A Very Fast Parallel Object Store for Very Fast Applications*

L. Böszörményi K.H. Eder
C. Weich
Institut für Informatik, Universität Klagenfurt,
Universitätsstraße 65 - 67, A-9020 Klagenfurt, Austria,
e-mail: {laszlo,carsten} @ ifi.uni-klu.ac.at

June 30, 1997

Keywords parallel, memory-resident, object-oriented database management system, object store, database architecture

Abstract

An architecture for a memory-resident, Parallel and Persistent ObjectSTore (PPOST) is suggested. Different object-oriented databases might be built on top of PPOST. The term memory-resident (or main memory based) means that the primary storage device is main memory. Persistence is guaranteed automatically by managing secondary and stable storage devices (such as main memory with uninterrupted power supply, discs and tapes). The architecture is able to take advantage of available main memory in a parallel or distributed environment. Thus, transactions can be actually performed with memory-speed, without being limited by the size of the memory of a given computer. Such an architecture is especially advantageous for applications requiring very fast answers, such as CAD or high-performance simulation.

*The implementation environment is partly supported by Digital Equipment Corporation (EERP contract number AU-035).

1 Introduction

The main application area of massively parallel processing is to make large (scientific) computations faster. Some efforts have been made to port existing databases, (such as Oracle) on parallel machines (such as nCube or KSR). Much less effort has been made in finding good architectures for database management systems, which are inherently parallel and thus can take full advantage of parallelism.

Main memory-resident databases [1, 9, 8] are often considered obsolete, because of their limited capacity of memory and their unability to scale up with growing needs. This objection is not true any more, if a memory-resident database is implemented on a parallel architecture, which not only can incorporate substantially large main memory (possibly several Gigabytes or even Terabytes), but scales even better than disc-resident databases. Adding new nodes adds not only more storage capacity, but corresponding processing power as well.

Moreover, in nowadays distributed computing environments not only a number of unused CPUs, but also a great amount of unused main memory is available. Taking advantage of this idle memory (in addition to the idle cycles) can make a parallel, memory-resident database even cost effective.

A memory-resident database can also coexist with other applications using virtual memory as main memory rather than real memory. In this case, however, overallocation must be avoided, otherwise the performance suffers from paging delay.

Another advantage of using parallelism in a memory-resident database is, that logging, checkpointing and archiving can be made in parallel with ordinary transactions serving the users. We make the explicit assumption that applications served by PPOST have substantially more read than update operations (this assumption defines a sufficiently large class of applications). With this assumption, transactions can be processed actually with the speed of the main memory, access to secondary storage can be done in background. In the case of object-oriented databases, we have the additional advantage that methods of a retrieved object can be immediately executed in its primary storage.

Such an architecture is surely not general enough to be the ideal solution for all applications. Its main advantage lies in giving very fast answers for processing-intensive queries. Engineering applications, such as CASE or CAD need this feature [7]. In the field of simulations there is a growing need to take use of databases [11], which is, how-

ever, restricted by slow answers of general database systems. PPOST is ideally suited for supporting high-performance simulations [4].

1.1 Goals

The main goals of PPOST are

- High performance is achieved due to storing all data in main memory and due to intensive use of parallelism.
- Safety and simplicity: Simplicity due to the fact that the object-store is freed from sophisticated disc optimizations. Safety is achieved by using a clean, type-safe object-oriented language by separating the conceptual schema from the external schemas and by simplicity.
- Flexibility: Flexibility means on the one hand, that applications of PPOST may control the degree of parallelism. On the other hand, the architecture can be implemented on different systems, can be adapted to different processing and communication parameters.
- Cost effectiveness: The architecture does not insist on special hardware, it can be implemented on any work-station cluster. It can, however, take advantage of special hardware, such as stable main memory or any high-performance MIMD machine

1.2 Data model

1.2.1 Separation of types and classes

PPOST supports an object-oriented data model. It consists of types, objects, typed object sets, classes, views and generic operations. The main idea is a clear separation between types and classes [5, 6]. Types specify the intensional aspect of objects while classes describe the extensional point of view. Classes and views are based on sets. Sets are well understood and their significance in hierarchical or relational database systems is well known. In addition, sets are inherently parallel.

An object is an instance of exactly one object type. Objects may be grouped in collections. In PPOST typed object sets build the base for all collections. A typed object set is a set of objects with base type specification. A class is an object container. Classes build a hierarchy. If an

object is inserted into a subclass, it automatically becomes a member of all superclasses (instance inheritance).

1.2.2 Separation of conceptual schema and external schemas

Object-oriented databases generally do not separate conceptual and external schemas. In PPOST, however the traditional layered architecture of databases is used [6]. The objects of the conceptual schema are stored in the objectstore of PPOST. The applications access the data via external schemas (views). A view is a named, derived virtual class, and for that reason a specialization of a class. Views are not materialized, i.e. objects are not stored physically in views. Therefore views require production rules to determine which objects exist virtually in them. These production rules must be declared at view definition time. These rules operate on a so called base, which can either be a class or again a view.

1.2.3 Language representation

The concepts of the data model are represented with the help of programming language constructs, in the form of typed, polymorphic objectsets. The schemes are represnted by set-types, classes and views are instances of such types. The integration of the language representation into an elegant, general-purpose programming language (Modula-3) is written in [3].

2 Architecture of PPOST

PPOST's main components are (Fig. 1):

- Objectstore (consisting of a number of object machines)
- Log machine
- Checkpoint machine
- Archive machine
- Users (consisting of a number of user machines)

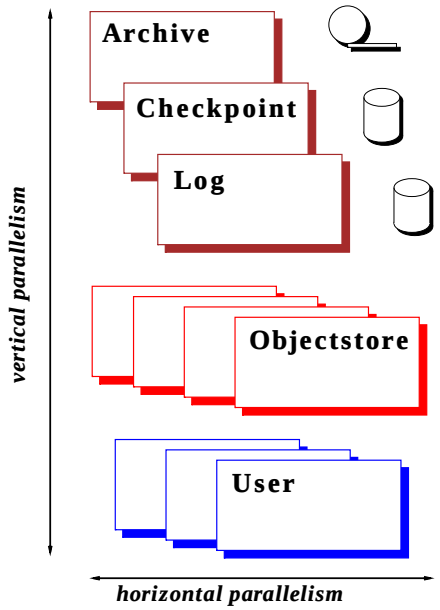


Figure 1: Architecture of PPOST

The ”machines” are logical processors with their own address space. They can be mapped on heavy-weight processes or on physical processors. The machines are able to manage light-weight threads inside the same address space.

PPOST's main purpose is to store and manage a large number of objects. The primary copy of the data are held in main memory. A backup image of the primary data and some log information are held on nonvolatile storage. The backup contains normally an older, but consistent state of the database. Applying the log information on the backup leads to a new consistent state of the primary image.

PPOST is transaction-oriented. Transactions are initiated by the user machines and processed by the objectstore. Issues of persistence are handled by the log, checkpoint and archive machines. The usual transaction properties (atomicity, consistency, isolation and durability) must hold.

Parallelism is used for three different, partly controversial purposes:

1. *Spatial* extensibility (storing capacity can be enlarged by adding nodes)

2. *Time-scale* extensibility (higher speed can be achieved by parallel algorithms)

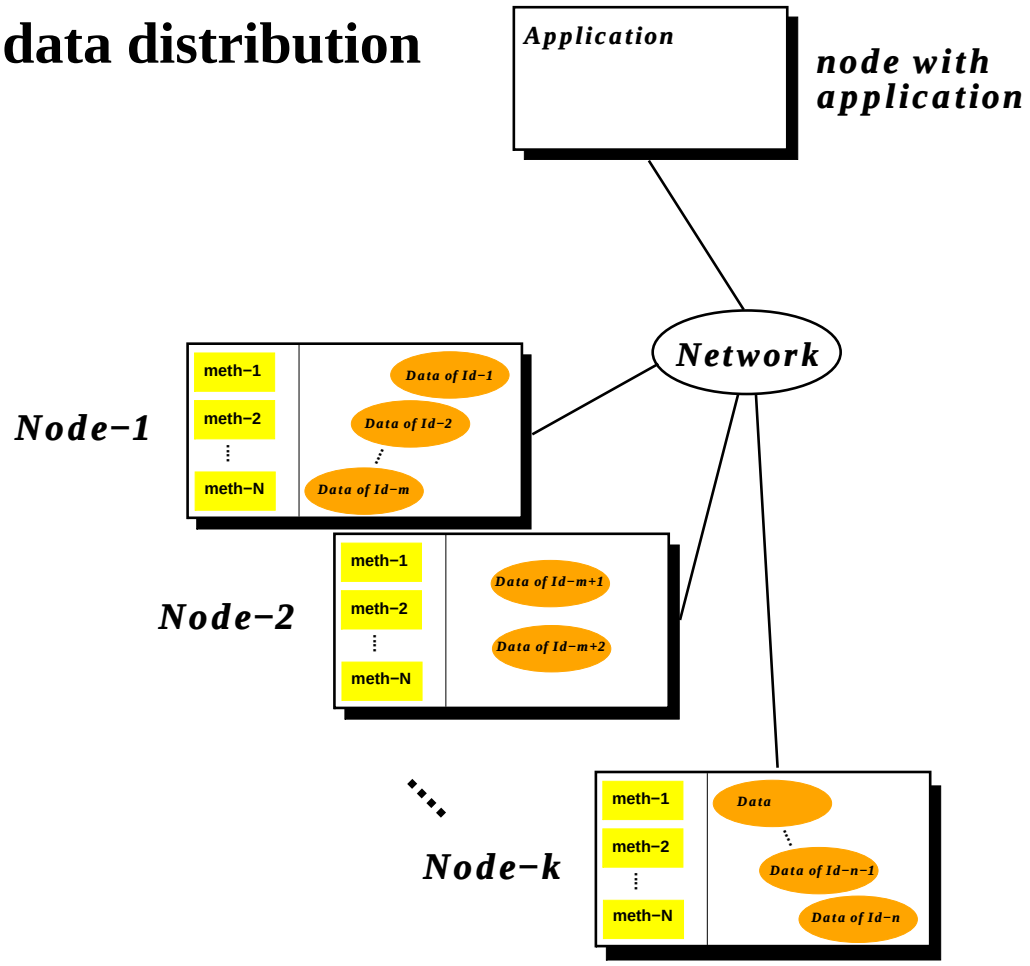


Figure 2: Distribution of data

3. Fast backup (secondary devices are managed in background)

The first two purposes regard to the object store (called ”horizontal” parallelism), the last aspect regards to the pipe-lined backup (”vertical” parallelism). The first two points are in a way controversial: Spatial extensibility requires a reasonably compact storing scheme, time-scale extensibility requires many redundant physical processors. This is a form of the usual time-space trade-off.

2.1 Horizontal parallelism

We organize sets of typecompatible objects in classes. Classes serve as object container (putting an object into a class to make it persistent) and as the starting point of any operation that reads or manipulates more than one of the objects stored. Classes in our object-oriented store have the same role as tables in a relational database.

To spread the data of the class among several nodes having all the methods available on every node we call *data distribution* (see Fig. 2). Operations like selecting certain objects of the class or starting a method of all objects in a class can be done in parallel when we distribute the data: The operation can be started on every node that holds data of the class. Section 3 describes benefits and costs of such distribution in more detail.

2.2 Vertical parallelism

The idea of vertical parallelism is to decouple normal transaction processing from issues of transaction undo and redo. This separation can be done not only conceptually but also physically. Normal transaction processing is done in the object store. Issues of transaction undo and redo are handled by the log machine, checkpoint machine and the archive machine.

2.2.1 Transaction undo

For transaction undo we use before-images or shadow copies in volatile storage. In the case of a system crash, the primary copy of the database is lost anyway. All not-yet-committed transactions are trivially ”undone”. Therefore, transaction undo is in accordance with the concept of a memory-resident database.

2.2.2 Transaction redo

In the case of a system crash an automatic recovery procedure must restore the content of the database.

Parallel logging

The necessary log information (see later) is sent to the log machine. The log machine would ideally store the log-tail in stable main memory.

In this case, transactions whose log information arrived in the log machine can be committed immediately. We do not insist, however, on the existence of a stable main memory. In the lack of this, we precommit [8, 10] the corresponding transactions and let run other transactions (locks are released). In the meantime, the log information is stored on disc in the form of simple sequential files (this can be done at full disc-speed). After that, precommitted transactions may be committed. In the case of a system crash precommitted transactions are handled as not-yet-committed.

Parallel checkpointing and archiving

The task of the checkpoint machine is to apply the logs on the last valid backup image [14]. After processing a certain amount of log information, a new backup is created, and the corresponding log files are deleted. Checkpointing is done by a separate machine, therefore its speed has no influence on the response time of the transactions. If the database is more or less quiescent, then the backup may come very close to (or even the same as) the primary copy. In the case of a heavy load, the backup might become relatively ”old”. In this case, the log files may become long and a restart maybe expensive. This is unlikely, however, because a database rarely has a constant heavy load over a long period of time (i. e. days). The newest backup generated by the checkpointer can be archived on additional nonvolatile storage (such as tapes). Archiving is considered as a normal activity, which does not reduce the response time of normal transactions.

Recovery

In the case of a system crash, a recovery must be executed. The backup image is loaded in main memory and the log is applied on it. Note that in this case the actual memory image is generated with ”memory speed” (instead of ”disc-speed”, as in the case of checkpointing).

Backup database

An interesting possibility of this architecture is to use an existing disc based database as nonvolatile storage medium. In this case a bidirectional mapping is needed between PPOST and the external database. The checkpointer must be able to generate appropriate calls to this database, on the basis of the log information. At recovery, the data

extracted from the database must be mapped on the PPOST primary store image. Such mappings are surely not trivial to find, and the external database should not be quite different from PPOST - e. g. it should be preferably an object-oriented one. If such a mapping can be found, then PPOST could serve as a kind of ”supercache” for some existing databases.

3 Costs of Data Distribution

In this section we want to demonstrate the feasibility of a distributed main memory object store: It is possible within certain limits to enhance throughput and speed of operations on the object store. That means we can manipulate larger sets of objects within the same time by adding new compute nodes, or on the other hand, do the same operation within shorter time.

Distribution of the data of a class makes it possible to accelerate operations that work on every object in the class. Typical operations that need to look at every single object are selecting objects that meet a certain criteria, computing a sum of a single attribute of every object and the like. The enhenced speed gained by parallelism has to make up for the time needed for communication. As we will see, every operation has—depending on the size of the class—an optimum number of nodes with which it runs fastest. Adding more nodes will decrease performance because the time gained by parallelism is less than the time needed for the additional network traffic. If either the class is too small or the network is too slow this optimum number of nodes is one: To distribute such a class with a given operation does not make sense.

A different problem is the increase of the size of a class. If the class is too large to fit in a single node, we have no choice other than to distribute it among several nodes. If the size of a class increases over the lifetime of the system, we would like to add new nodes to gain not only more storage room but also more computational power. As we will show, within certain limits the size of a class can increase without degrading performance if we add more compute nodes.

3.1 Calculating the costs

To calculate the number of nodes necessary for a given class with a given operation in order to meet a certain performance goal we have to know several parameters of the system, the class and the operation:

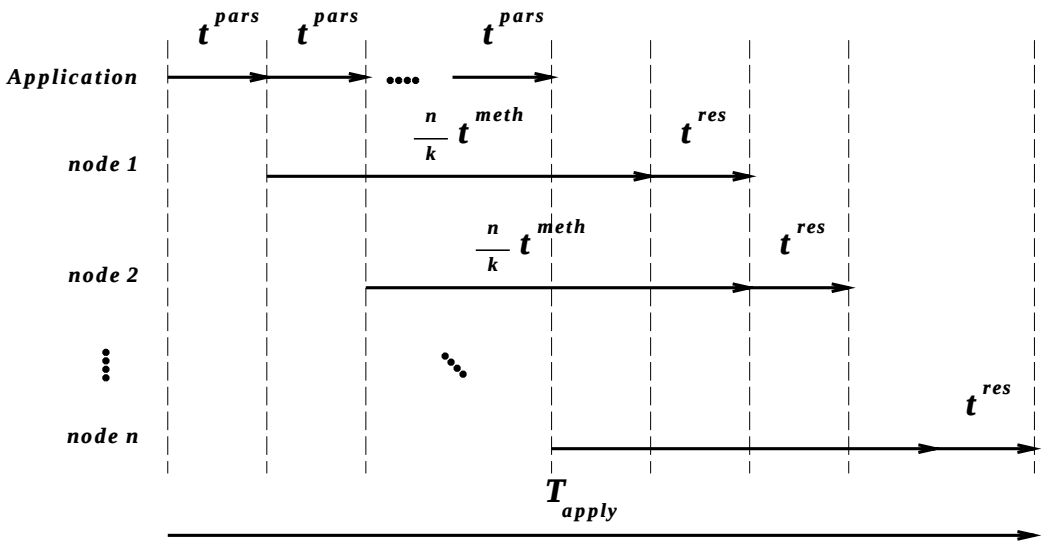


Figure 3: Time needed for a distributed operation

- Size of the class (i. e. number of objects in it) (n),
- Time needed to process a single object in main memory (t_{meth}),
- Time needed to transmit the parameters of the operation to a single node that holds part of the data (t_{par}),
- Time needed to transmit that part of the result that has been produced by a single node to the caller of the operation (t_{res}).

Example

Let us look at a typical operation upon a distributed class: A method shall be started on every object of the class. It has some constant parameters and produces some single valued scalar result. Such an operation might be to calculate a sum of a certain attribute of every object that meets a given criteria.

Fig. 3 shows the expense necessary for this operation. The parameters (together with the order to start the operation) have to be transmitted to all the nodes that hold part of the data of the class (t_{par}). As soon as a node has received these it will start its part of the operation in its main memory (t_{meth}). Afterwards every node will transmit its part of the result (t_{res}). The operation is finished when the last nodes

has transmitted its result. We call the time needed for the operation T_{apply}.

If the class is distributed among k nodes the parameter messages have to be retransmitted k times and k result messages have to be collected. As soon as the second parameter is transmitted, the first node starts to work[1]. Assuming that each node needs approximately the same time for its part of the operation, the resulting time is:

$$T^k_{apply} = kt_{par} + \frac{n}{k}t_{meth} + t_{res} \tag{1}$$

The acceleration due to parallelism is $(n - \frac{n}{k})t_{meth}$, the additional communication effort is $(k-1)t_{par}$. To describe the latency of our network we introduce a parameter L as the ratio of netcommunication-speed to process-speed in main memory:

$$L = \frac{t_{par}}{t_{meth}} \tag{2}$$

A large value of L means slow communication. A smaller value means faster communication or expensive methods. L is the number of objects that can be processed in a node before a single parameter message can be transmitted.

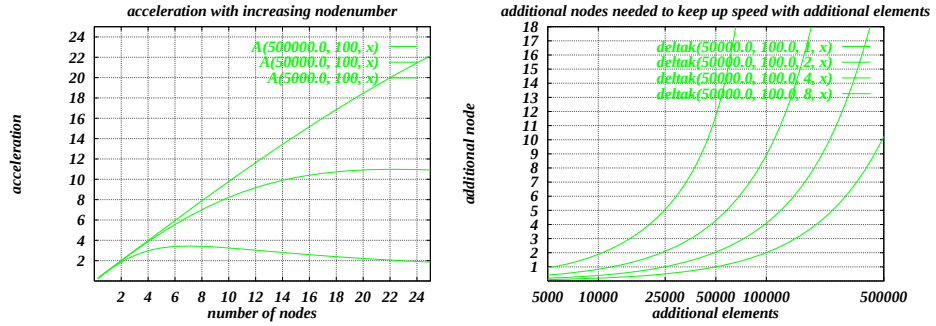
Acceleration of an operation

An operation is supposed to run faster if it is distributed over more than on node. But this acceleration is degraded by the time necessary to transmit the parameters of the operation to more than one node. We get:

$$AT^1_{apply} = T^k_{apply} \;\Rightarrow\; A = \frac{2L + n}{(k+1)L + \frac{n}{k}} \tag{3}$$

(A is the acceleration of the operation with data distribution compared with the operation running on a single node). Fig. 4(a) shows that with comparatively large class sizes and $L = 100$ the performance of the operation is increased first of all when we add nodes to the data distribution. But beyond a certain number of nodes the operation becomes

[1]We assume that the operation runs on a switching network, so no broadcast is available. A broadcasting network (like a ethernet bus) would permit to transmit the operation parameters in parallel. But the result messages are likely to collide on the bus after the operation has finished. That makes it more difficult to predict the total running time of an operation. Moreover, switching networks are usually faster than buses, so our assumption seams realistic.



(a) Acceleration

(b) Scalability

Figure 4: Distributed operation

slower when we add more nodes. We can differentiate Formular 3 to calculate the number of nodes $k_{fastest}$with which the operation should run fastest:

$$\frac{dA}{dk} = 0 \;\Rightarrow\; k_{fastest} = \sqrt{\frac{2Ln + n^2}{2L^2 + Ln}} \tag{4}$$

From an economic point of view, it does not make much sense to use $k_{fastest}$nodes for a data distribution. Fig. 4(a) shows that the performance gain between using 1 and $k_{fastest}/2$ nodes is much greater than the additional acceleration gained if more than $k_{fastest}/2$ nodes are used.

Scalability

A distributed main memory data store is able to keep the performance of a operation in spite of increasing class size by adding nodes to the distribution of the class. Let us look at the formulas above from a different point of view:

$$T^{k}_{apply} = T^{k+\Delta k}_{apply}(with\ n + \Delta n\ elements) \tag{5}$$

$$\Delta k = \frac{(\frac{n}{k} - kL) \pm \sqrt{(\frac{n}{k} - kL)^2 - 4L\Delta n}}{2L} \tag{6}$$

Given a data distribution for an operation upon a class of size n spread among k nodes: is the number of nodes we have to add to maintain the

speed of the operation if the class size is increased by factor m. The range of values of m for which a can be given depends on how "fast" the original distribution was. If the original class was distributed among a number of nodes that is close to $k_{fastest}$, increasing the class size is very expensive (Δk becomes large) and soon impossible without loosing performance. If the original distribution was less than optimal, we can add nodes to keep the performance of the original operation.

Fig. 4(b) shows values for in dependency of the relative increase of a class with $n = 50,000$, $L = 100$ for different distributions. The fastest distribution of that class is $k_{fastest} = 22$. The figure shows that extending the size of a class which was originally not distributed is inexpensive: It is possible to increase the size of the class and (nearly) keep the speed of operations in a single node. On the other hand, if the original distribution was speed optimized, it is much more expensive to increase the class size without performance loss.

3.2 Measurements

How fast would an operation in a PPOST based database actually run? What operation acceleration and size scalability can actually be expected? To answer these questions we define an example operation. We have simulated what would happen in a PPOST based database when this operation is started. What we have measured is presented in this subsection.

3.2.1 The example operations

Imagine a development department which works on optimizing some complicated machine (like the engine of a car). To monitor what happens when the machine runs, a lot of data is mesured with high frequency. This data consists of oil-, water-, exhaust-pressure and -temperature and other things like that. Also the exact time when certain events occur (like the completion of some cyclic action) is recorded.

Let us assume that the monitor collects data 1000 times per second on 16 channels. If the test runs 10 minutes, we get 10x60x1000 data records. We would like to answer the following questions:

- *How long does it take to insert all that data into a database?*
  Is it possible to insert the data on the fly in a crash safe fashion (if a powerfailure caused by the tested process happens, we do not want to loose the data collected immedeatly before).

- *How much additional space is required to store the data?*
  How much extra main memory not usable by application data do we need for storage structures?
- *How long do we have to wait for statistical data*
  We use the following example operations:
  1. Average and standard deviation of each chanel.
  2. List of averages between certain events.
  3. Distribution of data for each chanel (which requires sorting).
- *How do these costs evolve if the amount of data increases*
  Say we want to extend the test time from 10 minutes to 20 minutes. Beside the additional memory resources needed, how much performance is lost by extending the test?

3.2.2 The test setup

We will simulate this application on a farm of 12 Digital-Alpha/OSF-1 mashines (they are equipted with a 133 MHz processor with 128 MBytes of main memory; the nodes are connected with a fast switching FDDI net). The code to implement the test was written in Modula-3, a clean object-oriented language ([13, 2]). For comparison we also run similar operations on a standard comercial disk based database system (Oracle) on a single mashine. We implement:

- *Distribution*
  with a simple distributed hash table. There is a “master” on one of the nodes which cuts the hash table in equal parts. Data with a hash index belonging to a certain part is stored on one of eight data nodes.
- *Persistence*
  by creating a redo log of all insert and remove operations on a sequential file on the master node. From time to time we copy the contents of the main memory of all data nodes to their local disks. The redo log is created by the *stable object package*, the copy of the main memory data structures with the *pickle package*. Both packages are part of the Modula-3 library ([13, 12]).

3.2.3 Results

. . .

4 Conclusions

Traditional limits of memory-resident databases can be mastered by the use of parallelism. Given a relation between the processing speed of individual nodes and communication, the minimal size of classes can be stated, from which the memory-resident store scales nearly linearly. This actually means that we can either add nodes to store and process more information at constant speed, or we can add nodes to process the same information faster. Main memory based object-oriented databases have the additional advantage that methods stored in the database can be executed in their primary store.

Parallelism can be used in providing persistence as well: processing of log information, creation of disc backups and tape archives can all be done in parallel to normal transactions. Therefore, normal transactions are entirely decoupled from I/O on nonvolatile storage.

Acknowledgments

The authors thank J. Eder and M. Dobrovnik for many valuable suggestions.

References

[1] P. Apers and C. van den Berg. Prisma/db: A parallel, main memory relational dbms. *IEEE Transactions On Knowledge And Data Engineering*, 4(6), December 1992.

[2] L. Böszörményi. A comparision of modula-3 and oberon-2. *Structured Programming, Springer Verlag*, pages 15–22, 1993.

[3] L. Böszörményi and K.H. Eder. Adding parallel and persistent sets to modula-3. In *Proceedings of the Advances In Modular Languages (JMLC'94)*, pages 201–215. Universitätsverlag Ulm GmbH, September 1994.

[4] L. Böszörményi and A. Stopper. A distributed, object-oriented simulation system based on hints. In *Proceedings of the Eurosim'95*, pages 297–302. Elsevier, North-Holland, September 1995.

[5] M. Dobrovnik and J. Eder. A concept of type derivation for object-oriented database systems. In L. Gün et. al. (eds.), editor, *Proceedings of the Eight International Symposium on Computer and Information Sciences (ISCIS VIII)*, Istanbul, 1993.

[6] M. Dobrovnik and J. Eder. View concepts for object-oriented databases. In G. Lasker (ed.), editor, *Proceedings of the Fourth International Symposium on Systems Research, Informatics and Cybernetics*, Baden-Baden, 1993.

[7] J. Gray edited. *The Benchmark Handbook*. Morgan Kaufmann Publishers Inc., 1993.

[8] H. Garcia-Molina and K.Salem. Main memory database systems: An overview. *IEEE Transactions On Knowledge And Data Engineering*, 1(2), March 1990.

[9] H. Garcia-Molina and K.Salem. Main memory database systems: An overview. *IEEE Transactions On Knowledge And Data Engineering*, 4(6), December 1992.

[10] J. Gray and A .Reuter. *Transaction Processing – Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[11] P. Heywood, G. MacKechnie, R. Pooley, and P. Thanisch. Object-oriented database technology applied to distributed simulation. In *Proceedings of the Eurosim'95*, pages 291–296. Elsevier, North-Holland, September 1995.

[12] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful Modula-3 interfaces. Research Report 115, Digital Systems Research Center, Palo Alto, 1994.

[13] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[14] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. In *International Conference On Data Engineering*, pages 452–462, Los Angeles, 1989.