Use of inherent parallelism in database operations

T. Härder, Ch. Hübel, B. Mitschang
University Kaiserslautern

Abstract
Non-standard applications of database systems (e.g. CAD) are characterized by complex objects and powerful user operations. Units of work decomposed from a single user operation are said to allow for inherent semantic parallelism when they do not conflict with each other at the level of decomposition. Hence, they can be scheduled concurrently. In order to support this processing scheme it is necessary to organize parallel execution by adequate control units. Therefore, client-server processes and nested transactions are applied to hierarchically structure the DBS-operations. On the other hand, the DBS-code itself has to be mapped onto a multiprocessor system to take advantage of multiple processing units.

1. Introduction

Parallel computer architectures of various kinds - often referred as super-computing - offer huge instruction rates for processing a special sort of applications. They are implicitly tailored to data-intensive numerical applications for which such a tremendous demand of processing power is necessary to yield the required precision of the solution (e.g. finite elements) or the mandatory response time (e.g. real time applications). For these numerical applications, the transformation of (parts of) a sequential program - the so-called vectorization - to an equivalent program, which can take advantage of parallel facilities, is comparably simple because of the homogeneous data structures used, e.g. the distribution of a large matrix operation to a SIMD-architecture.
Database operations are also data-intensive and must be executed fast enough to support an interactive environment - in particular in "future" applications like engineering (CAD, CAM), or geographic information systems. Their decomposition to exploit parallelism on complex objects within a single operation is by far more complicated compared to numerical applications. In the first place, heterogeneous data structures and their interfering operations appear to be responsible for this difficulty. Therefore, up to now almost no efforts are made to gain some kind of "inherent" parallelism. Nowadays commercial database systems (DBS) execute the operation of a user's transaction (DML-operation) in a strictly sequential manner. In this context, parallelism at the level of DML-operation is only achieved by concurrent execution of requests in a multi-user mode.
Database applications do not allow for arbitrary kinds of syntactic parallelism; they rather require logical serialization of all committed transactions whose results must be equivalent to the results obtained by some serial schedule. Hence, a transaction is defined to be a unit of application-oriented work, of synchronization and of recovery. Its properties can be summarized by the ACID-principle of [HR83] (Atomicity, Consistency, Isolation and Durability). These requirements usually lead to a processing concept as sketched in Fig. 1. Parallel execution is only achieved between transactions subject to conflict-free data references controlled by some synchronization algorithm e.g. locking.

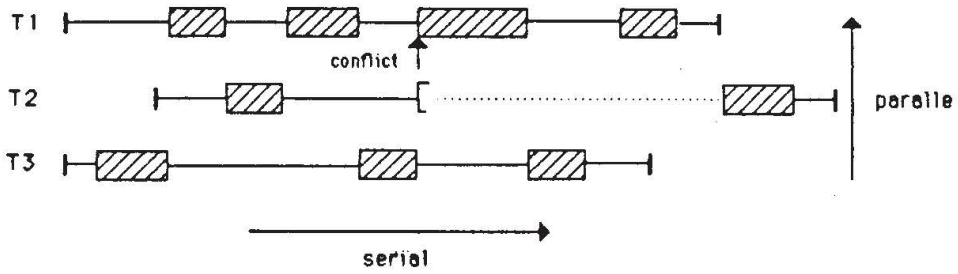

Fig. 1:
Inter-transaction parallelism

The use of parallelism within a single DBS-operation mainly focusses on two approaches explored in research projects: data-oriented parallelism for disk search in SIMD-architectures and parallel processing of relational operators in MIMD-architectures [DW79]. Both approaches have not been particularly successful.
Non-standard applications of DBS manipulate complex objects by powerful operations (ADT's). These objects are represented by simpler data structures (sets of heterogeneous tuples) handled by simpler operations. Hence, the decomposition of complex objects/operations to simpler ones offers a potential for parallelism to be investigated.

2. A new approach to parallel DBS-processing

2.1 Model of DBS-operation

Before sketching our approach we introduce a gross DBS-model to explain its operations. The architecture of a DBS may be conveniently represented by a multi-layer model describing the mapping hierarchy of data. A layer implements the objects and operations offered at its interface (level) to the above layer thereby using the services of the subordinate layer. For an NDBS (DBS for non-standard applications), the model may be illustrated by the type of operations and the level names indicating the type of objects:

levels	layers	operations
object level		ADT-operations
	application layer	
data model level		DML-operations
	data system	
record level		record/access path op.
	access system	
page level		page operations
	storage system	
disk level		disk accesses

An ADT-operation is processed in the application layer by using a number of DML-operations executed at the underlying layer which, in turn, are supported by primitive operations of the next subordinate layer. Each call at one level of implementation invokes a set of primitives at a lower level of control. In current DBS, typically all (sub-)operations are synchronously called and serially executed.

2.2 Control structures for system operations

ADT-operations, however, may consume quite a lot of DML-operations for their execution. For example, we have measured ADT's which have issued more than 10^4 DML's in a CAD-application. In order to reduce response time, efforts should be

made to call and distribute these operations in a synchronous or asynchronous manner and to execute them in parallel. Since the data structures are heterogeneous and dependent on the operations in quite complicated ways, such an approach must be planned carefully. Our idea is to decompose an ADT into a sequence of conflict-free operations. Then, sets of sequence-independent operations may be scheduled in parallel. Such decomposition units (DU's) are roughly equivalent to DML-operations of nowadays DBS. A set of DU's scheduled concurrently is called a parallel execution unit (PEU) (see Fig. 3). DU's are said to allow for inherent semantic parallelism when they do not conflict with each other at the level of decomposition. Currently, the decomposition is done "manually" (not supported by a compiler) by preparing sets of procedures for each ADT at the application layer. This job is facilitated by the fact that the DB-schema at the data model level is fixed for an application class.

For an ADT-operation, atomicity must be provided at the user interface ("all or nothing" has to be executed in case of arbitrary failures). Since the DBS processing is quite complex for an ADT (sometimes 10^7-10^8 instructions or more and lots of data references which may be serviced by disk accesses), there is a strong need for appropriate control structures. Nested system transactions have been proposed for this purpose [Wa84, WS84]. The nesting of sub-transactions (STA's) may be designed in accordance with the system layers. For example, at the application layer STA_1 is a proper structure to accept/return an ADT-call. The DU's used as primitives to the subordinate layer are, in turn, embedded into STA_{2i} ($1<i<n$) to take care of the atomicity of the service calls. Again, a DU may require some services from the storage structure layer which are organized by STA_{3j} and so on. Obviously, the atomicity of STA's greatly facilitates the design of the DBS. Two additional properties are required for our purpose:

- The operations within an STA must be synchronized against operations of other users (transaction principle). Furthermore, they must be isolated against the operations (DU's) within a PEU. Since they run in parallel, they might conflict at the physical level (common page).
- A committed STA_{1j} may be rolled back if a parent STA_{kl} ($k<i$) fails in order to reach the "nothing" state.

2.3 Using semantic parallelism - an example

Until now, we have sketched some abstract concepts allowing for decomposition and parallelization of ADT-operations. In order to demonstrate the practical usefulness of these approaches, we are going to introduce a simple but illustrative example. For this purpose, we have chosen the area of geometric solid modeling, where the user interface is designed for the construction of 3D-workpieces. Each user operation refers to solids and is either binary (UNION, DIFFERENCE and INTERSECTION) or unary (TRANSLATION, ROTATION and SCALING).

One important representation of such solids, especially for graphical output, is the so-called boundary representation scheme (BREP). There, each solid is represented by its faces, which are, in turn, composed of its border-lines, whereof each line is limited by its endpoints.

Using this modeling approach, each workpiece is represented by a heterogeneous structure as depicted in Fig. 2. A generalization of a decomposed ADT-operation using the above modeled BREP-scheme is shown in Fig. 3, where each DU is represented by a rectangle and the PEU's are built by means of dotted line rectangles.

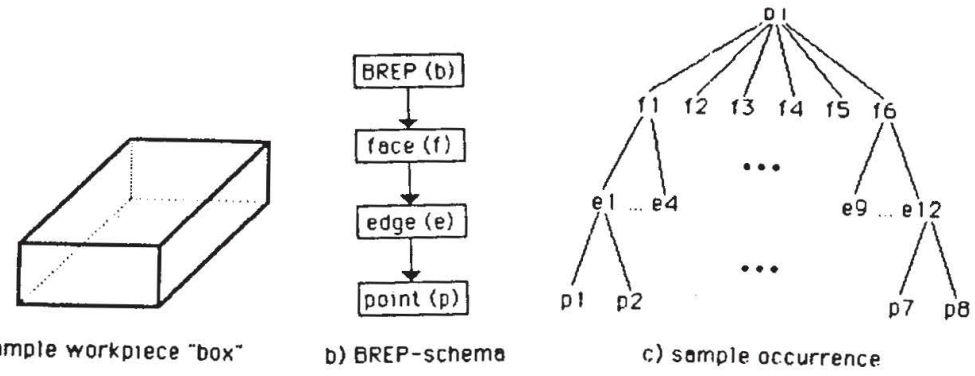

Fig. 2: Boundary representation

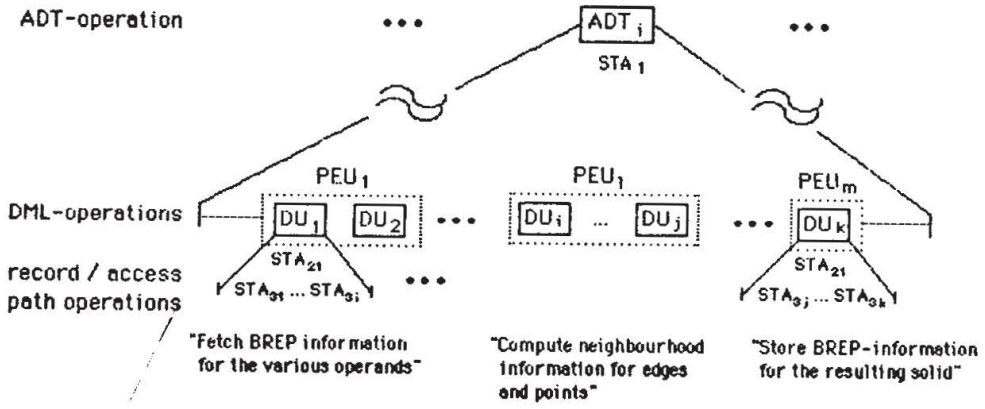

Fig. 3: Decomposition of an ADT-operation

3. Mapping of DBS-layers to software structures

Now we want to sketch some important concepts and constructs in the field of programming languages and operating systems necessary for the implementation of the ideas discussed above.

3.1 Use of process structures

The parallel execution of DU's anticipates the existence of concurrent processing units typically embodied by processes. From an OS point of view, they are the units of processor and resource allocation, protection and scheduling. Since these processes work together for a common job, there is an urgent need for effective cooperation which requires suitable mechanisms for communication and synchronization, e.g. service calls/results must be distributed/returned or access to shared data must be controlled efficiently. The concept of remote operations appears to be adequate for process communication when its semantics is extended for our purpose. A parallel activation of remote operations is a mandatory option for the concurrent execution of DU's within a PEU. An appropriate programming notation to express such a calling semantics is given by the **Parbegin...Parend** construct combined with the remote procedure call mechanism (Fig. 4a). **Parbegin** marks the begin of a set of parallel calls. **Parend** determines the "wait for result" point, i.e. the point of synchronization [AS83].

Our multi-layered architecture model describes the abstract mapping hierarchy of a DBS. Various ways for the DBS implementation are discussed in literature: object-, function- or layer-oriented. Here, we refer to the classic way of layer-oriented

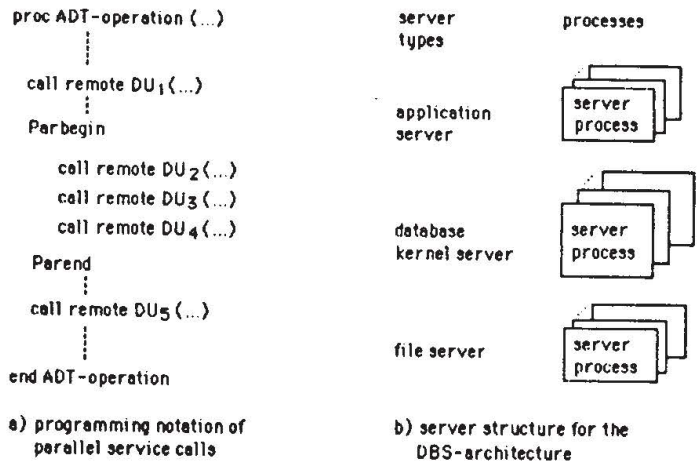

Fig. 4: Sample program and server structure

implementation. The system then has to be divided into parts and allocated to processes to appropriately support the anticipated parallel execution. We propose a separation of the mapping hierarchy into application, kernel and file server (Fig. 4b); their cooperation is performed by client-server relationships where a required service-call is directed to a task (generic processing structure for requests). In general, several client requests (from one or several processes) are issued concurrently (at each level). Hence, a suitable mapping of the above introduced tasks to their corresponding server processes must be provided. Among the conceivable solutions only multi-process/single-tasking (i.e. only one task within one process) or multi-process/multi-tasking (i.e. several tasks within one process) may be chosen for each of the three DBS components (Fig. 4b), when the full potential of parallelism in our hardware structure should be exploited (Fig. 6). Multi-tasking introduces an additional level of scheduling performed in a process. It is only justified for reasons of high process switching overhead or by the need to dynamically create cheap processing structures and tailored scheduling strategies.
The application server offers ADT-operations to the user. An application server process includes the execution of some model mapping functions and the distribution of DU's by activating several DML-operations. DU's within PEU's may be performed by asynchronous service calls. The kernel server executes DML-operations thereby using primitive operations offered by the file server. This mapping involves functions of the data system, access path system and parts of the storage system. Finally, the file server manages the disc accesses. Each of those three server types can be represented by several processes.
The kernel server works on behalf of one or more users on common data structures (records, access paths, locking/logging informaton). Message-oriented exchange of data/control would have an heavy impact on system performance. Therefore, common memory for these data is an important implementation requirement. Then, some mutual exclusion mechanism has to be provided to control access to these shared resources. The frequency of these events makes it necessary to use direct protocols by the participating process (e.g. semaphore-based).

3.2 Use of functional parallelism

The proposed client-server model exhibits a very simple structure for the sake of reduced inter-server complexity. As a consequence, especially the kernel server has a significant internal complexity which may be reduced by introducing some further

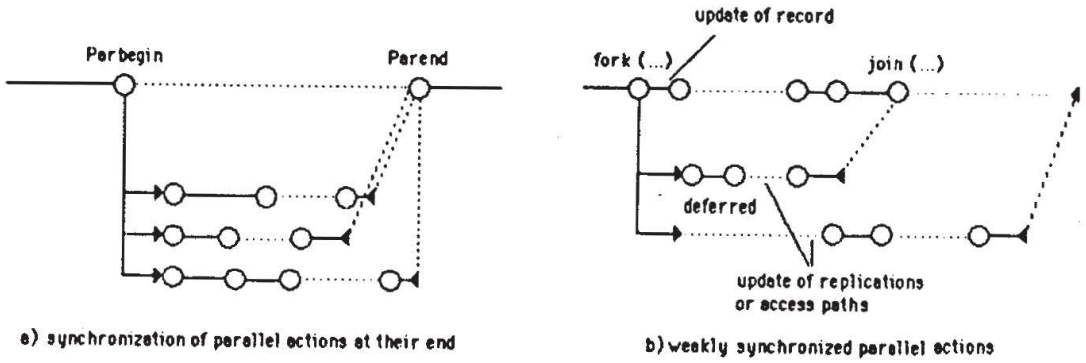

Fig. 5: Synchronization of parallel actions

server types. The expected gain of such subdivisions (e.g. simpler module structures) will often be paid off by increased communication costs. The intent to exploit parallelism, however, could be a strong motivation. On the other hand, a refined decomposition of operations and their parallel execution must be carefully considered for the kernel server, since shrinking operation granules coincide with a constant overhead for process communication and process switching.
Another use of parallelism, however, may be worth a closer look. Some DBS-functions and actions can be executed in parallel without the necessity of strict synchronization at their end (Fig. 5a). For example, locking and logging/recovery actions, integrity checking or access path/replica maintenance [HR85] are candidates for applying weakly synchronized parallel operations (Fig. 5b). Only at the end of the resp. STA or even later in case of deferred execution some acknowledgement has to be accepted. Obviously the price to pay is a new control structure for keeping track of such events. In addition, error detection and recovery become more complex. When a failure occurs, rollback is more difficult implying new and extended mechanisms for error recovery. This approach also allows for an optimistic attitude when concurrently calling functions. For example, record update can be initiated together with the request for the corresponding lock. If this asynchronous locking procedure does not succeed (lock is not granted), the depending record modifications must be undone. The rationale behind this optimistic idea is to try parallel actions with the (hopefully low) risk of bad luck.
Functional parallelism of this kind can be utilized to take advantage of refined server structures (or client-server relationships between kernel server processes) without being fully compensated by processing overhead. It requires some sort of inter-process communication to report the delayed end of operation. For this purpose, a less restrictive method (**FORK...JOIN** (AS83)) compared to Parbegin...Parend seems to be convenient. Nevertheless, the application of functional parallelism is less clear than the use of parallelism generated by semantic decomposition of operations. This idea deserves further attention.

4. Hardware architecture for parallel DBS-processing

Thus far, we have introduced the various levels and organizational concepts of our DBS-architecture, i.e. the computational model of the software system:

- the static view of the DBS-architecture expressed by hierarchic layers
- the organization of the execution by a nested transaction concept
- the mapping of the layers to server types
- the use of dynamic relationships (client-server) between process structures of the same or of different server types.

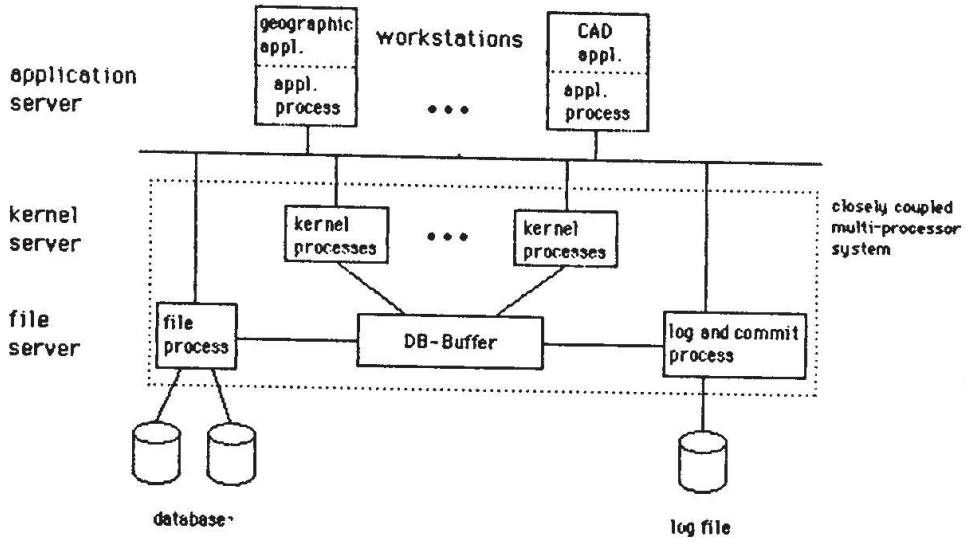

Fig. 6: Hardware architecture and server allocation

As a final step, mapping of processes to a multi-processor system must be accomplished adequately to preserve the inter-process parallelism as far as possible.
Our "ideal" hardware configuration and server allocation for the proposed type of parallel processing is illustrated in Fig. 6. This architecture is based on general purpose processors, for example of type MC 68020, each of them equipped with about 4 MBytes main memory. They are connected by a high bandwidth communication system (bus or ring) of about 10-100 MBits/sec. Some processors are closely coupled by a common memory partition. It is tailored to the particular processing needs of a DBS and serves as a common system buffer (having a size of preferably 10-? MBytes). All processors run an own copy of the OS and the DBS-components in their memory. The user process (e.g. for CAD/VLSI application) is allocated together with an application server process to a separate processor (e.g. workstation).
The prime mechanism for communication is message-based. Access to shared data by kernel server processes is extremely frequent; hence, a message protocol would have severe impact on performance. Therefore, shared data is located in a memory partition accessible for all kernel and file processes. Exclusive control is achieved by a semaphore protocol. File and commit processes are able to read/write pages (data and log information) to/from the system buffer. Kernel processes are directly referencing the contents of buffer pages by machine instructions.
Since data structures can be manipulated directly, DBS-algorithms for all important functions (e.g. buffer management, locking, logging) may be designed similar to the centralized case. Due to the server structure, service calls at various levels and granules must be distributed and assigned to processes in a suitable way. Hence, it is necessary to provide a load control function [Re86], in particular for the kernel server processes. Since these are accessing a common buffer, load control is comparably simple. Obviously, the use of special purpose processors would have complicated load control considerably.
All kernel server processes have the same view to the entire database - there is no need for distributing copies or dedicating processors to database partitions. Multi-processor database architectures without shared memory - e.g. DB-sharing vs. DB-distribution systems - are forced to use copies of data or are restricted to partitioned access [Hä86] - these properties make access to data and load control much more difficult. Of course, shared memory is then a limiting factor to

system's growth. On the other hand, the potential parallelism gained by decomposing a single user operation is not unlimited (up to 3-5 DU's in a PEU). In addition, synchronization between DU's of a PEU (at the physical level) and between user operations does not allow for arbitrary degrees of parallelism.

5. Conclusions

We have discussed a new approach to exploit parallelism on heterogeneous data structures. As opposed to homogeneous data structures of numerical applications, the transformation of a user operation to allow for parallel execution is much more complicated. We have proposed a decomposition of ADT-operations into units to be executed concurrently thereby using the inherent semantic parallelism.
The parallel execution is aimed at suboperations of a powerful user operation rather than separate user operations. Therefore, the concept of nested transactions (within a single operation) is needed to organize the dynamic flow of control. Based on adequate process and communication concepts two different kinds of parallelism are considered - semantic and functional parallelism with strictly synchronized and weakly synchronized communication. To maximize parallel actions the DBS-code has to be mapped appropriately to processes which in turn have to be assigned to processors in a suitable way. For DBS-processing of the particular kind, a closely coupled hardware architecture seems to be mandatory because references to shared data are very frequent.

6. References

AS83 Andrews, G.R., Schneider, F.B.: Concepts and Notations for Concurrent Programming, in: ACM Computer Surveys, Vol. 15, No. 1, March 1983, S. 3-43.

DW79 DeWitt, D.J.: DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems, in: IEEE Trans. on Computers, Vol. 28, No. 6, 1979, pp. 395-405.

Hä86 Härder, T.: DB-Sharing vs. DB-Distribution - die Frage nach dem Systemkonzept zukünftiger DB/DC-Systeme, in: Proc. 9. NTG/GI-Fachtagung "Architektur und Betrieb von Rechensystemen", Stuttgart, März 1986.

HR83 Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, in: ACM Computing Surveys, Vol. 15, No. 4, Dec. 1983, pp. 287-317.

HR85 Härder, T., Reuter, A.: Architektur von Datenbanksystemen für Non-Standard-Anwendungen, in: Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", März 1985, Karlsruhe (eingeladener Vortrag), IFB 94, S. 253-286.

Re86 Reuter, A.: Load Control and Load Balancing in a Shared Database Management System, in: Proc. Int. Conf. on Data Engineering, Los Angeles, Feb. 1986.

Wa84 Walter, B.: Nested Transactions with Multiple Commit Points: An Approach to the Structure of Advanced Database Applications, in: Proc. 10th Int. Conf. on VLDB, Singapore, 1984,, pp. 161-171.

WS84 Weikum, G., Schek, H.-J.: Architectural Issues of Transaction Management in Layered Systems, in: Proc. 10th Int. Conf. on VLDB, Singapore, 1984, pp. 454-465.