PARALLEL

Choose Concurrency-Friendly Data Structures

By Herb Sutter, June 27, 2008

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

What is a high-performance data structure? To answer that question, we're used to applying normal considerations like Big-Oh complexity, and memory overhead, locality, and traversal order. All of those apply to both sequential and concurrent software.

But in concurrent code, we need to consider two additional things to help us pick a data structure that is also sufficiently concurrency-friendly:

- In parallel code, your performance needs likely include the ability to allow multiple threads to use the data at the same time. If this is (or may become) a high-contention data structure, does it allow for concurrent readers and/or writers in different parts of the data structure at the same time? If the answer is, "No," then you may be designing an inherent bottleneck into your system and be just asking for lock convoys as threads wait, only one being able to use the data structure at a time.
- On parallel hardware, you may also care about minimizing the cost of memory synchronization. When one thread updates one part of the data structure, how much memory needs to be moved to make the change visible to another thread? If the answer is, "More than just the part that has ostensibly changed," then again you're asking for a potential performance penalty, this time due to cache sloshing as more data has to move from the core that performed the update to the core that is reading the result.

It turns out that both of these answers are directly influenced by whether the data structure allows truly localized updates. If making what appears to be a small change in one part of the data structure actually ends up reading or writing other parts of the structure, then we lose locality; those other parts need to be locked, too, and all of the memory that has changed needs to be synchronized.

To illustrate, let's consider two common data structures: linked lists and balanced trees.

1 2 3 Next

Related Reading

- News
- Commentary

- **Jelastic Docker Integration For Orchestrated Delivery**
- **A Datacenter Operating System For Data Developers**

Parallel Recent Articles

Dr. Dobb's Archive
Finding the Median of Two Sorted Arrays Efficiently
Matching Wildcards: An Empirical Way to Tame an Algorithm
Unified Memory in CUDA 6: A Brief Overview
Parallel In-Place Merge Sort

Most Popular

Stories Blogs

Lambda Expressions in Java 8
An Algorithm for Compressing Space and Time
A Simple and Efficient FFT Implementation in C++: Part I
Lambdas and Streams in Java 8 Libraries
Finding the Median of Two Sorted Arrays Efficiently

- **Docker Clocks In On Azure**
- **Sencha Licks Android 5.0 Lollipop, And Likes**
More News»

- Slideshow
- Video

- **Jolt Awards 2014: The Best Testing Tools**
- **Jolt Awards: Coding Tools**
- **Developer Reading List**
- **Developer Reading List: The Must-Have Books for JavaScript**
More Slideshows»

- Most Popular

- **Why Build Your Java Projects with Gradle Rather than Ant or Maven?**
- **So You Want To Write Your Own Language?**
- **Read/Write Properties Files in Java**
- **Unit Testing with Python**
More Popular»

More Insights
White Papers

- Securosis Analyst Report: Security and Privacy on the Encrypted Network
- Simplify IT With Cloud-Based Wireless Management

More >>

Reports

- Hard Truths about Cloud Differences
- Research: Federal Government Cloud Computing Survey

More >>

Webcasts

- Intrusion Prevention Systems: What to Look for in a Solution
- IT and LOB Win When Your Business Adopts Flexible Social Cloud Collaboration Tools

More >>

Most Recent Premium Content

Digital Issues

2014
Dr. Dobb's Journal
November - **Mobile Development**
August - **Web Development**
May - **Testing**
February - **Languages**

Dr. Dobb's Tech Digest
DevOps
Open Source
Windows and .NET programming
The Design of Messaging Middleware and 10 Tips from Tech Writers
Parallel Array Operations in Java 8 and Android on x86: Java Native Interface and the Android Native Development Kit

2013
January - **Mobile Development**
February - **Parallel Programming**
March - **Windows Programming**
April - **Programming Languages**
May - **Web Development**
June - **Database Development**
July - **Testing**
August - **Debugging and Defect Management**
September - **Version Control**
October - **DevOps**
November- **Really Big Data**
December - **Design**

2012
January - **C & C++**
February - **Parallel Programming**
March - **Microsoft Technologies**
April - **Mobile Development**
May - **Database Programming**
June - **Web Development**
July - **Security**
August - **ALM & Development Tools**
September - **Cloud & Web Development**
October - **JVM Languages**
November - **Testing**
December - **DevOps**

2011

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?

Linked Lists

Linked lists are wonderfully concurrency-friendly data structures because they support highly localized updates. In particular, as illustrated in Figure 1, to insert a new node into a doubly linked list, you only need to touch two existing nodes; namely, the ones immediately adjacent to the position the new node will occupy to splice the new node into the list. To erase a node, you only need to touch three nodes: the one that is being erased, and its two immediately adjacent nodes.

This locality enables the option of using fine-grained locking: We can allow a potentially large number of threads to be actively working inside the same list, knowing that they won't conflict as long as they are manipulating different parts of the list. Each operation only needs to lock enough of the list to cover the nodes it actually uses.

For example, consider Figure 2, which illustrates the technique of hand-over-hand locking. The basic idea is this: Each segment of the list, or even each individual node, is protected by its own mutex. Each thread that may add or remove nodes from the list takes a lock on the first node, then while still holding that, takes a lock on the next node; then it lets go of the first node and while still holding a lock on the second node, it takes a lock on the third node; and so on. (To delete a node requires locking three nodes.) While traversing the list, each such thread always holds at least two locks—and the locks are always taken in the same order.

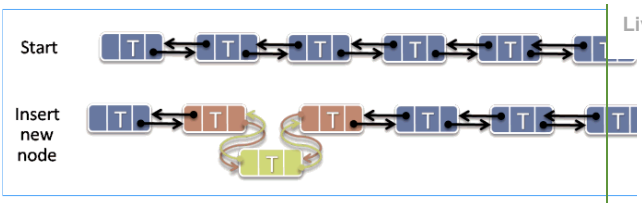

Figure 1: Localized insertion into a linked list.

Parallel Recent Articles

Dr. Dobb's Archive
Finding the Median of Two Sorted Arrays Efficiently
Matching Wildcards: An Empirical Way to Tame an Algorithm
Unified Memory in CUDA 6: A Brief Overview
Parallel In-Place Merge Sort

Most Popular

Stories Blogs

Lambda Expressions in Java 8
An Algorithm for Compressing Space and Time
A Simple and Efficient FFT Implementation in C++: Part I
Lambdas and Streams in Java 8 Libraries
Finding the Median of Two Sorted Arrays Efficiently

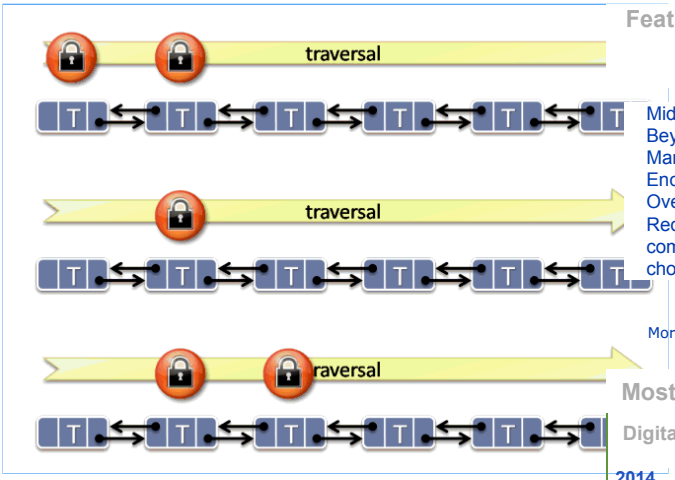

Figure 2: Hand-over-hand locking in a linked list.

This technique delivers a number of benefits, including the following:

- Multiple readers and writers can be actively doing work in the same list.
- Readers and writers that are traversing the list in the same order will not pass each other. This can be useful to get deterministic results in concurrent code. In particular, the list's semantics will be the same as if each thread acquired complete exclusion on the list and performed its complete pass in isolation, which is easy to reason about.
- The locks taken on parts of the list won't deadlock with each other, because multiple locks are acquired in the same order.
- We can readily tune the code for better concurrency vs. lower locking overhead by choosing a suitable locking granularity: one lock for the whole list (no concurrency), a lock for each node in the list (maximum concurrency), or a lock for each chunk of some fixed or variable length (something in between).

Aside: If we always traverse the list in the same order, why does the figure show a doubly linked list? Because not all operations need to take multiple locks; those that use individual segments or nodes in-place one at a time without taking more than one node's or chunk's lock at a time can traverse the list in any order without deadlock. (For more on avoiding deadlock, see [1].)

Besides being well suited for concurrent traversal and update, linked lists also are cache-friendly on parallel hardware. When one thread removes a node, for example, the only memory that needs to be transferred to every other core that subsequently reads the list is the memory containing the two adjacent nodes. If the rest of the list hasn't been changed, multiple cores can happily store read-only copies of the list in their caches without expensive memory fetches and synchronization. (Remember, writes are always more expensive than reads because writes need to be broadcast. In turn, "lots of writes" are always more expensive than "limited writes.")

Clearly, one benefit lists enjoy is that they are node-based containers: Each element is stored in its own node, unlike an array or vector where elements are contiguous and inserting or erasing typically involves copying an arbitrary number of elements to one side or the other of the inserted or erased value. We might therefore anticipate that perhaps all node-based containers will be good for concurrency. Unfortunately, we would be wrong.

Previous 1 **2** 3 Next

Related Reading

- News

- Commentary

- **Google's Data Processing Model Hardens Up**
- **Parasoft DevTest Shifts To Continuous**
- **Mac OS Installer Platform From installCore**
- **New Relic Continues Developer Data Apps Push**
  More News»

- Slideshow
- Video

- **Jolt Awards: Coding Tools**
- **Developer Reading List**
- **Jolt Awards: The Best Books**
- **Jolt Awards: The Best Testing Tools**
  More Slideshows»

- Most Popular

- **RESTful Web Services: A Tutorial**
- **Lambdas and Streams in Java 8 Libraries**
- **Lambdas in C++11**
- **Building GUI Applications in PowerShell**
  More Popular»

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*.

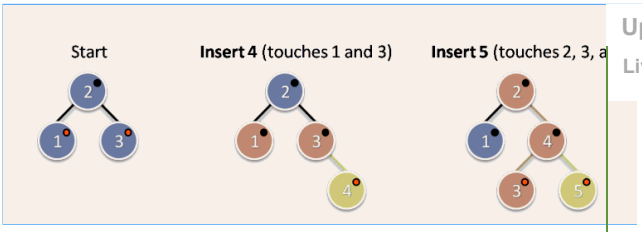

Figure 3: Nonlocalized insertion into a red-black tree.

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.

Previous 1 2 **3**

Login or Register to Comment