Title: Parallelizing Range Queries with Lock-Free SegTrees

Authors: Sebastian Dounchis, Neo Lopez
Web Page URL: https://sebcmu.github.io/parallel-seg-trees/

Summary

We are going to implement parallelized range queries and updates with coarse-grained locking, fine-grained locking and lock-free SegTrees. Then, we will analyze their performance on a diverse set of workloads, adding and comparing optimizations such as prefetching, vectorized instructions, and multi-threaded collaborative updates versus distributing updates across processors.

Background

Given an array A, a range query on that array asks about a property of some continuous range of that data from A[i] to A[j-1]. For example, one might be interested in the sum of the numbers from i to j. One approach to supporting such a range query is to loop over the range and compute the sum itself. A secondary approach is to precompute prefix sums and return the sum over a range as the difference A[j-1] - A[i-1], provided i > 0. Regardless of the approach, we would like to be able to support updating the array and re-querying. In the first approach, updates can be done in constant time, whereas in the second approach, updates must persist to all prefixes.

This gives the first approach, a manual range query, an O(1) update and an O(n) range query and the second approach, prefix queries, an O(n) update and an O(1) range query. In workloads where one operation is extremely rare, this may suffice, however, in workloads where updates and range queries are roughly balanced, then both operations are O(n) on average.

SegTrees implement O(log n) updates and range queries by using an array to represent a complete tree. On the bottom level is the array, on the next level is half the nodes of the lower level with sub computations A[0] + A[1], A[2] + A[3], and so on, and this pattern continues to the root of the tree which contains the result of the query on the entire array. Updates of array positions include updating all nodes on the path from leaf to tree that contain the array position. Range queries involve determining which blocks compose the range query.

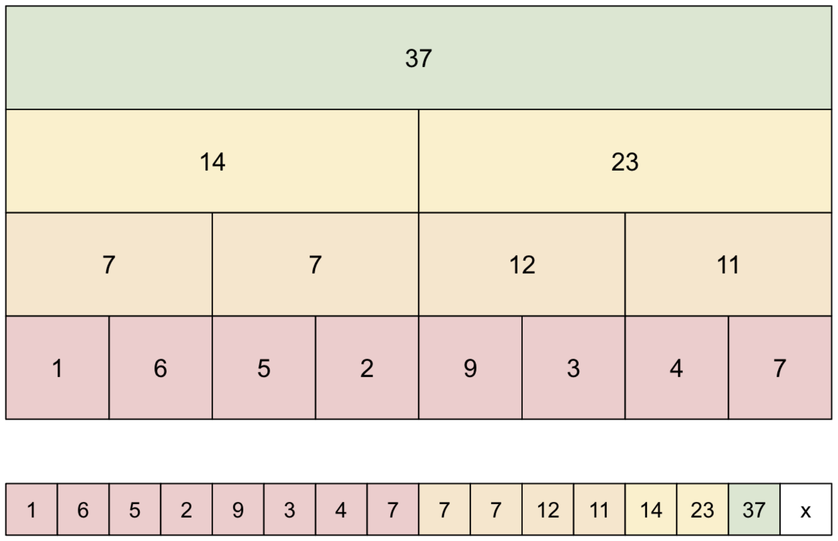

Figure 1: How a SegTree is represented as a tree and as an array.

With updates corresponding to traversing a tree from leaf to root, there is opportunity for parallelizing updates by having multiple processors perform separate updates along separate paths. Alternatively, parallelism could be attained by having multiple processors update separate array positions for a single update. Moreover, since range queries are read-only, multiple processors should be able to range query at the same time, provided no updates take place between subsequent range queries. These ideas present many opportunities for parallelism, as there are many ways to perform collaborative or per-processor updates.

Inspiration taken from the 15-451 textbook.

The Challenge

Since the memory accesses and modifications depend on the specifics of the operation (an update or a range query), the depth and width of necessary accesses and changes is unpredictable. Therefore, we will need to make decisions in work allocation, notably across sub-trees versus across operations, and synchronization of shared resources, such as the top, most commonly modified nodes, to scale performance with the number of processors.

With two update paths having the ability to cross at their lowest common ancestors in the SegTree, parallelizing updates requires a careful implementation to ensure updates to nodes in the SegTree are not lost due to memory operations not being atomic. This is where coarse-grained locking, fine-grained locking, and a lock-free implementation of the SegTree data-structure are important. Additionally, if range queries are overlapped with updates, we may not see the intended design of a range query after an update if the update has not finished before a query commences. This provides a challenge in dealing with the different operations quickly.

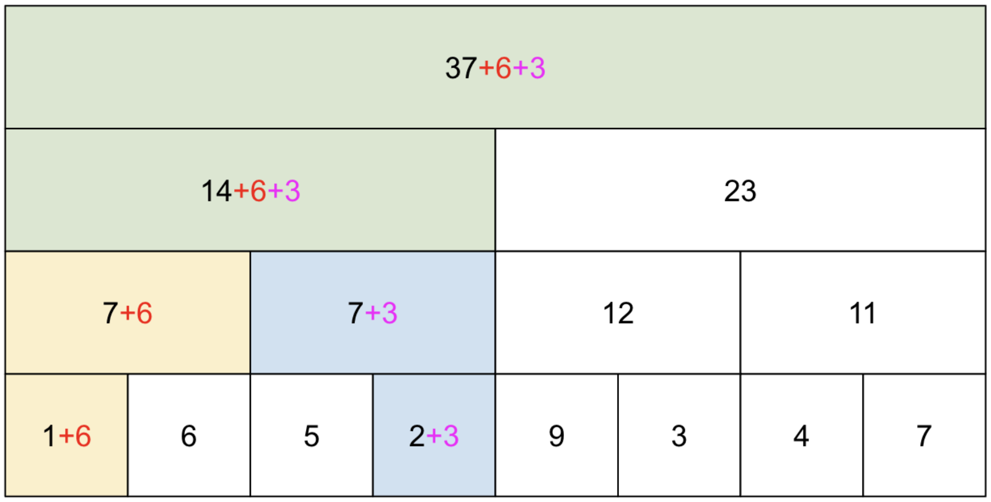

Figure 2: Update paths crossing. We need to ensure atomic updates to ensure updates aren't lost.

At the same time, updates may not overlap at all, at least until the top few layers. In this case, there are opportunities for prefetching up the SegTree path so that later computations can be performed more efficiently. If two update paths cross and invalidate each other's prefetched data, perhaps consolidating update paths could be beneficial.

Due to the array structure of SegTrees, cache coherence will be a prominent issue to deal with, decreasing performance as false-sharing causes our communication-to-computation ratio to increase. Opportunities for false sharing include the top few layers, where the nodes are clumped at the back of the SegTree array and in between each layer, where the rightmost node is adjacent in the array to the leftmost node of the next level. This specific challenge forces us to consider approaches like padding node layers to optimize performance. Further, we will need to hypothesize and test different padding strategies such as only post-layer padding versus different frequencies of intra-layer padding.

Each of the challenges presented might have a different "best" implementation dependent on the specific workload, i.e., update-heavy workloads, range query-heavy workloads, balanced workloads, and non-constant associative combination functions (instead of simply addition). This is something we will have to analyze and explore within our project.

For example, in workloads that are extremely update-heavy and extremely range query-heavy, perhaps it is better to implement the first two approaches we identified using CUDA technologies and vectorized instructions. This is something we plan to analyze within our project.

Resources

To create the initial serialized SegTree operation algorithms, we will use a small amount of starter code from the 451 textbook. In creating our file structure and parallel code, we will use previous assignments to guide us (i.e. creating the makefile, parsing files filled with operations). Additionally, we aim to analyze performance across different workloads, and we might use motivation from common characteristics of commercial workloads to drive our interest in how our optimizations could theoretically drive real-world improvements. We may also need to create a script to create trace files.

For testing, we will use the GHC machines to test at lower thread counts and experiment with the PSC machines for a wider range of thread counts.

Goals and Deliverables

Plan to Achieve:

- Coarse-grained locking, fine-grained locking, and lock-free implementations of SegTrees
- CUDA-based implementations of the manual range query and prefix range query approaches
- Parallel SegTree implementations distributing work within operations, across operations, and both within and across operations
- In-depth analysis of each parallel range query implementations on workloads of varying proportions of updates versus queries and workloads with non-constant update functions
    - Comparing idle time, synchronization stalls, computation time, and cache miss stalls
    - Graphing time of each implementation as a function of proportion of updates to queries within the workload (keeping the number of threads constant)
- In-depth analysis of the speedup of each implementation with different thread counts

Hope to Achieve:

- False-sharing optimizations: exploring high-frequency intra-level padding, low-frequency intra-level padding, and inter-level padding
- Prefetching optimizations: exploring different levels of aggression of prefetching
- Combining update paths: exploring the effect of consolidating update paths for processors who cross update paths

Platform Choice

We will use GHC machines to test and debug our implementations due to our immediate access to compute on these clusters. We will then use PSC machines on our experiments to provide consistent measurements of performance across thread counts varying from 1 to 256.

Our code will be written in C++ due to our familiarity with implementing parallel algorithms with this language. To create test files with specific proportions of updates to queries, we will use Python.

Schedule

- By 3/28: Create file structure, including makefile
- By 4/4: Implement coarse-grained and fine-grained locking queries and updates using SegTrees
- By 4/11: Implement lock-free SegTrees and CUDA-based manual range queries and prefix sum range queries
- By 4/15: Create a script for generating test files and perform initial testing on each implementation
- By 4/18: Begin drafting final report, experiment with false-sharing optimizations and prefetching optimizations
- By 4/25: Complete testing on PSC machines for important graphs, begin drafting final poster
- By 4/28: Finalize final report and submit, finalize poster, have practice run-throughs for presentation
- By 4/29: Present our final poster