Amortization Results for Chromatic Search Trees, with an Application to Priority Queues

Joan Boyar, Rolf Fagerberg, Kim S. Larsen

Odense University

Abstract. The intention in designing data structures with relaxed balance, such as chromatic search trees, is to facilitate fast updating on shared-memory asynchronous parallel architectures. To obtain this, the updating and rebalancing have been uncoupled, so extensive locking in connection with updates is avoided.

In this paper, we prove that only an amortized constant amount of rebalancing is necessary after an update in a chromatic search tree. We also prove that the amount of rebalancing done at any particular level decreases exponentially, going from the leaves towards the root. These results imply that, in principle, a linear number of processes can access the tree simultaneously.

We have included one interesting application of chromatic trees. Based on these trees, a priority queue with possibilities for a greater degree of parallelism than in previous proposals can be implemented.

1 Introduction

A chromatic search tree [15, 7] is a binary search tree for shared-memory asynchronous parallel architectures. It was introduced with the aim of allowing processes to lock nodes, in order to avoid inconsistencies from updates and rebalancing operations, without decreasing the degree of parallelism too much. The means for obtaining this was a new balance criteria, referred to as relaxed balance, along with new uncoupled operations for updating and rebalancing.

The rebalancing is taken care of by background processes in small independent steps; the processes do only a constant amount of work before they release locks and move on to another problem. This means that the traditional exclusive locking of whole paths or step-wise exclusive locking down paths, which would limit the amount of parallelism possible to the height of the tree, does not take place. Another advantage of the uncoupling of the rebalancing from the updating is that all or parts of the rebalancing can be postponed until after peak working hours. The disadvantage, of course, is that the tree can become very unbalanced if there are not enough background processes doing the rebalancing.

Since the rebalancing is done in small independent steps, which can be interspersed with other updating and rebalancing operations, an actual proof of complexity is not straight-forward, and the original proposal in [15] did not contain any such proof. In [7], the proposal of [15] was analyzed, and it was proven that some updating could give rise to a super-logarithmic number of rebalancing
operations. A modified set of rebalancing operations was proposed, and it was proven that the new set of rebalancing operations give rise to at most \(\log_2(n+i)\) rebalancing operations per insertion and at most \(\log_2(n+i) - 1\) rebalancing operations per deletion, if \(i\) insertions are performed on a tree which initially contains \(n\) leaves. Furthermore, the number of operations which actually change the structure of the tree is at most one per update. Compared to [15], a small constant number of extra locks per rebalancing operation are necessary in [7].

Having obtained logarithmic worst-case bounds, the next result to hope and search for when dealing with trees is an amortized constant number of rebalancing operations. It turns out that the proposal from [7] has these properties, though in order to get the best possible constant, one operation should be modified slightly. In this paper, we prove that, starting with an empty tree, \(i\) insertions and \(d\) deletions give rise to at most \(3i + d - 2\) rebalancing operations. We also show that the number of rebalancing operations which can occur at weighted height \(h\) is at most \(3i/2^{h-1}\). The latter result is especially important in a parallel environment, since many of the rebalancing operations require exclusive locks on the nodes they are accessing. The higher up in the tree a lock occurs, the larger the subtree which cannot be accessed by other operations. Our results imply that, in principle, \(O(n)\) processors can simultaneously access the tree, since searching does not require exclusive locking. The results are obtained assuming, as is standard in amortized analyses, that the structure is initially empty, though we also have results for the case where the structure is initially non-empty.

In the last part of the paper, we discuss one particularly interesting application of chromatic trees in greater detail. From the sequential case, it is known that in some cases search tree implementations of priority queues give better performance than heap implementations [11]. In our setting of a shared-memory architecture, it turns out that a variation of a chromatic search tree, used as a priority queue, allows for a greater degree of parallelism than in previous proposals for priority queues. The priority queue is suited for branch and bound and similar applications.

Our results supplement previous work on relaxed search trees. The chromatic search tree is a relaxed version of red-black trees [3, 8]. This structure was introduced in [15] and analyzed in [7]. A relaxed version of \((a, b)\)-trees [10] was analyzed in [14]. Both of the common B-trees [4], 2-3 trees [2, 9], and 2-3-4 trees [8] have relaxed versions, the properties of which are also discussed in [14]. The first relaxed version of a B-tree is from [16]. The analysis of the behavior of a relaxed version of AVL trees [1] introduced in [16] was given in [13].

2 Chromatic Search Trees

In this section, we describe chromatic search trees, noting a couple of minor changes from earlier definitions [15, 7]. Chromatic trees are leaf-oriented binary search trees, so the keys are stored in the leaves and the internal nodes only contain routers which guide the search through the tree. The router stored in a node \(v\) is greater than or equal to any key in the left subtree and less than