418 Final Report

Project Title

A Lock Free B+ Tree Implementation (Chenrui Shao and Cassidy Bolio)

URL

https://cbolio.github.io/15-418-Lock-Free-B-Plus-Tree/

Summary

We implemented a B+ tree data structure in C++ that supports lock free search, insertion, and deletion. We completed two different approaches to the structure, one using a chunk-based approach with state indicators and another using a batch-based approach (PALM trees). We analyzed the performance of these implementations compared to a basic sequential implementation and a public lock-based implementation on the GHC Intel Xeon 8-core machines. Our analysis looked at the difference between performance metrics of general benchmark cases, as well as search-dominant cases.

Background

We parallelized a basic B+ tree data structure. The encompassing data structure is the B+ tree itself. The public operations on the tree are the aforementioned search, insert, and delete. Additionally, the tree needs to maintain its balance for strong performance, so there are also split and merge functions that separate and combine nodes as needed during insertions and deletions. The tree structure contains inner-nodes/chunks which are their own structures. Each node/chunk contains an array of key-data pairs, where the key is a long and the data field is a pointer to the desired original data; in the chunk

approach, each pair entry also holds a pointer to another entry to simulate a linked list within each chunk. The nodes/chunks have their own smaller-scale search, insert, and delete operations; these are used to implement the overarching functions for the tree.

B+ trees are designed to be used on large amounts of data, often within database systems. As they need to handle a large number of queries, managing individual operations sequentially is far too slow for the real-life use cases. There are specific operations, like search, that can be easily performed in parallel to improve the overall runtime. In addition, through lock-based or lock free approaches, we can find a way to more safely overlap insertion and deletion with other operations. As independent threads will be working with the same B+ tree, there are many dependencies to consider. While concurrent queries can be performed safely, any other combination of an operation with insert or delete will potentially involve conflicting accesses to a part of the tree.

There is a high potential for parallelism, especially in query-dominated situations, but it will need to be managed carefully to avoid corrupting the data structure. Batches of operations will be distributed among threads, similar to the usage of OpenMP, allowing multiple operations to be performed simultaneously. While locality is not a large consideration for this structure, there are considerations for locality in leaf-node traversal in the chunk-based approach. As this involves varying operations on a data structure, this application would not benefit from SIMD.

Approach

We implemented both versions of our B+ tree data structure in C++, including the standard atomic library (with its compare-exchange functions) and the standard barrier library (to implement barriers). While we did not target any machines in particular, we knew we would be benchmarking on the GHC 8-core machines, and our implementations were both designed to scale well in environments with a high number of cores. We divided groups of operations into batches for individual threads to handle. Both implementations used a barrier-like structure to have the threads join and wait in necessary locations.

The specific mapping details differed between the two implementations. The chunk-based approach divided the operations with no considerations for the nodes they depend upon. This approach used “freeze states” which designated the current lifecycle of the node. These states were modified using atomic compare-exchange and helped to maintain the uncorrupted tree structure. The PALM approach divided query operations into batches for individual threads to handle. Then, other batched operations were managed at distinct levels of the tree, using barriers to prevent unsafe accesses. The PALM approach mapped insert and delete operations to threads based on the node they modified within the tree. This ensured that changes to a specific node were only being performed and observed by a single thread before all threads proceed through the barrier. The original serial algorithm was very simple, so we did not need to change it - rather, the two parallel implementations were so complex that the original serial algorithm is almost unrecognizable with the addition of concurrency features/structures.

The first approach that we implemented was the chunk-based approach. This implementation followed the 2012 paper *A Lock-Free B+ Tree* by A. Braginsky and E. Petrank, which was a C-focused design. The efficiency of this paper relied heavily on four details: packing data structures, chunk linked lists, master-slave relationships within the children of every node, and node freeze states. This implementation was designed under the assumption of a 64-bit cache line, as well as all addresses having zeros for the three least significant bits. Similar to malloc lab from 15-213, the paper used the least significant bits of pointers to pack flag bits, allowing entries and chunks to fit cleanly on limited cache lines. The original paper also only used 32-bit keys and assumes a base pointer to all data, storing a 32-bit offset value in the entry which can be combined with the base pointer to access the data.

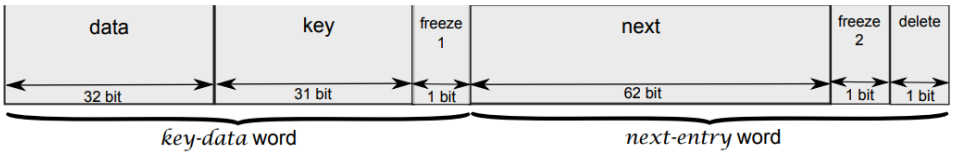

Figure 1: Entry data structure from original paper. Notice the 32-bit data pointer and LSB flags, allowing each entry to fit on exactly two cache lines.

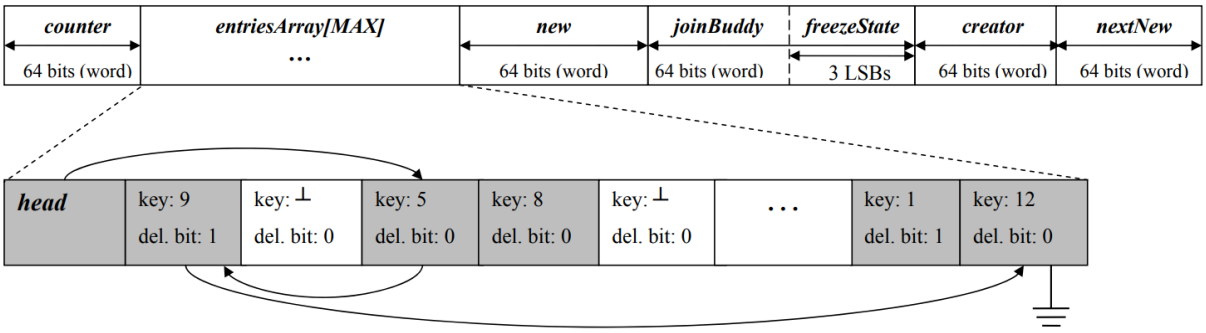

Figure 2: Chunk data structure from original paper.

We attempted to implement similar structures for our implementation. However, after running additional testing checking if the least significant bits were indeed zero, we found that we needed to use the C++ alignas keyword to ensure that the entries and nodes were 64-bit aligned if we wanted to use this packing approach. We also used 64-bit data pointers so that the original data did not need to be in a single contiguous array, which is how the base pointer approach would have worked. As a result, our entry data structure was three cache lines wide, rather than two: we used a 32-bit key, two 8-bit freeze flags, a 16-bit delete flag, a 64-bit data pointer, and another 64-bit next pointer. For the chunk structure, we made all chunks 64-bit aligned, so we could fit the freeze state into the LSBs. This was important as atomic compare-exchange was performed on those fields, and we were unable to find a reliable compare-exchange for any field larger than 64 bits.

The next implementation detail utilized by the paper was having entries within each array form a linked list. All entries are stored in an underlying array within the chunks for good locality. However, to prevent expensive reordering when inserting or deleting entries, the “order” of the entries is maintained as a linked list. This is demonstrated by the arrows within the entry array for the Figure 2 chunk. One key aspect of B+ trees is that all leaf nodes can be scanned to view all data without going through the upper-level nodes of the tree. In this implementation, this is enforced as chunks themselves are a linked list to each other. Then, to review all data, the structure can begin with the leftmost chunk, travel through its linked list of entries, then follow the pointer to the next

chunk, and proceed through its linked list of entries. The linked lists preserve strictly increasing ordering of keys, both within and across chunks. While this does add additional space requirements to hold the pointers, we found that this was far more efficient for insertion and deletion operations, especially due to the splitting and merging of nodes in some of those cases.

The third implementation detail was the maintenance of master-slave relationships within children of each node. For every node, each of its children had a designated “slave” node that was another child which was adjacent to it - in the chunk structure, this is the joinBuddy field. When a child node had too few entries and needed to be merged with another node, it was merged with its slave node. If two nodes were attempting to claim each other as slaves for a merge, the leftmost node (smaller keys) would be designated the slave. Additionally, if the slave node had too many entries for the merge to fit entirely in one new node, two new nodes were created that borrow keys from the master and slave, then the original master and slave were deleted. An important distinction was that master-slave nodes always have the same parent node. Freeze states included states to indicate when a node has claimed another as its slave; when a parent node was attempting to split into two nodes, it checked which of its children were in a master-slave relationship, ensuring that those pairs were distributed to the same split node. This maintained consistency for the tree structure, as we did not want chunks with different parents to merge. When chunks with the same parents merged, only the sole parent needed to be placed into a temporary freeze state.

The final and most critical implementation detail was the usage of freeze states. Freeze states indicated what lifecycle stage chunks/nodes were in, allowing operations to be built around those expectations. There were eight freeze states used in the paper: NORMAL, INFANT, FREEZE, COPY, SPLIT, SLAVE_FREEZE, REQUEST_SLAVE, and JOIN. Figure 3 below demonstrates the relationship between the different states.

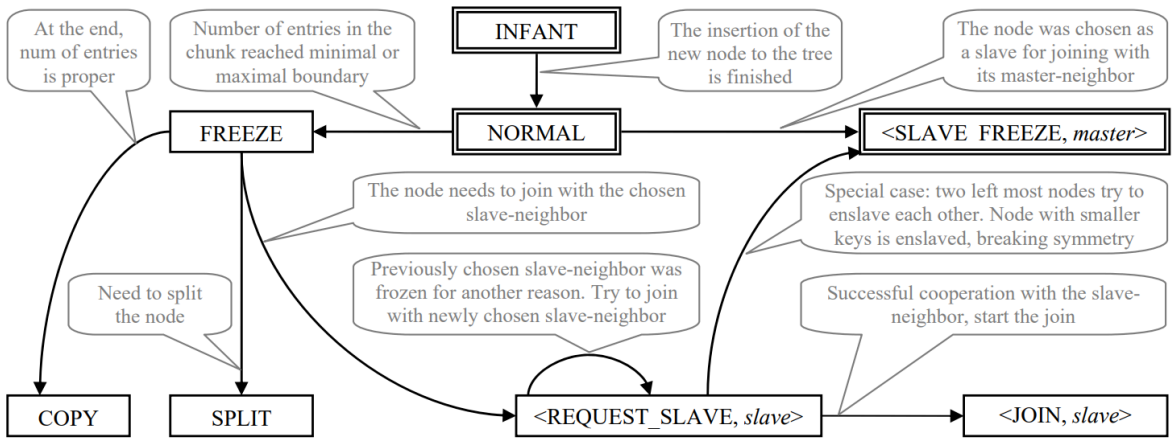

Figure 3: State diagram from original diagram showing state relationships.

New inserted nodes were given an INFANT freeze state; when a node triggered a merge or split, one or two new nodes were always created, rather than reusing the original node, and the original node was moved into the FREEZE state. Until the new nodes were moved from their INFANT state to the NORMAL state, their insertion was not complete, and there was no guarantee that all data had been moved from the original node into the new ones. Thus, search operations looked through both INFANT and FREEZE nodes to see all data. Insertion, deletion, split, and merge were not allowed on INFANT nodes, so nodes that wanted to perform those operations had to

execute a helpInfant function that finished moving the data to that node and changed its state to NORMAL. Nodes in the NORMAL state were able to have all operations performed on them. When a node had too few or too many entries, it was switched to a FREEZE state; when in a FREEZE state, a node could not be split or merged. Its entries were marked as frozen one by one; while frozen entries could not be modified, insertions and deletions could still be performed for unfrozen entries by other threads. Once all entries were frozen, the chunk could not be modified at all and would never be used again outside of searches.

Then, there was a reevaluation based on the current entries, as other threads may have inserted or deleted entries between the initial freezing and the final freezing. If the number of entries was now within the acceptable limit, the node was set to COPY, indicating that the data needed to be copied into one new node. If the number of entries was above the maximum entries per node, the node was set to SPLIT, indicating that the data needed to be divided between two new nodes. If the number of entries was below the minimum entries per node, the node was set to REQUEST_SLAVE, indicating that the data will need to be combined with a slave's data into a new node. In all cases, the original node's thread created new node(s) in INFANT stages with the original node as the creator. For the copy and split cases, the original thread worked on moving the data to the required new nodes. If other threads wanted to insert or delete entries from the new nodes, they would see the INFANT state, and helpInfant would be triggered where they assisted the original thread in transferring the data. Once the data transfer was completed, the original thread would change the new node(s) to NORMAL, and

delete the old node permanently.

The merge case was more complicated due to needing to ensure the slave node had not been frozen by other threads' operations. The original node started in a REQUEST_SLAVE state, where the original thread needed to check if the current slave was still available to be merged with - if it had been frozen by another thread, the original thread needed to find a new slave node to merge with. When a new non-frozen slave was found and its state was set to SLAVE_FREEZE, the original node usually moved into the JOIN state. The only exception was the aforementioned circular master-slave relationship when two nodes were trying to claim each other as their slaves. If the original node was the leftmost (lower keys) node, it would move into the SLAVE_FREEZE state, where no modification was allowed on the node; this was different from the regular FREEZE state, where insertion and deletion was allowed for unfrozen individual entries. The original thread of the JOIN node transferred its data to the new INFANT node it created. It also copied over the data from the slave node which had been completely frozen.

The ability to temporarily freeze entries and permanently freeze nodes allowed concurrent execution of potentially conflicting operations. Search operations were allowed to execute uninterrupted, greatly improving the runtime for search-dominant situations. Insertion and deletion was limited to nodes not being modified by other threads through the checking of FREEZE or other subsequent stages. The case where a search is unable to find a key when the entries are being transferred between two

nodes was prevented by not deleting the old frozen node until the data transfer was complete. Overall, the usage of states was very effective in maintaining an uncorrupted tree structure. However, the frequent checking and maintenance of states was extremely complex. Even with detailed pseudocode in the original paper, it took multiple weeks to implement a working version of the chunk-based version. Additionally, any attempt at optimization, even small changes, introduced a variety of issues for tree consistency. While the freeze states were a powerful framework, they made it very difficult to expand the implementation outside of what was designed in the paper.

The second approach we implemented was based on the literature "*PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors*." This approach stores the key and the pointers to the children of a node in C++ vectors, allowing the implementation to effectively search for the neighbors of a node. The number of children in a node and the depth of the node is also stored in the PALM approach. In this implementation, we assumed a 64-bit key will be used in the data structure and a 64-bit void pointer will be used as the key value. This key value would be the pointer to a child node for the internal nodes of the B+ tree, and would be the key value of the given key for the leaf nodes of the B+ tree.

The B+ tree data structure implemented by the PALM approach supports sequential insertion, deletion, and searching operations with simple code similar to that of a sequential implementation of the B+ tree. The B+ tree data structure implemented by the PALM approach also supports parallel execution of a mixture of insertion, deletion,

and searching operations effectively, without using latches or locks.

The implementation of sequential operations for the PALM B+ tree is trivial, nearly identical to that of a regular B+ tree with the need to maintain the depth and size of each node. The implementation of parallel operations for the PALM B+ tree relies on the delay of node splitting, rebalancing, and merging operations that is required to maintain the size of each tree node to be within a designated bound. In sequential implementation, the size of a modified node will be checked immediately after the insertion or deletion operation. A splitting operation will be performed if the node size is greater than the node capacity, and similarly a rebalancing operation will be performed if the node size is less than one half of the capacity. However, performing either a splitting operation, a rebalancing operation, or a merging operation involves modifying the node, at most one of its immediate neighbors, and its parent node. This is the source of conflicting operations, involving two types of conflicts: A node can be updated by its neighbor while it is also updating itself, or two nodes updating a shared parent node.

To prevent the conflicting operations from happening, the PALM implementation used a few designs to perform the query requests in a parallelizable, conflict-free way without using a lock to prevent conflicting operations from occurring. PALM only attempts to perform tree queries when they are treated as a batch with sequence. Our implementation assumes that it is supplied all together at the same time, but in theory the queries can be supplied one by one, as long as it is allowed to have a gap between two batches of queries during which node splitting, rebalancing, and merging can occur.

The process of performing one batch update in a PALM B+ tree is shown in Figure 4.

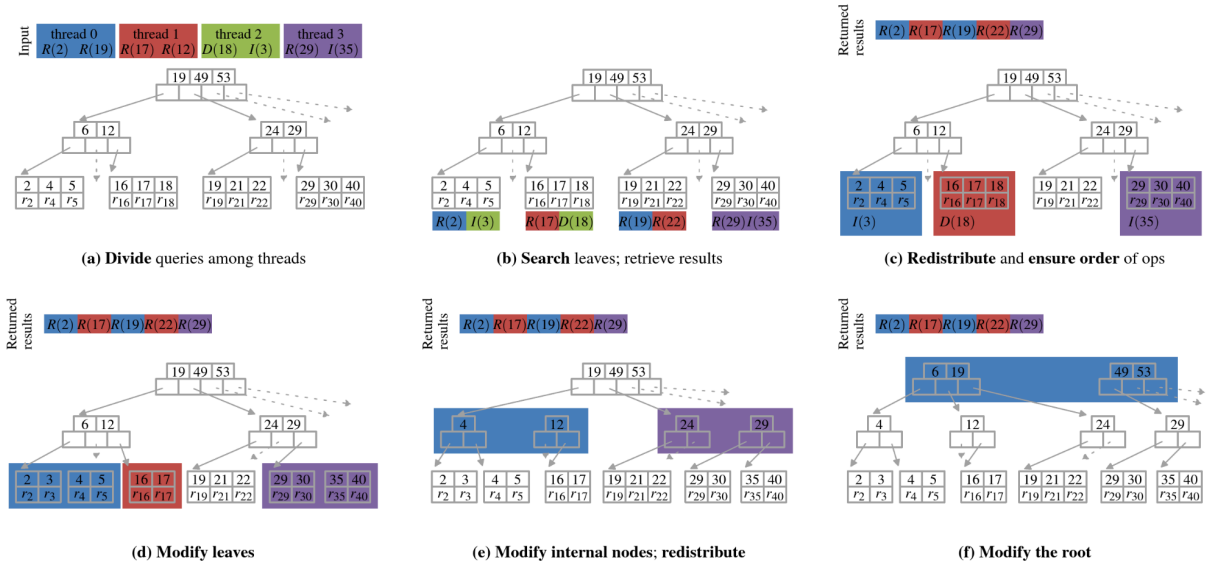

Figure 4: A sample diagram of the PALM B+ tree updating process

The PALM implementation first attempts to perform all queries without committing any changes to the tree; this step is labeled as “divide” and “search” in Figure 4. In this step, the leaf node that each tree query will operate on will be recorded in an ordered hash table. The hash table is indexed using a pointer to the leaf node. Because in these steps, the B+ tree will not be updated, all queries can be performed in parallel.

The workloads are distributed to threads using a hash table. The order of the query information added to the hash table here is being preserved. Therefore, after the query information for all queries were added with the leaf node pointer as the key, the list of query information for each leaf node is already in the order of the query. Then, each

thread can subsequently take the list associated with a few leaf nodes and the sequential order of the operations performed on each individual node are preserved. This step is labeled as “redistribute and ensure order”.

After all queries have been indexed to a leaf node, the tree will be able to update all leaf nodes in parallel, as long as the queries on each individual leaf node are committed in the correct order. The leaf nodes will not attempt to perform any splitting, rebalancing, or merging operations, instead each thread would check the number of children in a node after all updates are finished, and if any operation is needed it adds the leaf node to a set. Thus, each thread will always perform updates on separated leaf nodes. This step is labeled as “modify leaves” in Figure 4.

Once all leaf nodes are updated, the PALM B+ tree will perform updates on their parents in parallel. These nodes to be updated will have the same depth, starting with the nodes that have a depth of 1. Despite the possible conflicts, as the rebalancing and merging operations will not update neighbors that has a different parent due to the complexity of doing so and we instead will rebalance or merge with its other neighbor, we can guarantee that performing splitting, rebalancing, or merging on any node can only possibly update its parent and the children of its parent. Therefore, we can update all nodes with the same depth in parallel, each thread will take all nodes that need updating, at the same depth, and with the same parent node. This step is labeled as “modify internal nodes; redistribute” in Figure 4.

The update for all nodes that is allocated to the same thread is performed sequentially, and after all splitting, rebalancing, and merging operation of each thread is finished, the thread will check if the parent nodes allocated to it also need updates and add its parent to a set if either splitting, rebalancing, or merging operations is needed for the parent node. The process of updating all nodes with the same depth and proceeding to their parent nodes will be repeated until the root of the tree is reached, or the nodes in the next level that need either splitting, rebalancing, or merging operations do not exist. Thus, the “modify the root” step in Figure 4 may not be performed if there is no node that needs to be updated in a level.

Results

We constructed four basic benchmark cases to use on the various implementations for comparison. The number of operations generally coincides with the tree size. The first three benchmarks have a ratio of roughly 30% insertions, 30% deletions, and 40% searches with a varying number of operations: 1,000 operations for the small benchmark, 10,000 operations for the medium benchmark, and 100,000 operations for the large benchmark. The fourth benchmark has the same number of operations as the medium benchmark (10,000), but its ratio of operations is heavily search-dominated with roughly 5% insertions, 5% deletions, and 90% searches. We ran the sequential implementation, fine-grained lock-based implementation, and chunk-based implementation on 1, 2, 4, and 8 threads on the GHC machines. Full benchmarking data and results can be found at Final Project Benchmarking. The main benchmarks that

we focused on comparing were the medium benchmark and the search-dominant benchmark due to their matching sizes. Unfortunately, the PALM-based approach was not completed in time for benchmarking and submitting the paper, so there are no results for this implementation.

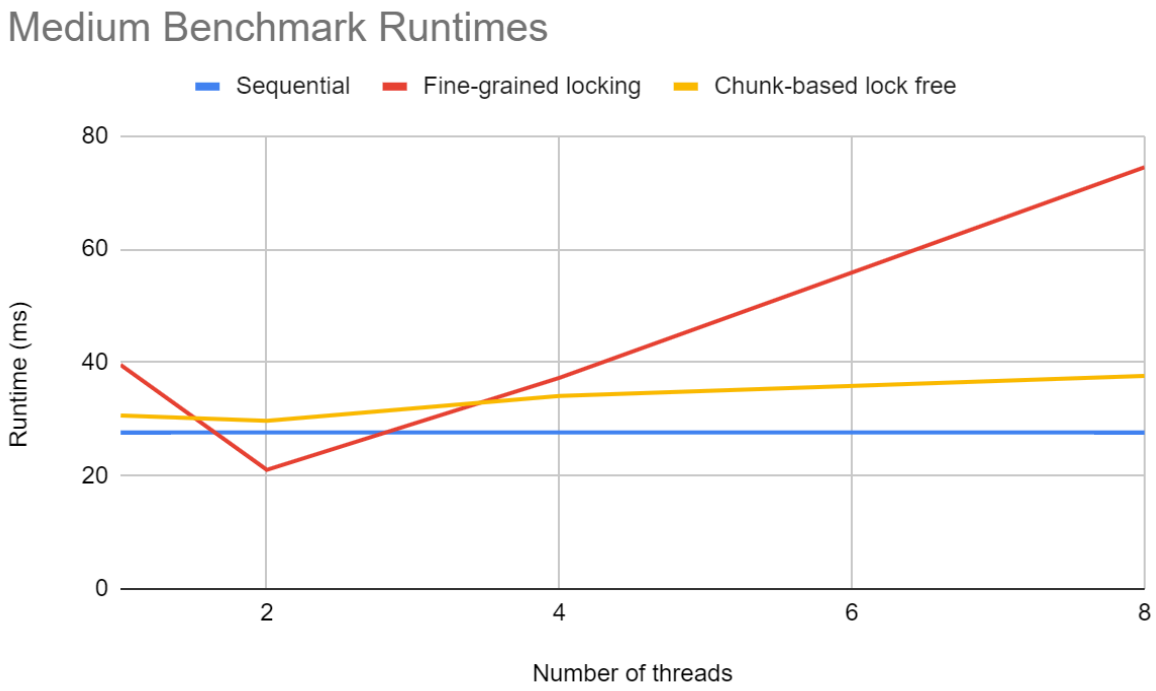

Figure 5: Runtime results in ms for the medium benchmark.

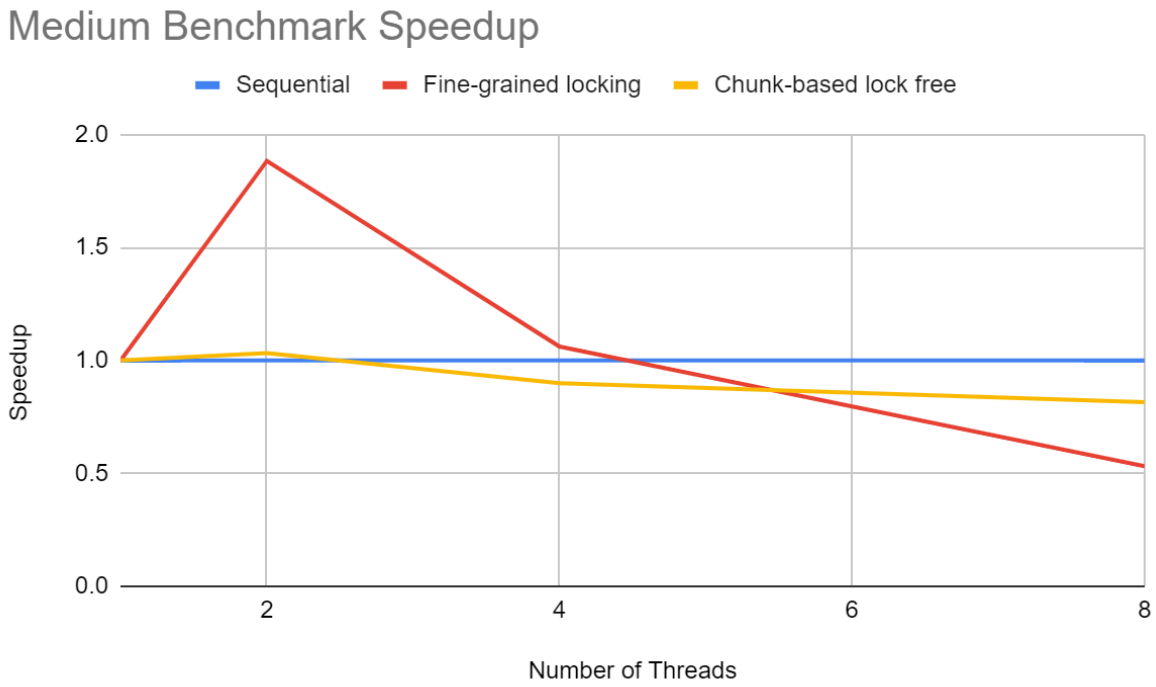

Figure 6: Speedup for medium benchmark.

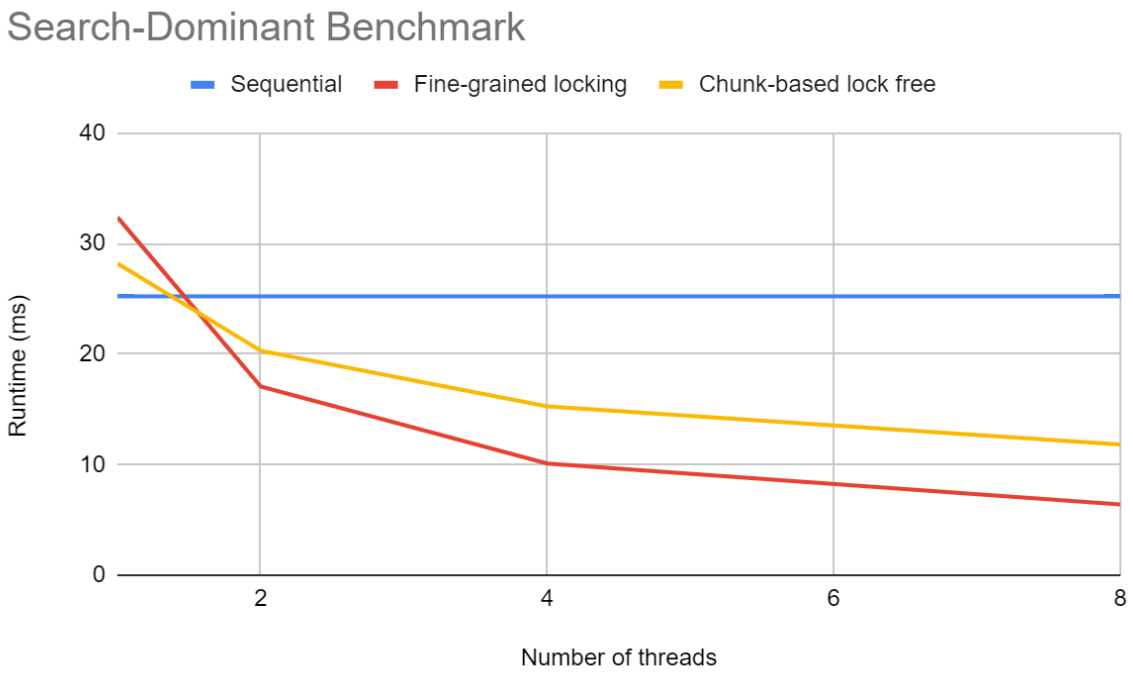

Figure 7: Runtime results in ms for search-dominant benchmark.

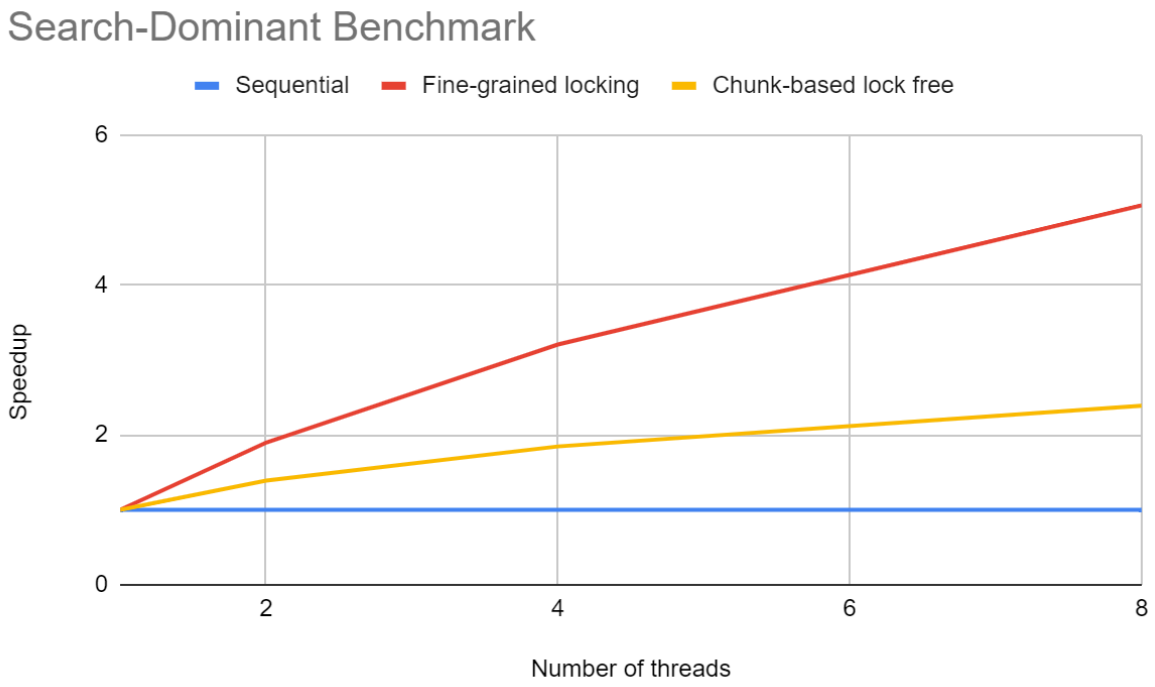

Figure 8: Speedup for search-dominant benchmark.

There is a strong difference between the performance of parallel implementations in a varied operation benchmark and a search-dominant benchmark. In the regular medium benchmark, we can see that outside of the fine-grained locking implementation on 2 threads, the sequential code performs the best. As the thread count increases, the fine-grained locking version decreases dramatically in performance. Our chunk-based lock free implementation does not outperform the sequential implementation on any thread counts, but its performance does not drop as significantly as the lock-based implementation does. However, the overhead costs of maintaining the node states and relationships appears to outweigh the benefits of having multiple threads working at once. Additionally, having more threads increases the risk of unnecessary node copying due to later modifications by other threads; as creating and transferring data to new

nodes is very expensive, this is likely the largest contributor to the lack of speedup.

The search-dominant benchmark is closer to what the usage might look like in a largely static database, where querying information is far more common than inserting or deleting data. In this case, we do see benefits from increasing thread counts in both the fine-grained locking implementation and our chunk-based lock free implementation. Unfortunately, our chunk-based version is not as effective as the fine-grained locking implementation in terms of runtime or speedup. Our chunk-based version does have less added overhead on one thread, but it does not even reach 2.5x speedup on 8 threads. However, it does not perform worse than the sequential implementation, like it did in the regular medium benchmark. While the overhead of maintaining states still seems to prevent significant speedup, there is no longer a drain from additional node copies that occur from having multiple threads insert/delete in a manner that triggers a FREEZE rebalancing, only for the entries to end up actually being balanced and all data is copied into a new node with no merging or splitting occurring.

Our hypothesis that the main source of overhead was the creation of new nodes does appear to be supported by further analysis. As seen in Figure 9 below, around 60% of the runtime for the medium benchmark is devoted to creating new nodes (initChunk), despite insertions only representing 30% of the operations. In comparison, 40% of the operations are searches, but only 9% of the runtime is spent on fulfilling queries. This also aligns with the results we saw from decreasing the ratio of insertions and increasing the ratio of searches: while still not comparable with the fine-grained locking

approach, there is no longer such an overhead of node creation that it entirely outweighs the benefits of parallelism. On the search-dominant benchmark, only around 30% of the time was spent creating new nodes, allowing the performance to beat the sequential implementation. However, 30% of time being spent in node creation when only 5% of the operations were insertions suggests that unnecessary nodes are being created from the COPY state.

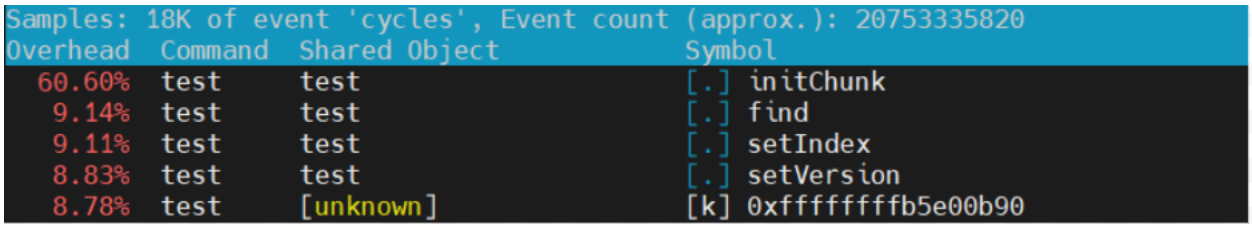

Figure 9: perf results of recording cycles on the regular medium benchmark for the chunk-based lock free implementation on 8 threads.

We attempted to improve the performance by removing additional node creation in the case of the COPY state. When this state occurs, no insertions, deletions, splitting, or merging are happening. The COPY node's data is transferred into a new node, which then replaces it. This state is reached when other threads insert or delete entries from the original node between its assignment to the FREEZE state and its full freezing of its entries, causing its entry count to fall back within the allowed bounds. However, we were unable to successfully remove or even limit the COPY occurrences without creating a large number of inconsistencies in our tree structure. It is imperative for maintaining tree structure consistency that any node that enters a FREEZE state is eventually deleted; allowing it to un-FREEZE in the COPY case causes other threads insertions and deletions to accidentally occur on the incorrect node. While the

complexity of freeze states allows for searches, insertions, and deletions to be performed more concurrently than in lock-based implementations, as seen in the lack of dramatic drop in performance on the regular benchmark, they make it extremely difficult to optimize or modify the code in any significant way without damaging the integrity of the structure and node relationships.

The PALM implementation does not contain a complicated mechanism to keep track of or update the states of tree nodes. This makes it more adaptable to different types of keys and values and can be more easily modified to incorporate other attributes in the tree data structure.

However, the PALM implementation also has a few drawbacks and implementation difficulties. The main drawback of the PALM implementation is it must delay all splitting, rebalancing, and merging operations to a designated operation stage, during which no new queries can be performed as the B+ tree is being updated without using a fine grained lock, so that it is uncertain whether the path of a query is being modified at this stage. This also led the PALM implementation to generate a different B+ tree than the sequential version of the B+ tree, as the two implementations split, rebalance, and merge the tree in different manner. The difference does not mean that the PALM implementation may violate the definition of B+ tree, but it makes debugging and testing the PALM implementation more difficult.

The PALM implementation also creates unconventional situations in the splitting and

rebalancing process. In a sequential implementation, splitting and rebalancing operations are done immediately after one insertion or deletion, this means the number of children in a node that needs to be split must be exactly one more than the upper limit, and the number of children in a node that needs to be rebalanced must be exactly one less than the lower limit. Thus, splitting the node is straightforward: The node only needs to be splitted in the middle and no further splitting might be needed. However, in PALM implementation, it is possible to overfill the node to over double the maximum capacity. While splitting the node in half is still feasible and would not create a broken tree, this is in violation of the B+ tree definition, as after the splitting in the middle, both nodes will still contain more children than is allowed. This means the splitting must be done in a significantly different way for the B+ tree definition to not be violated.

Similarly, the rebalancing is much more difficult to implement. In a sequential implementation, a rebalancing step will only involve moving one child from one of the neighboring nodes, and if it is impossible a merge must be possible. However, in the PALM implementation, a rebalancing step may move more than one child from the neighboring nodes of a node. This also makes it more difficult to implement and test. In our PALM implementation, this issue was found very late when we are preparing the code for benchmarks. While it does not make the insertion, deletion, and searching operations incorrect, it may make the B+ tree in violation of the B+ tree definition temporarily.

Apart from the difficulty in implementing the splitting and rebalancing process of the B+

tree, the PALM implementation is also not trivially parallel. A hash table is used in the searching and redistribution stage. The hash table is not a lock free data structure. If it is needed to perform the search step in parallel, the leaf node associated with each query cannot be inserted into the hash table directly. Instead, it must be kept in a set or a vector, and the process of building the hash table is inherently sequential due to the need of preserving the operation orders. Similarly, we also need to keep track of the nodes that need to be splitted, rebalanced, or merged in sets. If we are performing this step in parallel, we need to merge sets that are being created by each thread in the update of each level. This means the order of which the nodes are updated in each level is not deterministic. The non-deterministic order will not impact the correctness of the B+ tree, but it makes the debugging more complicated and creates additional overhead for set construction and merging.

Despite there being no time for us to improve the PALM B+ tree, we did identify a few points in the PALM implementation that can be improved. As the PALM approach already allows partial violation of the B+ tree definition between the two splitting / rebalancing / merging stages, it may be possible to implement the PALM in a way such that the queries can be separately launched and placed in a queue, the PALM would update the B+ tree with partial violation of the B+ tree until it finds a good time to perform splitting / rebalancing / merging operations, allowing the queries to be submitted not in a batch. However, this means that while the leaf nodes are being modified, as the parent nodes are not being modified until the splitting / rebalancing / merging stage, we need to keep a small cache for the committed operations, so that in

case of a previous operation is not visible in the tree due to partial violation of tree structures, it can still be performed correctly. However, maintaining a cache like this will likely reduce the performance significantly as we need to maintain additional structure for searching through the recent transactions, and this data structure is inherently sequential.

Another point for improvement is the implementation can treat the search queries in the same manner as insertions and deletions. In the original PALM implementation, they perform queries first when we are looking for the leaf nodes associated with each query. However this makes the query implementation extremely complicated, as we must find a way to keep track of the changes committed in the current query batch that may influence the searching results, and alter the searching results in the process of performing insertions and deletions, while preserving the order of all operations. In our implementation, we are treating the search queries in the same manner as insertion and deletion queries. Now the search queries will be performed in the correct order with the insertion and deletion orders. This mechanism means the queries results will always be correct, as updates on a single leaf node are processed in sequential order. In addition, it greatly simplifies the data structure by removing the need to look back into and modify the searching operations when we are processing insertions and deletions. This approach might also be more efficient than the PALM implementation when most of the operations are insertions and deletions. However, the PALM implementation will likely outperform our implementation if most of the queries are searching operations only.

References

We used the following papers to guide our implementations:

Chunk-based implementation

A. Braginsky and E. Petrank, *Locality-Conscious Lock-Free Linked Lists*.

https://www.researchgate.net/publication/220725315_Locality-Conscious_Lock-Free_Linked_Lists

A. Braginsky and E. Petrank, *A Lock-Free B+ Tree.*

https://dl.acm.org/doi/abs/10.1145/2312005.2312016

PALM implementation

J. Sewall, *PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors*. https://www.vldb.org/pvldb/vol4/p795-sewall.pdf

We used this public lock-based implementation by past CMU students Runshen Zhu and Ran Xian, who wrote a fine-grained locking implementation for their final project of 15-618 in 2016, for our performance comparisons, located at https://github.com/runshenzhu/palmtree/tree/master/fineTree.

Work Distribution

Chenrui: 50%

- Project proposal
- Sequential B+ tree implementation
- PALM-based lock free implementation
- PALM-based implementation benchmarking and analysis

- PALM-based implementation approach and results discussion

Cassidy: 50%

- Website creation and maintenance
- Milestone report
- Chunk-based lock free implementation
- Sequential, lock-based, and chunk-based benchmarking
- Non-PALM sections of final report