Transaction Management in Distributed Scheduling Environment for High Performance Database Applications

Sushant Goel[1], Hema Sharda[1], and David Taniar[2]

[1] School of Electrical and Computer Systems Engineering
Royal Melbourne Institute of Technology, Australia
s2013070@student.rmit.edu.au
hema.sharda@rmit.edu.au
[2] School of Business Systems, Monash University, Australia
david.taniar@infotech.monash.edu.au

Abstract. Synchronising the access of data has always been an issue in any data-centric application. The problem of synchronisation increases many folds, as the nature of application becomes distributed or volume of data approaches to terabyte sizes. Though high performance database systems like distributed and parallel database systems distribute data to different sites, most of the systems tend to nominate a single node to manage all relevant information about a resource and its lock. Thus transaction management becomes a daunting task for large databases in centralized scheduler environment. In this paper we propose a *distributed scheduling* strategy that uses a distributed lock table and compares the performance with centralized scheduler strategy. Performance evaluation clearly shows that multi-scheduler approach outperforms global lock table concept under heavy workload conditions.

1 Introduction

Continuously growing volume of data and expanding business needs have justified the needs of high performance database systems like *Distributed* and *Parallel Database Systems* (PDS) [2,8,10,11]. Distributed databases are the logically integrated systems that make the distribution of the data transparent to the user [11]. When the volume of data at a particular site grows large and the performance of the site becomes unacceptable, parallel processing is needed for data servers. PDS supports automatic fragmentation and replication of data over multiple nodes [2].

Single scheduler approach has been used as the correctness criterion for transaction management. With the increase in volume of data, message and locking overhead also increases in centralized locking approach. We will refer single scheduler transaction management approach and centralized locking interchangeably throughout the paper.

S. R. Das, S. K. Das (Eds.): IWDC 2003, LNCS 2918, pp. 120-130, 2003.

Most database management systems use single-scheduler transaction management approach [2,3,8,9,10,11]. But with increase in volume of data, managing transactions with single scheduler may become difficult and computationally expensive. Single scheduler strategy may not meet the requirements of high performance database systems as the database scales to terabyte sizes. Distributed lock managers have been proposed in the literature to distribute the load among multiple sites [9,10,11]. Unfortunately distributed lock manager nominates one node to manage all the information about the resource and its locks. The distributed lock manager still has to manage a global lock table [11], thus the architecture is not truly distributed and the number of messages in the network is also high due to internode communications. With increase in amount of the data stored, some of the single scheduler inherent weaknesses become prominent, e.g. big lock table, increased number of messages in the system, deciding the coordinator for the transaction etc.

This motivated us to distribute the scheduling responsibilities of the data items to their parent nodes where the data is residing and consider *distributed schedulers* for transaction management and to maintain the consistency of data items. Migrating from single-scheduler transaction management approach to distributed scheduler transaction management approach may pose a threat to the consistency of the data items. The underlying problem in direct migration is discussed later in the paper. We use multi-scheduler and distributed scheduler interchangeably in this paper.

We discuss the problem in migrating the single scheduler algorithms to multi-scheduler environment in the following sections. We will discuss the case for *distributed memory* also known as *Shared-Nothing* parallel database systems [2] in this paper but this concept can be modified to meet the requirements of other high performance data-centric applications.

We would like to highlight that mainly three types of transaction management strategies are used for distributed memory architecture in the literature [10]: 1) *centralized locking* 2) *primary-copy locking* 3) *distributed locking*. All three strategies either manage the locking table centrally or manage a portion of the *global lock table*. In this paper we focus to reduce the load of global lock table and remove the requirement of centralized scheduling.

Rest of the paper is organized as follows: Section 2 discusses the related work done in distributed database system and multidatabase systems, Section 3 explains our working environment and elaborates the proposed algorithm, section 4 shows and the performance comparison for single and multi-scheduler strategies. Finally, Section 5 concludes the paper and discusses the future extension of the work.

2 Related Work

In this section we discuss different high performance database systems like multidatabase, distributed database and parallel database systems. We also very briefly discuss the environment of transaction management required by these database systems.

2.1 Multidatabase Systems

Multidatabase system is an interconnected collection of autonomous databases. A multidatabase management system is the software that manages a collection of autonomous databases and provides transparent access to it [9]. Multidatabase management system provides Database Management System (DBMS) at two different levels – *i)* local *ii)* global. Transaction manager at local sites can guarantee correct schedules for local transactions only. A global transaction manager spanning all sites has to be designed to meet specific requirement. For detailed discussion please refer [9]. Multidatabase system (MDS) is a good option when individual databases have to be combined logically. If large volume of data has to be managed and data distribution is an important factor in performance statistics, then MDS may not be the preferred design option. MDS is best suited when autonomy is the key but certain transactions may span more than one database.

2.2 Distributed Database Systems

Data can be manually partitioned over geographically separated databases but can still appear as one unit to the user. The data is transparently distributed over nodes of distributed database. Distributed DBMS manages transactions by global transaction manager and global lock manager [10,11]. Distributed databases have high locking and message overhead due to the distributed nature of application. Lock table may become a performance bottleneck despite the distribution of data over multiple sites due to its global nature. Global lock table maintains the locking information about all data items in the distributed database thus the scheduling strategy act as a centralized scheduling scheme. Thus, despite most of the applications distribute data to multiple sites they still use a centralized scheduling scheme and maintain a global lock table.

2.3 Parallel Database Systems

The enormous amount and complexity of data and knowledge to be processed by the systems imposes the need for increased performance from the database system. The problems associated with the large volumes of data are mainly due to: 1) Sequential data processing and 2) The inevitable input/output bottleneck. Parallel database systems have emerged in order to avoid these bottlenecks. Different strategies are used for data and memory management in multiprocessor environment to meet specific requirements. In s*hared memory architecture* [2], processors have direct access to the disks and have a global memory. In s*hared disk architecture* [2], processors have direct access to all disks but have private memory per processor. *Shared-nothing architecture* [4] has individual memory and disk for each processor, called *processing element (PE)*. The main advantage of shared-nothing multi-processors is that they can be scaled up to hundreds and probably thousands of PEs that do not interfere with one another, refer [1, 2, 4, 5, 6] for detailed discussion.

Though shared-nothing database systems have individual data partitions, to achieve correctness of schedule and consistency of data these database systems also rely on central transaction management schemes. Looking at the drawbacks of centralized

transaction management schemes we distribute the transaction management responsibilities to individual PEs.

2.4 Basic Definitions and Problem Identification

We would like to briefly define the *transaction* and properties of transactions before we discuss the problem in direct migration from single scheduler to distributed scheduler environment. From the user's viewpoint, a transaction is the execution of operations that accesses shared data in the database; formally, a *transaction* T_i is a set of read (r_i), write (w_i), abort (a_i) and commit (c_i). T_i is a partial order with ordering relation $\prec_i$ [7].

A *history* or *schedule* indicates the order in which the operations of the transactions were executed relative to each other. Formally, let $\boldsymbol{T} = \{T_1, T_2, \ldots T_n\}$ be a set of transactions. A complete history (or schedule) H over $\boldsymbol{T}$ is a partial order with ordering relation $\prec_H$ [7].

A history H is *Serializable (SR)* if its committed projection, *C(H)*, is equivalent to a serial execution H_s. A database history H_s is serial iff

$$(\exists p \in T_i, \exists q \in T_j \text{ such that } p \prec_{Hs} q) \text{ then } (\forall r \in T_i, \forall s \in T_j, r \prec_{Hs} s).$$

For detailed description of the definitions refer [7]. A history (H) is serializable iff serialization graph is acyclic. A transaction must possess ACID properties [7] and these properties must be enforced by the concurrency control and recovery management techniques implemented in the DBMS.

Problem identification: The algorithms developed for single scheduler might produce contradictory serialization order at different nodes in multi-scheduler environment, if two transactions access more than one processing elements simultaneously. We have already discussed the inherent weaknesses of single scheduler algorithms applied to high performance database systems [13]. We propose a new serializability, *Parallel Database Quasi-serializability,* and compare the performance of this distributed scheduling strategy with the centralized scheduling strategy in this paper.

3 Proposed Algorithm

Before discussing the algorithm we first present the working environment for our model in subsection 3.1, we then briefly discuss the correctness criterion for the multi-scheduler concurrency control algorithm namely *Parallel Database Quasi-serializability* (PDQ-serializability) in subsection 3.2. Finally, subsection 3.3 explains the proposed *Timestamp based Multi-scheduler Concurrency Control (TMCC)* algorithm. We have already proposed the algorithm in [13]. Here we give an overview of the algorithm, and we focus on the performance comparison between the two strategies.

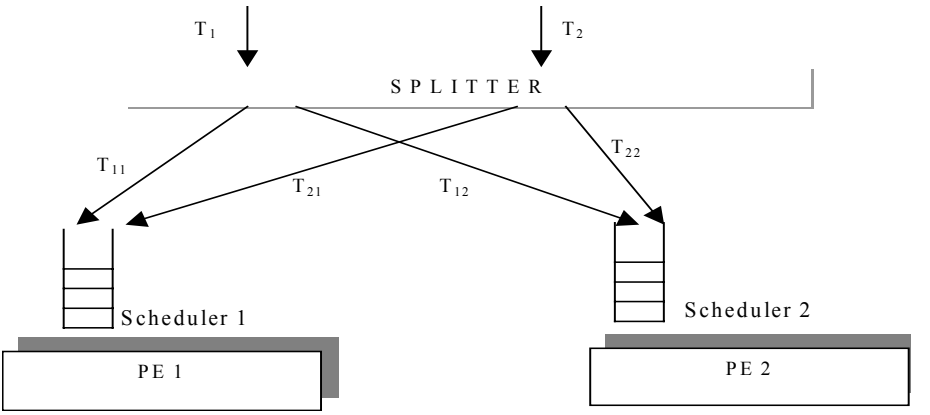

Fig. 1. Conceptual model of multi-scheduler concurrency control

3.1 Working Environment

The model presented in this paper has additional scheduling responsibility of transactions per processing element. The traditional model shown has centralized scheduling responsibility and a global lock table. We try to distribute the scheduling responsibility to respective PEs, where data is located, (see fig. 1) in order to take maximum advantage of the abundant computing power available.

We have already said that single scheduling strategy may produce incorrect serialization order in multi-scheduling environment [13]. For the sake of simplicity we consider only two processing elements to demonstrate our algorithm. We assume that local schedulers are capable of producing serializable schedule.

3.2 PDQ-Serializability

Scheduling responsibilities in PDS is distributed to respective PE according to the partitioned data. Problem in migrating from single-scheduler to multi-scheduler have been discussed in [13]. Thus, we conclude that operation level concurrency control cannot ensure a correct schedule to be produced in multi-scheduler environment. Even, subtransaction level granularity in itself is not sufficient to produce correct schedule. Some additional criterion has to be enforced to ensure multi-scheduler concurrency control in addition to subtransaction level granularity.

A new serializability criterion is proposed that separates two types of transactions – transactions having only one subtransaction and transactions having more than one subtransaction. The following definition states the correctness criterion for PDS.

Definition 1: A *Multi-Scheduler Serial (MS-Serial)* history is considered correct in parallel database system.

Definition 2: A history in multi-scheduler environment is PDQ-serializable iff it is equivalent to a *MS-Serial* history (definition of equivalence ($\equiv$) from [7], page 30).

The PDQ-serializability of the history is determined by analysing *Parallel Database Quasi-Serializability* (PDQ-serializabilty) graphs. Only the committed projection [7] of the history *C(H)* is considered in the definition. The following definition describes the PDQ-serializability graph.

Definition 3: At any given instance, histories of all the schedulers at each PE can be represented using a directed graph defined with the ordered three: ($\mathbf{T}^1$, $\mathbf{T}^n$, $\boldsymbol{A}$). The graph would be referred as *Multi-scheduler Serializability Graph (MSG)*.

$\mathbf{T}^1$ and $\mathbf{T}^n$ are the set of labeled vertices representing transactions with one *subtransaction* and more than one *subtransaction* respectively. $\boldsymbol{A}$ is the set of arcs representing the ordering of transaction in each PE. In rest of the paper we would only consider transaction having more than one subtransaction and denote that as *T*, without the *superscript*. Transactions with single subtransaction are taken care by the individual PE and do not pose any threat to the concerned problem.

Based on the definition of MSG we next formalize the following theorem:

Theorem 4: A history in multi-scheduler environment is PDQ-Serializable iff MSG is acyclic.

Proof: Please refer [13] for proof.

Our earlier work [13] discusses the above definitions and theorem in detail. In this paper we focus on the performance evaluation of the proposed algorithm and comparison with global lock table (single scheduler) approach.

3.3 Timestamp Based Multi-scheduler Concurrency Control Algorithm

In this section we propose a *Timestamp based Multi-scheduler Concurrency Control* (TMCC) algorithm that enforce *total order* in the schedule to ensure PDQ-serializability. *Total order* is required only for those conflicting transactions that accesses more than one PE being accessed by other active transactions. Functions that the algorithm uses are *Split_trans(T_i)*, *PE_accessed(T_i)*, *Active_trans(PE)*, *Cardinality()*, *Append_TS(Subtransaction)*. Function names are self-explanatory, definitions of the functions can be found in [13].

Working of the TMCC algorithm is explained below:

1. When transaction arrives at the splitter, *split_trans(T_i)* splits the transaction into multiple subtransactions, depending on allocation of data.
2. If there is only one subtransaction required by the transaction, the transaction can be submitted to the PE immediately without any delay.
3. All the transactions having more than one subtransaction are added to the **Active_Trans** set.
4. If multiple subtransactions are required by the transaction, the *splitter* appends a timestamp with every subtransaction.
5. If there are active transactions that access one or less than one PEs accessed by the transaction being scheduled then the subtransactions can be scheduled immediately. This condition does not pose any threat to the serializability.

6. If there is active transactions that access more than one of the PEs accessed by the transaction being scheduled then the subtransactions are submitted to the PE's *wait_queue*.
7. When all subtransactions of any transaction complete the execution at all the sites, the transaction commits and is removed from *Active_trans(PE)*. (*Active_trans(PE)* takes the processing element as an argument and returns the set of transactions running at that PE)

4 Performance Evaluation

In this section, we discuss the results obtained by simulation for the two scheduling strategies namely single scheduler and multi-scheduler. We run our simulation for various sizes of the system, under various loading conditions (heavy and light) and also at different write probability of the transaction. We used CSIM [12], a process-oriented, discrete-event simulation package to obtain our results. The simulation code was written in C++. We discuss the performance metrics and parameter settings next, before the performance experiments and results.

4.1 Performance Metrics and Parameter Settings

Performance metrics for most of the high performance database systems are *response time* and *throughput*. But our perspective of this research is different and hence the performance metrics are also different. In this paper our focus is to evaluate the performance of single and multi-scheduler.

Table 1. Parameter settings for simulation

Description	Value
Number of Processors	2-8
Write probability	15%-60%
Minimum number of data items accessed	4
Maximum number of data items accessed	12
CPU access time for data (*cpu_accesstime*)	1.5 millisecond
Disk access time for data	35 millisecond
Lock factor (*lock_factor*)	10 % of disk access time
Read time for a data object	((cpu_access_time) + (lock_factor * *lock_table_size*))
Write time for a data object	25 % more than read time.
Maximum wait before the transaction aborts	4 second
Number of pages per node (*Num_pages*)	125, 185
Data partitioning strategy	Range partitioning
Inter-arrival time between transactions (exponential distribution with mean *m)*	1 – 3.5 seconds.

Our first performance metric is the *abort-ratio*. Abort ratio is calculated by dividing the number of aborting transactions by the number of submitted transactions. We measure the abort ratio against different loading conditions (light and heavy). Depending on type of transaction, transactions can have different *write-probability*. We consider different write-probability to measure the blocking of transactions. Write-probability of the transaction can be thought as: how much percentage of read data item is being written. Finally, we increase the number of processors and measure the *utilization* of the processors for both scheduling strategies.

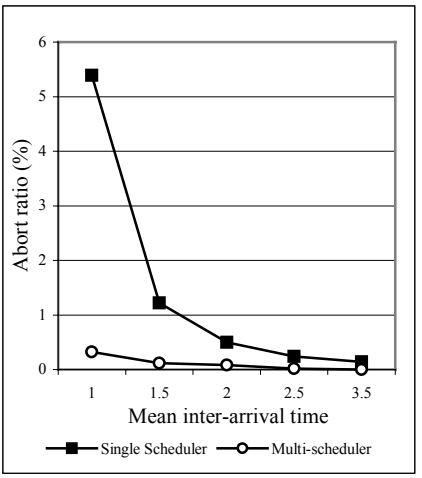

Fig. 2. Num_page = 125

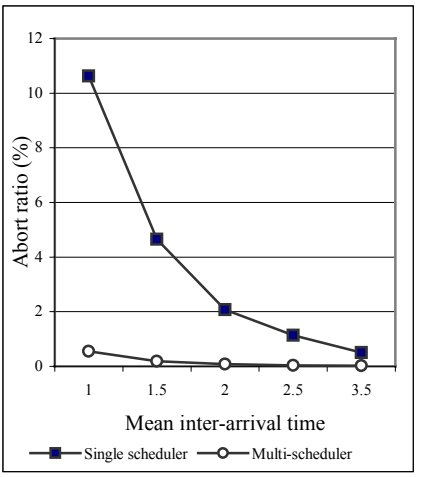

Fig. 3. Num_pages = 188

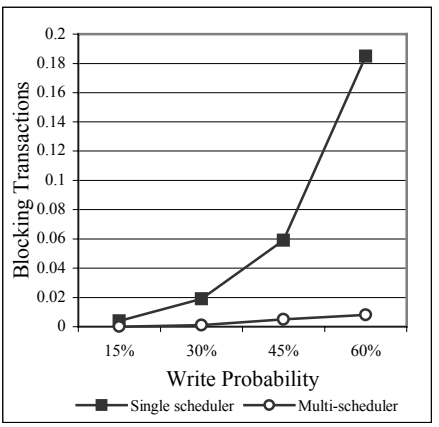

Fig. 4. Blocking Transactions

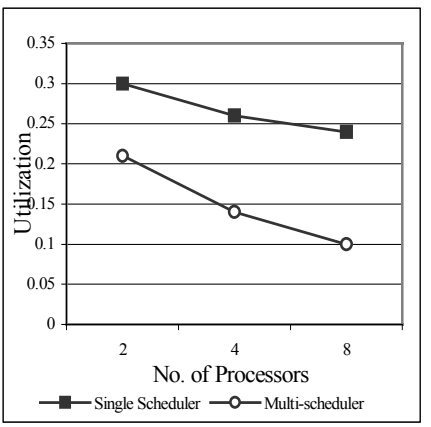

Fig. 5. Utilization

Table 1 shows the parameter settings for the experiments. Specific parameter setting values are again mentioned in the concerned experiment. OLTP (On-Line

Transaction Processing) workload is assumed for the system. We assume the 80-20 rule for data access. 80-20 rule states that 80% of the transactions access 20% of the data items and they are known as hotspots. Inter-arrival time for the transaction is assumed to be *exponential distribution* with mean *m*. Data is partitioned across all the PE's using *range partitioning* with equal number of data objects in each node.

4.2 Results and Discussions

Fig. 2 and 3 shows the performance results for 2-node case. Fig. 2 shows the variation of *abort ratio* with different loading condition for database size of 125 pages. We note that as the *mean* of inter-arrival time approaches to the maximum waiting time, abort ratio for both the strategies coincides. Under heavy loading conditions (i.e. with less inter-arrival time) data contention increases, thus the locking overhead also increases and consequently the abort ratio increases for single scheduler as well as for multi-scheduler strategy. We note that the rate of increase of abort ratio is much higher for single scheduler. The graph clearly shows the effect of locking overhead in single scheduler strategy is higher than multi-scheduler strategy.

Next we run the experiment under same environment by increasing the data volume by 1.5 times, that increases the number of pages per node from 125 to 188 (see fig. 3). The nature of the graph is same as that of fig. 2 but an interesting fact to note is that the rate of increase in abort ratio is less in multi-scheduler strategy. Under heavy loading conditions (inter-arrival time with $m = 1$) the increase in abort ratio for single scheduler is 97.22% but for multi-scheduler strategy the increase in abort ratio is 71.87%. Thus increasing the size of the database adversely affects the performance of single scheduler as discussed in previous sections.

Fig. 4 presents the relation between write-probability and percentage of blocking transactions. Number of nodes for this experiment was 4 and the mean of the inter-arrival time was 2.5 second. Depending on the type of transaction e.g. update, read-only etc., write-probability of the transaction may vary. Fig. 4 shows the effect of write-probability on the percentage of transactions that are blocked. We vary the write-probability from 15% to 60% and measure the percentage of blocking transactions for both single and multi-scheduler strategy. Observation shows a marginal difference between the two strategies at higher write-probability. At lower write-probability of 15% the number of blocking transactions for both strategies are approximately equal but with increasing write-probability the separation between the two strategies increases exponentially. Write operations acquire locks in exclusive mode and also tend to keep locks for longer duration. Hence, with increasing write-probability lock contention increases and centralized or single scheduler's performance deteriorates greatly.

The next experiment was motivated to test the affect of distributing the scheduling responsibilities on scale-up issues of high performance database systems. We vary the number of processors to 2, 4 and 8, then measure the average utilisation of the processors (see fig. 5). As expected the utilisation of the processors reduces in both the strategies but the rate of reduction in both the cases are different. Decrease in average percentage utilisation for the single scheduler case is 13% for 4 processors and 20% for 8 processors (percentage reduction is measured with respect to 2

processors). For multi-scheduler strategy the decrease in average percentage utilisation is 33% for 4 processors and 52% for 8 processors. Thus in multi-scheduler strategy, with increase in number of processors the average utilisation of processors reduces faster and the processor can be used to serve more transaction and thus achieve scale-up.

5 Conclusion

This paper aimed to evaluate and compare the performance of single and proposed multi-scheduler policies in high performance database applications. The motivation behind this work was to reduce the message and locking overheads of centralized scheduling scheme. Though the performance of both strategies is comparable under light workload, the performance of single scheduler deteriorates exponentially under heavy loading conditions. Thus, we can conclude that multi-scheduler algorithms show better chances of scalability and performance under heavy workload conditions. The reason for enhanced performance of multi-scheduler algorithm (TMCC in this case) is due to better distribution of work to individual nodes.

In future we plan to test the sensitivity of multi-scheduler strategy to different data distribution strategy e.g. hash partitioning and round robin partitioning. We also intend to investigate the direct relation of speedup and scaleup with the multi-scheduler strategy. The performance of the algorithm may also be examined for different loading conditions of the application such as decision support queries and mixed workloads (OLTP and decision support queries).

References

[1] Bhide, “An Analysis of Three Transaction Processing Architectures”, *Proceedings of 14th VLDB Conference*, pp. 339-350, 1988.

[2] D.J. DeWitt, J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems", *Communication of the ACM*, vol. 35, no. 6, pp. 85-98, 1992.

[3] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[4] M. Stonebraker, "The Case for Shared-Nothing", *IEEE Data Engineering*, vol. 9, no. 1, pp. 4-9, 1986.

[5] P. Valduriez**, “**Parallel Database Systems: The Case For Shared Something”, *Proceedings of the International Conference on Data Engineering*, pp. 460-465, 1993.

[6] P. Valduriez, "Parallel Database Systems: Open Problems and New Issues", *Distributed and Parallel Databases*, vol. 1, pp. 137-165, 1993.

[7] P. A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addision-Wesley, 1987.

[8] T. Ohmori, M. Kitsuregawa, H. Tanaka, “Scheduling batch transactions on shared-nothing parallel database machines: effects of concurrency and parallelism” *Data Engineering, Proceedings, Seventh Intl. Conference on,* 8-12, pp: 210 –219, Apr 1991.

[9] K. Barker, “Transaction Management on Multidatabase Systems”, PhD thesis, Department of Computer Science, The university of Alberta, Canada, 1990.

[10] T. Ozsu, P. Valduriez, "*Distributed and Parallel Database Systems*", *ACM Computing Surveys,* vol.28, no.1, pp 125-128, March 1996.

[11] M.T. Ozsu and P. Valduriez, editors. *Principles of Distributed Database Systems* (Second Edition). Prentice-Hall, 1999.

[12] CSIM, User's Guide CSIM18 Simulation Engine (C++ Ver.), Mesquite Software, Inc.

[13] S. Goel, H. Sharda, D. Taniar, “Multi-scheduler Concurrency Control Algorithm for Parallel Database Systems”, *Advanced Parallel Processing Technology, Lecture Notes in Computer Science*, Springer-Verlag, 2003.