Vol. 45 No. SIG 14(TOD 24) IPSJ Transactions on Databases Dec. 2004

Regular Paper

PO-P: A Concurrency Control Protocol for Parallel B-trees

Damdinsuren Amarmend,† Masayoshi Aritsugi†
and Yoshinari Kanamori†

A number of parallel and distributed B-tree structures for shared-nothing parallel database systems have been proposed in connection with the related technological advancements such as clustered computers. We focus on the performance of the concurrency control protocols for such parallel B-trees and consider the possibility and the suitability of introduction of preparatory operations such as preemptive node split or merge to them in this paper. Since the early top-down concurrency control protocols were only designed for shared-everything architecture, it is inappropriate to directly apply them to the distributed environment because the way update processes traverse the B-tree may cause significant degradations in the system performance. In this paper, we propose a protocol, named preparatory operations-parallel (PO-P), in which such preparatory operations are used during traversals of update processes. The results of our implementation of PO-P on a parallel B-tree in comparison with the existing protocol INC-OPT prove that the PO-P can serve as a good alternative protocol for distributed parallel B-trees.

1. Introduction

Shared-nothing parallel database systems are being considered suitable for large-scale data processing applications with high performance and high capacity and availability requirements. An instance of such a system consists of some independent processing elements (PE) each of which has its own memory, storage and CPU while interconnected through high speed networks.

To provide effective data processing and management in such systems, parallel distributed B-tree structures have been proposed (e.g., Refs. 10), 12)). The effectiveness of the concurrency control protocol applied to the parallel B-tree is crucial for the systems performance.

Although there have been several concurrency control protocols proposed for the ordinary B-trees, none of them are considered to be suitable for parallel B-trees because their direct application is potential to create deadlocks and may require expensive synchronizations between distributed nodes in case of update operations[6]. Therefore, an appropriate concurrency control method is needed for concurrently running transaction processes on parallel B-trees to meet the high performance requirement. An incremental optimistic protocol named INC-OPT[6] has been proposed for this purpose.

In this paper we aim at improving the INC-OPT protocol, in which every update operation is done only after all the necessary latches are obtained on the whole update scope. For distributed parallel B-tree, performing an update operation in this manner is potentially expensive. Because its synchronization with other PEs can block many other processes which want to traverse the parallel B-tree in parallel until the whole update operation is complete. For example, all updates whose scope is from the leaf to the root node of the tree create such long waits.

This situation can be improved by using top-down methods in which preparatory operations (POs)[3),9)] are used. When the POs are introduced to the parallel B-tree, there will not be any highly cascaded update operations. Because update processes preemptively split or merge nodes through their traversals. That means, the POs can help to get more concurrent processes run on the tree than in the INC-OPT. However, the original top-down protocols such as POM[9], TD-OPT[3] can not be directly applied to the parallel B-tree because their use of the X, S and SIX latches on the upper part of the tree may cause unnecessary expensive synchronizations between PEs.

In this paper, we consider the possibility and the suitability of using the POs in parallel B-trees, propose an optimistic protocol named preparatory operations-parallel, or PO-P, and evaluate its effectiveness in comparison with the INC-OPT.

The rest of the paper is organized as follows:

† Faculty of Engineering, Gunma University

In Section 2, we discuss the related work in detail. Section 3 describes the PO-P protocol. Section 4 presents the implementation details and explains the results of experiments on that. Finally, Section 5 concludes the paper.

2. Related Work

In Ref. 6), the concurrency control protocols such as B-X, B-SIX, B-OPT[1)], OPT-DLOCK[11)], ARIES/IM[7),8)] and B-link[5)] for conventional B-tree are considered to be not suitable for parallel B-trees, and the INC-OPT[6)] has been thus proposed as a new optimistic protocol for parallel B-trees.

The INC-OPT is based on the B-OPT, but differs from it in the second phase of the update process. An update process in the INC-OPT traverses the tree from the root to the desired leaf node using the IX latch-coupling[2),6)~8)]. If the process finds that the leaf node needs any structure modification operation (SMO) such as node split or merge of nodes, the process releases the all acquired latches. In the second phase, the process retraverses from the root by the IX latch-coupling until the process reaches the top node of the update scope. From that point, it traverses getting X latches on the whole update scope. In this way, the process may have to traverse again if the top node of the scope needs any SMO. In the worst case, the update scope could be the path from the root to the desired leaf and latched by X latches. In brief, the INC-OPT uses the latches at appropriate places.

However, there is a case where the INC-OPT's performance may degrade. In an update intensive environment, there will be much occurrences of SMO which involves more than two levels of the parallel B-tree. Due to the nature of the distributed parallel B-tree, some nodes of the tree can be copied on several PEs for the sake of parallel access[4),12)]. Also, it is appropriate to have more copy nodes at the upper levels of the tree while less copy nodes at the lower levels because updates are likely to occur often in the lower levels and retrievals are likely to occur often at the upper levels[6)]. During any update, which involves these copy nodes, they must be synchronized and reflected.

Upon this facts, if the update scope is higher and involve more copy nodes, then the INC-OPT may block other local and remote processes waiting for longer time. For example, if the update scope equals the path from the leaf to the root, other PEs can not even access the root node until the update is finished.

To improve this situation, we study the suitability of introduction of preparatory operations into parallel B-tree.

If a node has full or insufficient entries then it is referred to as an unsafe node otherwise as a safe node. If an update process finds any unsafe node on the path, it can preemptively split the node or merge with another node making the node safe. These operations are referred to as preparatory operations. This idea was already embodied in the top-down protocols such as POM[9)], TD-X, TD-SIX, and TD-OPT[3)]. However, they were only designed for conventional B-trees.

In the TD-X, update processes use the X latch-coupling during their traversals, not allowing other processes to read the latched nodes. At each level the process checks the child node and performs PO if the node is unsafe.

Update processes in the TD-SIX use the SIX latch-coupling allowing other retrieval processes to read the latched node but disallowing other update processes. If the process encounters any unsafe node, it tries to convert the SIX latches into the X latches.

In contrast, an update process in the TD-OPT has two phases when it faces an unsafe node. In the first, it traverses using the IX latch-coupling from the root to the leaf node. If the leaf node is unsafe then the process releases the latch and traverses again from the root using the SIX latch-coupling like in the TD-SIX. As for the performance, the TD-OPT is better than the other two protocols, but in the second phase it still blocks other update processes in the upper part of the tree.

The direct application of the TD-OPT to the parallel B-tree is inappropriate because using the SIX latch on the upper part of the tree is expensive. It means, for example, the root node will be latched often while the root node is not included in the actual update scope. Thus, it may keep other remote processes unnecessarily waiting for a long time.

Therefore, we combine the PO with the INC-OPT's use of the latches in order to effectively use the POs for parallel B-trees.

3. PO-P Protocol

We propose an optimistic PO-P (preparatory operations-parallel) concurrency control as an alternative protocol for parallel distributed B-tree structure. The PO-P is a deadlock free protocol as proved in Theorem 1. The PO-P employs the POs only for update processes. For retrieval processes, they traverse using the IS latch-coupling as in other optimistic protocols.

A PE which initiates a PO is called an initiator. A PE which has a copy of a node is named a copy PE. The initiator always has all the addresses of the copy nodes of its local nodes. That address consists of a copy PE number and its local node address. Before doing any PO, the parent node and its copies and the unsafe child node and its copies must be latched with X latches. A PO in the PO-P is rather different than the conventional one because it deals with distributed nodes.

A PO in the PO-P is defined as follows:

(1) Inform all the copy PEs of the unsafe node to start the PO. It means, the initiator sends a special message to each copy PE including the copy node address.

(2) Each copy PE, including the initiator, performs the POs which can be split or merge, and updates the parent nodes. These POs are assumed to be done in parallel. Each participating PE has a process to perform the PO.

(3) They also define the state(s) of the resulting node(s) from the PO to be local or remote. If a node does not contain any local pointer, it is referred to as a remote node, and it must be removed from the PE. Otherwise, it is referred to as a local node and will be kept locally.

(4) Each copy PE sends the address of the resulting node(s) together with their state(s) to the initiator PE.

(5) The initiator PE collects the information, then sends all the information to the copy PEs. The initiator must also send the information to the copy PEs of the parent node, which have remote pointers to the unsafe node but have no local copy of it. They update the parent node and its remote child pointer.

(6) Based on the information, each PE adjusts the pointers in the parent. If the child node is remote, the pointer in the parent must point to the address of its copy nodes. Also, the addresses of the copy nodes of the child node(s) must be updated.

To obtain latches on the copy nodes, the PO-P uses linear order latching as used in the INC-OPT. According to this method, for example, PE1 will have the highest priority to get a latch on the root.

Based upon the definition given above, we describe the protocol in **Fig. 1**. Let H be the height of the tree. The variable *height* is used to indicate the level of any unsafe node while the variable $unsafe\_node$ is used to indicate the occurrence of any unsafe node. Initially, *height* is set to H and $unsafe\_node$ is to No.

At first, each update process optimistically traverses the tree down using the IX latch-coupling. The target leaf node will be latched with an X latch. If the leaf node is safe, the process updates it and completes its operation by unlatching the leaf. This part is described in Main Module in Fig. 1 in detail. If the leaf node is unsafe, then the process sets $unsafe\_node$ to Yes and decreases *height* by one to get an X latch on the parent of the leaf. Then, the process releases the leaf's latch and restarts from the root as described in Module 2 in Fig. 1. According to Module 2, the process now pessimistically checks every node on the way whether it is safe or not, even though the leaf node could happen to be made safe by another process during the restart. If any unsafe node is encountered, the process makes it safe. In order to do that, it first gets X latches on the parent and its copies and on the child node and its copies in the retraversal(s). Only after all the necessary latches are obtained, the process starts the appropriate PO as defined above. After finishing the PO, the process releases the X latches and continues its traversal. The update process completes finally updating the leaf node. Until then several retraversals can be made.

The fundamental property of the PO-P is the adaptation of the conventional POs for the parallel B-tree. The PO-P allows multiple processes to traverse the parallel B-tree concurrently, even performing multiple POs unless their update scopes do not affect each other.

The PO-P is similar to the INC-OPT except for the fact that it uses the POs for update operations. The update scope of the INC-OPT may reach from the leaf to the root node while any update of the PO-P involves only two levels of the tree. For a distributed environment in which multiple PEs participate, performing highly cascaded SMOs is considered to be expensive. Therefore, the PO-P is expected to have more degree of concurrency than INC-

```
1.  height := H; unsafe_node := No;
2.  Parent := NULL; Child := ROOT;
3.  h := 1;
3.  while h < height do begin
4.     latch Child with IX;
5.     unlatch Parent;
6.     Parent := Child;
7.     Determine nextChild;
8.     Child := nextChild;
9.     h := h+1;
10.   end;
11. latch Child with X;
12. unlatch Parent;
13. if Child is unsafe then  begin
14.    height := h-1;
15.    unsafe_node := Yes;
16.    unlatch Child;
17.    goto Module 2.
18.    return;
19. end
20. else begin
21.    update Child; unlatch Child;
22. end.
```

Main Module

```
1. Parent := NULL; Child :=ROOT;
2. while h < height do begin
3.    latch Child with IX; unlatch Parent;
4.    if Child is unsafe then
5.       begin
6.          height := h-1; unsafe_node := Yes;
7.          unlatch Child; goto 1;
8.       end
9.    else begin
10.         Parent := Child; Determine nextChild;
11.         Child := nextChild; h := h+1;
12.        end;
13.   end;
14.  end;
15.  if unsafe_node == Yes then begin
16.   latch Child and its copies with X;
17.   unlatch Parent; Parent := Child;
18.   Determine nextChild; Child := nextChild;
19.   latch Child and its copies with X;
20.   perform the POs on Child and its copies;
21.   unlatch Parent's copies, Child and its copies;
22.   Determine the appropriate Child;
23.   unsafe_node := No; height := H;
24.   h := h+1; goto 1;
25.  end;
26.  latch Child with X; unlatch Parent;
27.  if Child is unsafe then  goto 5;
28.  else begin
29.   update Child; unlatch Child;
30.  end.
```

Module 2

Fig. 1 PO-P protocol.

OPT in the update intensive cases.

The second phase of the TD-OPT uses the SIX latch-coupling starting from the root to the target leaf node. Thus it needs to synchronize many PEs while it is actually unnecessary. This will create an intolerable degradation in the system performance. In contrast, the PO-P uses the IX latch-coupling until the update scope to avoid the above situation.

Theorem 1 The PO-P protocol is deadlock free.

Proof. The latching remote copy nodes are deadlock free, since the linear order technique never creates cyclic waits. Therefore, we should consider deadlocks that involve different nodes. Suppose any two processes, say P and Q, the latter preceding the former along the same path. If a deadlock occurs between these two processes, then that must be a situation in which Q waits for a node on which P already has a latch. However, this situation never happens because the B-tree does not have cyclic links, and the latch-coupling and the PO-P allow latches only in descendant order. Even the retraversals of Q cannot create such waits because, according to the PO-P, Q must release all the latches it had acquired in the previous traversal. Hence, Q preceding P never waits for the node on which P already has a latch. □

Although the PO-P is expected to be more efficient than the TD-OPT in performing the POs for parallel B-trees, it has a problem to be solved because of its optimistic behavior. Below, we only mention about the problem and its possible solution. In the future work, this aspect will be discussed in detail.

Suppose an update process, say P, finds its target leaf node unsafe. In the retraversal, P may also find an unsafe internal node, say U. According to the PO-P, P makes U safe. If a safe node, say S, exists along the path between U and the leaf, the PO on U can be considered unnecessary. If the number of such costly POs is high, it will hurt the performance. Moreover, during the retraversal of P after making U safe, U can be made unsafe in repetition by the effect of other update processes. This means that those unnecessary POs can create a chance that P can fall into an endless loop. This chance can be reduced by increasing the node size. In contrast, the INC-OPT performs the SMOs on the exact update scope, and thus the retraversals of its update processes are limited to the height of the B-tree in the worst case.

The problem can be fixed by learning from the previous traversal. From the first traversal path, the process P can find a safe node which is the closest to the leaf node. We call this *a safe point*. P traverses using IX latch-coupling until the safe point and acquires X latch on it. From the safe point to down, the process assumes that every node, including the leaf, on the path is unsafe. Then the process performs the cascading POs on all of them not doing any retraversal. This method eliminates the unnecessary POs, and also limits the retraversals to the height of the B-tree as the INC-OPT does.

4. Experiments and Results

In order to see the PO-P's performance practically, we implemented it together with the INC-OPT on Fat-Btrees[6),12)] on a shared-nothing parallel machine. Our shared-nothing parallel machine consists of 8 PCs each of which is running Red Hat Linux version 9 and connected through a Gigabit Ethernet. We used LAM/MPI 7.0.3 implementation for the point-to-point communication between PEs.

According to the Fat-Btree's structure definition each node can have separate pages (let us name it as ptrPage) in which the addresses of copy nodes and remote children nodes are contained. A copy node address consists of two parts:[PE number, local node address].

In our implementation, we put the ptrPage's address in the location of the remote child's address so that the process is able to recognize if the determined next child is remote or local node. Then, we inserted the position of the remote child's address in the right position in the ptrPage along with the corresponding [PE number, local node address] pair.

We implemented two Fat-Btrees, in which one had small size nodes while another had larger size nodes. The former was used for observing an update intensive environment where multiple SMOs occurred often while the latter was used for simulating environments where SMOs happened rarely.

For the small size nodes, we used 133 B, and, an index node can thus have $\lfloor(133-32-4)/8\rfloor = 12$ entries. 32 B is saved for the administration section of a node. For the larger size nodes, we used 4096 B. Thus, $\lfloor(4096-32-4)/8\rfloor = 507$ is the maximum number of entries in an internal node. A leaf node can have $\lfloor(4096-32)/208\rfloor = 19$ tuples at the maximum as a tuple takes 208 B.

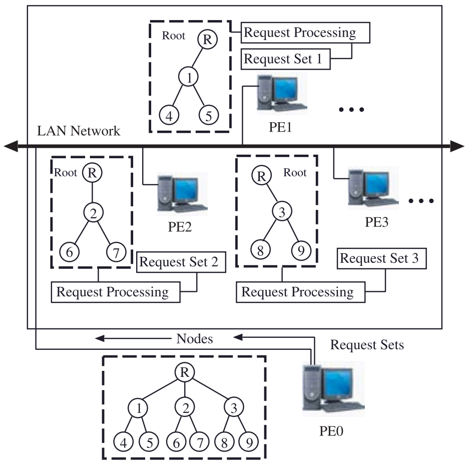

Fig. 2 System structure.

Table 1 Specifications of PEs.

Processing Elements	PE0-PE7
CPU speed	1.4 GHz
Memory size	512 MB
OS	Linux Red Hat 9.

4.1 System Structure

The general system structure is illustrated in **Fig. 2**. The PEs numbered 0 through 7 have the same specifications which are shown in **Table 1**. All the implementations are done in the main memory of the machines because we focused only on measuring the performance of concurrency control protocols. We used PE0 for the tree initialization, tree distribution and also request set generation. PE0 does the followings:

(1) Initialize the B-tree.
(2) Distribute the tree to PE1 through PE7 according to the right value ranges.
(3) Produce request sets and request type sets (retrieval or update) according to the given update ratio.
(4) Send appropriate request sets with the corresponding type sets to the PEs according to the value ranges.
(5) Start PE1 through PE7 to process their assigned sets of requests.
(6) When all the PEs complete their job, they inform PE0. Then PE0 finishes the PEs.

We assigned the request sets to the PEs before the actual request processing starts to highlight the effectiveness of parallelization.

In **Table 2**, the initial B-tree configurations used in the experiments are shown. The first

Table 2 The initial B-tree configurations.

Tuples	H	$N_n(1)$	$N_n(2)$	$N_n(3)$	$N_n(4)$
6 K	4	1	8	77	753
2 M	3	1	254	64516	

Table 3 System performance.

Communication throughput of the network	37.6 Mbps
Message setup time	93 μs
Access time to a page in memory	1.5 μs
Latch overhead in local PE	2.5 μs
Remote latch overhead	419 μs

column is the number of initially inserted tuples and the subsequents are the tree height and the number of nodes in the corresponding levels. We set the initial tree nodes with 50–100% occupancies in all the experiments.

A request consists of:
(1) Request type (retrieval | update)
(2) Key (tuple's attribute value).
The keys were randomly chosen integer numbers from the range $[1, 2^{25}]$. We assigned 4 M of the requests for the Fat-Btree with the nodes of small size in order to create an update intensive environment. We assigned 2 M of the requests for the Fat-Btree with the nodes of larger size. We assumed here only insert operations for update and read operations for retrieval. In **Table 3**, the system's initial performance data is given. The remote latch cost was rather high than the local latch cost because of the message cost plus the waiting time for the latch releases.

4.2 Performance Results

We ran the two protocols (the INC-OPT and the PO-P) on different number of PEs under increasing update ratio in the request sets. The graphs **Figs. 3, 4, 5** and **6** illustrate the results of the experiments which are done on the Fat-Btree with the nodes of small size. The graphs **Figs. 7, 8** and **9** show the results for the Fat-Btree with the nodes of larger size.

The vertical axis of the graphs denotes the throughput which shows the number of operations done per second. The horizontal axis of the graphs but Fig. 6 describes the changes of the ratio of update operations in the request set. The update ratio increases from 0% to 100% by 10.

In the first type of the experiments, the root node split occurred three times, which means the tree height increased from 4 to 7.

In Fig. 3, the results of an experiment in which 2 PEs participated are shown. From this figure, we can see that the PO-P has slightly better performance than the INC-OPT when update ratio increases from 50% to 100%, although almost the same in the first half range. When the number of PEs is lower, the number of copy nodes is also lower according to Fat-Btree's design. That means copy node splits occur less frequently in the configuration in which fewer PEs participated than that of more PEs participated in. In one PE case, we have also conducted a similar experiment that shows the PO-P has slightly better performance than the INC-OPT. But in that case all the operations are locally done without doing any communication with other PEs, and its throughputs are the lowest among the other configurations due to less processing elements.

In Figs. 4 and 5, the results of the same experiment but the number of PEs are 5 and 7, respectively, are shown. In these experiments, the performance difference is almost constant between 20% to 80% of update ratios. However, it slightly increases between 80% to 100% of update ratios. During the experiments, only 10% of the node splits led to the intermediate node splits and 0.3% of them involved copy node splits or copy parent reflections. Therefore, any big differences between the two protocols are not seen when the update ratio is low. On the other hand, the Fat-Btree's structure has its own character, that is, it has more node copies in the upper part of the distributed tree but less copies in the lower part. Then performing any SMOs in the upper part is much more costly than doing in the lower part because of the communication cost.

Fig. 6 shows the compared performance of the two protocols under different number of PEs while the request set consists of 100 percent update operations. From the figure, it is seen that the difference is gradually increasing when the number of PEs increases under a high update ratio.

Finally, we show the results of the experiments with the nodes of larger size in Fig. 7 through Fig. 9 where the number of PEs are 2, 5 and 7. It is clear that the larger the nodes are, the fewer the SMOs or the POs occur. In the experiments, only 3% of the total insert requests led to the SMOs. From the experiments we saw that the performances of the PO-P and the INC-OPT were identical. The larger size nodes kept the tree height low. This precluded any highly cascaded SMOs which involve many copy nodes. Therefore, any differences

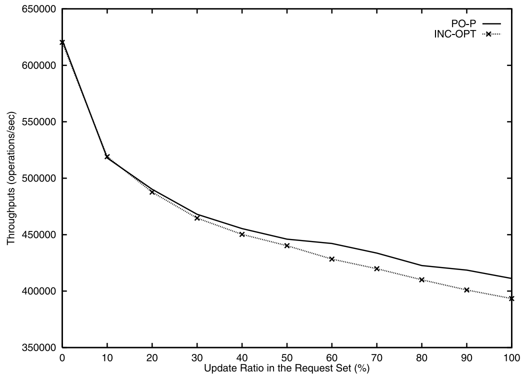

Fig. 3 Performance comparison of the two protocols under 2 PEs (node size: 133 B, number of tuples inserted: 4 M).

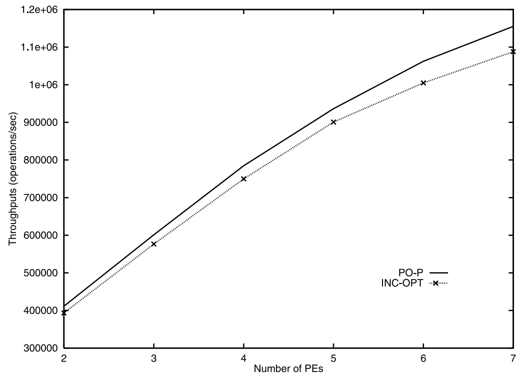

Fig. 6 Performance comparison at the highest update rate (node size: 133 B, number of tuples inserted: 4 M).

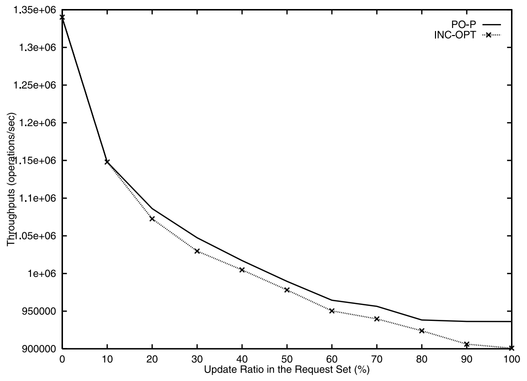

Fig. 4 Performance comparison of the two protocols under 5 PEs (node size: 133 B, number of tuples inserted: 4 M).

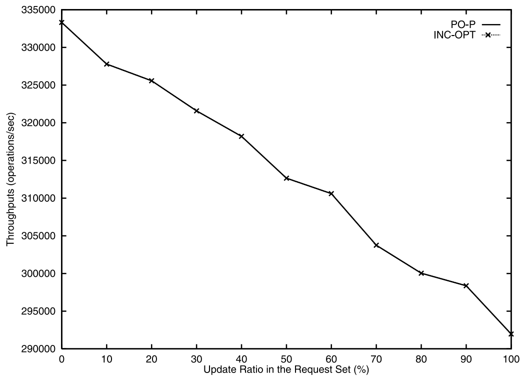

Fig. 7 Performance comparison of the two protocols under 2 PEs (node size: 4 K, number of tuples inserted: 2 M).

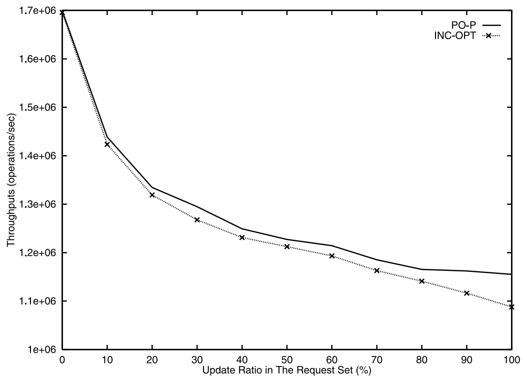

Fig. 5 Performance comparison of the two protocols under 7 PEs (node size: 133 B, number of tuples inserted: 4 M).

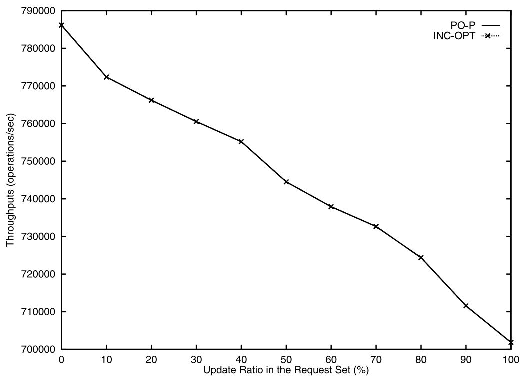

Fig. 8 Performance comparison of the two protocols under 5 PEs (node size: 4 K, number of tuples inserted: 2 M).

will not be observed between the protocols in cases where SMOs rarely occur.

5. Conclusions

We have proposed a protocol named PO-P for shared-nothing parallel B-trees such as Fat-Btree as an alternative. In our protocol we adapted preparatory operations for update processes when they encounter unsafe nodes. By using this method any update operations are done in small atomic operations which require only two levels' nodes latched at a time on the

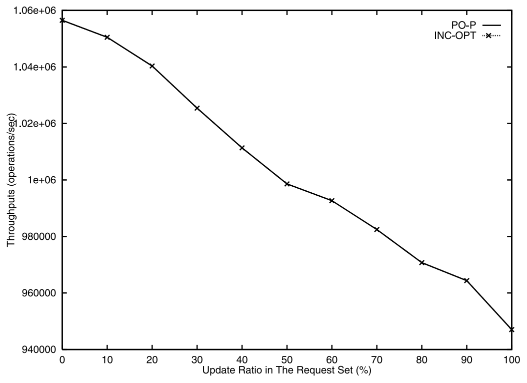

Fig. 9 Performance comparison of the two protocols under 7 PEs (node size: 4 K, number of tuples inserted: 2 M).

global parallel B-tree. Therefore, the PO-P increases the degree of parallel operations by decreasing the latch wait time for processes which run in parallel on many PEs.

In contrast, the current method INC-OPT creates longer wait times for processes in high SMOs occurences. Our experimental results show that the PO-P works at least at the same performance with the INC-OPT but shows better performance in the cases in which highly cascaded updates occur often when many PEs are involved. The main merit of this paper is that it showed that top-down method is not only applicable for ordinary B-trees but can also be used for parallel B-trees with more effective results.

Our further work will consider the versions of the implementations involving cache usage and also recovery issues. The future work will also include the implementation of the solution for the problem mentioned in Section 3.

Acknowledgments The authors wish to thank the editor and anonymous reviewers for their valuable comments and suggestions.

References

1) Bayer, R. and Schkolnick, M.: Concurrency of Operations on B-trees, *Acta Informatica*, Vol.9, No.1, pp.1–21 (1977).

2) Gray, J. and Reuter, A.: *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

3) Haritsa, J.R. and Seshadri, S.: Real-time Index Concurrency Control, *IEEE Trans. Knowledge and Data Engineering*, Vol.12, No.3, pp.429–447 (2000).

4) Kröll, B. and Widmayer, P.: Distributing a Search Tree Among a Growing Number of Processors, *Proc. 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 24–27, 1994, Snodgrass, R.T. and Winslett, M.(eds.), ACM Press, pp.265–276 (1994).

5) Lehman, P.L. and Yao, S.B.: Efficient Locking for Concurrent Operations on B-Trees, *ACM Trans. Database Syst.*, Vol.6, No.4, pp.650–670 (1981).

6) Miyazaki, J. and Yokota, H.: Concurrency Control and Performance Evaluation of Parallel B-tree Structures, *IEICE Trans. Inf. & Syst.*, Vol.E85-D, No.8, pp.1269–1283 (2002).

7) Mohan, C.: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes, *Proc. 16th International Conference on Very Large Data Bases*, August 13–16, 1990, Brisbane, Queensland, Australia, McLeod, D., Sacks-Davis, R. and Schek, H.-J.(eds.), Morgan Kaufmann, pp.392–405 (1990).

8) Mohan, C. and Levine, F.: ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging, *Proc. 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 2–5, 1992, Stonebraker, M.(ed.), ACM Press, pp.371–380 (1992).

9) Mond, Y. and Raz, Y.: Concurrency Control in B+-Trees Databases Using Preparatory Operations, *VLDB'85, Proc. 11th International Conference on Very Large Data Bases*, August 21–23, 1985, Stockholm, Sweden, Pirotte, A. and Vassiliou, Y.(eds.), Morgan Kaufmann, pp.331–334 (1985).

10) Seeger, B. and Larson, P.-Å.: Multi-Disk B-trees, *Proc. 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado*, May 29–31, 1991, Clifford, J. and King, R.(eds.), ACM Press, pp.436–445 (1991).

11) Srinivasan, V. and Carey, M.J.: Performance of B+ Tree Concurrency Algorithms, *VLDB J.*, Vol.2, No.4, pp.361–406 (1993).

12) Yokota, H., Kanemasa, Y. and Miyazaki, J.: Fat-Btree: An Update-Conscious Parallel Directory Structure, *Proc. 15th International Conference on Data Engineering*, 23–26 March 1999, Sydney, Australia, IEEE Computer Society, pp.448–457 (1999).

(Received June 20, 2004)
(Accepted October 14, 2004)

(Editor in Charge: *Jun Miyazaki*)

Damdinsuren Amarmend received his B.E. in computer science from Computer Science and Management School of Mongolian Technical University, Mongolia, in 1996, and his M.E. degree in computer science from Gunma University, Japan, in 2003. He is currently a Ph.D. student at the Department of Computer Science, Gunma University. His research interests include database systems, specially parallel and distributed database systems.

Masayoshi Aritsugi received his B.E. and D.E. degrees in computer science and communication engineering from Kyushu University in 1991 and 1996, respectively. Since 1996, he has been working at the Faculty of Engineering, Gunma University, where he is now an Associate Professor. His research interests include database systems and parallel and distributed data processing. He is a member of IPSJ, IEICE, IEEE-CS, ACM, and DBSJ.

Yoshinari Kanamori received his D.E. degree from Tohoku University in 1975. Since 1991, he has been a Professor at the Department of Computer Science, Gunma University. His research interests include database systems and image processing. He is a member of IPSJ, IEICE, ACM, and IEEE-CS.