Performance Analysis: An Implementation of Lock-free B+ Trees in GoLang

Srisaidhruti Kuchibhotla (srisaidk), Adhvik Kanagala (akanagal)

Carnegie Mellon University, Pittsburgh, PA

Abstract

In this paper, we present a lock-free implementation of B+ trees, a widely used data structure for maintaining sorted data in databases and file systems, and compare its performance to that of sequential, global lock, and latch crabbing implementations of the same data structure. Lock-free data structures aim to improve concurrency and scalability by allowing multiple threads to operate on the structure without contention. Our proposed lock-free B+ tree is designed to provide efficient find, insert, and delete operations in a concurrent environment. It is implemented in GoLang for ease of thread management.

We evaluate the implementations using a variety of workloads and concurrency levels, measuring key performance metrics such as throughput and scalability across experimental setups on both the Apple M1 architecture, and the PSC Bridges-2 supercomputing cluster. Our experimental results demonstrate that the lock-free B+ tree achieves competitive performance, especially under high levels of concurrency, outperforming its sequential and globally locked counterparts, and performing comparatively to its latch crabbing counterpart.

Our code is available at https://github.com/dhrutik/15418-final-project.

Fall 2023

Final Project for 15-418 (https://github.com/dhrutik/15418-final-project).

Background

B+ trees are M-way search trees, where M represents the maximum number of children a node can have, that are widely employed in the realm of database systems. They provide an efficient means of database indexing, storage, and retrieval of data. Key operations such as `insert`, `find`, and `delete` are able to be executed in $O(\log(n))$ time. B+ trees are balanced tree data structures where each node contains a certain number of keys and pointers. The keys are sorted, allowing for efficient search, insertion, and deletion operations.

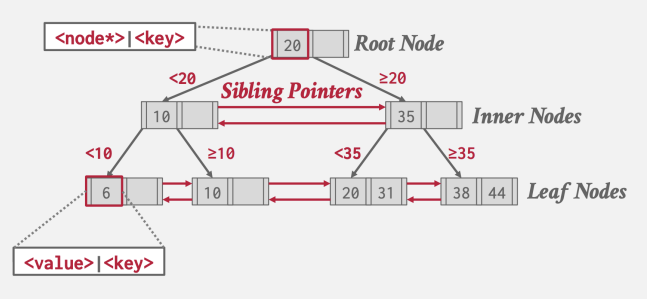


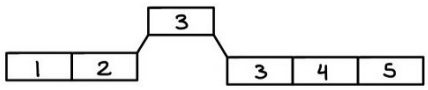
Figure 1: Generic B+ Tree structure. [5]

The tree's structure ensures that data is stored in a way that optimizes range queries and sequential access. B+ trees maintain the following invariants:

- **Balance Property**: The tree is perfectly balanced (i.e., every leaf node is at the same depth).
- **Half-full Property**: Every inner node, other than the root, is at least as full as a predefined lower bound (typically half full).

- **Separator Keys Property**: Every inner node with k keys has $k + 1$ non-null children.

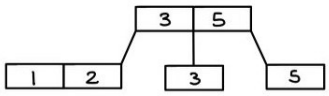
While these invariants ensure efficient execution of operations, their enforcement results in write-heavy operations, such as node splits and merges whose effects can propagate throughout the tree. These events occur when an invariant is temporarily violated mid-write, like in an Insert or Delete. Splits occur when a node has reached capacity, and therefore must "split" its data across itself and a new node.



The rightmost node splits on insert 6, and these changes propagate up.

Figure 2: Diagram of how an `insert` operation can result in node split.

Merges/redistributions occur when a node has fewer keys than a predefined lower bound on capacity, so it coalesces itself with a neighboring node.



The leftmost node merges/redistributes on delete 3, and these changes propagate up.

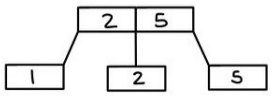


Figure 3: Diagram of how a `delete` operation can result in node merge/redistribution.

The main data structure to note within the B+ Tree structure is the Node structure. There are two types of nodes, those being Inner and Leaf nodes. Inner nodes contain keys and pointers to child nodes. These nodes help guide the search to the appropriate leaf node during lookups. Leaf nodes store the actual data entries (or pointers to such). These nodes are also linked together to facilitate range queries.

Latch Crabbing

The locking procedure that is most widely used on these trees is commonly known as "latch crabbing." This is a technique often used in concurrent B+ tree implementations to reduce contention and improve parallelism over aforementioned write-heavy operations. Traditional, or naive lock implementations can lead to contention issues, especially in high-concurrency scenarios. The latch crabbing technique allows for the minimal subsection of the tree to be contained in the critical section, to decrease the amount of contention, and latency lost as a result of it. Specifically, instead of holding one single, large lock for the entire duration of the operation, latch crabbing involves temporarily releasing locks as the operation progresses. The concurrent thread moves through the tree, releasing locks on nodes that are no longer deemed "critical" to the current operation. This allows other threads to access those nodes concurrently.

To illustrate a small example, if the left-most leaf node is to be split, this change may not propagate all the way to the right-most leaf, so we can deem the right-most leaf to be non-critical, and release the latch on it, so other threads can potentially perform operations on this remaining portion.

Computational Expenses

However, this sort of latching can be difficult to scale. The process of acquiring and inspecting such latches can introduce significant overhead and uptick in communication costs. In multi-core environments, to acquire a latch, the system must fetch the memory associated with a latch into the processor cache of the thread requesting it, which may also necessitates this latch's memory being flushed

from all other threads. In the vent where there is a critical section whose latch is modified by several threads, or is simply frequently accessed, this latch's memory will bounce around between caches, resulting in high latency and bandwidth costs [6].

Because of these sorts of complications, this paper builds on existing concurrent implementations that utilize latching to isolate critical sections whose computation can be parallelized, due to lack of dependency between sections. The latch crabbing approach has illustrated that parallelization is possible, with some clever segmentation of work. The dependencies lie in propagation of node splits/merges up through the tree, upon insertion/deletion. The locality exists within each node, and between parent/child node relationships. This paper exploits such dependencies and locality, and presents a latch-free implementation of B+ Trees, building on ideas from PALM [2], a novel technique for performing these read/modify queries on an in-memory B+ tree in a bulk synchronous fashion, without the use of latches. Bulk Synchronous Parallel (BSP) algorithms proceed in a sequence of steps, each with a local computation, communication, and barrier synchronization phases [3].

This implementation is somewhat amenable to SIMD execution, although Go does not have builtin support for SIMD – we cannot necessarily apply it everywhere, but we do have a few locations where all items in a vector could be processed independently of each other. This is in stage 2 of the lock-free algorithm (explained below). Here, each thread is responsible for processing a vector of requests. These requests are usually a mixture of insert, delete, and find queries. While the processors work to figure out which leaves are touched by the insert/delete queries, they also work to find the values that are being requested by the find queries. Each query can be processed in parallel, because it is a simple matter of reading data and writing data to independent locations. There is not much work to be done for non-Find queries, so we would likely end up masking those lanes pretty quickly. However, for a find-heavy batch of queries we could achieve nearly linear speedup with SIMD.

Approach

We initially took a barebones B+ tree implementation, and added a global lock to the tree, to introduce the concept of concurrent execution in the simplest way we could think of. We then implemented latch crabbing, the state-of-the-art in terms of concurrent execution protocols. The motivation behind this implementation was twofold, with the first being a way to allow us to familiarize ourselves with both the codebase and the Go environment, so that our actual lock-free implementation process might go smoothly, and the second being that doing so allowed us to produce our first iteration of optimization. The latch crabbing protocol is the most commonly used concurrency optimization, and one of the most efficient, on this data structure to date. Completing this implementation allowed us to now have 3 different types of optimization iterations to compare our lock-free implementation's benchmarking too. This allowed us to properly do a comparative performance analysis over the different styles of B+ Trees.

Iterations of design

Our lock-free implementation also took several iterations to reach its current state. We began by researching other lock-free protocols for more traditional data structures that often tend to utilize locks in their execution. In our literature review, we read papers on lock-free implementations of Binary Search Trees [1] , and Lock-free Monte Carlo Search Trees [4]. Through this process, we started reasoning about an almost naive approach to this lock-free implementation. Our main takeaway, was that this implementation needs to be able to isolate/group queries that do not share dependencies, so that individual threads can execute said queries on this portion of the tree

that is strictly under their ownership, thus eliminating the need to explicitly lock these portions.

Initially, we detailed an extremely naive design that attempted to batch queries. Given a particular write operation, we were able to determine up to which level of the tree these writes would propagate to, similar to how we determined the minimal critical sections in our latch crabbing implementation. Given this, we attempted to batch our queries, such that all queries that affect a particular subtree would be grouped together and assigned to a thread, so that this thread could execute all such queries without having to worry about dependencies on other portions of the tree. However, this proved ineffective, due to the fact that it was difficult to settle on a "happy medium" for subtree size/assignment. These subtree determinations had a lot of overlap, requiring great nuance and fine granularity, so we either had subtrees that were too small, which would have generated results extremely similar to the latch crabbing protocol, or subtrees that were so large they were essentially the whole tree, which would generate results similar to the global lock protocol.

Our next iteration built upon the theory provided in the PALM paper put out by the Intel Corporation. In reading this paper, we gained some intuition for what portions of the B+ Tree invariants are useful to exploit in determining thread assignments, so that individual threads do not overlap in what changes they make to the tree. The implementation we settled on, and what we ran our final performance analysis on is as follows.

Our initial train of thought was correct, in that we wanted to batch our queries, and isolate execution of those that wouldn't effect each other. An important assumption we made, was that queries made at a single time would all be operating on the same tree state, meaning the input we've defined to a single operation, was a batch of queries which would all be meant to operate on the same tree state at the start of the operation. This allows us to realize, that no matter the order of the queries, we are able to frontload all read operations (`Find` calls), i.e. we can service these first because they do not change the state of the tree, and all the information we need to service them is already present.

Implementation: Stage 1

Given this batch of queries, the first step we took was conducting a search call on all such queries. Search here is a variant of our find call, but while the `Find` operation returned the actual value associated with the key we were looking for, searching for this key returned the leaf node that contains this key. Because every query in our batch references some key, each of which is part of some node's set of keys, we reasoned that searching for all keys can produce a list (or slice, in Go terms) of affected nodes by this specific set of queries.

We parallelized the construction of this list in a naive fashion, seeing as all searches were conducted on the same tree state, and no modifications were being made. We evenly split the number of queries over the number of threads we had, allowing for a well-balanced load across threads as well.

Implementation: Stage 2

Upon construction of this, we had each thread return back this "affected leaves" list to the main thread, in a shared array across all threads, again with each one being made to only modify one index of it, so that locking is unnecessary.

Here, we redistributed work among the threads. Using the paper as a guide, we constructed the following scheme. We had each thread compare its own "affected leaves" list to all such `L` constructed by those threads with lower index number than itself, assigning itself only those

nodes that are present in its affected list, but not present in any affected list prior to it. This can be seen in the following:

$$L_i^{'} = \{\lambda \in L_i | \lambda \notin L_j, \forall 0 \leq j \leq i\}$$

Where i is the thread index, L_i is ith thread's initial affected leaves list, and $L_i^{'}$ is the new one. (Note: This protocol was determined from the aforementioned PALM paper)

After constructing these lists, we see that each thread has been assigned a list of nodes that only it is solely responsible for, as in it has ownership over this node. Where we went wrong in our initial design, was we tried to do thread assignment using subtrees, which required communication and locking between threads. Here, we have node-wise assignments to threads, allowing each one to perform operations affecting only this node, eliminating the need for communication across threads.

From here, each thread performed its `Find` queries first, then performed the write operations, `Insert` and `Delete` that would change the state of our tree. At this stage, the only nodes being affected are leaves, however on these write operations, there is a chance that changes need to be propagated up through the tree. Instead of doing this immediately, we incorporate ideas from lazy computation, and delay these operations by storing what needs to be done in a modifications map, with the key being an affected node, and the value being a list of modifications that it has caused. Each thread then returns these modifications maps in a similar fashion to Stage 1, to be accumulated into a shared list across all threads that the central thread hosts. [potentially go into semantics of the struct???]

Implementation: Stage 3

At this point, we have made all of our updates to the leaves, as well as computed the results of all Find requests. The next task to deal with involves inserting or removing any leaves that were split or merged. From stage 2, we have that each thread has constructed a map, mapping parent nodes of leaves to all of the modifications that need to be made to that parent node. This is very similar in overall structure to the Insert/Delete queries on the leaves – we are given a series of items that need to be added to/removed from each node in the lowest level above the leaves.

The immediate response might be to just have each thread manage all of its parents. However, this does not immediately work - it is possible for multiple leaves of the same parent to have been edited. This means that multiple threads will have constructed a set of updates that must be made to the same parent. Thus, we must again redistribute all of our updates such that all of the updates for any given node are carried out by a single thread. This is how we ensure thread safety.

To do this, we first do another redistribution phase identical to the one used in stage 2 for leaves. Once that has been done, each node has been assigned to a single thread. Each thread carries out the task of updating its nodes. This takes the process of adding in any new child leaves, and removing any leaves that were merged.

Similarly to the update process for leaves in stage 2, our updates to their parents could cause the core invariants to be broken – we might end up with too many or too few elements in a node. This would force a split or a merge. Upon performing this split/merge, we would then have to make updates to the parent node. Thus, we would essentially need to repeat the modification process for the next layer.

This would happen for each layer all the way up to the root,

as changes may or may not get propagated up. One difference is that we would always have to propagate something up, specifically to address the case of orphaned keys. The orphaned key phenomenon happens when a node becomes too small. In a traditional B+ tree, the remaining elements in the node would be somehow merged with values from neighboring nodes in the same layer. However, we cannot safely interact with other nodes in a lock-free design. Instead, we figure out which key/value pairs are left in the children of the remaining values in the node, and we aggregate them over all of the operations of the tree. We effectively silently delete them by deleting the node that points to them. Then, after our current batch of queries is complete, we rerun PALM to insert all of the orphaned keys in bulk.

Implementation: Stage 4

While propagating node modifications up to their parents, we eventually arrive at the root. Here, we have only one thread read all of the modifications to the root from each thread, and make all of those updates on it own. This thread also handles any updates that must be made to split the root or merge it back with one of its children. Stage 4 makes these updates, and finishes off by calling PALM on any orphaned keys. If there are no orphaned keys, it simply returns.

Technologies Used

The language we used to complete this project is GoLang. We made this decision because of the robustness and ease with which we can spawn/manage threads, with the use of the inbuilt "goroutines." These are lightweight threads managed by the Go runtime, allowing us simpler thread management, so that we can concentrate our focus on the interesting task at hand, implementing the lock-free implementation of B+ Trees. Other languages may have required more overhead with regards to thread management, both implementing and understanding it, whereas using Go allowed us to bypass this. However, because both of us were fairly new to the Go space, there was a fairly steep learning curve in terms of constructing our benchmarking protocols and understanding how to translate our design to code, given the strange data typing in the language. We were able to overcome this pretty quickly, due to the various iterations we went through, with our global lock and latch crabbing implementations, as well as iterations of our final lock-free implementation.

We built our code off of an existing barebones B+ Tree implementation found at:
https://github.com/collinglass/bptree.

The machines we targeted were our local machines, Apple's M1/M2 architectures, and the PSC Bridges-2 supercomputers, to allow us to run experiments that scaled across cores.

Synchronization: Wait Groups

To ensure that we did not run into race conditions, we needed to implement some form of synchronization between threads. As there is no locking, we needed a way to make sure there were points in our code at which all computation converged, so that at any given time, all threads were working on the same initial tree state. This necessitated the use of something like MFence instruction we learned about in class. Go has similar simple functionality for constructing these sync points. We utilized this something called a WaitGroup. When we spawned goroutines, we incremented the counter associated with our WaitGroup. Inside each goroutine we called `defer wg.Done()`. This defers the decrementation of the count until all the work we want to do in this routine is done, signaliing to the sync that this thread has completed. After spawning all goroutines, we called `wg.Wait()`, to wait for all threads to signal that they are done.

We determined the optimal sync points would be right after each stage detailed in our approach. However, because each thread had some individual computation that it needed to return and contribute to some shared array, we had to make a couple design decisions about where/how to spawn these threads. We constructed wrapper functions, for each stage which served as that stage's "main" thread. This is where we hosted the shared lists that aggregated information across threads. We spawned goroutines inside of these wrapper functions, so that each stage became self-serving, handling its own thread synchronization independent of the others. This abstraction not only helped in organization, and provided clarity when it came to debugging synchronization and correctness issues, but it also allowed us to easily pair program, and work on different components of our implementation without causing merge conflicts. In a way, removing dependencies to parallelize our real-life process of work if you will.

We did not build our lock-free B+ tree to target any specific hardware. Instead, we focused on scalability so that it would work with any number of CPU cores. The algorithm is discussed in detail above. In Go, we implemented the B+ tree by first defining a general API for each of our implementations. Each B+ tree aside from the lock-free B+ tree was required to support an Insert, Find, and Delete operation. We then built a generic benchmarking tool that could take any tree and run a sequence of insert, find, or delete queries on it. We built multiple trees using 1-64 (or 128 for PSC) threads to build each tree. We calculated how long it took to run each group of operations on the tree, giving us a throughput in keys per second. We then compared this throughput to the same sequence of operations on the original sequential tree, giving us a clear speedup coefficient for each of our implementations.

In order to map the solution to target parallel machines, we chose to use Golang's goroutine framework. Goroutines are very lightweight threads in Golang that are very easy spin up and manage. Golang provides a synchronization primitive known as a channel, which is an extremely powerful tool for thread-safe data sharing and thread management. Alongside the waitgroup discussed above, we also used channels in stage 3 as a barrier synchronization primitive to force synchronization than threads after each layer of the tree had been updated.

The serial algorithm is very different from the PALM algorithm. The immediate change is that the serial algorithm processes an entire query at once before processing another. On the other hand, PALM is explicitly designed to process requests in batches. Here, all requests are processed in parts, at the same time. This required a very different way of thinking about operations. This was especially true when it came to inserting/removing elements.

With the previous sequential version, we were only ever inserting or removing a single element at a time from any given node. This means that rebalancing would only require that we transfer one value from the left or right neighbors of a node. In the lock-free version, we are given series of updates to apply to any given node. In an extreme case, the node could increase in size to more than double the original max. In those cases, we are forced to insert multiple nodes into parent, which could in turn cause multiple splits in the worst case. This led to a lot of thinking on our part about how we could cleanly implement this multiple-split algorithm. We eventually ended up using the algorithm provided by the PALM authors – essentially we would process all operations at once, and in the worst case would split our oversized node into multiple smaller nodes after all operations on a node had completed. Then we would insert all of our new nodes into the parent node at once. This could in turn cause multiple splits, but the problem was solved.

Results

We managed to achieve our base goals of building a lock-free B+ tree and comparing its performance to other versions of a B+ tree. Our four versions were a sequential tree, a globally-locked tree, a latch crabbing tree that used fine-grained synchronization, and finally the latch-crabbing tree that uses BSP to achieve thread safety without any sort of actual locking of resources.

We were able to benchmark each of our implementations on a series of Insert, Find, and Delete queries. We chose to measure performance by the throughput that each tree supported as a function of the number of threads used to execute the operations. Throughput was measured in units of keys per second. For each benchmark we generated 1,000,000 key/value pairs, where each key was between 0 and 1000000. We partitioned these million requests across each of the threads, so that each thread had less queries to make as the number of threads went up. We chose to stop at 64 threads when running locally, and stopped at 128 threads when testing on PSC. This is because PSC's CPUs support 128 hardware cores. We usually noticed a big falloff in performance on our local machines past 16 threads, so we did not feel the need to test beyond 64 threads because the performance degradation was fairly obvious.

Another concern we wanted to handle was the possibility that having a series of requests that were sorted by the key might cause a skewed series of operations. This would cause all of our splits to always happen in the rightmost leaf of the tree, because we would always be adding a new value that was larger than all the others. To solve this, we generated all of our requests in advance and then shuffle them randomly so that we could get a more balanced workload across threads. We chose to use 1 million keys for these benchmarks, as we felt it was a large enough number for the average throughput to be fairly represented (division not subject to underflow etc).

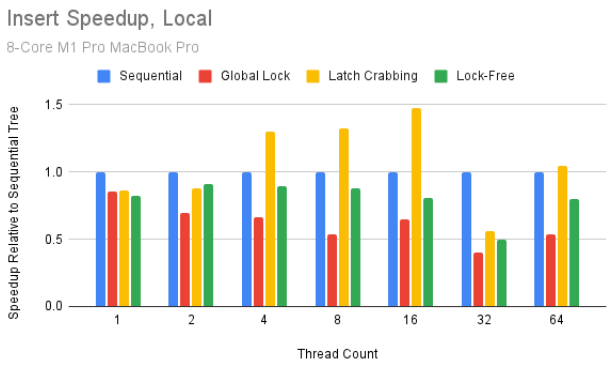


Figure 4: Results of running the Insert benchmark on local machine (M1 architecture)

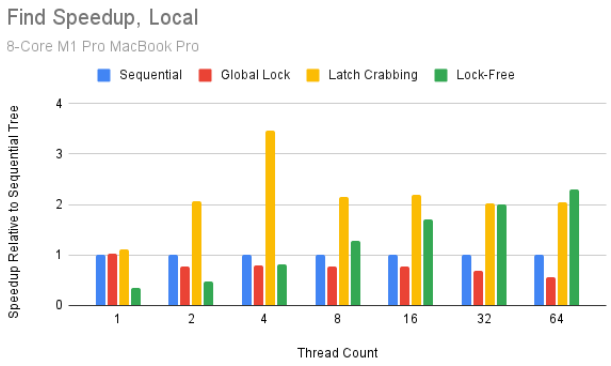


Figure 5: Results of running the Find benchmark on local machine (M1 architecture)

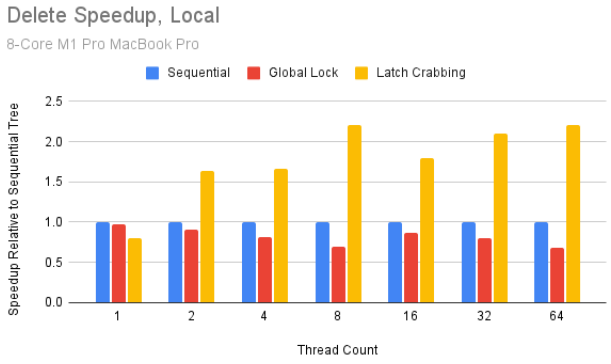


Figure 6: Results of running the Delete benchmark on local machine (M1 architecture)

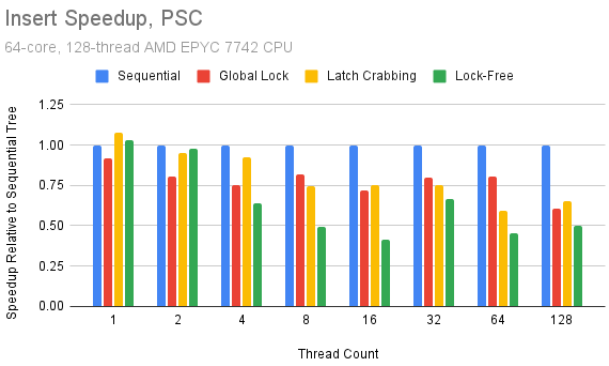


Figure 7: Results of running the Insert benchmark on PSC (AMD Epyc)

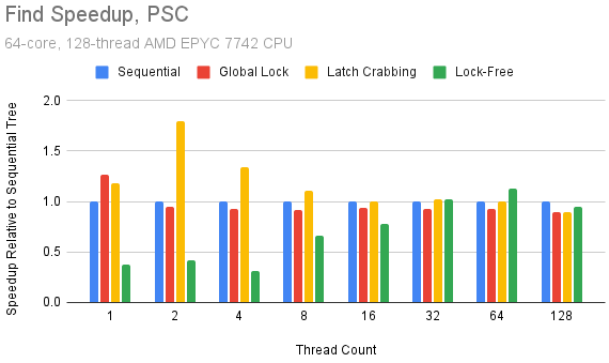


Figure 8: Results of running the Find benchmark on PSC (AMD Epyc)

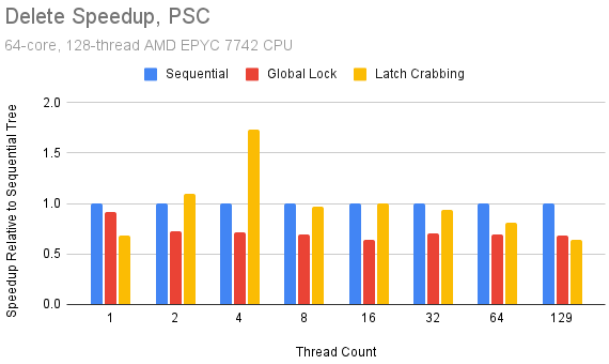


Figure 9: Results of running the Delete benchmark on PSC (AMD Epyc)

Looking at our graphs, we were both a bit unhappy with the low-seeming speedup we archived for the latch crabbing and lock-free implementations. In calculating speedup, we compared the throughput of each tree with a specific thread count to the original sequential B+ tree's throughput. The goal was to observe how increasing thread counts would increase the throughput. Breaking down the performance by tree type:

- Global lock trees were initially surprising to us, but we ended up completely understanding why they behaved like they did. The core issue with the global lock trees was the level of contention for access to the tree's main lock. Especially as we increased the thread count, we observed a significant loss in throughput, dropping down below a 0.5x speedup in the worst case.

- Latch crabbing was also less performant than we had expected. The original idea for a fixed size of 1 million keys per tree was meant to provide enough time for the tree to grow and reach a state where it was rarely being split or merged with other teams/lives. Instead, we found the following:

  – There was usually some speedup with latch crabbing, but there was still overhead in taking a lock.

  – Another potential issue is that we may have been too conservative with our lock-holding policy. This primarily showed up in cases where we held a sequence of parent/child nodes down to a leaf. Once we split a leaf and inserted it into its parent, we were safe to release the lock. However, we instead waited until all of the insertions into parent nodes had completed before releasing the leaf locks and their parents until after the entire transaction was complete. This likely led to some threads waiting much longer than necessary.

- This left the lock-free implementation, which we had hoped would scale very easily. Instead, we found that PALM is generally slower at lower core counts. This seems to be because of the large amount of work that needs be done to reshuffle various nodes. We observed that as the number of threads went up, we were more easily able to distribute this reshuffling work across each thread. Thus, we observed slightly better speedup with increased parallelism locally, although this once again

fell off past 16 threads. For the Find operation, we observed a very promising trend of improved performance over time – this is likely because the Find operation was comprised entirely of read operations, and thus there was minimal work to be done past the initial shuffling phase of PALM. Here, we took advantage of multiple threads to greatly speed up the work redistribution, and then each thread was able to run to completion without waiting for any other threads. We even observed that the Find operation specifically was able to scale with increased thread counts locally – it finally was able to beat the latch crabbing implementation on 64 threads, locally. On PSC we osberved somewhat similar behavior, but we believe there were other confounding factors that led to our multithreaded code being slower than the sequential version. We discuss this below.

Limitations and Discussion

We initially built, tested, and ran all of our code on our local machines, which are 8-core M1 Pro and M2 chips by Apple. Both run on the ARM architecture, and natively support Golang. Once we felt confident about our implementations, we compiled our code into x86 binaries on our local machines and then copied those binaries over to the Bridges2 machines. Then we simply ran the binary, which would benchmark each tree and output the throughput.

Our main, immediately obvious observation was that any multithreaded benchmark was almost always slower (or similar at best) than the base sequential tree. This did not match the data we collected from local benchmarking, where we observed that the latch crabbing tree could usually beat the base tree pretty handily. We are unsure what the exact cause of this could be. Some of the confounding variables are:

- Lack of a native Go runtime on PSC
  - This could be an issue. It is possible that running the code natively might have produced better outputs. We had to make guesses about some environment variables (the main guess being the maximum number of processors to compile with - we chose 128), and a native go installation might have had additional and parameters defined that would lead to a better-performing binary.
- x86 architecture rather than ARM
  - It's unclear what impact this would have, if any. But it is an obvious difference in how exactly the code is run.

Conclusion

Overall, this was an interesting project that allowed us to better understand B+ trees, lock-free programming, and performance profiling. All of our projects in this class have provided the benchmarking infrastructure for us, which allowed us to simply focus on building the actual parallel projects and easily calculate the speedup. We spent a lot of time in the beginning of this project simply setting up all of the infrastructure we needed, so that we could easily change parameters like thread counts and key counts and observe performance differences. In hindsight, we might have been better served using Go's native benchmarking infrastructure, but we felt that it was too large of a learning curve to be able to properly use for our project.

We also feel that the project demonstrated the tradeoff that must be made between usability and understandability, and performance when building lock-free code. We found it much harder to build our lock-free B+ tree than any of the others that came before, and we also found it much easier to debug our lock-based implementations than the lock-free version. In the end, we were also disappointed by the lack of performance improvement we observed with the lock-free tree. We definitely believe that latch crabbing is the optimal way to support concurrent operations in B+ trees, because

it is a light enough modification to the original implementation that it maintains understandability and usability while also allowing for improved performance.

References

1. Bapi Chatterjee, Nhan Nguyen, & Philippas Tsigas. (2014). Efficient Lock-free Binary Search Trees.

2. Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. Proc. VLDB Endow. 4, 11 (August 2011), 795–806. https://doi.org/10.14778/3402707.3402719

3. L. G. Valiant. A bridging model for parallel computation. Commun. ACM,

4. Mirsoleimani, S. Ali, et al."A Lock-Free Algorithm for Parallel MCTS - Leiden University." A Lock-Free Algorithm for Parallel MCTS, Leiden Centre of Data Science, Leiden University, liacs.leidenuniv.nl/ plaata1/papers/paper_ICAART18.pdf.

5. Pavlo, Andy, and Jignesh Patel. "Lecture #08: Tree Indexes" Carnegie Mellon University, 15445, In https://15445.courses.cs.cmu.edu/fall2023/slides/08-trees.pdf, slide 12, 2023.

6. S.Kumar,D.Kim,M.Smelyanskiy,Y.-K.Chen,C.J.Hughes,C.Kim,etal. Atomic vector operations on cmps. In ISCA, pages 441–452, 2008.

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT:

The credit should be split fairly 50%/50%. The way we split up our work was pretty even, though it did deviate from the plan listed on our website a little. Dhruti was responsible for the implementation and test of both Stage 1 and Stage 2 in the code, and Adhvik was responsible for the same with Stage 3 and Stage 4. In terms of paper construction, Dhruti was responsible for the Abstract and Background sections, while Adhvik was responsible for the Results section. Both parties worked on the Approach section. Both parties also worked on constructing a robust test/benchmarking suite.