B+ Forest – A Comprehensive Analysis of Parallel B+ Tree Algorithms

Yutian Chen, School of Computer Science, yutianch@andrew.cmu.edu
Yumeng Liu, School of Computer Science, yumengli@andrew.cmu.edu

Abstract

Our 15-418 final project – B+ Forest – focuses on the implementation and analysis of different B+ tree structures, an essential data structure in database management systems. We implemented and investigated the performance of 5 versions of B+ Trees: (1) sequential B+ Tree, (2) coarse-grained B+ Tree, (3) fine-grained B+ Tree, (4) latch-free B+ Tree, and (5) distributed B+ Tree under different settings.

In this project, we embodied multiple levels of parallelisms in the program. We have utilized the SIMD instructions, multithread, multi-process and even multi-machine distributed system to improve the throughput and scalability of B+ trees.

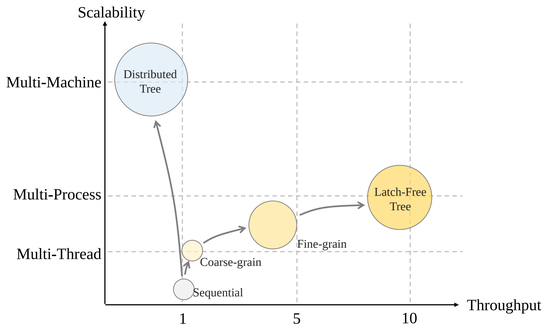

Fig 1. 5 Different B+ Trees implemented in this project and their relative speedup/scalability. Each tree satisfy a specific scenario.

Data Structure Design

Sequential B+ Tree

We implemented the sequential B+ tree following the most native and standard version. For a tree with order n, every node can have at most $n-1$ keys and n children. One unique characteristic of B+ tree is the guarantee of balance – all leaf nodes in the tree have same height and all nodes are at least half-full (has $\geq n/2$ keys).

Coarse Lock B+ Tree

The sequential tree did not support the multithread usage. In coarse lock B+ tree, we add a global mutex lock on the sequential version to eliminate the data racing problem.

Fine-grain Lock B+ Tree

As a naïve adaptation, the coarse lock B+ tree is enough for many low-throughput scenario. However, in highly parallel and request-intensive use case, the global mutex is overly conservative. To increase the availability of tree and increase parallelism, we can break the global mutex into a per-node reader-writer locks.

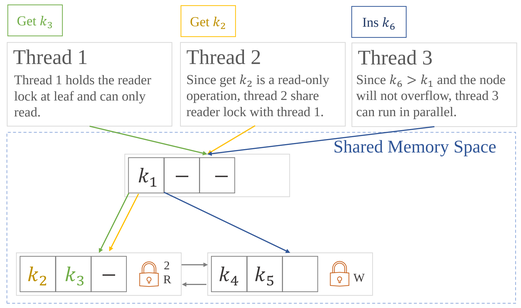

Fig 2. Fine-grain lock B+ Tree, using reader-writer lock on each node.

The reader lock from root to leaf is obtained by the read-only operation like get, while the writer lock is obtained if current write operation (insertion or deletion) might cause tree to rebalance and will affect current node.

Latch-free B+ Tree (PALM Tree)

Fine-grain lock improves the parallelism of data structure, but every request will lock and unlock for $\log_o n$ times, which becomes a large overhead for throughput-bounded applications. Also, the fine-grain lock algorithm has limited scalability due to the contention problem on root node. To further increase the throughput and scalability, Jason et al. proposed the PALM algorithm for a latch-free parallel B+ tree. The PALM utilizes parallelism by batching requests and let all thread perform local changes iteratively through the tree.

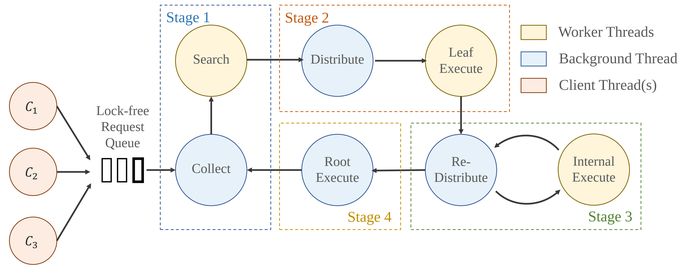

Fig 3. 4 Stages of the PALM algorithm

During scheduling, the background thread will orchestrate N worker threads and maintain an internal task queue. At each iteration, the tasks of each thread will be completely irrelevant with other threads.

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. In the distributed B+ tree, every node maintains a local and partial B+ tree. Inter-process communication will occur only during *get* and *delete* operation.

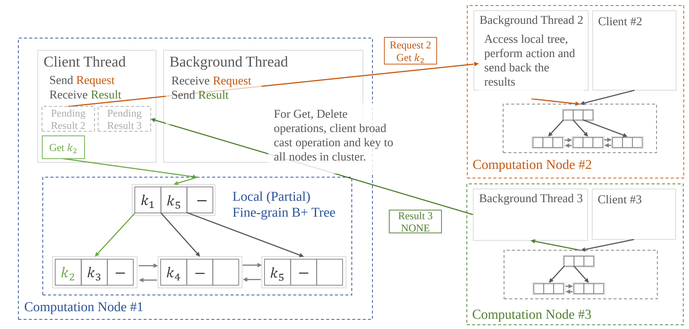

Fig 4. Distributed B+ Tree Algorithm

Results

We evaluated the algorithms in different working condition and with different number of threads. In intense read scenario, fine-grain lock algorithm performs the best, while in intense write scenario, the latch-free algorithm performs the best.

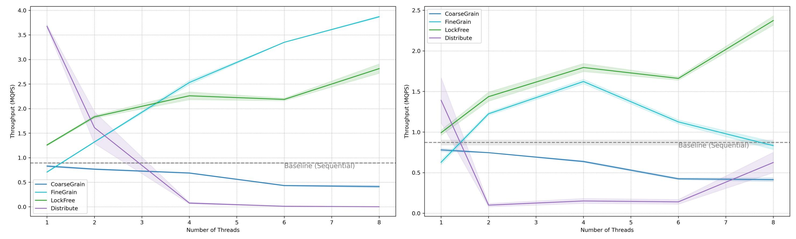

Fig 5. Benchmarking result – Intense Read (Left) and Intense Write (Right) on Heavy B+ Tree (with depth ≈ 8)

However, the parallel B+ tree algorithm is effective only when the tree is sufficiently large. Fig 5. shows the performance when tree has 10^6 elements on average. (That is, tree depth is approximately 8)

When the tree depth is approximately 3, the performance of Sequential algorithm is significantly better than all parallel algorithms (Fig 6).

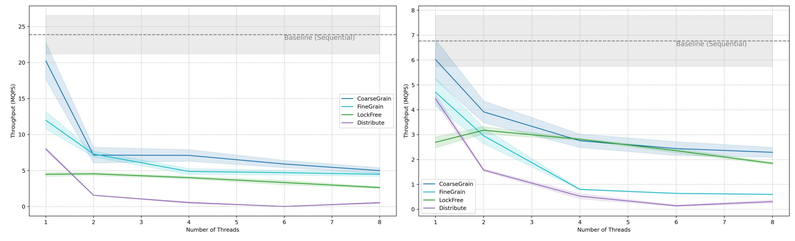

Fig 6. Benchmarking result – Intense Read (Left) and Intense Write (Right) on Lightweight B+ Tree (with depth ≈ 3)