15418 Final Project Report
B^{+} Forest - A comprehensive comparison of Parallel and Distributed B^{+} Trees

Yutian Chen, Yumeng Liu
{yutianch,yumengli}@andrew.cmu.edu
Carnegie Mellon University

https://markchenyutian.github.io/15-418-Final-Project/

December 16, 2023

1 Abstract

The B+ Forest is an in-depth exploration and comparative study of various B+ tree architectures, a pivotal element in the realm of database systems. Our investigation encompasses the development and performance evaluation of five distinct B+ tree variants, namely: the Sequential B+ Tree, Coarse-Grained B+ Tree, Fine-Grained B+ Tree, Latch-Free B+ Tree, and Distributed B+ Tree, each tailored for specific computational frameworks from single-threaded applications to distributed environments. These structures are all capable of executing `insert`, `delete`, and `get` functions.

Starting with the Sequential B+ Tree as the baseline, the subsequent trees, with the exception of the Latch-Free B+ Tree which adopts a unique approach, evolve from this foundational model. The Coarse-Grained and Fine-Grained Trees enhance the base with lock mechanisms and multithreading capabilities, respectively. The Latch-Free Tree, employing the PALM algorithm, provides a latch-free alternative conducive to environments demanding high concurrency. The Distributed Tree utilizes the OpenMPI interface to facilitate parallel processing across nodes or processes within high-performance computing contexts.

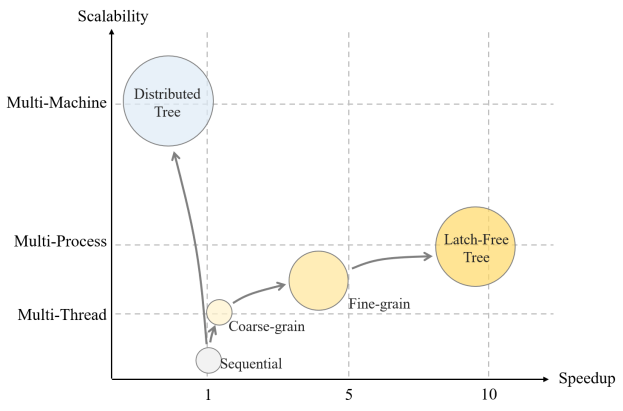


Figure 1. 5 Different Types of B+ Trees studied by the B+ Forest Project and their qualitative throughput / scalability

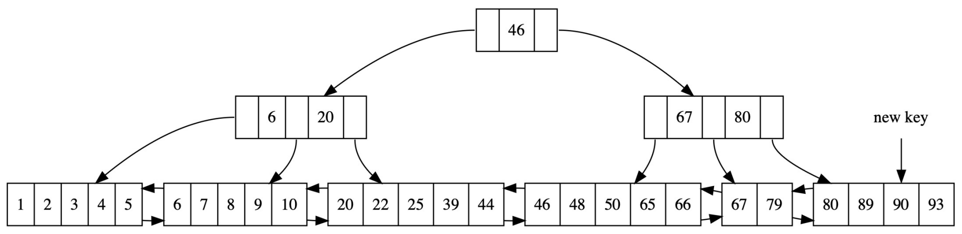


Figure 2. An exemplary B+ Tree with order of 5

In our extensive benchmarking, we assess the performance spectrum of these B+ tree variants under a range of operational conditions to draw conclusions about their relative efficiencies and optimal application scenarios.

Figure 1 qualitatively depicts the five B+ Trees that the B+ Forest Project examines, showcasing their respective scalability and speed enhancements.

2 Background

2.1 B+ Tree

The B+ tree is a type of self-balancing tree that keeps data sorted and supports searches, insertions, deletions, and sequential access operations in logarithmic time complexity. It is a variant of the binary search tree optimized for storage systems. Due to its inherent balance and proficiency in facilitating range queries, the B+ tree is a common choice for indexing in relational database management systems. Within this structure, internal nodes house key ranges that direct to subtrees populated with data, while leaf nodes exclusively hold key/value pairs. (Wikipedia, 2023)

B+ tree is characterized by its order, which dictates the acceptable number of keys and children within its nodes. Every non-root node must have keys numbering in range $[(order - 1)/2, order - 1]$. For root nodes, the maximum number of keys is $order - 1$. The number of child pointers in an internal node always exceeds the number of its keys by one. All keys in the i-th child of an internal node are less than the internal node's i-th key, and if there is a preceding key, they are at least as great as the $i - 1$-th key.

The keys within any node are sorted, and at the same depth, nodes are connected through `prev` and `next` pointers to form a double linked list that enables sequential traversal. Additionally, there is a two-way relationship between nodes: children have references to their parents, and parents have pointers to their children.

Figure 2 is an illustrative B+ Tree of order 5, dealing with integer keys with the values not shown.

2.2 Split, Merge and Borrow

Before moving to the detailed implementation and parallelism, we would first focus on the three auto-balancing operations in B+ tree - `split` , `borrow` , and `merge` .

Split - When a B+ tree node reaches capacity, it must undergo a `split` operation to maintain the balanced structure and efficient performance. This auto-balancing operation involves several steps (shown in figure 3):

1. *Median Selection* - The middle key-value pair within the overflowing node is chosen as the median. This element will become the root of a new B+ tree node sibling to the original node.

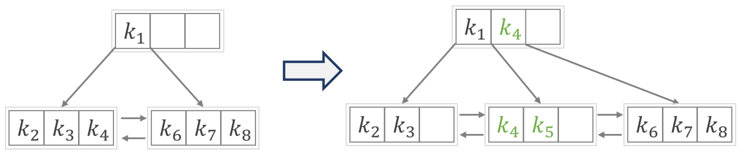


Figure 3. Illustrates how a "split" operation in the B+ tree

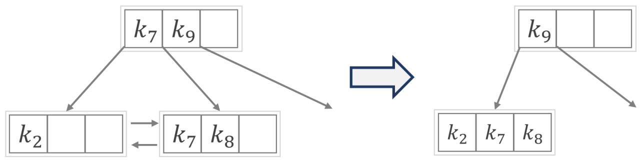


Figure 4. Illustrates how a "merge" operation in the B+ tree

2. *Key Distribution* - All keys less than the median are relocated to the original node, while all keys greater than or equal to the median are shifted to the newly created sibling node. Both nodes maintain the B+ tree structure with appropriate key ranges and minimum number of key-value pairs required.
3. *Parent Update* - The median key from the overflowing node is inserted into the parent node, potentially causing it to overflow as well. If the parent overflows, the split operation propagates recursively up the tree until a node with sufficient space to accommodate the new key is found.

This self-balancing mechanism ensures that B+ trees remain approximately balanced during insertion, resulting in $O(\log n)$ complexity for other operations. The efficient distribution of keys across nodes and minimal reorganization during splits contribute to the B+ tree's robust performance in real-world applications.

Merge - While B+ trees utilize splitting to maintain structural balance and efficient performance, the opposite scenario, a node falling below a minimum key-value threshold, also requires attention. This situation calls for a `merge` operation to consolidate resources and prevent performance degradation. Here's a breakdown of the merge process (shown in figure 4):

1. *Sibling Selection* - The B+ tree identifies a suitable sibling node with sufficient space to accommodate the keys from the underpopulated node. In our implementation, we will **always try to merge with the sibling in same subtree (share same direct parent) with current node**. Since this can minimize the impact to unrelated parts of the tree, thus reducing the risk of data racing and contention (in parallel implementation).
2. *Key redistribution* - All key-value pairs from the underpopulated node are transferred to the chosen sibling, maintaining the sorted order and proper key distribution within the B+ tree.
3. *Parent update* - The key separating the two nodes in the parent is removed, as the merged node now encompasses the combined key range. If the parent node falls below the minimum key count due to this removal, the merge operation may propagate upwards, potentially merging it with its own sibling in a cascading manner.

Borrow - The borrow operation in B+ trees offers a third solution for maintaining balance and efficient performance when dealing with nodes that fall below the minimum key threshold. Instead of merging entirely, a B+ tree can "borrow" a key from its immediate sibling to increase its own key count and avoid falling under the minimum requirement (fig 5). This technique offers several advantages:

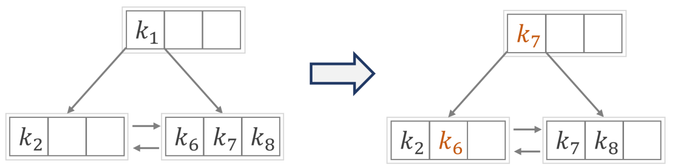


Figure 5. Illustrates how a "borrow" operation in the B+ tree

1. *Minimized disruption* - Unlike a complete merge, borrowing only involves transferring a single key-value pair, limiting the reorganization and potential performance impact on operations involving the affected nodes.
2. *Efficient redistribution* - Borrowing prioritizes transferring keys that maintain the sibling's balance while ensuring the underpopulated node reaches the minimum key count with minimal disruption to the overall key distribution.

The borrow operation, along with splits and merges, forms a crucial part of the self-balancing mechanism in B+ trees. By dynamically adjusting key distribution and node occupancy based on data volume and access patterns, B+ trees achieve consistent performance for a wide range of operations, making them a popular choice for storing and retrieving sorted data efficiently.

2.3 Parallelism with OpenMPI for Distributed B+ Tree

The distributed variant of the B+ Tree in our project utilizes OpenMPI, a prominent message-passing interface that adheres to the MPI standard, crucial for high-performance computing applications. OpenMPI allows for effective parallel processing across various nodes or processes through sophisticated communication techniques including both point-to-point and collective messaging. Its renowned for its performance, adaptability, and scalability, making OpenMPI a cornerstone for intricate computational tasks in distributed systems.

3 Design and Implementation

We implemented various B+ tree data structures for efficient storage and retrieval of sorted data, with a focus on concurrency control and performance in parallel environments. This section details the design choices and considerations for each B+ tree type.

3.1 Interface

To make the project's deliverable more user-friendly and maximize the reuse of infrastructure codes like benchmarking engine, test engine, all versions of B+ trees, except for the latch-free B+ tree, support the `ITree` public interface with three operations.

```
template <typename T>
class ITree {
    public:
        void insert(T key);
        void remove(T key);
        std::optional<T> get(T key);
};
```

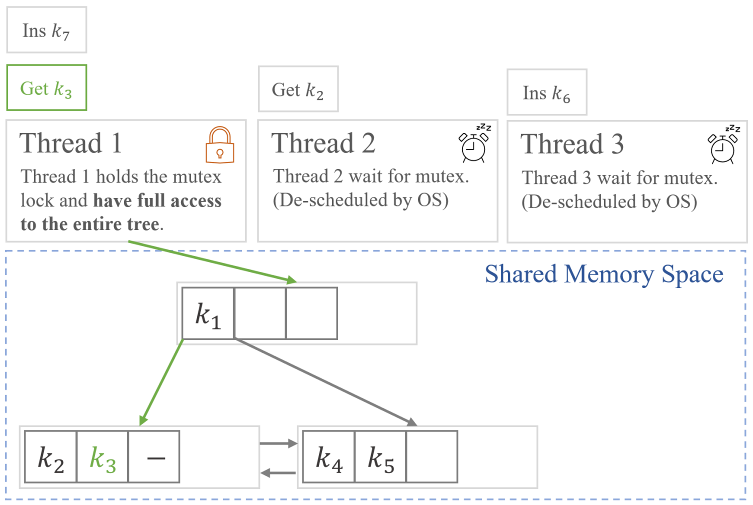

Ins k_7
Get k_3
Get k_2
Ins k_6
Thread 1
Thread 1 holds the mutex lock and **have full access to the entire tree**.
Thread 2
Thread 2 wait for mutex. (De-scheduled by OS)
Thread 3
Thread 3 wait for mutex. (De-scheduled by OS)
Shared Memory Space
k_1
k_2 k_3 –
k_4 k_5

Figure 6. Coarse-Grained B+ Tree

3.2 Sequential B+ Tree

The Sequential B+ tree serves as the foundation for all other implementations. It focuses on efficient operations without concurrent access considerations. This version provides a stable and predictable foundation for understanding the complexities of concurrency control mechanisms and performance optimizations employed in the other B+ tree implementations.

For the sequential B+ tree, all three operations, the `insert` , `remove` and `get` , are implemented in a very straight forward manner. The algorithm will first find the leaf node the provided key is (or should be) in, then perform operation on the leaf node's `key` vector. After leaf operation, balancing operations like `split` , `borrow` and `merge` will be triggered depending on the status of leaf node.

If the balancing of leaf node cause parent node to overfill or underflow, the balance operation will be called recursively until the root of tree is reached. If the root node is overfilled, the root node will split and the tree's height will increment by 1. If it underflows (have no key at all), the original root node will be removed and its only child will become the new root.

3.3 Coarse-Grained B+ Tree

The coarse-grained B+ tree employs a global mutex lock to synchronize the accesses to the tree. This simple approach ensures atomic execution of concurrent operations but hinders parallelism as only one thread can access the tree at a time. Each operation acquires the mutex lock before modifying the tree and releases it afterward. This design is easy to understand and implement but suffers from scalability limitations due to the single point of contention (as shown in figure 6).

3.4 Fine-grained B+ Tree

To improve parallelism, the fine-grained B+ tree utilizes reader-writer locks (implemented as `std::shared_mutex`) on each node (called a latch). Reader locks allow concurrent read-only operations on non-conflicting sub-trees, maximizing concurrency. For write operations, the tree acquires write locks along the access path from root to leaf only if the operation might trigger a split, borrow, or merge. This enables concurrent read and write access to different parts of the tree, boosting throughput compared to the coarse-grained approach.

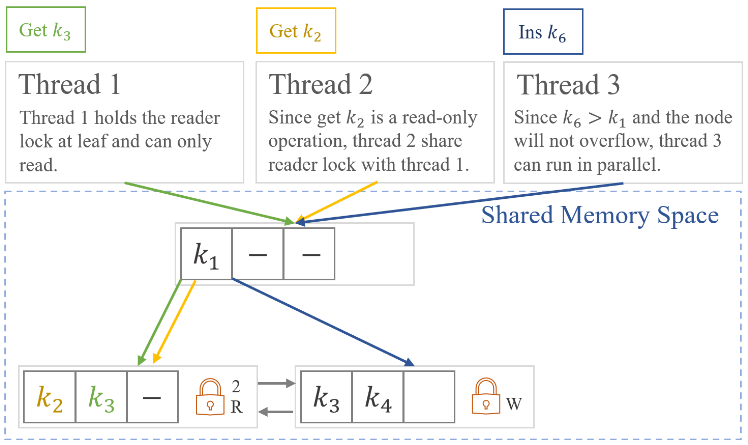

Get k_3
Get k_2
Ins k_6
Thread 1
Thread 1 holds the reader lock at leaf and can only read.
Thread 2
Since get k_2 is a read-only operation, thread 2 share reader lock with thread 1.
Thread 3
Since $k_6 > k_1$ and the node will not overflow, thread 3 can run in parallel.
Shared Memory Space
k_1
k_2 k_3
2 R
k_3 k_4
W

Figure 7. Fine-Grained B+ Tree can improve utilization by allowing concurrent read-only access and exclusive write access on certain subtree

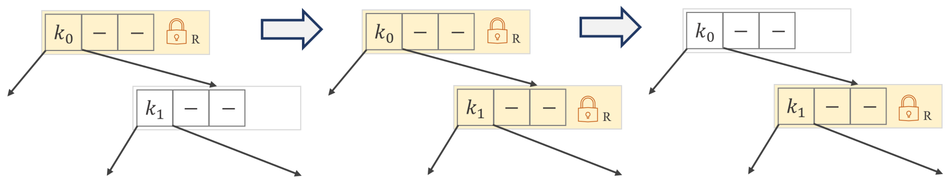


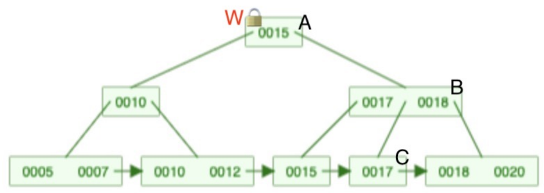
Figure 8. Hand-over-hand locking for `get` operations in fine-grained B+ tree

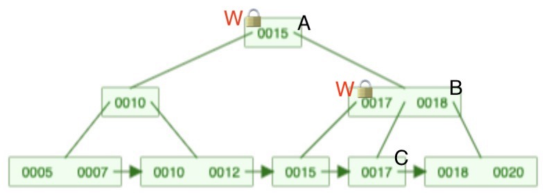
Below we will focus on the detailed lock-acquiring and releasing strategy adapted for each of the `get`, `insert`, and `remove` operation to maximize threshold under parallelism.

Get operation - The locking strategy employed in this operation ensures that only one node in the tree is locked by a single thread's `get` operation at any given time. Strategy begins by locking the root. As the function traverses down the tree towards the leaf node where the key might be located, it acquires a lock on the child node before releasing the lock on the current (parent) node. This lock acquire-release pattern, known as lock coupling or hand-over-hand, ensures that the function maintains exclusive access to the relevant part of the tree while traversing, but quickly frees resources to allow other operations to proceed in parallel on other parts of the tree. The strategy is further illustrated in figure 8.

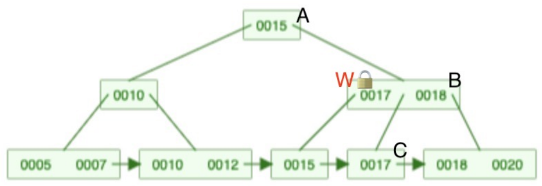
Remove operation - This method initiates at the root node with an exclusive lock acquisition, guaranteeing controlled access. As the operation progresses towards the target leaf node, the strategy evaluates the necessity of maintaining the lock based on the node's occupancy level. Specifically, if a node is determined to be more than half full, the lock on it is released after child's lock is acquired, thereby optimizing resource utilization since the balancing action will not propagate through it. This conditional locking mechanism, hinging on the occupancy state of each node, effectively mitigates excessive lock contention and overhead, enhancing the system's overall efficiency.

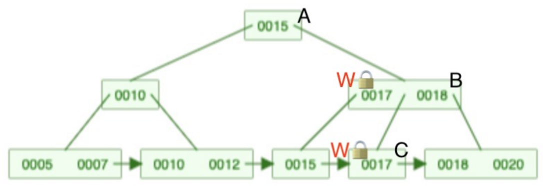
> **Example.** *Suppose we want to* `Remove` *the key* 17*, we need to get to the second to last leaf node.*





At this point, the reason we can release the exclusive write lock on A is because node B is still at least half full even if its children sub-trees merge with each other (making node B to lose one key), i.e. the number of keys of node B is strictly greater than $(order - 1)/2$ - threshold of half-full).





But here, we cannot release the exclusive write lock on node B because node C is not at least half full when it removes the key 17, i.e. it will need to either borrow from its siblings or merge with its siblings, and thus making changes to its parent node B.

Insert `Insert` is similar to `Remove`, but the criterion on whether the thread can release ancestors' exclusive write locks is that the current node is less than full, i.e. the number of keys of the current node is strictly less than ORDER $- 1$.

3.5 Latch-Free B+ Tree

Traditional fine-grain locking methods, while adept in managing concurrency to a certain extent, often encounter limitations in high-performance, many-core processing environments. These constraints predominantly arise from the overhead associated with lock management, and the susceptibility to contention scenarios, especially under high transaction loads and frequent write operations. To address these issue and pushing the limit of parallel B+ tree, Jason et al. introduces a latch-free approach, which is inherently designed to align with the parallel architecture of modern many-core processors.

The latch-free algorithm, unlike all other B+ trees implemented in the scope of this project, provides an

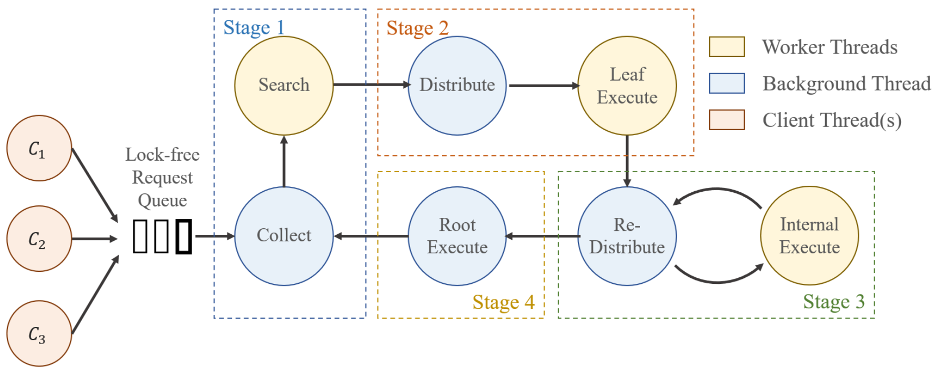


Figure 9. A cycle for latch-free B+ tree. Dashed boxes show the 4 stages in the original PALM paper.

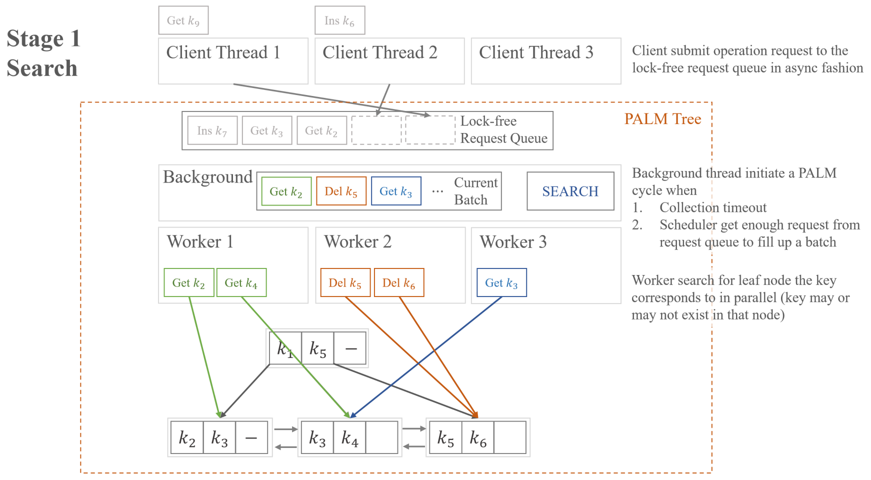


Figure 10. Search state of latch-free B+ tree - all workers search for leaf node in parallel

asynchronous API for the user due to its special scheduling mechanism to maximize the utilization of parallel architecture. Specifically, when user calls `get`, `insert` or `remove`, the client only submitted a request into a lock-free request queue to the tree. The background thread will read from the request queue, compose a batch, and resolve one batch of request in a single PALM cycle with multiple worker threads.

The algorithm can be divided into 4 stages in the original paper, but when implementing, we divide the algorithm into 7 different states (as shown in figure 9).

1. **Collect State** - at this state, all the worker threads are busy waiting for the signal from background thread. The background thread will collect requests from the lock-free queue in `boost::lockfree` library.

2. **Search State** - at this state, background thread will busy wait for the signal from worker threads. The worker threads will begin to search in parallel for the leaf correspond to each Request and store the leaf pointer. (Shown in figure 10)

   Note that though there are no locks/latches in the tree, the parallel access of tree is not considered as data racing since all threads are performing read-only searching operation at this stage.

3. **Distribute State** - at this state, the worker threads will busy wait for background thread to collect leaf nodes searched and group requests based on the leaf node pointer they owned. Then, all requests within a single node will be assigned to a single thread so no data-racing is possible.

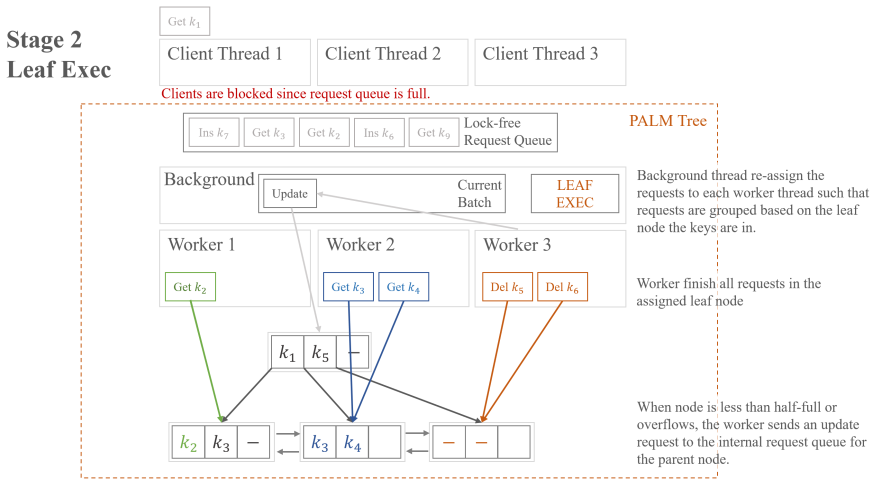

Stage 2
Leaf Exec
Get k_1
Client Thread 1
Client Thread 2
Client Thread 3
Clients are blocked since request queue is full.
PALM Tree
Ins k_7
Get k_3
Get k_2
Ins k_6
Get k_9
Lock-free Request Queue
Background
Update
Current Batch
LEAF EXEC
Background thread re-assign the requests to each worker thread such that requests are grouped based on the leaf node the keys are in.
Worker 1
Worker 2
Worker 3
Get k_2
Get k_3
Get k_4
Del k_5
Del k_6
Worker finish all requests in the assigned leaf node
When node is less than half-full or overflows, the worker sends an update request to the internal request queue for the parent node.

Figure 11. Leaf execute state of latch-free B+ tree - each worker receive request on a unique leaf node and perform operation in parallel

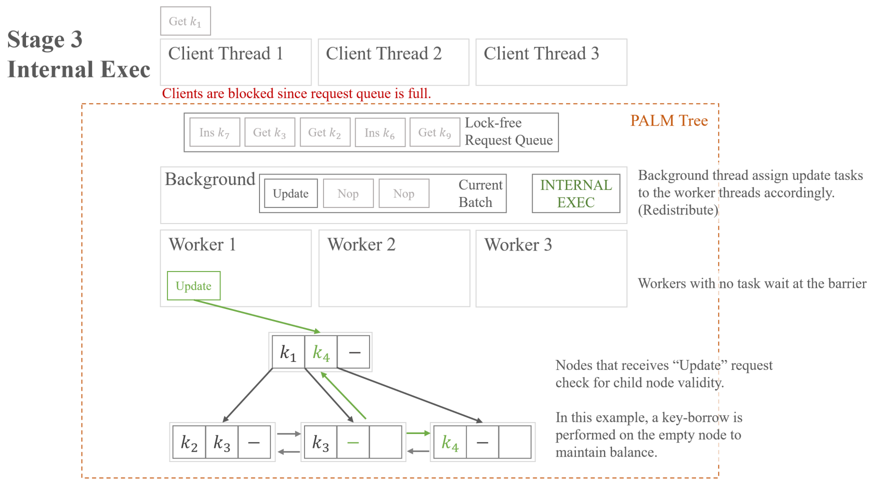

Stage 3
Internal Exec
Get k_1
Client Thread 1
Client Thread 2
Client Thread 3
Clients are blocked since request queue is full.
PALM Tree
Ins k_7
Get k_3
Get k_2
Ins k_6
Get k_9
Lock-free Request Queue
Background
Update
Nop
Nop
Current Batch
INTERNAL EXEC
Background thread assign update tasks to the worker threads accordingly. (Redistribute)
Worker 1
Worker 2
Worker 3
Update
Workers with no task wait at the barrier
Nodes that receives "Update" request check for child node validity.
In this example, a key-borrow is performed on the empty node to maintain balance.

Figure 12. Internal execute state of latch-free B+ tree - worker receive update request and merge, borrow or split children

4. **Leaf Execute** - workers execute the requests in leaf node. Insert or remove the key from the leaf node. If the node is overfilled (number of keys more than or equal to order of tree) or underflows (less than half of the order), the worker thread will push a `UPDATE` request to the scheduler to notify the inconsistency in current subtree. (Shown in figure 11)

5. **Redistribute** - scheduler collect the `UPDATE` request from the worker nodes, remove duplicate `UPDATE` requests, and re-assign them to each worker.

6. **Internal Execute** - worker receive the `UPDATE` request for parent nodes and execute borrow, merge, or split operations on the children nodes. (Shown in figure 12) During the update, it is possible that the parent get more/less key and requires a recursive split/merge. In this case, the worker will send a new `UPDATE` request to the background thread and update the current node in next iteration.

7. **Root Execute** - if the root node need to be modified, the background thread will handle this modification directly to reduce the overhead of communication between worker and background threads.

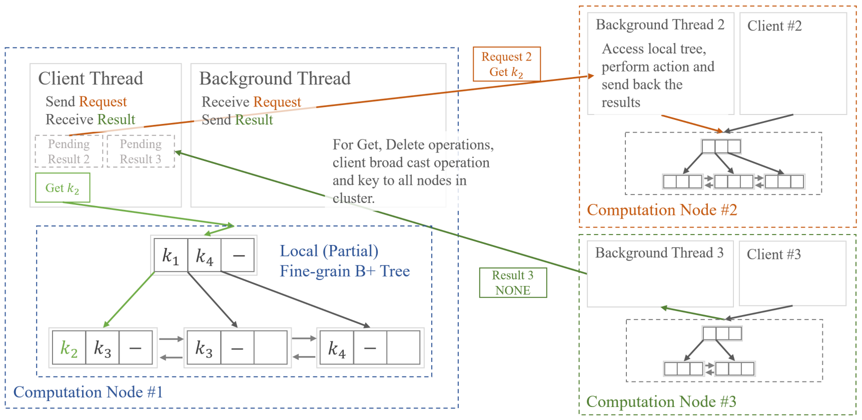


Figure 13. The distributed B+ tree - the client send request and the background thread process requests

3.6 Distributed B+ Tree

Though Latch-free B+ tree can utilize parallelism on arbitrarily many threads, there are cases where a single machine does not have enough RAM to store all keys or scenarios where B+ tree need to be shared between processes. To fill up this corner of the throughput-scalability plot, we implemented a distributed B+ tree based on OpenMPI.

When an MPI program create a `DistributeBPlusTree` object, a background thread will be created and act as a "server" for the local B+ tree (shown in figure 13). The background thread runs in an infinite loop listening for the requests from other processes, execute the requests on local tree, and send back response. The "client", or the control flow for caller of distributed tree, will send requests to all other processes when `get` and `delete` operation occurs. Since both client thread and background thread need to access the local tree, we used the fine-grained lock B+ tree as the local tree.

Communication reduction - To reduce communication between threads, the `insert` operation is not broadcasted within the cluster, the process will insert the new key in local tree silently. When other process broadcast a `get` request, the local tree can then response with existence of key.

Destruction Strategy - The destructor of the distributed B+ tree is designed to operate in a blocking manner to preserve the system's integrity. On activation, it issues a `STOP` request to all connected processes, signaling the end of incoming client requests. The client thread then waits for the background thread to join. The background thread, however, does not terminate immediately. It maintains a counter, tracking the number of `STOP` requests received. Termination of the background thread occurs only when the count of `STOP` requests equals the total number of nodes in the 'MPI World Comm', indicating all computational nodes are offline. This strategy guarantees a coordinated and complete shutdown of the distributed B+ tree.

3.7 General Performance Optimizations

Using `std::vector` + SIMD or `std::deque` for `key` and `children` We initially used `std::deque` for both `keys` and `children`. After experimentation, we decide to use `std::vector` for `keys` and `std::deque` for `children`. Although `std::deque` has $O(1)$ cost for push_front and pop_front, it is not using contiguous memory; for `std::vector`, we can accept $O(n)$ cost for push_front and pop_front, since they are not common in the operations (only `splitNode`, `removeMerge`, and `removeBorrow` are manipulating keys). A major

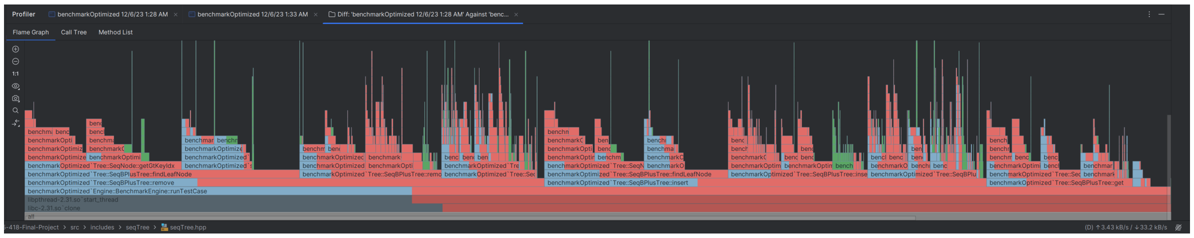


Figure 14. Comparison of using `std::deque` (before) and `std::vector` (after) for `keys`

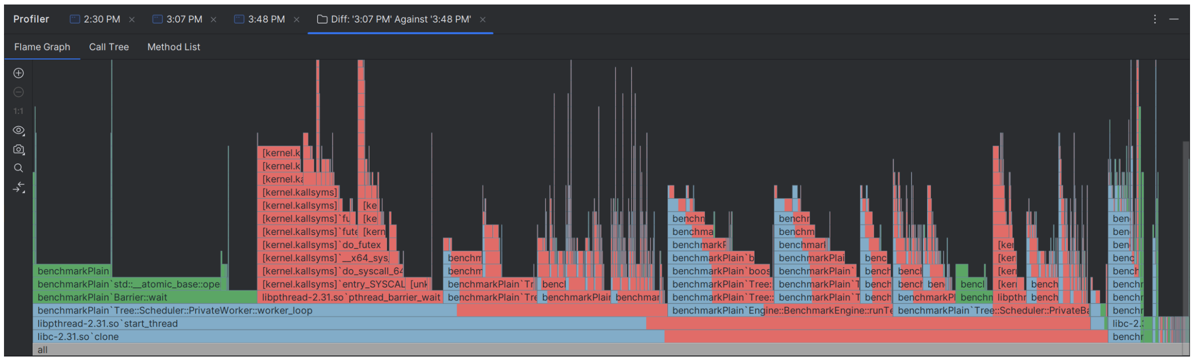


Figure 15. Comparison of using `pthread_barrier_t` (before) and `Barrier` (after) for Latch-Free B+ Tree

advantage of using `std::vector` is that we can apply SIMD acceleration when we search keys on nodes, which is the most frequent operation in `Get`, `Insert`, and `Delete`.

Figure 14 show the diff of profiler flame graph before and after applying SIMD in leaf-finding process. As indicated on the graph, there is a significant improvement on the timing after IPL acceleration is applied.

SIMD for processing assignment Latch-Free B+ Tree From runtime benchmarking and analysis on latch-free B+ Tree, we noticed that the Distribute state of background thread consumes most time. So similar to speeding up the search of keys in nodes for all versions of B+ Tree, we also use SIMD in the (Re)-Distribute state of latch-free B+. When the background thread assigns the requests in the same node to the same threads, we iterate through `std::unordered_map` which map the node to a `std::vector` of requests in that node in SIMD pattern, which speedup the assignment process.

Refined `LockManager<T>` for allocation & deallocation of locks for Fine-Grain B+ Tree For fine-grain B+ Tree, we refactored `LockManager<T>` to speed up the acquisition and release of shared and exclusive locks on nodes by multiple threads when they want to operate on the tree in parallel; specifically, we use statically allocated array instead of `std::deque` for storing locks on nodes, and this reduces the overhead of creating and deleting elements in `std::deque`.

Customized `Barrier` for Latch-Free B+ Tree For Latch-free B+ Tree, we use a customized `Barrier` - Spin lock (instead of `pthread_barrier_t`) for coordination between background thread and worker threads to reduce system call to Linux kernel, significant speedup for palm tree!

4 Results

4.1 Experiment Setup

The experiments are conducted on two different machines. All experiments on sequential tree, coarse grain lock tree, fine-grain lock tree and latch-free tree are conducted on the personal PC, while all experiments on distributed B+ tree are conducted on the PSC Bridges2-RM machine using `sbatch` jobs on partition `RM`.

The detailed hardware configuration for each machine is listed below:

1. Personal PC
    - CPU: AMD Ryzen 7 5800H @ 3.67GHZ 8 physical core, with hyperthread
    - Memory: 64GB, 3200MHz
    - Operating System: Ubuntu 20.04 LTS
2. Pittsburgh Super Computing (PSC) Bridges-2 Regular Memory (RM) machine
    - CPU: AMD EPYC 7742 64-core CPU $\times 2$, with 8 cores allocated in `sbatch` job.
    - Memory: 256GB
    - Operating System: Linux

4.2 Correctness Test

Prior to delving into performance comparisons, rigorous correctness testing ensured the functionality and integrity of all five B+ tree implementations. An automated test generator script crafted diverse input sequences, containing a wide range of insertion, deletion, and get operations on varying sizes. These generated test cases, combined with hand-crafted unit test cases were run against each B+ tree variant. These tests meticulously verified fundamental operations, key validation, tree structure consistency, and adherence to expected B+ tree behavior. Only after confirming 100% passing rates for all tests could we proceed with confidence to the benchmarking stage, knowing that the observed performance differences stemmed solely from design and optimization choices, not underlying implementation flaws. This thorough correctness testing laid the foundation for a reliable and insightful performance analysis.

4.3 Benchmarking Configurations

The benchmarking are performed under four different sets of working conditions. Two access patterns are simulated using an automated test generator.

- **Get Pattern** - in this case, the user is mostly trying to retrieve keys from the tree, with very little and sparse modifications like `insert` and `remove`. In the Get pattern, the distribution of `Get`, `Insert` and `Remove` is 98%, 1%, and 1%. This case is a comparatively light-loaded since the `Get` is a read-only operation on B+ tree.
- **Mixed Pattern** - this access pattern represents a more extreme use case of the B+ tree, the user is frequently modifying the keys in the tree - the proportion of `insert` and `remove` in operation sequence is significantly larger than the Get pattern. In the Mix pattern, the distribution of `Get`, `Insert`, and `Remove` is 90%, 10%, and 10%.

And two different tree sizes (“pre-filled” size) are also considered in the testing to represent different workload of the B+ tree data structure.

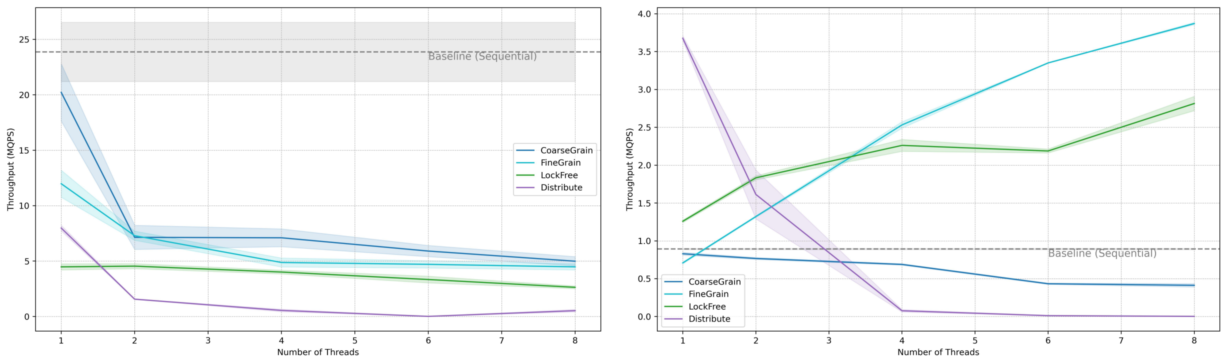


Figure 16. Benchmarking result on the Get access pattern, with light workload (left) and heavy workload (right). Shaded area indicates the range of ± 1 standard deviation.

- Light workload - the tree is initially empty when running the benchmark.
- Heavy workload - before running the benchmark, 10^6 random keys are inserted into the tree.

The combination of 2 variables form 4 different benchmarking configuration. For each configuration, we generate 5 random access trajectories and ran every type of tree for 5 times on each file with each number of thread used. The average and standard deviation of throughput (million queries per second, MQPS) is then measured and reported in figure 16 and 17. The trees are all set to have order of 5 to maximize the utility of SIMD (on both machines, the SIMD width is 4 when using `int` as key).

4.4 Benchmarking Result

Get Pattern We conducted benchmarking on get-intensive access pattern under both light and heavy workload configuration and the resulted number of thread - throughput graph is shown in figure 16.

In the **light** workload configuration, the baseline model (sequential tree) performs the best, reaching an extreme throughput of 24MQPS. while all parallel algorithms show a negative correlation between the throughput and number of threads used. This is due to: 1) the overhead of parallelism in orchestration, synchronization, and communication; 2) on a tiny B+ tree with depth approximately < 4, the threads may content with each other for the ownership of lock or exclusive access to root node.

In the **heavy** workload configuration, the fine-grain lock model performs the best, reaching a speedup of approximately $4\times$ in 8-thread configuration. Since in fine-grain lock B+ tree, a subtree can be accessed simultaneously by multiple threads if all of them are performing read-only operation (i.e. `get`). In the get-intensive access pattern, the tree is almost completely paralleled. (the $\times 2$ difference between theoretical speedup and practical speedup is caused by the locking/unlocking) The latch-free algorithm, even though having higher performance initially, eventually becomes slower than the fine-grained lock algorithm due to the expensive overhead of orchestration and communication in search-distribute-execute cycle.

It is noteworthy that in both workload configuration, the performance of coarse grain lock B+ tree and distributed B+ tree shows negative correlation with number of threads used. For the coarse grain lock B+ tree, the contention on global mutex lock increases linearly as the number of threads increases. For the distributed B+ tree, the inter-process communication increases as number of nodes increases, which makes the application bandwidth-bounded.

Mix Pattern We conducted benchmarking on mixed access pattern under both light and heavy workload configuration and the resulted number of thread - throughput graph is shown in figure 17.

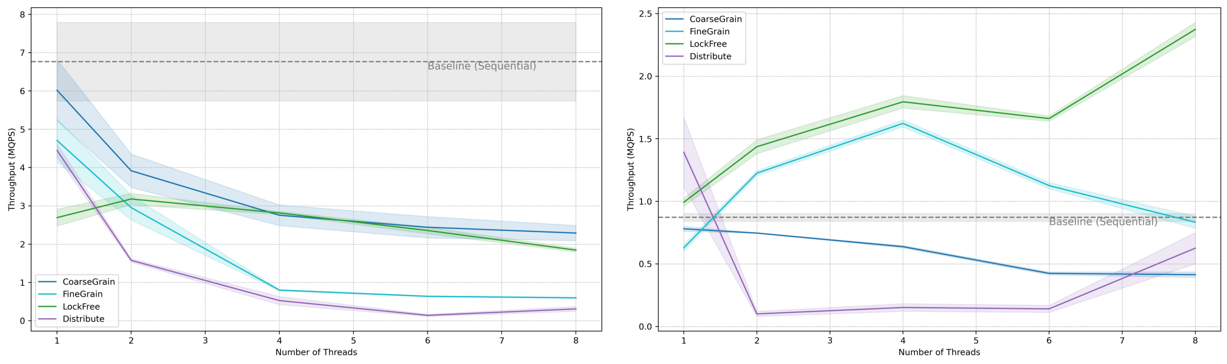


Figure 17. Benchmarking result on Mix access pattern, with light workload (left) and heavy workload (right). Shaded area indicates the range of ± 1 standard deviation

In the mixed access pattern, a similar trend to get-intensive access pattern in **light** workload is shown - all parallel algorithms show negative correlation between number of threads and the throughput of data structure. Still, this is caused by the contention and overhead of parallel algorithm on tiny tree.

However, in the **heavy** workload setup, the benchmarking result becomes more interesting. First, we will analyze and explain the performance fluctuation of **fine-grain lock B+ tree**. On the right side of figure 17, we can see that the fine-grain lock B+ tree's throughput increases initially but significantly decreases after having more than 4 threads. This might be due to the contention of locks in the B+ tree. Since the mixed access patter have much more frequent write operation, the exclusive ownership is frequently required by individual thread and this competition of resource increases as number of threads increases.

Then, we would provide a reasoning for the performance of **latch-free B+ tree** in heavy workload setup. Though fluctuated a bit when number of worker thread equals to 6, the overall trend of latch-free B+ tree's throughput is increasing as number of threads increases. This is because the distribute-execute cycle in PALM algorithm allows to maximize the parallelism in tree without any inter-worker communication like lock or flag, and processing requests in batch allows the tree to reduce redundant work. For instance, in PALM tree algorithm, there are `BigSplit` , `BigMerge` function, which may perform multiple balancing operations in one iteration without additional cost of searching for node from root repetitively.

4.5 Additional Experiment on Tree Depth

In the previous reasoning, we mentions the contention problem in tree for multiple times. Since all parallel algorithms perform parallelism on a node-level (i.e. each node is processed by one thread at a time), and the amount of node in tree is exponential to the depth, we hypothesized that **the key factor for deciding whether to use parallel B+ tree or not is the expected depth of the tree**.

The following experiment is conducted to test this hypothesis: under the heavy workload configuration, the tree order is set to 9 instead of 5 as in the benchmarking experiments above.

The results are shown in figure 18. While the fine-grain lock's performance did not degrade significantly in the get-intensive access pattern, it degenerates significantly comparing to result with $order = 5$ and is even slower than the coarse-grain lock B+ tree under mix access pattern. This indicates that the speedup achieved by the fine-grain lock significantly depends on the amount of write operations and tree-depth.

On the other hand, the performance of latch-free algorithm (PALM) is more robust against the change of order. while its absolute throughput decreases, it remains to gain a $\times 2$ to $\times 3$ speedup comparing to the sequential algorithm and shows a positive correlation between number of thread and throughput in this case.

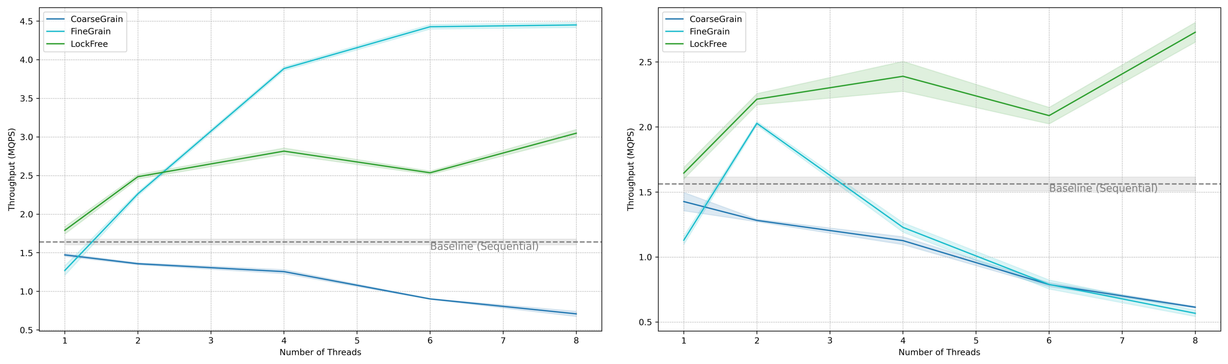


Figure 18. Benchmarking result for Get-intensive pattern (left) and Mix pattern (right) under heavy workload, orders are set to 9

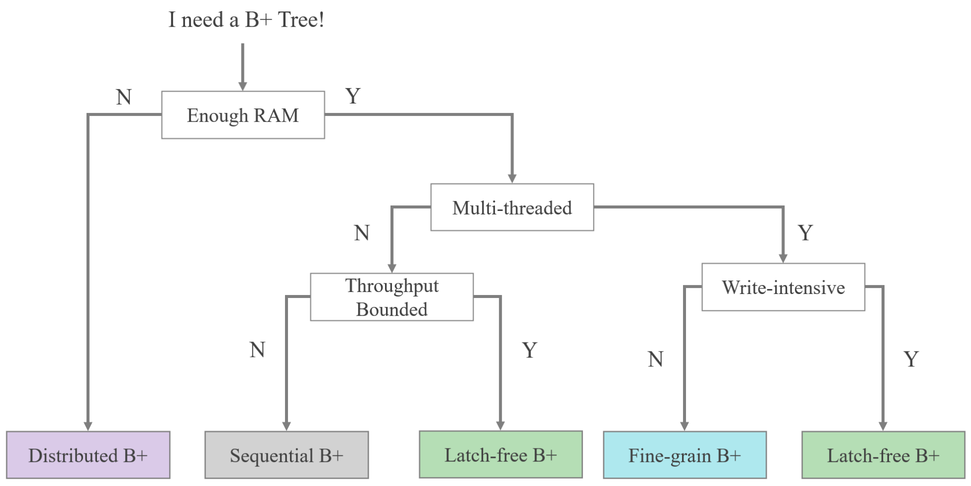


Figure 19. Guide for selecting the appropriate B+ tree for data intensive applications

5 Summary

In conclusion, the B+ forest project implemented 5 different B+ trees, namely sequential, coarse-grain lock, fine-grain lock, latch-free and distributed B+ trees, covering a wide range of need for scalability and throughput under intensive operations. During implementation, we embodied various techniques in parallel programming like instruction level parallelism, multi-threading, etc. and fully utilize the parallel architecture provided by the hardware platform.

We also conducted extensive benchmarking on all 5 types of B+ trees under different access pattern and working conditions. In general, multi-threaded parallel B+ Trees (fine-grained and latch-free) perform significantly better than sequential B+ Trees under heavy workload. This is because under heavy workload, the tree size is large enough and most operations will only affect a fraction of the entire tree. This leads to lower resource contention, allowing more operations to be done in parallel in different sub-trees.

Our benchmarking result can be summarized into a single decision tree shown in figure 19.

6 Future Works

We present B+ Tree framework that support `Get` , `Insert` , `Remove` on keys of different types. For a more general purpose B+ Tree, supporting leaf nodes with key/value pairs is one of our future directions.

Also, our latch-free B+ Tree is not actually lock-free in the sense that (1) worker threads are assigned tasks statically and (2) if one worker thread aborts/dies, the whole process would be in deadlock since we are using `Barrier` . So, in order to make it lock-free, (1) worker threads should be assigned tasks dynamically in case some die during execution, and (2) there should be timeouts in the coordination of threads so that some other threads can take over.

References

1. **Experiment Platform**
   Since we plan to use OpenMPI to construct a large-scale parallelism N-body simulator, we will need to use the Bridge2-Regular Memory (Bridge2-RM) nodes in Pittsburgh Super Computing (PSC) for benchmarking. For the remaining parts of this project, we can conduct experiment on GHC cluster machines.

2. **Papers on B+ tree concurrency**

   [1] Bayer, R., Schkolnick, M. Concurrency of operations on B-trees. Acta Informatica 9, 1–21 (1977). `https://doi.org/10.1007/BF00263762`

   [2] Sewall, J., Chhugani, J., Kim, C., Satish, N., & Dubey, P. (2011). PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. Proceedings of the VLDB Endowment, 4(11), 795–806. `https://doi.org/10.14778/3402707.3402719`

   [3] Srinivasan, V., Carey, M.J. Performance of B+ tree concurrency control algorithms. VLDB Journal 2, 361–406 (1993). `https://doi.org/10.1007/BF01263046`

3. **Reference in final report**

   [1] Wikipedia. (2023, March 19). B+ tree. In Wikipedia, The Free Encyclopedia. Retrieved from `https://en.wikipedia.org/wiki/B%2B_tree`

Contribution

- We collaborated on most of the work, including brainstorming, implementing, and debugging (1) sequential B+ tree (2) coarse grained B+ tree (3) fine grained B+ tree (4) latch-free B+ tree, and (5) distributed B+ tree together using the live-share extension of VS code.
- Aside from the above collaboration, Yutian (Mark) Chen is in charge of setting up the website, cleaning & refactoring the codebase, writing test engines & benchmarking, and optimizing the fine grained B+ tree and lock-free B+ tree.
- Aside from the above collaboration, Yumeng (Acee) Liu is in charge of writing most of the reports, and added some optimization for apple M1 chip.