Choose Concurrency-Friendly Data Structures

By Herb Sutter, June 27, 2008

Post a Comment

Linked Lists and Balanced Search Trees are familiar data structures, but can they make the leap to parallelized environments?

Balanced Search Trees

The story isn't nearly as good for another popular data structure: the balanced search tree. (Important note: This section refers only to balanced trees; unbalanced trees that support localized updates don't suffer from the problems we'll describe next.)

Consider a red-black tree: The tree stays balanced by marking each node as either "red" or "black," and applying rules that call for optionally rebalancing the tree on each insert or erase to avoid having different branches of the tree become too uneven. In particular, rebalancing is done by rotating subtrees, which involves touching an inserted or erased node's parent and/or uncle node, that node's own parent and/or uncle, and so on to the grandparents and granduncles up the tree, possibly as far as the root.

For example, consider Figure 3. To start with, the tree contains three nodes with the values *1*, *2*, and *3.* To insert the value *4*, we simply make it a child of node *3*, as we would in a nonbalanced binary search tree. Clearly, that involves writing to node *3*, to set its right-child pointer. However, to satisfy the red-black tree mechanics, we must also change node *3*'s and node *1*'s color to black. That adds overhead and loses some concurrency; for example, inserting *4* would conflict with adding 1.5 concurrently, because both inserts would need to touch both nodes *1* and *3*.

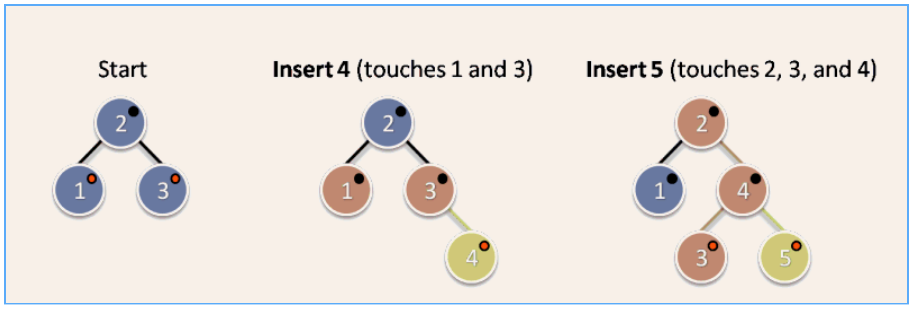
Start
Insert 4 (touches 1 and 3)
Insert 5 (touches 2, 3, and 4)
2
1
3
2
1
3
4
2
1
4
3
5

Figure 3: Nonlocalized insertion into a red-black tree.

Next, to insert the value *5*, we need to touch all but one of the nodes in the tree: We first make node *4* point to the new node *5* as its right child, then recolor both node *4* and node *3*, and then because the tree is out of balance we also rotate *3-4-5* to make node *4* the root of that subtree, which means also touching node *2* to install node 4 as its new right child.

So red-black trees cause some problems for concurrent code:

- It's hard to run updates truly concurrently because updates arbitrarily far apart in the tree can touch the same nodes—especially the root, but also other higher-level nodes to lesser degrees—and therefore contend with each other. We have lost the ability to make truly localized changes.
- The tree performs extra internal housekeeping writes. This increases the amount of shared data that needs to be written and synchronized across caches to publish what would be a small update in another data structure.

"But wait," I can hear some people saying, "why can't we just put a mutex inside each node and take the locks in a single direction (up the tree) like we could do with the linked list and hand-over-hand locking? Wouldn't that let us regain the ability to have concurrent use of the data structure at least?" Short answer: That's easy to do, but hard to do right. Unlike the linked list case, however: (a) you may need to take many more locks, even all the way up to the root; and (b) the higher-level nodes will still end up being high-contention resources that bottleneck scalability. Also, the code to do this is much more complicated. As Fraser noted in 2004: "One superficially attractive solution is to read-lock down the tree and then write-lock on the way back up, just as far as rebalancing operations are required. This scheme would acquire exclusive access to the minimal number of nodes (those that are actually modified), but can result in deadlock with search operations (which are locking down the tree)." [2] He also proposed a fine-grained locking technique that does allow some concurrency, but notes that it "is significantly more complicated." There are easy answers, but few easy and correct answers.

To get around these limitations, researchers have worked on alternative structures such as skip lists [4], and on variants of red-black trees that can be more amenable to concurrency, such as by doing relaxed balancing instead of rebalancing immediately when needed after each update. Some of these are significantly more complex, which incurs its own costs in both performance and correctness/maintainability (for example, relaxed balancing was first suggested in 1978 but not implemented successfully until five years later). For more information and some relative performance measurements showing how even concurrent versions can still limit scalability, see [3].

Conclusions

Concurrency-friendliness alone doesn't singlehandedly trump other performance requirements. The usual performance considerations of Big-Oh complexity, and memory overhead, locality, and traversal order all still apply. Even when writing parallel code, you shouldn't choose a data structure only because it's concurrency-friendly; you should choose the right one that meets all your performance needs. Lists may be more concurrency-friendly than balanced trees, but trees are faster to search, and "individual searches are fast" can outbalance "multiple searches can run in parallel." (If you need both, try an alternative like skip lists.)

Remember:

- In parallel code, your performance needs likely include the ability to allow multiple threads to use the data at the same time.
- On parallel hardware, you may also care about minimizing the cost of memory synchronization.

In those situations, prefer concurrency-friendly data structures. The more a container supports truly localized updates, the more concurrency you can have as multiple threads can actively use different parts of the data structure at the same time, and (secondarily but still sometimes importantly) the more you can avoid invisible memory synchronization overhead in your high-performance code.

Notes

[1] H. Sutter. "Use Lock Hierarchies to Avoid Deadlock" (*Dr. Dobb's Journal,* January 2008).

[2] K. Fraser. "Practical lock-freedom" (*University of Cambridge Computer Laboratory Technical Report #579,* February 2004).

[3] S. Hanke. "The Performance of Concurrent Red-Black Tree Algorithms" (*Lecture Notes in Computer Science*, 1668:286-300, Springer, 1999).

[4] M. Fomitchev and E. Ruppert. "Lock-Free Linked Lists and Skip Lists" (*PODC '04*, July 2004).

Herb is a software development consultant, a software architect at Microsoft, and chair of the ISO C++ Standards committee. He can be contacted at www.gotw.ca.