subpages:	0.35

Table 3: Number of Messages and Lock Conflicts Using 32 Nodes

flict rate but increases the number of concurrency control messages. When the system grows, lock-conflicts impede its scalability and that is exactly what we are avoiding with the subpage algorithms. The disadvantage of producing more concurrency control messages can to a great part be overcome using a customized hierarchical locking scheme. This will substantially reduce the number of concurrency control messages of our algorithms. Developing a customized locking protocol is a main point of our current work.

To conclude, in this subsection we have seen that there are performance gains, and improved scalability when using subpages. Undoubtedly a new, customized locking scheme will improve performance even further and make our approach more adaptive to low-conflict rate workloads.

3.3 VDBC

In this subsection we present the MIDAS cache-management component architecture (**V**irtual **D**ata**b**ase **C**ache), which resulted from spending a serious amount of time with DBSIM to fine-tune the new algorithms.

3.3.1 The Architecture of VDBC

Figure 4 shows the architecture of the VDBC: Every node in the system contains a *cache/lock server* (Pageserv, LckMan and ShmMan) and a local database cache (LDBC) in its local memory. Hence, VDBC is constituted by all *cache/lock server*s and the local database caches. Each *cache/lock server* can interact with multiple interpreters on the same node concurrently through the VDBC interface. The Pageserv module of the *cache/lock server* is responsible to provide this interface, which is subpage-based. Pageserv modules are responsible for maintaining cache coherency, thus exchanging database pages and their subpages between the nodes.

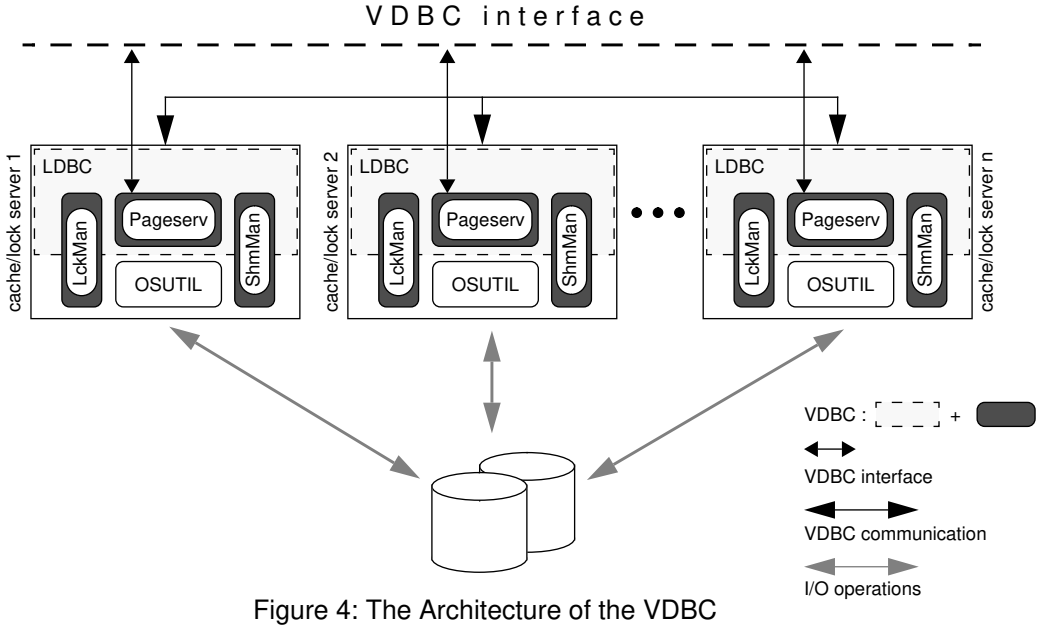


Figure 4: The Architecture of the VDBC

3.3.2 Maintaining Cache Coherency

Whenever a database page is needed at a node, the node gets its own copy in its LDBC. In order to provide a coherent view of the copies in the LDBCs, VDBC makes use of two different concepts: the *page owner* and the *page administrator*. Furtheron, VDBC's invalidation mechanism and the fixing of subpages are detailed as well.

Page owner. We characterize as page owner that Pageserv which has the page buffered in its node's LDBC and the exclusive right to write that page onto disk. Whenever a page is accessed for the first time it has to be loaded into VDBC from disk. The requesting Pageserv buffers the page in the LDBC and becomes owner of that

page. The page owner privilege is not bound to this Pageserv but is transferable through the replacement policy (cf. Section 3.3.3.). Thus we use a dynamic page ownership approach.

Page administrator. Because of the dynamic ownership scheme we have to keep track of the current owner of a page. For that, we partition the page space disjointedly among all Pageserv modules using a predetermined hash function. Each Pageserv maintains a directory with owner information about its part of the page space. The Pageserv holding the owner information for a page is called its administrator. That means, we use a fixed distributed directory scheme for maintaining page owner information.

Fixing subpages. As mentioned before, the interface between the VDBC and the interpreters is subpage-based. The main services provided at this interface are fixing and unfixing a subpage. In a frame of a LDBC a whole database page is buffered and the corresponding subpages can be either *valid* or *invalid*. Whenever a fix request for a subpage arrives at the VDBC interface of a node, the following different cases have to be considered:

(1) The page containing the requested subpage is not buffered in the node's local cache. In this case Pageserv on that node determines the page administrator and asks for the owner of the page. The owner gets the request and sends the desired page back to the requesting Pageserv. If the page is not in the VDBC at all, the administrator instructs the requesting Pageserv to load the page from disk and to become its owner. In every case all subpages in the received page are in the state *valid*.

(2) The page containing the requested subpage is buffered locally and the subpage's state is *invalid*. As in case (1) Pageserv will ask the administrator for the owner of the subpage which is the owner of the corresponding page. In our approach it is always guaranteed that the owner has a *valid* copy of the subpage [Li94]. That means another node must be the owner of the subpage, since the subpage's state at the requesting Pageserv is *invalid*. Therefore, the administrator will instruct the subpage owner to send the requested subpage to Pageserv. Receiving the subpage, Pageserv replaces its *invalid* copy with the *valid* one and proceeds with the fix operation.

(3) The same as (2) but the subpage's state is *valid*. In this case Pageserv can satisfy the request locally.

Invalidation mechanism. In a shared disk system changes made by a committed transaction must be visible at all nodes. In our approach we invalidate remote copies of changed subpages during commit processing and before unlock operations are performed.That means, Pageserv on the committing node sends each changed subpage to its owner at commit time. The owner is then responsible for doing the invalidation. For that, the owner uses a special data structure, called *copy set*, where it stores all nodes having copies of the corresponding page. Using the copy set, the owner is able to perform an invalidation operation without using broadcast. It sends invalidation messages only to the members of the copy set. Thus, we use a multicast invalidation approach.

3.3.3 Replacement Strategy

As mentioned before, VDBC uses subpages as the unit for fixing and for maintaining cache coherency, whereas as replacement unit it uses pages. Assuming that the size of a page is equal to the block size of the disk, writing a page to disk is much faster than writing all subpages of a page individually to disk. Whenever a local cache is full and we need a free frame in there, we have to decide which page can be replaced. Since VDBC should support high transaction rates and short response times by reducing the physical I/O costs during normal processing to a minimum, we keep modified pages and therefore their subpages in the database cache at least until the end of the transaction. This corresponds to a NOSTEAL approach [HR83] and allows us to limit operations for aborting a transaction to main storage accesses only.

Our approach is based on the fact that interprocessor communication is more efficient compared to disk I/O. Therefore, the main strategy is to keep locally purged pages in the VDBC as long as possible. There are two different cases to consider: owner pages and non-owner pages. In the latter case we simply purge out the page without any other action, because we can be sure that the VDBC still has a valid copy of the page (at the owner node). In the former case, copies of the page may exist in other nodes. In this case, we purge the page and the page-owner privilege to a node that owns a copy. For efficiency reasons, we send the page to the administrator and let him asynchronously search for a new owner (using the copy set information). If the search does not succeed, the administrator can keep the page and become its new owner, or throw it away (e.g. if its buffer is full). Since VDBC uses a NOFORCE approach, a page chosen for replacement may contain more recent data than the corresponding page on disk. In that case, if the replacement happens at the owner site, the page has to be written to disk before we can send it to the page administrator for new owner search. Details can be found in [Li94].

3.4 Conclusion

In this section we have shown that VDBC as it is now implemented in MIDAS is a carefully designed and (partially) performance-tuned cache manager. Its subpage-based approach offers greater flexibility, scalability and performance than page-based approaches. Further it turned out that DBSIM is an important tool that allows to firstly test promising scenarios before implementing them into the prototype. Currently we are using DBSIM to evaluate hierarchical locking protocols optimized for VDBC's subpage-based algorithms.

4. Current and Future Testcases

As already reported above, we have reached our first goal in the development of the MIDAS prototype, namely the support of inter-transaction parallelism. In addition to analyzing the performance of this inter-transaction parallelism, we aim at effective for intra-query parallelism in MIDAS. To reach this goal we already integrated some mechanisms into the prototype. In TransBase the complete query processing was done by a single kernel process. We split this into two process classes, the *application servers* realizing the logical database processor and the *interpreters,* serving as the physical database processor. By means of this new process structure we are now able to execute a single query by several *interpreters*. The VDBC is readily suited to support a set of *interpreters* to execute a single query in parallel on different nodes of the system. So VDBC isolates the *interpreters* from distribution aspects.

The realization of intra-query parallelism will subsume three main steps, namely the development of a parallelizing component, a well suited transaction model, and some new communication features.

Query optimizer/parallelizer The main problem within the logical database processor concerning intra-query parallelism is the generation of an optimized PQEP. To stay with our transformation and reuse approach we divided this plan generation into two phases [MPTW94]. The first phase consists of query compilation and generation of an optimized sequential QEP. This is already done by the existing TransBase components. A second phase will be implemented as a new component of the *application server*, the *parallelizer.* Its task consists of splitting the operator tree represented by the sequential QEP into several subtrees (see Figure 5). Each of these operator subtrees is sent to one or more *interpreters*. The latter case becomes important, as soon as data parallelism is employed during query execution. The *interpreters* run in parallel on different nodes or on the same node. There are many important aspects of the parallelizer that can be investigated using MIDAS. We plan to examine this two-phase approach. We are quite optimistic that a two phase plan generation will match our requirements since MIDAS is a shared-disk architecture. Compared to a shared-nothing architecture we do not

have to care about data allocation in the optimization phase because all data is directly accessible from every node. We also intend to investigate the different kinds of parallelism (e.g. data parallelism, pipelining) making extensive use of MIDAS's testbed feature. Finally we will investigate scheduling strategies optimizing the parallel query execution. In this context, dynamic choice of PQEP-alternatives, dynamic load balancing and the use of a special transaction model for parallel query execution have to be considered.

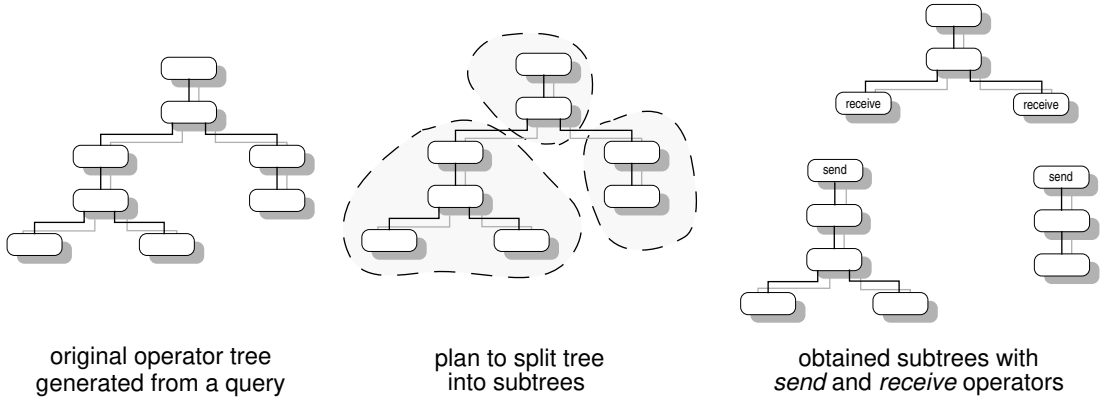


Figure 5: Splitting an Operator Tree

Transaction model We plan to specialize the generally applicable nested transaction model [GR93] to our needs. Our adapted transaction model views subtransactions as a means to structure and coordinate the parallel execution of operations in MIDAS. Fault tolerance is another important requirement for parallel query processing, that will be a main topic of our investigations.

Transfer of intermediate results between *interpreters* Each subtree being sent to one *interpreter* in the context of a subtransaction, has to be equipped with special communication operators. These operators manage the transfer of intermediate result tuples between the parallel running subtree evaluations. The root of each child tree is represented by a *send*-operator which passes its results to a *receive*-operator at a leaf of the parent tree (see [Gr90] for an implementation of a similar communication operator, called exchange) as seen in Figure 5. The result of the execution of each subtree is viewed as a temporary relation. So, *send* and *receive* are implemented to write respectively read from segments. These are special segments called communication segments that are handled by the *cache/lock server* and transparent to the *interpreters*.

Communication segments in VDBC Communication segments used by *send* and *receive* will have special semantics with respect to normal segments especially concerning their concurrency requirements. Moreover, communication segments are known only to the communicating *interpreters*. That means for the VDBC that coherency control, locking, as well as logging/recovery algorithms can be adapted to the needs of communication segments in order to increase efficiency. Currently only communication segments for one *send*/*receive* pair are implemented, but extensions to n:m-communication are under way. We are very optimistic that the VDBC algorithms can be easily adapted to efficiently support various kinds of communication segments. Here again, DBSIM is a very useful tool to test candidate scenarios. Communication segments are generally viewed as a kind of temporary segments. In order to achieve fault tolerance at the query execution level, we plan to treat communication segments as (temporarily) non-volatile data structures known to the transaction model.

Network Technology Besides our goal of intra-query parallelism, we also focus on exploiting new network technologies, since in the last couple of years interesting concepts have been developed in this area. ATM technology seems to be very promising. We believe that MIDAS will find ATM quite interesting and will profit from advantages like shorter latency, broadcasting support and scalable network design as well as bandwidth. We observe the current development efforts in the ATM area and intend to use the technology as soon as it matures.

At the moment we use DBSIM to check out ATM scenarios thereby reevaluating our VDBC design decisions. Meanwhile PVM supports ATM technology (PVM-ATM 3.3.2.0). As soon as our DBSIM simulations are finished, we want to investigate the usage of PVM for ATM within the MIDAS testbed. In addition to ATM, PVM also supports a parallel I/O subsystem, called PIOUS [Su94]. We plan to get this new parallel I/O system into MIDAS, as well. We view both as key technologies for enhancing the performance and parallelism in our system environment. Since both are provided by PVM, we assume an easy integration into the MIDAS prototype.

5. Conclusions

One important goal of our MIDAS project is to investigate the suitability of the evolutionary approach to the design and engineering of a shared-disk PDBMS. We reported on the most important design decisions as well as on first results and experiences with the MIDAS testbed PDBMS. So far our approach seems to be very promising, as we were able to reuse existing source code to a great extent. Our main programming efforts have concentrated on the realization of the distributed database cache facility VDBC and communication features. These are mandatory components that enable the MIDAS testbed to run on our shared-disk architecture consisting of a cluster of commodity UNIX workstations. Inter-transaction parallelism is already available in MIDAS and intra-query parallelism is currently under development.

The next steps comprise the completing of the transformation from TransBase to MIDAS and, of course, a careful evaluation of our implementation. In order to reach at a high-performance PDBMS, locking, logging as well as recovery have to be adapted, and optimization, scheduling, and load balancing for efficient parallel execution have to be integrated into the MIDAS testbed. Whenever new algorithms are designed, we firstly evaluate their characteristics w.r.t. scalability and performance using our simulation system DBSIM. Only if these preliminary evaluations are satisfactory, we consider an integration into the MIDAS testbed. For validation of the implementation efforts, extensive performance measurings testing a variety of transaction processing profiles will be conducted. In this context we strive to show the usability of a shared-disk PDBMS for simultaneous OLTP and complex query processing. We will use this series of experiments to fine-tune our system components and load balancing schemes. Additionally we will have a close look at new concepts promising a further increase of our system performance, as e.g. ATM-technology and novel query optimization and processing techniques.

6. Literature

ACM95 Carey, M., Schneider, D. (eds.), Proceedings of the International Conference on Management of Data, ACM SIGMOD Record, Vol.24, No. 2, 1995.

Ap92 Apers, P, et al.: PRISMA/DB: A Parallel Main-Memory Relational DBMS, in: IEEE Trans. on Knowledge and Data Engineering, Vol.4, 1992.

BFG95 Baru, C., Fecteau, G., Goyal, A. et al.: DB2 Parallel Edition, in: IBM Systems Journal, Vol. 34, No. 2, pp. 292-322, 1995.

BHR80 Bayer, R., Heller, H. Reiser, A.: Parallelism and Recovery in Database Systems, in: ACM TODS, Vol.5, No.2, pp. 139-156, 1980.

Bo89 Boral, H. (ed.): International Workshop on Database Machines, Deauville 1989, in: Lecture Notes in Computer Science, Springer, 1989.

Bo90 Boral, H., et al.: Prototyping Bubba: A highly parallel database system, in: IEEE Trans. on Knowledge and Data Engineering, Vol. 2, No.1, pp. 4-24, 1990.

De90 DeWitt, D, et al.: The Gamma Database Machine Project, in: IEEE Trans. on Knowledge and Data Engineering, Vol. 2, No.1, pp. 44-62, 1990.

DG92 DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems, in: CACM, Vol.35, No.6, pp.85-98, 1992.

Fr87 Freytag, J.C.: Translating Relational Queries into Iterative Programs, LNCS 261, Springer, 1987.

Ge94 Geist, A. et al.: PVM 3 User‘s Guide and Reference Manual. Technical Report ORNL/TM12187, Oak Ridge National Laboratory, May 1994.

Ge95 Gerber , B.: INFORMIX Online XPS, in [ACM95], p. 463, 1995.

Gr90 Graefe, G.: Encapsulation of Parallesim in the Volcano Query Processing System, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 1990.

Gr95 Gray, J.: A Survey of Parallel Database Techniques and Systems, in: Tutorial handout at Int. Conf. on Very Large Databases, 1995.

GR93 Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

Hi90 Hirano, M., et al.: Architecture of SDC, the super database computer, in: Proc. of JSSP, 1990.

HR83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, in: ACM Computing Surveys 15(4), pp. 287-317, 1983.

Hs83 Hsiao, D. (ed.): Advanced database machine architecture, Prentice-Hall, 1983.

HS93 Hong, W., Stonebraker, M.: Optimization of Parallel Execution Plans in XPRS, in: Distributed and Parallel Databases, Vol.1, No. 1, pp. 9-32, 1993.

INTEL89 iPSC/2 User's Guide: Intel Scientific Super-Computer Division, Intel Corporation, 1989.

LARS92 Lomet, D., Anderson, R., Rengarajan, T., Spiro, P.: How the Rdb/VMS Data Sharing System Became Fast, DEC Technical Report CRL 92/4, 1992.

LDHSY89 Lorie, R., Daudenarde, J., Hallmark, G., Stamos, J., Young, H.: Adding intra-transaction parallelism to an existing DBMS: Early experience, in: IEEE Data Eng. Newsletter, Vol.12, No.1, 1989.

Li93 Linder, B.: Oracle Parallel RDBMS on Massively Parallel Systems, in: Proc. Int. Conf. on Parallel and Distributed Information Systems, pp. 67-68, 1993.

Li94 Listl A., Using Subpages for Cache Coherency Control in Parallel Database Systems. in: Proc. of the International PARLE'94 Conference, Athens, pp. 765 - 768, 1994.

LB95 Listl A., Bozas G., Performance Gains Using Subpages for Cache Coherency Control, Technical Report, TUM-INFO, SFB-Bericht Nr. 342/21/95 A, 1995.

LPRBL95 Listl A., Pawlowski M., Reiser A., Bozas G., Lehn R., Architektur des parallelen Datenbanksystems MIDAS. in: Proc. of BTW'95 Conf., Informatik Aktuell, Springer, 1995 (in german).

LSF94 Listl, A., Schnekenburger, T., Friefrich, M.: Zum Entwurf eines Prototypen in MIDAS, Technical Report, TUM-INFO, SFB-Bericht Nr. 342/1/94 B, 1994 (in german).

MMK90 Bemmerl, T. Ludwig, T.: MMK - A Distributed Operating Systtem Kernel with Integrated Dynamic Loadbalancing, CONPAR 90 - VAPP Conf., Zürich, Schweiz, 1990.

MN92 Mohan, C., Narang, I.: Efficient Locking and Caching of Data in the Multisystem Shard Disks Transaction Environment, in: Proc. 3rd Int. Conf. on Extending Database Technology, Vienna, 1992.

MPTW94 Mohan, C., Pirahesh, H., Tang, W., Wang, Y.: Parallelism in Relational Database Management Systems, in: IBM Systems Journal, Vol. 33, No. 2, pp. 349-371, 1994.

Ra91 Rahm, E.: Concurrency and Coherency Control in Database Sharing Systems, Technical Report ZRI 3/91, Dec. 1991, University of Kaiserslautern.

Ra94 Rahm, E.: Mehrrechner-Datanbanksysteme - Grundlagen der verteilten und parallelen Datenbankverarbeitung, Addison Wesley, 1994 (in german).

Sk92 Skelton, C. et al.: EDS: A Parallel Computer System for Advanced Information Processing, in: Proc. 4th In.t PARLE Conf., LNCS 605, Springer, pp. 3-18, 1992.

St94 Stonebraker, M.: Readings in Database Systems, 2nd Edition, Morgan Kaufmann Publ. 1994.

Su94 Sunderam, V. et al.: A Parallel I/O System for High-Performance Distributed Computing, in: Proc. of the IFIP WG 10.3, 1994.

TB95 TransBase System and Installation Guide, Version 4.2, TransAction Software GmbH, 1995.

Va93 Valduriez, P.: Parallel Database Systems: Open Problems and New Issues, in: Distributed and Parallel Databases, Vol.1, No. 2, April 1993, pp.137-166.

VLDB95 Dayal, U., Gray, P., Nishio, S. (eds.) Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, 1995.

WCK93 Witkowski, A., Carino, F., Kostamaa, P.: NCR 3700 - The Next-Generation Industrial Database Computer, in: Proc. of 19th Int. Conf. on Very Large Databases, 230-243, 1993.

Ze90 Zeller, H.: Parallel Query Execution in NonStop SQL, in: Proc. of IEEE Compcon Spring, 1990.

ZZB93 Ziane, Z., Zait, M., Borla-Salamet, P.: Parallel Query Processing in DBS3, in: Proc. Int. Conf. on Parallel and Distributed Information Systems, pp. 93-102, 1993.