in: Proc. 3rd Annual Parallel Processing Symposium, Fullerton, CA, 1989, pp. 564-578.

Evaluation of Hardware Architectures for Parallel Execution of Complex Database Operations

T. Härder, H. Schöning, A. Sikeler
University of Kaiserslautern, P.O. Box 3049, D-6750 Kaiserslautern, West-Germany

Abstract

New database applications, primarily in the areas of engineering and knowledge-based systems, refer to complex objects (e.g. representation of a CAD workpiece or a VLSI chip) while performing their tasks. Retrieval, maintenance, and integrity checking of such complex objects consume substantial computing resources which were traditionally used by conventional database management systems in a sequential manner. Rigid performance goals dictated by interactive use and design environments imply new approaches to master the functionality of complex objects under satisfactory time restrictions. Because of the object granularity, the set orientation of the database interface, and the complicated algorithms for object handling, the exploitation of parallelism within such operations seems to be promising.

Our main goal is the investigation and evaluation of different hardware architectures and their suitability to efficiently cope with workloads generated by database operations on complex objects. Apparently, employing just a number of processors is not a panacea for our database problem. The sheer horse power of machines does not help very much when data synchronization and event serialization requirements play a major role during object handling. What are the critical hardware architecture properties? How can the existing MIPS be best utilized for the data management functions when processing complex objects? To answer these questions and related issues, we discuss different kinds of architectures combining multiple processors: loosely-, tightly-, and closely-coupled. Furthermore, we consider parallelism at different levels of abstraction: the distribution of (sub-)queries or the decomposition of such queries and their concurrent evaluation at an inter- or intra-object level. Finally, we give some thoughts as to the problems of load control and transaction management.

1. Introduction

Database management systems (DBMS) are software systems of some functionality and complexity. They are designed to provide the data management functions for a wide spectrum of applications. Although successful in many business and administration applications, today's DBMS are unable to meet the high performance requirements and the increasing need for enhanced functions of applications that also would like to use a DBMS. Particular challenges are posed by high performance transaction systems (HPTS) where the goal is to process workloads of more than thousand transactions/sec (>1 KTPS [1]; these transactions are typically very short and executed in banking or reservation systems. Another fast growing area of emerging DB-applications which combine performance and functionality requirements is the class of so-called non-standard applications. Typically, all engineering (e.g. CAD/CAM, VLSI design) and knowledge-based applications (e.g. expert systems) are considered to be candidates for this class. Here, emphasis is not given to highly concurrent processing of short transactions, but to fast execution of operations on complex objects thereby only dealing with moderate concurrency among long transactions. Because of the interactive nature of design transactions, response time is a major or even the critical issue.

All these ambitious goals for providing performance and/or function are undermined by the concept of utilizing parallelism to speed up DBMS processing. Reference patterns of transactions in HPTS typically prohibit parallel running of execution paths belonging to the same transaction. Therefore, it is attempted to optimize inter-transaction parallelism by employing multiple processors operating on the incoming transaction load in a parallel fashion. Three classes of computer architectures are designed and investigated for this purpose: shared everything, shared disks, and shared nothing [2]. If high availability and modular growth are important design issues, two candidates are often recommended - by controversial arguments: the DB-sharing architecture where a complex of independent processors share the database (DB) at the disk level [3] and the DB-distribution architecture where the database and the set of disks are partitioned and where each partition is assigned to an individual processor [1]. Here, we cannot elaborate on these architectures.

Intra-transaction parallelism, i.e. to concurrently execute "independent" parts of a user operation, seems to be highly desirable for many applications. Early DBMS interfaces (e.g. network-oriented data model of CODASYL) didn't encourage such objectives at all, because their tiny navigating operations on single records made all corresponding efforts meaningless. DBMS interfaces permitting set-oriented specification of DB requests point principally in the right direction and are a prerequisite for exploiting DBMS parallelism for single operations. For example, the relational model allows for the accessing of homogeneous sets of tuples (representing simple objects). Various possibilities to utilize intra-transaction parallelism for ad hoc queries/search tasks in the framework of the relational model were investigated in numerous database machine projects [4]. Some proposals were concerned with a "disk search engine" and others with the use of (simulated) associative memory. Still others facilitate concurrent execution of subqueries on the same or on different relations [5, 6] by distributing the data across multiple disks (declustering [7]) in order to partition the processing problem and to assign disjoint data sets to the available processors.

Non-standard applications execute operations on complex objects, that is, on objects with an internal structure - subcomponents may be complex objects, too. The representation of such an object may consist of a potentially large set of heterogeneous tuples connected in a way seas to reflect the inner object structure. Operations on such objects may search or maintain their object representations or check complex integrity restrictions on them. Due to the tightly interrelated object representations neither the search engine/associative memory approaches nor declustering techniques seem to be successfully applicable. Rather, clustering techniques guarantee effective operations on such complex objects, since physical clusters support locality of reference and help to minimize the number of disk accesses. Nevertheless, access to a complex object provokes a long execution path. A set-oriented DBMS interface enables the specification of operations on sets of complex objects thereby causing even longer execution paths within the DBMS.

So far, the discussion reveals that the processing characteristics of non-standard applications provide good opportunities for new DBMS to exploit parallelism on data management functions. However, the approaches to be chosen have little in common with those of numerical applications. Handling complex objects means

- to search large sets of heterogeneous tuples by contents (supported by appropriate access paths)
- to dynamically construct a "particular view" of complex objects or to momentarily derive a specific conclusion by using a set of rules
- to keep physical clusters for effective support
- to maintain large quantities of "useful redundancy" which might be provided to speed up various kinds of search operations [HSS88].

Hence, these prime tasks of DBMS processing lead to the conclusion that parallel execution of complex DB operations is quite different from other applications which traditionally utilize parallelism (e.g. solving numerical problems). The essential characteristics may be sketched as follows:

- 'data intensive' instead of 'computing intensive';
- heterogeneous, irregular, and highly dynamic data structures in contrast to homogeneous, regular, and static data structures;
- dynamic parallelization as against preplanning.

We investigate the exploitation of parallelism during the processing of DBMS operations on complex objects handled in non-standard applications. For this purpose, we classify the ways in which parallelism may be achieved on complex objects and identify their potential uses. Our main goal is the comparison of various underlying hardware architectures as to their suitability for supporting the most performance-critical DBMS functions such as data synchronization, logging, buffer management, process communication, etc. Furthermore, we discuss other important operational issues concerning load control and transaction concepts in the different environments. To start with, we introduce a model describing the DBMS architecture in some general way.

2. Parallelism within a DBMS

2.1 An Architecture Model for DBMS

In order to discuss the DBMS-specific issues of parallel processing, we introduce a fairly general model of DBMS architecture. For various reasons, every DBMS exhibits multiple, hierarchically ordered layers which are dictated by a stepwise abstraction process. At the bottom, the database consists of variable-length bitstrings allocated on disks. With each level of abstraction, the objects become more complex, allowing more powerful operations and being constrained by a larger number of integrity rules. Finally, the uppermost layer supports the complex application objects visible at the DBMS interface. A multi-layered architecture with well-defined internal interfaces embodies essential advantages for every complex system: modularity, data independence, and extensibility in the various layers are especially important for DBMS. Therefore, we refer to an architecture model, as shown in Fig. 2.1.

We use the term NDBS to describe a database management system which is tailored for supporting non-standard applications [8]. The overall architecture consists of a so-called NDBS kernel and a number of different application layers which map particular applications (e.g. 3D solid modeling) to the data model interface of the kernel. Our kernel is considered to be application-independent. Its layers roughly correspond to those of most set-oriented database systems (see, for example, System R [9]):

- The storage system is responsible for the management of the DB buffer and for the mapping of its contents onto external storage. It provides segments divided into pages of uniform size (within a segment) at its upper interface.
- The access system typically offers a 'record at a time' interface for direct access as well as for sequential access along various access paths (scan operations). Units of access provided by this system layer are called 'basic objects'; they are connected to each other in various ways representing specified relationships of the data model. A number of special storage structures such as record clusters, containers for long objects (e.g. images or texts), or spatial access paths should be made available to support performance-critical operations at the physical level. Furthermore, basic objects may be stored redundantly in different storage structures to speed up object retrieval.
- The data system maps the complex objects and operations of the data model interface to the basic objects available at the access system interface thereby translating, optimizing, and executing the DBMS

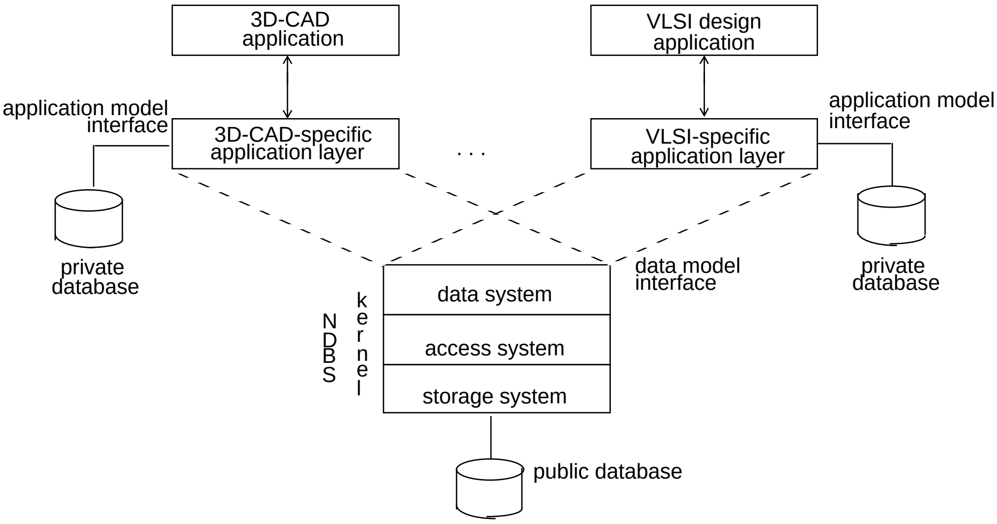


Fig. 2.1: Framework of an NDBS architecture

requests. It is responsible for dynamic construction of complex objects and for set-oriented delivery of result sets to the requesting application layer.

The natural 'breakpoint' in such an architecture is the data model interface. It should provide neutral, yet powerful mechanisms for simultaneously supporting a multiplicity of different applications. DB-based engineering applications are typically running in a workstation/server environment where the kernel is allocated to a multiprocessor complex and the individual application layers to separate workstations. Note, the properties of the data model play a key role in determining the usefulness of such a decision. A set-oriented data model interface is a prerequisite for the separation of kernel and application layer as well as for enhanced utilization of intra-operational parallelism within the kernel.

2.2 Parallelizing Query Processing

In the following, we look at different ways to parallelize query processing [HSS88]. To exemplify the various choices, we introduce a very simple geographic application, which is illustrated by means of the entity-relationship model in Fig. 2.2. It may be used for the management of a county's area, which is assumed to be completely divided into parcels of land. Each parcel is described by its edges, which in turn are defined by their starting and ending points (corresponding to survey points).

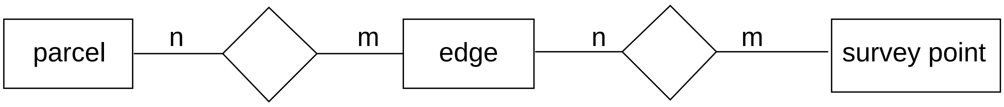


Fig. 2.2: Entity-relationship-diagram of a sample application

Both relationships introduced are of type many-to-many (m:n), because each edge belongs to two parcels, and each survey point may be shared among several edges. Thus, parcels together with their edges and points form non-disjoint complex objects.

2.2.1 Parallelizing the Data System

All operations at the interface of our database system (DBS) work on sets of complex objects. The most obvious way to introduce parallelism in the processing of these queries is to run different queries on separate processors. Unless the queries reference the same objects, they can run totally independently and truly in parallel. Otherwise, they are synchronized along the underlying transaction concept.

Due to the complexity and the large number of objects handled by one query, it is natural that we investigate parallelism within the processing of a single query. For this purpose, we must take a small glance at the execution model. A query is transformed into an operator tree, which consists of nodes representing operators on complex objects and of edges which indicate the data flow among the operators during bottom-up evaluation. In many cases, the children of an operator tree node are independent of one another, and therefore can be processed in parallel. Their result may be consumed by the parent operator in a pipelined way, hence increasing parallelism further. Operators in the data system transform and combine complex objects. One basic operator, however, is required to build up complex objects from the basic objects provided by the access system. All leaves of the operator tree represent this kind of operator. Because of its complexity and its frequent appearance in an operator tree, this operator should exploit parallelism, too. This is possible in two ways, which are exemplified by means of the query

> "Which parcels have at least one point with coordinates inside the rectangular (x1,y1,x2,y2)?".

To answer this query, the appropriate points must be searched for and the corresponding edges have to be fetched, in order to find the qualifying parcels. Once a point in the given range has been found, the edges sharing this point can be accessed in parallel. The same holds for the parcels belonging to these edges. On the other hand, the whole processing can be pipelined: here, the first stage of the pipeline searches for the points, the second one is responsible for the edges, and the third stage fetches the parcels. Of course, both methods may be combined.

Parallelizing the Access System

The techniques described above obviously result in a huge number of access system calls. Therefore, it is desirable to enable parallel processing of these calls. Besides this inter-operational parallelism on the access system level, there are some chances for the parallelization within one single access system operation, in particular, for manipulation operations. As mentioned earlier, the physical representations of the basic objects managed by the access system can be stored redundantly in order to support efficient access according to a bunch of different criteria. Manipulation has to be aware of this redundancy because all representations of one basic object have to be kept in the same state, and hence have to be updated. This can be done in parallel for all representations, as long as there is a synchronized access to the central address structure which contains information about the various physical representations of the same object. A detailed discussion of the problems involved in this approach can be found in [10].

Parallelizing the Storage System

Since even basic objects in non-standard applications may be very large, it may be necessary to allocate them in a sequence of pages rather than in a single page. The access to a basic object involves a couple of pages, which may not reside in main memory, and therefore have to be read from disk. Depending on the mapping of page sequences to different devices, this can be done in parallel. However, the most obvious way to achieve parallelism within the storage system is by utilizing asynchronous I/O operations. Thus, the storage system may handle different overlapped page requests. Nevertheless, both asynchronous I/O operations and locating pages on different devices only support low-level parallelism within the storage system, which is not really a new and challenging affair.

2.3 Activation of Concurrent Requests

So far, we have sketched various possibilities to exploit intra-operational parallelism in all layers of our DBMS kernel. In the following, we only focus on parallel activities in the data and access systems because we anticipate satisfactory results due to their sufficiently large operation granules.

The run-unit in our system is called 'process'. The system's components have to be mapped to a set of co-operating processes before a given request can be executed. Cooperation among processes is achieved by a client-server mechanism; the calling process as the client (e.g. the data system) issues a request to the server process (e.g. the access system) which, in turn, may act as a client when calling a server (e.g. the storage system). To obtain maximum parallelism, our model allows for asynchronous processing, that is, a client may invoke one or more servers and proceed after the invocation (parent/sibling parallelism).

3. Mapping the Database System onto Hardware

Our database system as described in the previous chapter consists of a number of components which may run in parallel. True parallelism, however, is only achieved by mapping these components onto a parallel hardware architecture. In the following, we want to discuss the pros and cons of various ways to map the DBMS kernel onto different hardware architectures. As a first step, we outline the problems - general as well as DBS-specific - which have to be considered throughout this discussion.

3.1 Important Operational Issues

Experience in today's multiprocessor systems show that substantial degradation in performance has to be anticipated as compared to a uniprocessor system [11]. This is even true for 'non-related' applications where a perfect partitioning of resources is possible. It is difficult to accomplish 'almost' linear speed up at the application level (not in MIPS numbers) due to resource management overhead, unbalanced resource allocation, and coordination efforts. Many more problems are caused by data sharing applications. As indicated in section 1, DB operations of non-standard applications deal with complex objects which typically consist of meshed, network-like representations. Hence, data partitioning is not applicable. Rather, the system often has to cope with clustered structures for performance reasons. As a consequence, one may observe early saturation and asymptotic system behavior if wrong or inadequate design decisions are made. Since object partitioning may be very difficult for complex objects, we focus on system architectures where every processor can share all objects of the database. In other words, we believe that multiprocessor architectures sharing the entire database at the disk- or DB-buffer-level incorporate appropriate solutions for our type of DB-problem.

General Problems

Distributing a software system among several processors in order to achieve parallelism always involves some typical problems. In general, the components on different processors have to communicate with each other. Since this can be a crucial point for system behaviour, the appropriate communication mechanisms should be as efficient as possible, particularly in a message-based communication scheme. If the communication is not fast enough for the affected process to perform a synchronous wait, a process switch is enforced. Message passing without the usage of shared memory typically causes the data to be copied, which takes many instructions. In case of large data objects the number of messages together with their size is a critical factor requiring a high bandwidth of the communication medium.

Processes are typically expensive. The cost of switching a context (process switching) may add up to several thousand instructions. In the case of synchronous and remote server requests (e.g. for synchronization protocols) the anticipated processing overhead is particularly high:

- The client has to be deactivated.
- The message may be delayed for buffered message transfers.
- After message handling/queuing, its transmission to the server side is performed.
- Again, after message handling/queuing the server is activated to produce the answer.
- The return path of the message creates the same overhead.

Obviously, certain DBMS configurations are not feasible. High frequency requests which require process deactivation and crossing of processor boundaries must be avoided. In general, minimizing such remote requests for the given database/transaction profiles is a good design principle.

Furthermore, process switching overhead is a critical parameter for the degree and type of parallelism to be achieved. Assume synchronization primitives such as SIGNAL/WAIT. If the system-related path length for executing SIGNAL is much larger than the application-related execution between two such events, then the frequency of such critical events must be reduced by choosing larger processing granules (reducing the parallelism) or a more efficient synchronization mechanism may be found.

Another typical problem is the balancing of the load among the processors. The system should guarantee that no processor is overloaded, while others are idle. For this purpose, each new task entering the system can be assigned to a specific processor depending on the global system state. If the tasks to be assigned are large or long-lasting, a second mechanism is required: Load must be able to migrate from one processor to another, for example by shifting the process dealing with it. Thus, process migration should be achievable with reasonable costs.

Cheap processes and efficient messages are the "two myths of computer science". Their availability effectively determines the quality of every solution for a multiprocessor DBMS.

Specific problems

Running a DBMS in a multiprocessor environment causes some additional, DBMS-specific problems: The key concept in database systems is that of transactions as units of atomicity, consistency, isolation and durability (ACID principle [12, 13]). These goals require synchronization on commonly used objects as well as logging of each manipulating action (in order to guarantee UNDO and REDO of the transaction). To achieve atomicity of a transaction despite failures the work of all participating components must be "glued" together by a special commit protocol. It is important to note that a transaction has the unilateral right to abort its work before commit processing, that is, its effects have to be removed from the database as if it were never started. Such an isolated rollback on DB data requires kind of backward recovery (with log information); forward recovery by means of process pairs, for example, is not sufficient.

Intra-transaction parallelism strongly requires a refinement of the 'flat' transaction concept. In order to provide more decomposable control units and finer grained control of concurrency and recovery the concept of nested transactions has been proposed. It introduces the ability to invoke transactions from within transactions [14]. These subtransactions are atomic and isolated, but have weaker consistency requirements. At commit, they pass on their resources (e.g. locks and log data) to their parent transaction. Since these subtransactions as units of parallel execution are supposed to be small processing granules, efficient activation and commit processing becomes mandatory, because due to their frequency these events play a critical role for the overall system performance.

Synchronization is done by locks in most database systems. This requires either a central lock manager or distributed lock control. In both cases, communication with other processors should be minimized in order to make lock requests cheap [15]. Note, the aggregated time delays for lock requests (see message/process switching costs) extend the allocation of the transaction's resources, thereby obstructing or blocking other transactions. Hence, lock waits of a transaction T may also increase the lock waits of other transactions waiting for locks held by T. It is worth mentioning that many DB-applications contain some very active data elements sometimes called 'high traffic' or 'hot spot' data elements [16]. For special resource usage patterns or high traffic situations, they can serialize the entire transaction processing (exclusive access for every transaction). Therefore, such points of contention need special mechanisms [1].

Log information to deal with a system crash must be written onto a non-volatile storage medium, commonly a disk. This can be done on one disk for the whole system causing message exchange for logging, and may result in a bottleneck on this disk. If logging is done on local disks, however, recovery may become more complicated [12]. Some recovery schemes, for example, require a single file with all log information in commit sequence of all transactions, that is, the local files have to be merged.

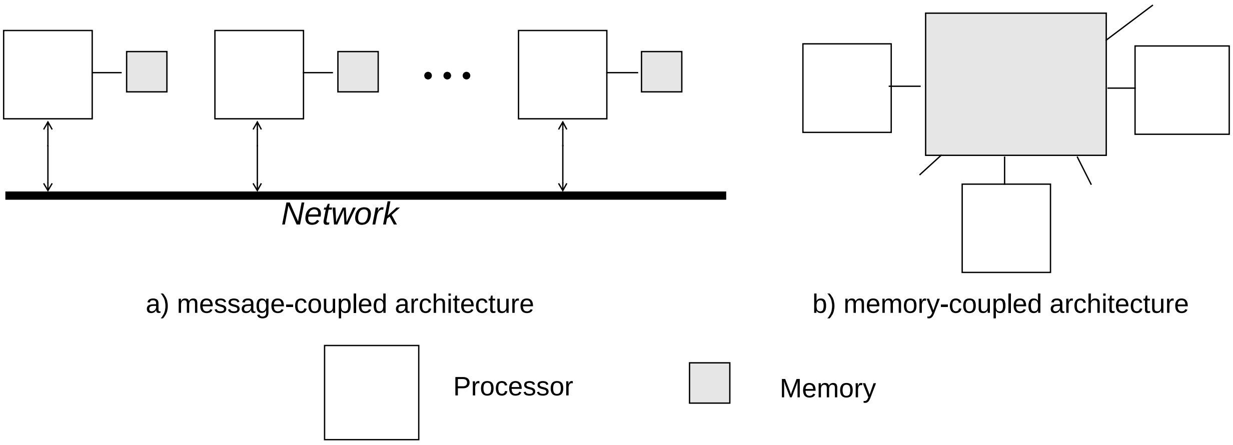
At the end of a transaction (Commit), several actions have to be performed in order to keep the transaction repeatable and to release the resources occupied by the transaction. If the transaction is distributed among several processors, this requires a sophisticated two-phase commit protocol. Such a protocol governed by a coordinator guarantees the atomicity of a distributed transaction execution. However with n participating components (subtransactions), a successful commit involves 4n messages and 2(n+1) synchronous disk writes which also substantially burden the communication and data paths [17]. Furthermore, allocation of resources (primarily locks) has to be prolonged until commit processing reaches a final decision about the fate of the transaction.

Due to the high locality of reference inherent in data access patterns of database systems, a DB buffer is required to obtain satisfactory performance by minimizing the number of disk accesses. Given a sufficiently large buffer and a suitable page replacement algorithm, the DBMS tries to keep all data pages possessing a certain rereference probability in the buffer. As opposed to virtual memory, the DB buffer is not managed by the operating system, but by the DBMS itself - for various reasons [18]. This DB buffer is a virtually infinite linear address space with visible page boundaries, which must be directly addressable by the database system. In an environment without shared memory among processors, this can only be achieved by distributing the DB buffer among the processors. Since the same pages may be used by several processors, this leads to the well-known buffer invalidation problem [15]. Depending on page update frequencies, such page copies in different DB-buffers could cause a tremendous message overhead when identifying the 'current' version of a page.

In the following considerations, we assume a number of processors in the range of 4 to 30, which seems to be sufficient for our class of problems. We further assume that our DBMS is not distributed over different geographic locations; high bandwidth communication (processor bus or LAN) is supposed to be mandatory for every reasonable approach.

3.2 Hardware Architectures

We first investigate two fundamental hardware architectures without regard to special technical properties: The first one assumes no shared memory among the processors, i.e. is purely message-coupled ("loosely-coupled", Fig 3.1a). This makes the system behaviour especially sensitive to the size and number of messages exchanged among the processors. The second is based on a shared memory which can be reached from all processors for almost the cost of a normal main memory access (Fig. 3.1b). In order to control access to common data a hardware-supported synchronization mechanism (e.g. COMPARE AND SWAP) must be available. In either case, we assume that each processor can access all disks.



a) message-coupled architecture

b) memory-coupled architecture

Fig. 3.1: Fundamental hardware architectures

In the following, we investigate several ways to map the database system model of section 2.1 onto these two hardware configurations.

3.3 Distributing the Database System

We approach our solution in a stepwise fashion starting with a monolithic kernel allocation to a configuration where single components of the kernel are independently assigned to processors, thereby maximizing the degree of parallelism for complex-object operations. For this purpose, we distinguish inter- and intra-operational parallelism. Inter-operational parallelism in the kernel or a specific layer means that independent operations may be invoked concurrently at the corresponding (kernel or layer) interface. In addition, intra-operational parallelism enables the decomposition of each operation within one layer (see section 2.2).

Monolithic Kernel Allocation

A first approach allocates the whole kernel to each processor. This only achieves inter-operational parallelism among complex-object operations because the entire kernel code is considered to be monolithic. Since each kernel handles a complex-object operation on its own, no parallelism is accomplished on lower levels. Thus, intra-operational parallelism is not supported. In the message-coupled case (Fig. 3.2a), there is locality of data shared among the three system layers, resulting in a cheap communication from one layer to another, if shared memory segments can be used within a processor. However, different complex-object operations may work on overlapping data, if the corresponding storage structures belong to common pages. Therefore, the buffer invalidation problem may appear, whenever those operations are directed to different processors. For the same reason, a primary copy approach or similar synchronization optimizations [15] are not adequate. Hence, a centralized synchronization scheme is required leading to inter-processor communication for each non-local lock request. Furthermore, there is no need for a distributed two-phase commit, since a transaction within the kernel is not spread over different processors. Due to the complexity of the high-level, set-oriented complex-object operations it is also very hard to decide whether or not they work on overlapping data. Therefore, good load balancing algorithms to allocate such operations on the same processor are hard to find.

In the memory-coupled case (Fig. 3.2b), the buffer invalidation problem does not appear, because there is only one DB buffer, shared by all processors. The same holds for the synchronization problem, since the appropriate data structures (e.g. lock tables) are kept in the shared memory. In contrast to the message-coupled case, the attached allocation of the three kernel layers yields no extra benefits.

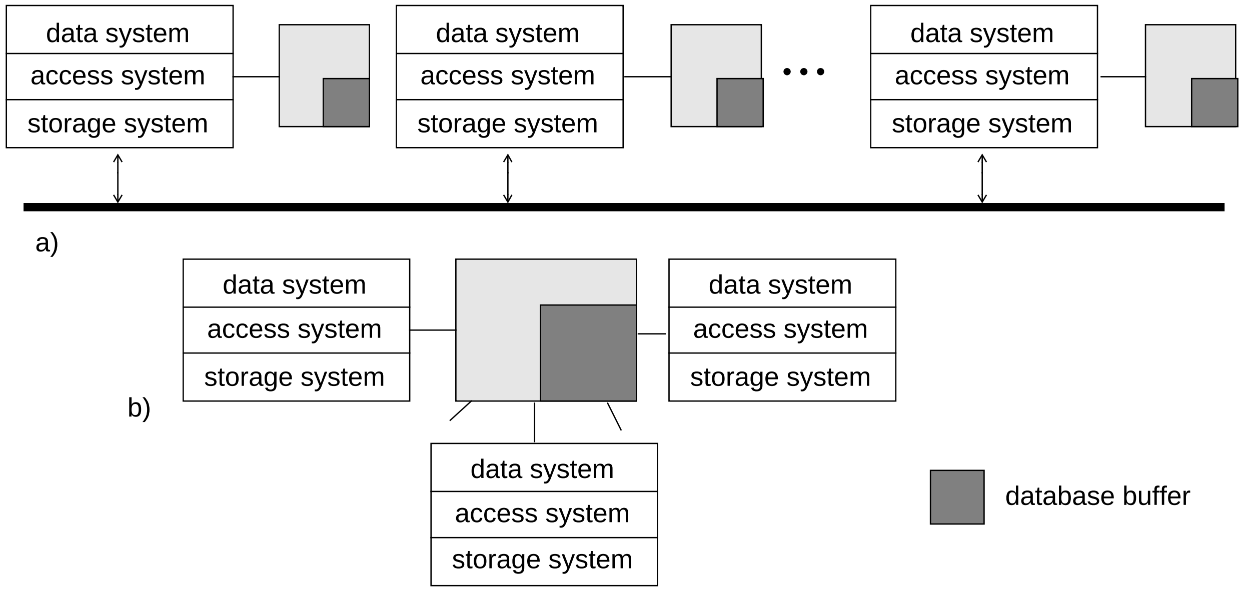


Fig. 3.2: Allocation of the whole kernel code to each processor

Layer-Oriented Kernel Allocation

To achieve inter-operational parallelism on lower levels, too, we consider the mapping of single system layers to processors, thus establishing a client-server relationship among the different layers. Intra-operational parallelism inside the layers is not supported by this proposal. The message-coupled architecture, however, does not permit the separation of access system and storage system (Fig. 3.3a), because the DB buffer (managed by the storage system) is to be directly addressable by the access system, which requires common memory. Thus, inter-operational parallelism is achieved only on the data system and access system level. If there is more than one processor holding the access system and the storage system (which is a prerequisite for parallelism on this level), the buffer invalidation problem as well as the synchronization problem still exist. Additionally, a two-phase commit is required. Requests resulting from the same data system operation can be distributed among several access system/storage system processors, unless they refer to previous calls (conversational interface, e.g. for a scan). Moreover, data system and access system must exchange large data volumes via messages.

In the memory-coupled case (Fig. 3.3b), the access system and storage system may also be separated, allowing inter-operational parallelism on storage system level, too. Changing the number of processors allocated to a specific layer is enabled by the cheapness of process migration, if all local data of a process are situated in the shared memory.

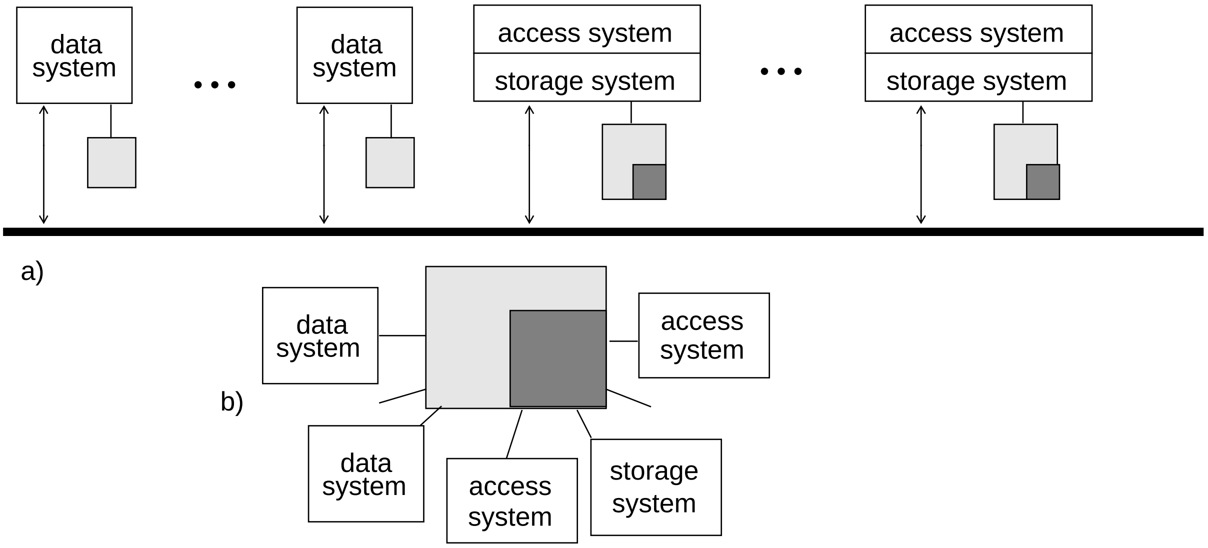


Fig. 3.3: Allocation of layers to processors

Kernel Allocation by Components

In order to allow intra-operational parallelism inside the levels, a finer granule of processor allocation has to be used. In the data system, each operator can be assigned to a processor. In the case of a complex-object constructing operator, it even makes sense to distribute it in a way that enables pipelining within the operator (see section 2.2). In the access system, the components responsible for the management of redundancy can be units of processor allocation. The message-coupled case (Fig. 3.4a) implies an extremely high message exchange among the components. In a similar way to the proposal above, each part of the access system must share the processor with a part of the storage system in order to be able to address the DB buffer directly. Thus, the same problems as above occur.

The memory-coupled case (Fig. 3.4b) seems to be the most promising approach discussed so far:

- It supports inter-operational as well as intra-operational parallelism in all three layers of the database system.
- It allows for cheap synchronization based on common data structures located in the shared memory.
- Two-phase commit can be done efficiently for the same reason.
- Buffer invalidation does not occur.
- Dynamic migration of processes (i.e. operators or components of the layers) is even easier, since the portions to be shifted are smaller. Therefore, short-time tasks like compilation and optimization can allocate a processor dynamically and release it after their task is finished.

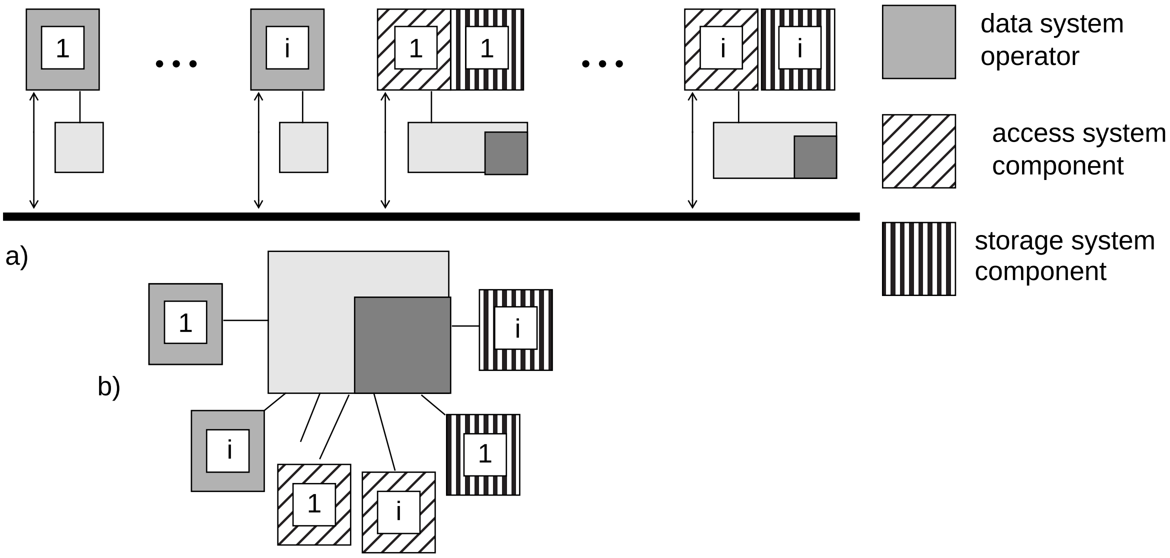


Fig. 3.4: Allocation of smaller components to processors

3.4 Refinements

Our favorite approach presented in the previous section still shows some drawbacks: Until now we have not particularly considered the memory location of the operating system (OS) code and the DBMS code. If they reside in the shared memory ("tightly-coupled") this burdens the memory with unnecessary access conflicts and may lead to a bottleneck in the communication with the memory. This problem is commonly relaxed by the introduction of special caches, but the working set behaviour of a database system may reduce the usefulness of this caching drastically. Fortunately, this kind of data can easily be replicated. For this reason, we propose a “closely-coupled" architecture (Fig. 3.5), where each processor has its own memory containing the OS code as well as the DBMS code, and additionally shares a certain amount of memory with all other processors. Thus, only truly shared data must be allocated in the shared memory, leading to lower traffic on the shared memory. Furthermore, we expect a higher degree of failure isolation by this separation compared to tightly-coupled systems.

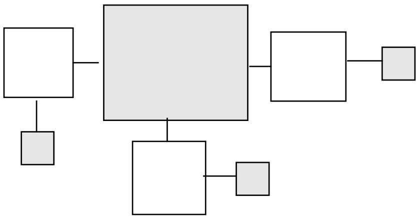


Fig. 3.5: Closely-coupled architecture

4. Further design considerations

So far, we have voted for a closely-coupled multiprocessor architecture. Main arguments were that data partitioning cannot be achieved effectively for complex object processing and that message-based synchronization and subsequent data access to typically small data elements deteriorate system performance due to extreme frequency of these events. The question we are now going to address is whether specialized processors should be used to speed up the DBMS work.

Special function vs. general purpose

A processor is called specialized if it is tailored to a special function or component, that is, it can run the corresponding function very efficiently. As opposed to it, a general purpose processor can execute the entire OS/DBMS code, the code of a single layer, or that of a special component; in any case, it is using an 'unoptimized' standard instruction set. In a multiprocessor DBMS context, for example, it often seems to be desirable to use a 'lock engine' as special purpose hardware [3]. However, we believe that a homogeneous architecture consisting of general purpose processors outperforms a mixed set of processors.

- First of all, much more process switches have to be anticipated with specialized processors as compared to general purpose ones. Every tiny function call for locking, logging, etc. would cause a synchronous interrupt of the main line DBMS processing whereas a process on a general purpose processor carrying the DBMS or a layer code could proceed and perform these functions on the shared memory.
- Load distribution is simple for specialized hardware. However, load balancing is much more difficult. A homogeneous set of processors has more degrees of freedom and more opportunities for optimization. Furthermore, there are no single points of failure.

Load balancing and control

On the other hand, load balancing and load control are very important for many performance criteria in DBMS processing (utilization, response time, etc.). Nevertheless, their role in a DBMS is not well understood because complex dependencies exist among data, memory use, blocking times, synchronization need, recovery issues, etc. For example, increased parallelism may be useless if it leads to higher deadlock and rollback rates; in such cases, the extra work gained by parallel execution is sacrificed to repeatedly process certain transactions. Therefore, load balancing and control may sometimes drastically improve the DBMS performance 'by reducing system utilization', that is, by decreasing the degree of parallelism among conflicting transactions competing for the same set of data [19].

Process migration may be an effective vehicle to balance the processor load. Our closely-coupled scheme facilitates this kind of action because the shared data (DB buffer, lock tables, etc.) remain unchanged.

Distributed systems are, in general, characterized by a lack of global knowledge. Therefore, every balancing and control decision is difficult to obtain. According to Fig. 1, we also have to consider remote workstations which generate high-level database requests in their application layers. As a consequence, hierarchical workload management schemes seem to be appropriate where relatively coarse decisions are made at the global level (application layer), but refinement and adjustment is performed locally.

5. Summary and Outlook

We have presented an evaluation of hardware architectures and their suitability for parallel execution of database operations on complex objects. It includes the use of loosely-, tightly-, and closely-coupled multiprocessor architectures.

The focus of the paper has primarily been the support of parallelism in a DBMS kernel of non-standard database systems. Since NDBS are tailored to workstation/server environments, multiple engineering applications invoke DBMS requests obtaining inter-transaction concurrency on the shared DB resources. For organizational and failure isolation reasons, workstations and servers are strongly separated at the hardware and software level. Adjusted processing models in the workstation allow for asynchronous DBMS requests which fetch bulks of data for long-term local processing (checkout/checkin). Due to the data granules to be fetched, the application requests may be decomposed in the application layer and shipped as "independent" tasks to the DBMS kernel [10]. In such a way, another kind of intra-transaction parallelism may be generated at a higher level of abstraction.

The concepts discussed in this paper are investigated at the University of Kaiserslautern. For this purpose, we have implemented an NDBS called PRIMA according to our multi-layered architecture [8]. It serves as a testbed for a practical evaluation of the proposed kinds of parallelism.

Bibliography

[1] J. Gray, B. Good, D. Gawlick, P. Homan, H. Sammer: One Thousand Transactions per Second, Proc. IEEE Spring CompCon, San Francisco, pp. 96-101, 1985.

[2] A. Bhide: An Analysis of Three Transaction Processing Architectures, Proc. 14th VLDB Conference, pp. 339-350, Aug. 1988

[3] K. Shoens: The AMOEBA Project, Proc. IEEE Spring CompCon, San Francisco, pp. 102-105., 1985

[4] Special Issue on Database Machines, IEEE Transactions On Computers, Vol. C-28, No. 6, 1979.

[5] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, M. Muralikrishna: GAMMA - A High Performance Dataflow Database Machine, in: Proc. VLDB 86, pp. 228-237, 1986.

[6] P. Neches: The Anatomy of a Database Computer System, in: Proc. IEEE Spring Complon, San Francisco, Feb. 1985.

[7] G. Copeland, W. Alexander, E. Boughter, T. Keller: Data Placement in Bubba, Proc. SIGMOD Conference 1988, SIGMOD RECORD, Vol. 17, No. 3, pp. 99-108, Sept. 1988

[8] T. Härder, K. Meyer-Wegener, B. Mitschang, A. Sikeler: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. VLDB 87, pp. 433-442, 1987.

[9] M. M. Astrahan et al.: System R: Relational Approach to Database Management, ACM TODS 1 : 2, pp. 97-137, 1976.

[10] T. Härder, H. Schöning, A. Sikeler: Parallelism in Processing Queries on Complex Objects, in: Proc. Int. Symposium on Databases in Parallel and Distributed Systems, Austin, Texas, pp. 131-143, 1988.

[11] T. Härder, P. Peinl: Evaluating Multiple Server DBMS in General Purpose Operating System Environments, Proc. Conf. on 10th VLDB, Singapore, 1984, pp. 129-140.

[12] J.N. Gray: Notes on Database Operating Systems, Operating Systems - An Advanced Course, Lecture Notes in Computer Science 60, Bayer, R., Graham, R.M., Seegmueller, G. (eds.), Springer-Verlag, 1978, pp. 393-481.

[13] T. Härder, A. Reuter: Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys 15:4, 1983, pp. 287-318.

[14] J.E.B. Moss: Nested Transactions: An Approach to Reliable Computing, M.I.T. Report MIT-LCS-TR-260, M.I.T., Laboratory of Computer Science, 1981.

[15] E. Rahm: Design and Evaluation of Concurrency and Coherency Control Techniques for Database Sharing Systems, submitted to: ACM TODS, 1989.

[16] M. Blasgen, J. Gray, M. Mitoma, T. Price: The Convoy Phenomen, in: ACM Operating System Review, Vol. 13, No. 2, pp. 20-25, April 1979.

[17] C. Mohan, B. Lindsay: Efficient Commit Protocols for the Tree of Processes Model of Distributed Transacitons, In: Proc. 2nd ACM SIGACT/SIGOPS Symp. on Principles of Distributed Computing, Montreal, Canada, Aug. 1983.

[18] W. Effelsberg, T. Härder: Principles of Database Buffer Management, in: ACM TODS 9:4, pp. 560-595, 1984.

[19] A. Reuter: Load Control and Load Balancing in a Shared Database Management System, Proc. Conf. on Data Engineering, Los Angeles, CA., 1986.