Parallel Segment Trees: High Performance Range Queries and Updates

Stefano Lopez
Sebastian Dounchis
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

The segment tree data structure allows for high performance range queries and updates by storing information for contiguous subsequences in higher nodes of the tree. In this paper, we explore parallel CPU and GPU implementations of segment trees, including fine-grained locking and lock-free approaches. Further, we compare optimizations such as delaying updates to higher levels of the tree, prefetching node data on updates, and padding node data to fill cache lines. We tested various different workloads and optimization combinations, analyzing different tradeoffs. Our findings suggest that the lock-free approach with the delayed updates and prefetching achieves a 9x speedup over serial with a high update proportion and a 5x speedup over serial with a high query proportion and constant combining functions.

Keywords

parallel, locking, contention, segment trees, queries, updates, workloads, threads, fine-grained, lock-free, implementation

1 Background

1.1 The Range Query Problem

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 $A[j] - A[i-1]$, 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.

1.2 Introduction to Segment Trees

Segment trees (SegTrees) implement $O(\log n)$ updates and range queries by using a complete tree to represent an array. On the bottom level of the tree 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. This pattern continues to the root of the tree which contains the result of a query on the entire array: query$(0, n)$, where n is the size of the array. Updates of array positions include updating all nodes on the path from leaf to the root. Therefore, updates write to $O(\log n)$ nodes. Range queries involve determining which leaf and interior nodes, which represent contiguous subsequences, compose the range query and reading data from these nodes to complete a query. Since each query reads at most two nodes per level, queries are also $O(\log n)$ [2].

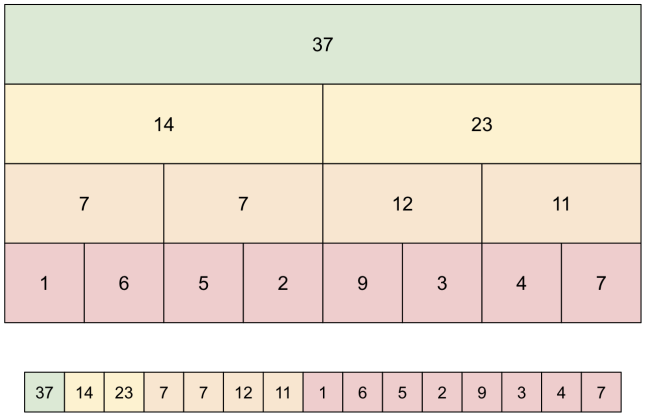

Figure 1: SegTree representation of an array.

In figure 1, the top visual shows how an array is represented as a SegTree using a sum combining function. The leftmost node on the orange level is a sum of its two children. Generally for SegTrees with combining function `f`, a node u on level k equals `f(leftChild(`u`),rightChild(`u`))` with these child nodes on level $k+1$. The bottom visual shows how these SegTrees are implemented as arrays in implementation. This allows for simple traversal of the SegTree. If the index of a node is i, it's parent has index $\lfloor \frac{i-1}{2} \rfloor$, left child has index $2i+1$, and right child has index $2i+2$ [2].

SegTree combining functions determine the outputs of any range query by applying the combining function f to the queried contiguous subsequence: `query(`$i, i+3$`)` $= f([i], f(A[i+1], A[i+2]))$ and `query(`i, j`)` $= f(A[i], f(A[i+1], ..., A[j-1]))$. With $f = +$, `query(`i, j`)` $= A[i] + A[i+1] + ... + A[j-1]$ = sum of subsequence from i to j-1, inclusive.

An associative function `f` has the property `f(`a`,f(`b`,`c`))` = `f(f(`a`,`b`),`c`)`. SegTrees store the results of subsequences in higher level nodes of the array and support fast queries by reading at most $O(\log n)$ nodes to minimize data reading. Therefore, on queries, the combining function is called on pre-combined sequences, so it must be associative for correctness [2].

1.3 Parallelism Opportunities and Challenges

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.

We would like a query at operation index k to reflect all updates at operation indices $0, ..., k-1$, which we'll refer to as serial correctness. In any operation sequence, a sequence of updates and range queries, only contiguous subsequences (chunks) of updates or queries can be processed in parallel for serial correctness. This is because each query must be completed before any subsequent updates. Otherwise, a later update might overwrite a query read, causing an incorrect query output. Moreover, each update must be completed before any query. Otherwise, the query might read stale data. We refer to this as the SegTree consistency problem. When considering parallelizing over operations, the SegTree consistency problem implicitly requires synchronization of threads after each update or query chunk is processed.

Chunking constrains the set of possible workloads we consider, but is necessary to parallelize the problem. In a workload without this chunking pattern, the SegTree consistency problem suggests that a serial implementation will have the best performance since there is no guarantee of chunks of large independent computations that parallel implementations can exploit. Thus, considering chunking is necessary to produce meaningful parallel implementations.

Additionally, a call to `update(`i`,`x`)` usually assigns $A[i] = x$ and propagates this update up the path from leaf to root [2]. This style of updating makes updates inherently sequential: query(i,j) only relies on the most recent updates to indices between i and j. To allow for parallelization across updates, we choose the style of updating that increments or combines the previous array value with x. In our SegTrees, `update(`i`,`x`)` assigns $A[i] = f(A[i], x)$ where `f` is the chosen combining function. When `f` = +, `update(`i`,`x`)` increments $A[i]$ by x: $A[i] = A[i] + x$. This change to the problem, combined with the associative property of the combining function, allows for update chunk parallelization: all permutations of updates within a chunk result in the same SegTree.

2 Approach

In each implementation, we scan through the sequence of operations, updating the SegTree data structure on updates and populating a query results array on queries. All query and update operations use logic inspired from the 451 textbook [2].

2.1 Trace Generation

Our main code parses input trace files, which contain the inputs necessary for any of our implementations. Each trace file first defines the array size, number of updates, and number of queries. Next, trace files contain the operations we wish to perform: updates on index i with delta value x are represented by `u i x` and queries from index i to j are represented by `q i j`. Finally, each trace file concludes with the results we expect to achieve when querying, which we can compare against our actual results to validate correctness.

The use of trace files made testing much easier. We wrote code to automatically generate traces given user-definable inputs such as the chunk size of operations, the combining function to be used, array size, number of operations, query-to-update ratio, and more. For more information on trace generation, please refer to our code.

2.2 CPU Implementations

For our CPU implementations, we used `C++` code and targeted high performance on both GHC and PSC machines.

2.2.1 Serial implementation. Our first implementation was serial, meant to serve as a baseline with which to compare our other implementations. In this approach, the CPU scans through the operations, completing one at a time.

2.2.2 Coarse-grained locking implementation. We then moved to a coarse-grained locking implementation, which locks the entire SegTree on each operation. Threads, created via the `C++` Standard Library's `std::thread` class [4], dynamically fetch operations to execute by atomically incrementing a shared operation index. We tracked the correct position to update in the query results array with a second atomic. We did not further optimize the coarse implementation, since its locking makes it slower than any serial implementation. The implementation was mainly to learn `std::thread` usage.

2.2.3 Fine-grained locking implementation. After implementing coarse-grained locking, we moved onto fine-grained locking. We initially thought that this would be as simple as locking over array positions, however, due to the SegTree consistency problem, updates and queries cannot overlap. This led us to change our trace generation code to include a chunk size parameter, which defines the size of each update or query chunk that we could parallelize over. With chunk processing requiring synchronization between batches, we could track which section of the query results array we needed to populate on a query batch. Statically mapping operations to threads enabled threads to compute the exact index of the query results array to update, allowing us to omit the shared atomics that we had previously used to track the operation and query results indices.

We initially performed separate chunks and synchronized between chunks by spawning update workers or query workers, and joining these workers after each batch. This implementation had too much thread overhead, ideally, we'd want to spawn threads once and delete them after all operations. We were able to implement this by using barriers as our form of between-batch synchronization.

In terms of actual operation processing, updates require each spawned thread to lock over the node currently being updated to ensure races do not occur. Queries can be performed without locking, since they are read-only. There is synchronization in the form of a barrier between update and query batches to eliminate the SegTree consistency problem.

Initially, we were locking over the node being updated, N_c, and its two children, C_l and C_r, thinking that if children were updated

during the update of N_c, there could be races due to the memory consistency problem. This is due to the fact that if N_c were updated with a stale child, the thread who caused staleness in the child did so via an update, and thus it must pass through N_c since all updates persist to the root. Locking over only the node being updated imposes a sequential order on threads who pass through that node. Therefore, though at any given time the SegTree could reflect staleness, after an entire batch of updates, the SegTree will be correct due to associativity of the combining function. Removing the locks over the children caused a speedup in the code without affecting correctness.

2.2.4 Lock-Free implementation. To move from our fine-grained implementation to a lock-free implementation, we simply changed lock-based updates to use compare-and-swap (CAS). With high overhead due to locking, this should provide a speedup since implementing CAS makes both queries and updates lock-free.

Initial testing indicated our lock-free code had races. We identified that this was due to children being updated as their parent is being updated, similar to the issue we'd identified while implementing the fine version. The difference in the two implementations is that locking over nodes imposes a sequential order of how each node will be passed through by threads, while using CAS for node updates does not guarantee this.

Our solution was to check that children of a node had the same values after an attempted CAS update as before the CAS update. If this were not the case, we would reperform the CAS update. This method, inspired by a double CAS, corrected the race issues [1].

2.2.5 Delaying Updates for Higher Levels: The Levels Saved Optimization. Although we'd implemented chunked operations and fine-grained and lock-free implementations that could use multiple threads, timing revealed the parallel implementations were still slower than the serial version of the code. We attributed this to lock contention and contention over which thread progresses in the CAS implementation. To avoid this contention, we introduced the levels saved optimization. A levels saved parameter of k indicates to the parallel implementations that the top k levels of the SegTree will not be computed individually by threads during updates. The top level corresponds to level 0, so a levels saved parameter of k means that updates occur as described up to level k, then we sequentially perform updates per level after all updates in an update batch have occurred.

This achieves a speedup since we no longer have to lock over the top nodes, which otherwise are highly contended over. Instead, we wait via a barrier for all updates to complete through the kth level, then persist these changes to the root. At last, our parallel implementations of the problem were finally outperforming the sequential version.

Interestingly, we found it best (via timing code) to assign nodes in the second sweep contiguously to threads in the fine implementation but interleaved to threads in the lock-free implementation. We believe this is due to the fact that the lock-free implementation uses an atomic integer vector while the fine-grained implementation uses an integer vector. Exactly why this is the case is outside the scope of this project. Moreover, some later implementations perform the second sweep with one thread, so we do not focus on this optimization.

2.3 GPU Implementations

For our GPU implementations, we used C++ and CUDA code, targetting high performance on the GPUs on the GHC machines.

2.3.1 CUDA Prefix-sums implementation. As discussed in the background, one way to perform $O(n)$ updates and $O(1)$ queries is to maintain a prefix sums array. Updating position i then requires updating positions $i+1, ..., n-1$ with the same delta. We hypothesized that a GPU could very quickly perform updates using warps and vectorized instructions, since updates to a prefix sums array would be to contiguous sections.

This implementation allowed the SegTree to quickly be computed on the device, but the allocation of the SegTree on the GPU (device) meant that data movement was necessary. Either we could compute query results on the device and move them to the host, or we could move the SegTree from the device to the host and compute query results on the host. We tried both methods, even trying asynchronous memory movement for the query results (since query results are not necessary until the trace is finished), however, we could not beat even our serial implementation.

2.3.2 CUDA Levels implementation. We originally planned on creating a CUDA manual range query implementation where we simply keep the array, performing updates in $O(1)$ and manually querying in $O(n)$. However, due to the poor performance of our CUDA Prefix-sums implementation, we pivoted away from this.

Instead, we created a CUDA Levels implementation. In this implementation, we use the SegTree array representation. Updates to the leaves are performed in parallel by different warps which use an atomic operator to perform individual updates. After all of these updates have been performed, we compute each remaining level in parallel with warps, all the way to the root.

More specific timing code identified that the SegTree's maintenance (updating) and querying were fastest in this implementation. However, this implementation is also severely limited by the amount of data movement between the host and device that must occur. This implementation was about 4x faster than our prefix-sums implementation.

2.4 Further Performance Optimizations

Timing code indicated our CUDA implementations were slower than the serial, fine-grained, and lock-free implementations. Even though the CUDA levels implementation was faster than the CUDA prefix implementation, it was still anywhere from 2-8x slower than our lock-free implementation. More specific timing code indicated that this was due to the data movement between the host and GPU. We thus explored further performance optimizations on our threaded CPU implementations: lock-free and fine-grained locking.

2.4.1 Prefetching. Within a single SegTree update, writes must occur on every node on the path from the leaf corresponding to the update index to the root. This also involves reading the children of updated nodes on the path. With the levels saved optimization, this path is compressed to the leaf to its first saved ancestor. Looking back to Figure 1, each tree level is stored as a contiguous subsequence of the array data structure. Therefore, traversing the path

involves jumping back in the array, highlighting the lack of spatial locality on an update. This suggests that prefetching these predictable reads would improve performance.

Additionally, once we reach our saved levels portion of the update, we iterate across nodes within the same level, rather than jumping levels. Without padding, reading a node on level k will bring the subsequent nodes on level k into the cache. This inherent cache line prefetching within our algorithm removes the need to prefetch during the levels saved portion of updates.

In contrast, the sequence of interior and leaf node reads necessary for queries is unpredictable and dynamically computed to achieve $O(\log n)$ query complexity. Therefore, we did not implement prefetching for queries.

To reach our final prefetching strategy, we had to make two major decisions: what do we prefetch and how early do we do it. Each node in an update must read its two children and combine them. Our initial algorithm prefetched the current node's parent and it's sibling, the other child of the parent. However, prefetching this sibling is unnecessary since an earlier prefetch to the current node will bring the sibling to the cache via cache line prefetching and the adjacency of siblings. To find the optimal aggression of prefetching, we added a command line argument that specifies how many levels up to prefetch. The outcome of the prefetching performance boost and testing different levels of prefetching aggression are shown and discussed in the results section.

2.4.2 Padding. False-sharing occurs when two different threads repeatedly attempt to gain exclusive access to different pieces of data on the same cache line. Despite the data having no dependencies to each other, cache lines are the smallest level of granularity for achieving this exclusive access, so the threads must falsely share. With respect to our CPU implementations, this occurs for both lock-free and fine-grained locking approaches: updating a node invalidates the entire cache line. It is very likely that another thread wants to update a node on the same level and now must read the data and request exclusive access again. This same reasoning is also true for locks on nodes if they are not padded. Therefore, we padded our mutex locks for our padded fine-grained locking and lock-free implementations.

Padding also has costs: increasing the size of every array position to a full cache line increases the data structure size and cache footprint. The set of memory being worked with is much larger, decreasing the number of nodes that can fit in each cache. Therefore, an average memory access could actually take longer, even after removing false sharing. In our results section, we discuss the outcome of padding on these CPU implementations.

3 Testing

For testing, we began hyperparameter tuning by testing different amounts of levels saved on the GHC machines. After this, we largely used scripts on the PSC machines. We measured implementation performance varying metrics such as thread counts, levels of prefetching aggression, etc., which will be described more in the results section.

3.1 Traces

Our traces varied the array size, the number of operations, and the chunk size, as shown in Table 1. For all of these traces, the combining function is addition, the query proportion is 0.5, and the update delta values are randomly generated in the range (-20,20), with the intent of keeping the partial and total sums about zero.

Name	Array Size	Operations	Chunk Size
Small	$2^{16} = 65536$	$2^{18} = 262144$	$2^{10} = 1024$
Medium	$2^{18} = 262144$	$2^{18} = 262144$	$2^{11} = 2048$
Large	$2^{20} = 1048576$	$2^{20} = 1048576$	$2^{12} = 4096$

Table 1: The main trace files for testing.

By testing at different array sizes, we could see how themes such as contention and speedup are affected by different array sizes. We note that the CUDA implementations were profiled on the GHC machines since we were not given GPU compute capability on the PSC machines.

3.2 Nonconstant Combining Functions

One workload we wanted to explore during this analysis was our implementations' performance using non-constant combining functions. A usual problem with analyzing non-constant combining functions for a SegTree workload is that the root of the SegTree, representing the combination of all values in the array, can become too large. For our nonconstant combining function, we still use the same traces (operation: +), but we simulate the function taking longer for larger levels.

Let the time of the actual computation required to compute a node under the sum implementation be k seconds. Our nonconstant combining function simulates nonconstancy by taking k seconds to acutally combine the values and update the parent then sleeping for an additional $k \cdot l$ seconds, where l is 1 at the leaf level, 2 at the next level up, and so on until l is equal to the number of levels at the root. This has the effect of making our combining function $O(l)$ where l is the current level (however, in this definition, the leaves correspond to level 1 whereas we've previously used level 0 to define the root node).

4 Results

Our goal in this project was to create optimized parallel implementations of a data structure supporting updates and range queries that minimized wall-clock time and compared favorably to the performance of a serial algorithm across a variety of different workloads. When discussing wall-clock time, we focus on the computation time because initialization time is the same across implementations.

4.1 Speedup Graphs

We show our speedup graph for the computation speedup as a function of the number of threads for the small, medium, and large traces in Figure 2. Speedup here is defined as `CompTime(mode,threads=1) / CompTime(mode,threads=k)` where we vary k and the mode is either fine or lock-free. Therefore, while it may seem as though at each array size, the fine implementation out scales the lock-free

implementation, this is due to the raw speed of the lock-free implementations at a thread count of one. Each implementation scales better as the array size increases, which shows how contention limits speedup, since contention is less frequent on larger arrays. We attribute our poor scaling at high thread counts to the fact that

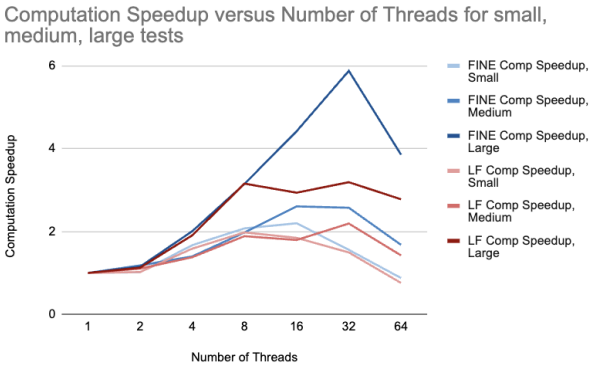

Figure 2: Computation speedup over one thread.

the workload with a constant combining function is memory-bound as opposed to compute-bound. Therefore, the overhead due to additional threads is not worth splitting the computation. However, this is discussed more in the nonconstant results section.

4.2 Impact of the Levels Saved Optimization

More comprehensively discussed in the approach section, this optimization saves the top l levels for a second sweep of updates. We discuss the results of differing l in this section. Since we wanted to find the optimal levels saved hyperparameter before testing on the PSC machines, these tests are the only tests in the results section conducted on the GHC machines.

4.2.1 Fine-Grained Locking v.s. Lock-Free. In Figure 3, the two graphs show the computation speedup of the fine-grained locking and lock-free implementations across 4 and 8 threads for the medium and small traces on the GHC machines. Speedup here is defined as `CompTime(mode,threads,levels_saved=0)` divided by `CompTime(mode,threads,levels_saved=k)` where we vary k.

In the medium trace, the SegTree has 19 levels, and performance is optimized at 16 levels saved for fine and 14 levels saved for lock-free. In the small trace, the SegTree has 17 levels, and performance is optimized at 14 levels saved for fine and 12 levels saved for lock-free. Across both traces and thread counts, the optimal number of unsaved levels (the number of total levels less the optimal number of saved levels), including the leaves which must be updated, is 3 for fine and 5 for lock-free.

The difference between these two implementations lies in performance under contention. The fine-grained locking implementation uses `std::mutex` locks and the lock-free implementation uses compare-and-swap loops, specifically `std::compare_exchange_weak` with relaxed memory order [3, 5]. Every update must persist to the roots, so the locks in levels closer to the root are more contended for than levels closer to the leaves. In contrast, the CAS loops continuously, trying to win access to the contended data, requiring

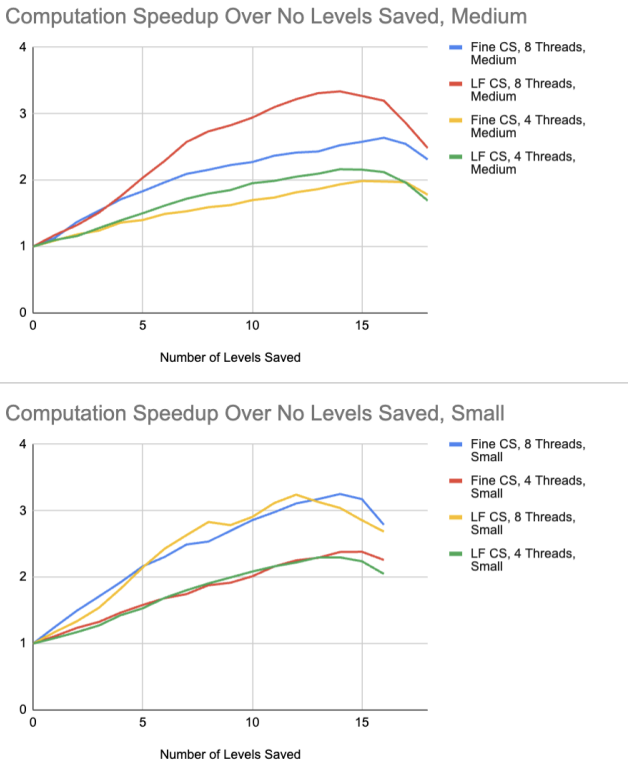

Figure 3: Computation speedup over no levels saved.

less locking overhead. Since CAS loops provide atomic updates at a lower overhead than locking, it makes sense that a lock-free implementation can use multiple threads to perform atomic updates under higher levels of contention than a locking equivalent.

Interestingly, it didn't matter whether the array was small or medium, the optimal number of unsaved levels was the same. We attribute this to the fact that the workload is memory-bound, which we discuss more in the nonconstant results section.

4.2.2 Impact of Levels Saved on Prefetching. Combining prefetching with levels saved allows for quicker updates on the path to the lowest saved level. This changes the optimal number of levels saved under different prefetching strategies.

Figure 4 shows the computation speedup over no levels saved by the prefetching optimized fine-grained locking and lock-free approaches. Speedup here is defined as `CompTime(mode, threads=16, P, levels_saved=0)` divided by `CompTime(mode, threads=16, P, levels_saved=k)`, similarly to the previous section. Therefore, the speedup graph does not compare performances across implementations, but it does allow for comparison across prefetch aggressions and across different levels saved within implementations.

We performed this testing on the PSC machines with the large trace and 16 threads. We used the large trace because its in larger arrays that prefetching will face less contention, and the thread count was chosen as a baseline. The large trace has 21 total levels.

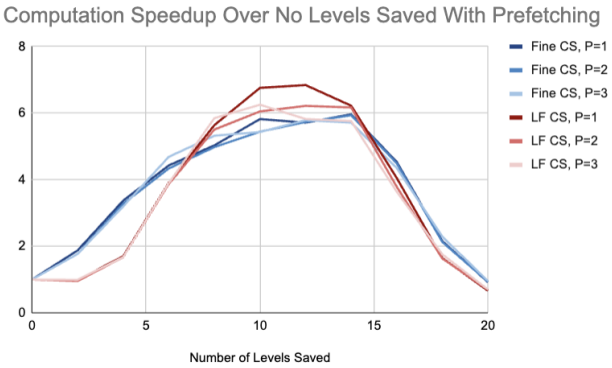

Figure 4: Computation speedup over no levels saved with prefetching.

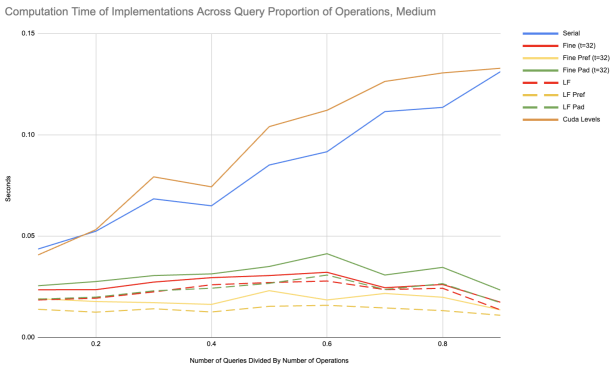

Figure 5: Computation time across query proportions on medium trace.

$P = 1$ corresponds to a node prefetching their parent while it's being updated, while $P = 3$ corresponds to a node prefetching their great-grandparent. Prefetching more aggressively, as in the latter case, offers a higher guarantee that the data will have reached the processor by the time it is needed, but this strategy is faces a higher risk that the prefetched cache line will be evicted or have its exclusive access downgraded via contention from another thread by the time it is used. This is because the higher a node in the SegTree, the more leaves and thus updates are in its subtree. Prefetching less aggressively has the opposite effects. The data may not reach the cache in time, but it has a lower risk of being evicted or downgraded from exclusive access.

As seen in Figure 4, the optimal lock-free levels saved hyperparameter under each prefetch strategy is less than the fine-grained optimal levels saved hyperparameter, consistent with our findings from the previous section. Looking at the fine-grained locking lines, all 3 are incredibly similar, and there does not seem to be a clear relationship between prefetch levels and optimal levels saved. However, the variation among the lock-free lines is much greater. The obvious computation speedup winner is lock-free with $P = 1$, achieving a computation time of 60.6 milliseconds which is 5 fewer than the fastest of P=2 and P=3. The data suggests that the risk of contention and evicting or downgrading prefetched cache lines outweighs the risk of data not arriving in time. The tradeoff here is between saving fewer levels, so that prefetching can take a greater effect due to longer leaf-to-last-unsaved-ancestor-paths, and saving more levels, so that contention over the top levels, and thus lock overhead, does not dominate.

4.3 Varying Query-to-Update Ratio

A crucial factor distinguishing update and range query workloads is the ratio of queries to updates.

Figure 5 shows the computation time of our best SegTree implementations across the proportion of updates that are queries, holding all other workload variables fixed.

With parameters including levels saved, number of threads, and prefetching aggression, we used an iterative approach to find the optimal configurations for each implementation. First, we tested all possible levels saved across 2, 4, and 8 threads on the GHC machine to find the optimal levels saved hyperparameter input. For the prefetching implementations, we tested all levels saved with P=1, P=2, and P=3 prefetching aggression. Then, we tested all possible number of threads with levels saved and prefetching aggression held constant on the PSC machines (we omit this graph, it is a happy medium with a peak at a thread count of 32, similar to the speedup graphs).

This testing used the PSC machines and the optimal number of levels saved and threads for the fine and lock-free implementations. The algorithms completed the medium workload.

4.3.1 Query Ratio Affects Parallel Speedup Over Serial. At low query proportions, it is clear that the fine and lock-free implementations are faster than the serial implementation by about 2-4x. As the query proportion increases, the computation time of the serial algorithm increases linearly while the other CPU implementations stay relatively constant. On query heavy workloads, the parallel CPU implementations are 5-9x faster than the serial algorithm.

The serial implementation struggles with queries for multiple reasons. First, accesses at each tree level are not contiguous for queries. If a range query from i to j contains a block, or a node at a higher tree level, then subsequent levels must search outside of this block to fill in the range query gaps. This hurts spatial locality compared to updates which read neighboring children. Second, queries recursively search through the tree, often using multiple recursive calls in a single level of the recursion, spawning recursive branches. Meanwhile, an update has a straight path from the leaf to the root.

When parallelizing across operation chunks, threads must contend for nodes during updates since they require an exclusive cache state to write to the node. On queries, there is no resource contention. This difference between operations mitigates the speed advantage that updates inherently have over queries in SegTrees as shown in the serial line. This results in no significant relationship between query-to-update ratio and computation time for the parallel CPU implementations.

4.3.2 CUDA Levels Struggles With Thread Divergence in Queries. The performance of the parallel GPU implementation, CUDA levels,

is very similar to the serial implementation for slightly different reasons. CUDA levels performs updates on the device in kernels one level at a time, saving all levels but the bottom, the leaf nodes. Only the bottom level relies on the actual indices being updated. All other levels update all nodes on their level. It performs the queries on the host due to the divergence of queries. In an earlier version, CUDA levels called kernels to allocate queries to threads, but the divergence of the CUDA threads and overhead of kernel calls outweighed the gained parallel arithmetic resources. With respect to CUDA threads' strengths, there is no similarly parallel aspect of query chunks compared to update chunks.

4.3.3 Parallel CPU Implementations Performance Analysis. As shown in Figure 5, lock-free with prefetching ($P = 1$) has the best performance on the medium trace across all query ratios with each implementation using their optimal hyperparameters. Additionally, there is a consistent ordering of performance among the three lock-free and fine-grained approaches: prefetching, then normal, then padding. This suggests that the extra memory overhead and thus increased working set size from padding dominates correcting the false sharing issues. This intuitively makes sense with our levels saved optimization avoiding the frequently contended cache lines of higher levels on the SegTree.

For the lock-free implementation, prefetching achieves a 25% to 107% speedup over the non-prefetching approach. This peaks at a 2.07x speedup when the query proportion is 0.4. For the fine-grained locking implementation, prefetching achieves a 13% to 81% speedup over the non-prefetching approach, also peaking when the query proportion is 0.4.

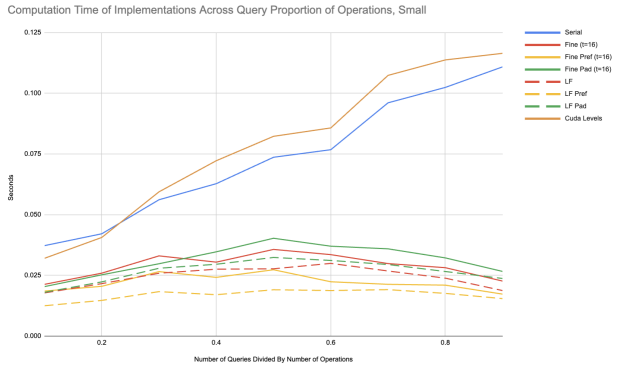

Figure 6: Computation time across query proportions on small trace.

Figure 6 shows the same computation time comparison across query proportions as Figure 5 but on the small trace. The graphs are incredibly similar, highlighting the consistent speedup that our parallel CPU implementations generate over the serial implementation across different problem sizes.

4.3.4 Lock-free Prefetch Timing Breakdown. Table 2 shows an execution time breakdown of different components of lock-free prefetch. This timing was done with the medium trace on the GHC machines with the optimal hyperparameters. The components leaf update, path update, query traversal, and barrier wait are average times across all threads. Since the levels saved is computed by 1 thread, that metric is summed. Total comp represents total computation time.

Component	2 Threads	4 Threads	8 Threads
Leaf Update (ms)	3.10	2.05	1.40
Path Update (ms)	11.99	8.07	5.19
Query Traversal (ms)	31.29	15.95	8.17
Barrier Wait (ms)	1.92	2.44	3.37
Levels Saved (ms)	1.58	1.72	1.76
Total Comp (ms)	61.85	37.28	24.22

Table 2: Lock-free prefetch time per component.

At lower thread counts, query traversal takes up a higher percent of the total computation time: 51%, 43%, and 34% at 2, 4, and 8 threads respectively. As discussed earlier in this section, threads do not compete for shared resources (locks, CAS access) since queries are read only, which describes why these percentages decrease.

Total leaf update times are 6.2, 8.2, and 11.2 ms while total path update times are 24, 32.3, and 41.52 ms. Total query traversal times are 62.7, 63.8, and 65.4 ms. This matches the expectation that total query time would stay near constant with more threads while total update time would increase due to contention.

Component	2 Threads	4 Threads	8 Threads
Leaf Update (ms)	2.99	2.14	1.49
Path Update (ms)	28.67	21.49	14.65
Query Traversal (ms)	31.15	16.20	8.02
Barrier Wait (ms)	0.63	1.58	5.13
Levels Saved (ms)	0.04	0.05	0.07
Total Comp (ms)	87.26	58.19	42.76

Table 3: Lock-free prefetch time per component with fewer levels saved.

Throughout this paper, there has been an emphasis on the impact of the levels saved. Table 3 displays the same information as Table 2 but with 8 levels saved instead of 14. Obviously, the total time spent on updating the saved levels decreased to near 0, but this comes with the path update time more than doubling for each thread count. This tradeoff severely hurts total computation time, demonstrating the clear impact of the optimization.

4.4 Nonconstant Combining Function Workload

In this section, we show results that differ from the constant combining function results. The computation speedup graph in Figure 7 shows near perfect speedup. We attribute this to the fact that the nonconstant combining function makes the workload compute-bound.

4.4.1 Impact of a memory-bound workload on computation speedup. In our constant combining function analysis, Figure 7 portrayed poor scaling at higher thread counts. Our new analysis tells us that this could be because the constant combining function workload becomes memory-bound at higher thread counts, as the threads have less computation to share at higher thread counts.

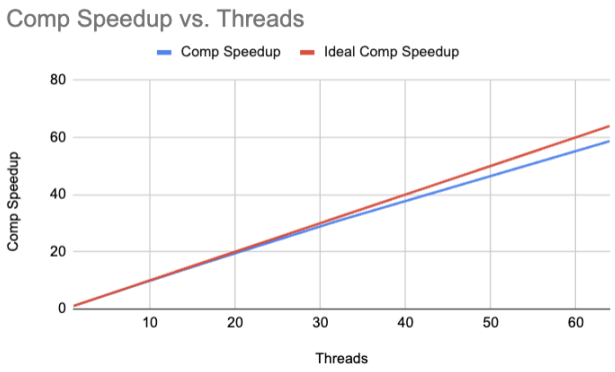

Figure 7: Computation speedup over 1 thread, nonconstant workload.

4.4.2 Impact of a memory-bound workload on the optimal levels saved parameter. Identifying that our workloads in the constant combining function analysis are memory-bound allows us to explain results that seemed initially confusing. For example, whether the array size was small or medium did not seem to affect the optimal levels saved parameter. In both cases, the number of unsaved levels was equal across array sizes, being 3 unsaved levels for the fine implementation and 5 unsaved levels for the lock-free implementation. This is counterintuitive since there are more nodes in the medium array 2 levels up from the leaves (65,536) than there are nodes in the small array 2 levels up from the leaves (16,384). Therefore, there should be more contention in the smaller array, assuming equal thread counts.

The fact that there are the same number of memory loads when, independent of array size, we do not save 3 levels can explain this phenomenon. This is further reinforced by our results in finding the optimal number of levels saved, k, for the nonconstant workload, shown in Table 4. These results show that the optimal computation speedup is at different amounts of unsaved levels for different sized arrays.

k	FINE, med	FINE, small	LF, med	LF, small
0	1	1	1	1
2	1.0647	1.0760	1.0630	1.0736
4	1.1396	1.1613	1.1381	1.1632
6	1.2237	1.2581	1.2225	1.2646
8	1.3187	1.3701	1.3175	1.3786
10	1.4200	1.4815	1.4189	1.4914
12	1.4992	1.5157 *	1.4967	1.5257
14	1.4288 *	1.2450	1.4274	1.2525
16	0.9903	0.6327	0.9892	0.6353
18	0.4121		0.4109	

Table 4: Comparison of fine and lock-free computation speedups at different levels saved.

4.4.3 Impact of a memory-bound workload on prefetching. We show our results for the computation speedup over zero levels in Figure 8. We see that there is a smaller degree of speedup (less than 2x speedup over 0 levels saved) compared to the same result for the memory-bound workload (greater than 6x speedup over 0 levels saved). This is telling us that the speedup due to prefetching in the memory-bound workload was much larger than that of the compute-bound workload, which makes sense, since prefetching helps hide time due to memory reads (by prefetching into our cache) and writes (by prefetching in exclusive mode).

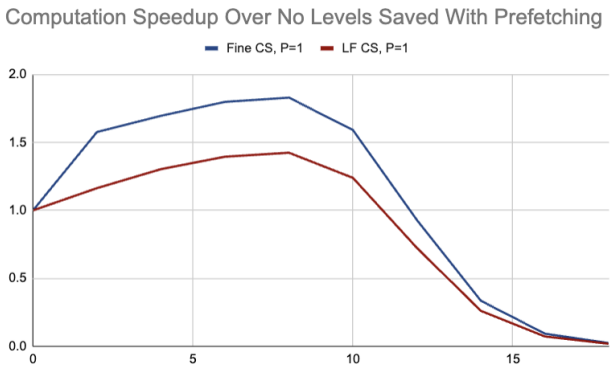

Figure 8: Computation speedup over no levels saved with prefetching, nonconstant workload.

4.4.4 Final Performance Results. Figure 9 shows the computation time comparison across query proportions. This graph is incredibly similar to the same results for constant combining function workloads, indicating that no matter whether the workload is compute or memory bound, the implementations provide a large speedup over the serial implementation.

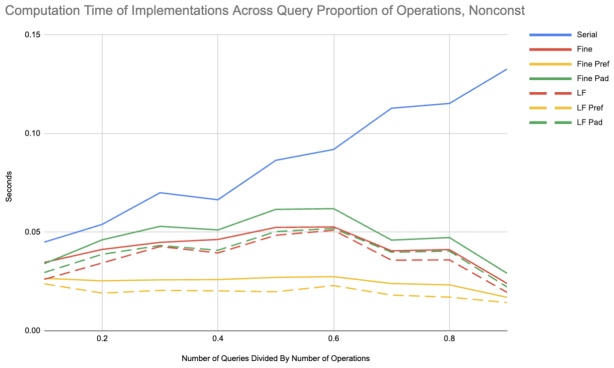

Figure 9: Computation time across query proportions on medium trace.

4.4.5 Nonconstant analysis takeaways. The nonconstant analysis allowed us to further analyze constant analysis, and we learned that our constant combining function workloads were highly **memory-bound**.

5 Work Distribution by Student

Table 5 shows distribution of work to each student by task. Almost every minute we spent working on this project was done together.

Tasks	Sebastian	Stefano
Overall	50%	50%
File Structure	25%	75%
Main File	25%	75%
Gen Test File	30%	70%
Serial Impl.	80%	20%
Coarse Impl.	30%	70%
Cuda Levels Impl.	85%	15%
Cuda Prefix Impl.	60%	40%
Fine Impl.	50%	50%
Lock-free Impl.	50%	50%
Add Padding	65%	35%
Add Prefetch	65%	35%
Add Nonconst	80%	20%
GHC Testing	30%	70%
PSC Testing	0%	100%
Graph Creation	20%	80%
Graph Analysis	50%	50%
Project Proposal	50%	50%
Milestone Report	50%	50%
Final Report	50%	50%
Poster Creation	50%	50%

Table 5: Distribution of work by task.

References

[1] Carnegie Mellon University 15-418. 2025. Lecture 18: Fine-grained synchronization lock-free programming. https://www.cs.cmu.edu/afs/cs/academic/class/15418-s25/public/lectures/22_lockfree.pdf. Lecture Notes, Accessed: 2025-04-28.

[2] Carnegie Mellon University 15-451. 2024. Range Query Data Structures: 15-451 Algorithm Design and Analysis Lecture Notes. https://www.cs.cmu.edu/~15-451/lectures/. Lecture Notes, Accessed: 2025-04-28.

[3] cppreference.com. 2023. std::atomic<T>::compare-exchange-weak - cppreference.com. https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange Accessed: 2025-04-28.

[4] cppreference.com. 2023. std::thread - cppreference.com. https://en.cppreference.com/w/cpp/thread/thread Accessed: 2025-04-28.

[5] cppreference.com. 2024. std::mutex - cppreference.com. https://en.cppreference.com/w/cpp/thread/mutex Accessed: 2025-04-28.