Brief Announcement: Tree Decomposition for Faster Concurrent Data Structures

Johannes Schneider
Computer Engineering and Networks Laboratory
ETH Zurich
8092 Zurich, Switzerland
jschneid@tik.ee.ethz.ch

Roger Wattenhofer
Computer Engineering and Networks Laboratory
ETH Zurich
8092 Zurich, Switzerland
wattenhofer@tik.ee.ethz.ch

Abstract

We show how to partition data structures representable by directed acyclic graphs, i.e. rooted trees, to allow for efficient complex operations, which lie beyond inserts, deletes and finds. The approach potentially improves the performance of any operation modifying more than one element of the data structure. It covers common data structures implementable via linked lists or trees such as sets and maps. We demonstrate its simplicity and its effectiveness using a concurrent sorted linked list. We achieve a speedup of up to 250% even for small divisions.

Categories and Subject Descriptors: E.1 Data structures
General terms: algorithms, performance
Key Words: concurrent data structures, graphs

1. INTRODUCTION

With the rise of multi-core processors efficient and simple parallel data structures gain more and more importance. Though there is a rich literature on concurrent data structures, outside of the database community surprisingly little work has been carried out to deal with general operations beyond inserts, deletes and finds. For instance, any common operation such as an update query on a subset of all elements in a list is often not handled well with conventional techniques, meaning that either the whole data structure is locked or each modified element is locked. In the first case, all operations are sequential, and in the second case the overhead due to acquiring and releasing locks frequently induces an unwanted time penalty. In databases this issue has been addressed from the 1970s onwards through various mechanisms such as hierarchical locking. Our proposal is less complicated and does not involve hierarchies. We group elements together, such that only one lock per group must be acquired.

2. RELATED WORK

For database systems, predicate locks are an approach to logically lock elements [1]. A predicate, such as "all data records with field x larger some value", defines all tuples that are of interest for a transaction. In theory and practice, finding intersecting predicates is a difficult undertaking and has not been used in real systems [2, 4], whereas hierarchical locking is standard since the 1970s. In database systems only whole blocks with several data records can be locked due to physical constraints such as the addressable size of a block on a hard drive, whereas for concurrent data structures any object can be locked. Our approach intentionally introduces a minimum granularity for locking objects. We argue that the complexity of hierarchical locking is well justified for large centralized database systems where hundreds or thousands of operations are executed in parallel, but it is not necessary in a typical multi-core system, where the number of actual concurrent transactions is in the order of the number of cores.

Many implementations of concurrent data structures which support only insert, delete and find operations have been proposed. For example, in [5] a concurrent binary tree implementation is given such that locks are only acquired for modified objects (as for the lazy concurrent list in [3]). Our algorithm and the one in [3] need one lock for such an operation, but we have another book keeping overhead (i.e. remembering the leader of the currently traversed group) and furthermore, in general, we slightly restrict the potential parallelism. Thus, for the mere standard operations insert, delete and find, these algorithms are somewhat more efficient than our approach. However, the proposed strategies [3, 5] are not well suited for operations beyond inserts, deletes and finds. In particular, if one iterates through a list or tree modifying many objects, all these objects must be protected (e.g. locked). Thus there is "no free lunch" in the general case.

3. DATA STRUCTURE DECOMPOSITION

We distinguish two types of nodes: group leaders and group members. All group members that are reachable from a leader by member objects form a group (together with the leader). See Figure 1.

A traversal of the data structure starts from the root, which is always a leader. When an object to be modified is encountered, any thread working on some object in the same group must also have passed by the group leader. Therefore, it suffices to lock leaders, i.e. before a group member is modified its group leader is locked. This holds for lists as well as for all kinds of tree traversals such as depth or breadth first. The choice of group leaders can be made on different basis. For example, when a new node is inserted it is randomly chosen to be either a group leader or a group member. On the positive side, if several elements are modified in the same group, only one lock needs to be acquired. On the negative side, if several threads operate on different elements in the

PODC'10, July 25–28, 2010, Zurich, Switzerland.
ACM 978-1-60558-888-9/10/07.

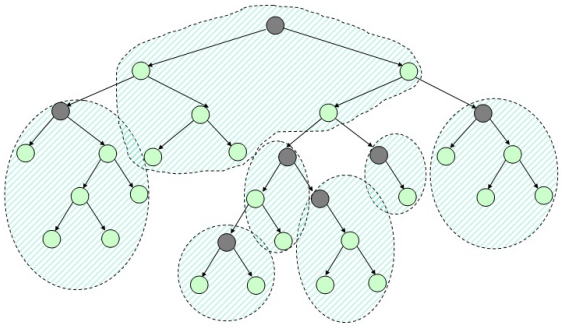

Figure 1: Decomposition of a directed rooted graph, where dark vertices indicate leaders and pale vertices show group members. A group is shown by a striped area.

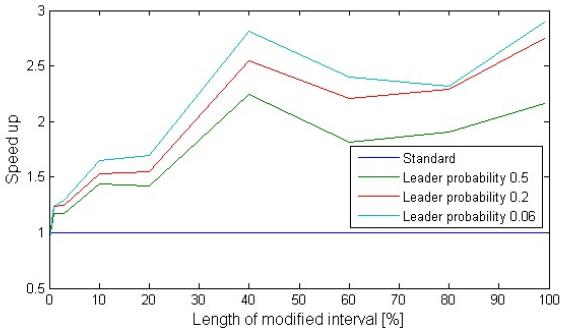

Figure 2: Speedup of a concurrent linked list when using groups of expected sizes 2, 5 and 17. The x-axis gives the length of the modified interval in per cent of the total list length.

same group, they face a conflict and cannot run concurrently. Thus, if the size of the group is large, e.g. the whole tree in the most extreme case, the restriction of possible parallelism might have a bigger impact than the savings due to using less locks. However, since our technique is simple and comes with little overhead, it results in a performance gain already for small groups. The overhead is low, since membership relations of the groups are implicit, i.e. group members do not need to know their own group (i.e. their leader) and a leader object does not need to know the nodes in its group. If this was not the case then any deletion of a leader caused an overhead proportional to the size of the group, since all members must be updated. In our case leader deletions also cause complications, i.e. if a leader L is deleted then the group led by L is implicitly merged with the prior group. Though, an operation can notice that L got deleted when it attempts to lock L (e.g. by having a deleted flag), it might be unaware of its new group leader, but it could retraverse the list/tree to find it. When a new leader L gets inserted, a group might be split. To find out, if the leader is still current, versions can be used, i.e. whenever a group is split by inserting a new leader, the version of the original group leader is incremented. Thus, by checking the version of a leader L, an operation can be sure that the group has not been changed. However, there are also other ways to deal with these issues, i.e. a simple way to deal with deletions is to use "dummy" nodes as leaders that cannot be deleted. In future work, we will evaluate several strategies. A series of deletions and inserts might degenerate the decomposition, i.e. create a few very large groups and many small groups. In future work, we intend to develop and test rebalancing strategies for groups.

For evaluation we used Java with 16 threads on a 16 core system, i.e. four quad-core Opteron 8350 processor. We implemented a sorted concurrent linked list running Algorithm *StdModifyRangeValues(int x,int y)*. The algorithm traverses the list and changes all objects, which have a key larger than x and smaller than y. If we have found the first object O with key larger than x (and smaller y), then we cannot just lock object O but must lock the last accessed leader.

We added all numbers in the range [0,10000]. Then Algorithm *StdModifyRangeQuery* was executed on an interval $[x, y]$ of varying width $y-x$, start element x and with varying group sizes (see Figure 2). If no elements are modified our

Algorithm StdModifyRangeValues(int x,int y)

```
1: Node curr = list.getFirst(); Node lastLeader = null; {1st
   object is always a leader, e.g. dummy with key −∞}
2: while curr.getValue()<y do
3:    if curr.isLeader() then lastLeader = curr; end if
4:    if (curr.getValue() > x) or (curr.getValue() < y)
      then
5:       if lastLeader!=null then lastLeader.lock() endif
6:       Change some field of curr
7:       if curr.getNext().isLeader() then
         lastLeader.unlock(); lastLeader = null; endif
8:    end if
9:    curr = curr.getNext();
10: end while
11: if lastLeader!=null and lastLeader.isLocked() then
    lastLeader.unlock() endif
```

algorithm is about 5% slower due to the overhead of remembering the leader. It holds that the larger the groups and the modified interval, the larger the speedup (up to some point). If an interval containing one percent of all elements is modified, the performance improvement is about 25% for groups of average size 5. If 10% of all elements are altered the gain is about 65% for group sizes still below 20. For much larger group sizes (e.g. 200) the throughput does not increase compared to group sizes of around 20 – possibly, since the running time for groups beyond 20 is mainly dominated by the list traversal and locking operations account only for a small amount of the overall running time.

4. REFERENCES

[1] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11), 1976.

[2] G. Graefe. Hierarchical locking in b-tree indexes. In *BTW*, 2007.

[3] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm. *Parallel Processing Letters*, 17(4), 2007.

[4] H. B. H. III and D. J. Rosenkrantz. The complexity of testing predicate locks. In *SIGMOD Conference*, 1979.

[5] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1), 1988.