Multi-threaded implementation of Merkle Tree Operations

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Rishitha Surineni   Shloka Somalaraju   Guide: Sathya Peri

Indian Institute of Technology Hyderabad

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Introduction

A Merkle Tree is a binary tree that uses cryptographic hashes to efficiently verify large datasets, with a root hash summarizing all the data.

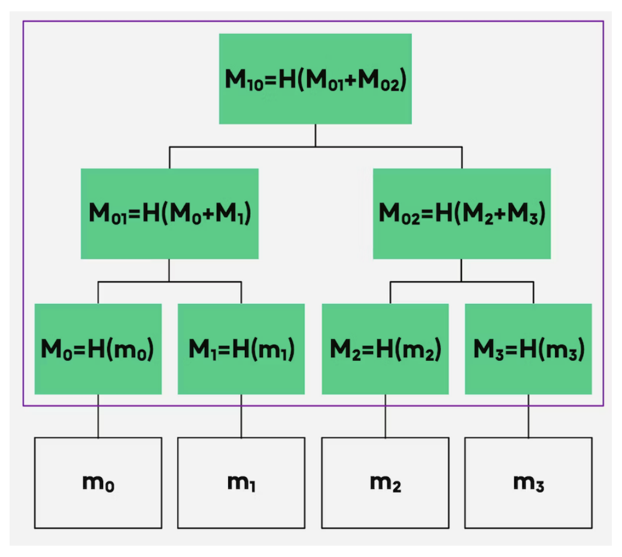


Figure 1. Merkle Tree of Depth 2.

It is commonly used in blockchains like Bitcoin and Ethereum, and file-sharing networks to ensure data integrity and efficient verification.

Motivation

- As datasets grow massively in size, sequential updates and read operations on Merkle Trees become a bottleneck.
- Parallel updates risk inconsistencies in the root hash if not synchronized properly.
- Without isolation, read operations risk capturing the tree in an **inconsistent mid-update state**, rendering generated membership proofs untrustworthy.

- **Problem Statement**: Our main objective is to parallelize updates in a Merkle Tree while keeping the system live and consistent under heavy loads. The goal is to design a system that efficiently distributes workload across threads and maintains uninterrupted availability for both operations.

Prior Work and their Limitations

Trillian's implementation

- Locking the entire tree during each update.
- **Limitations:** By not utilizing multi-core architectures, it forces serial execution and struggles with high update rates.

Angela implementation

- Identifies **Conflict nodes** (deepest common ancestors) in a batch.
- Only locks these nodes while updating.
- Visitation check at each conflict node so no stale values go up.
- **Limitations:** Batch processing which prevents real-time processing, making it unsuitable for live applications requiring immediate state updates and low-latency responses.

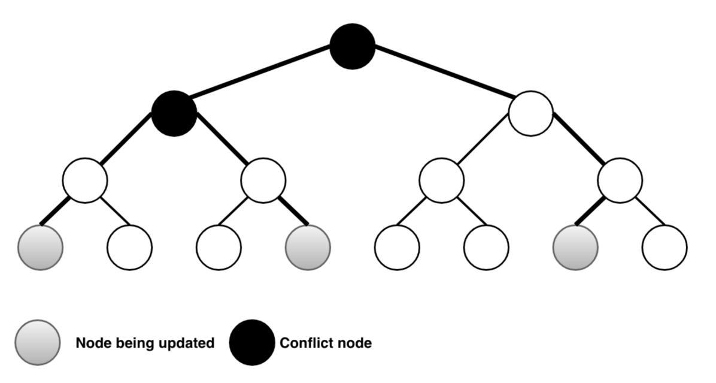


Figure 2. Angela implementation - locking

Proposed Solution

- Generated random workloads which are update heavy.
- Developed a thread pool to process operations via a synchronized queue, mimicking real-time request handling.
- Acquiring parent's lock before locking any children ensured the **prevention of deadlocks**.

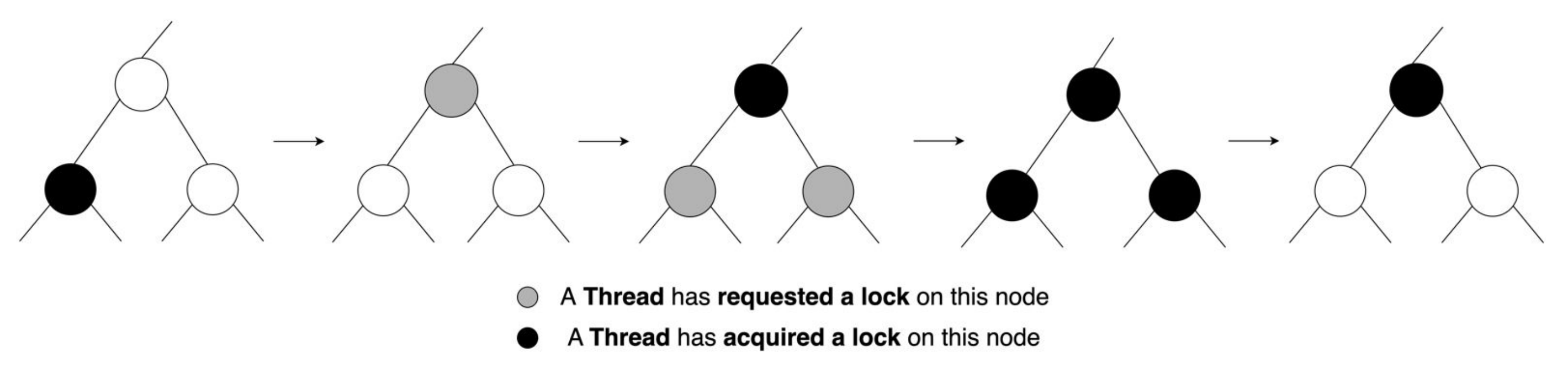

A **Thread** has **requested a lock** on this node
A **Thread** has **acquired a lock** on this node

Figure 3. Locking procedure in our model.

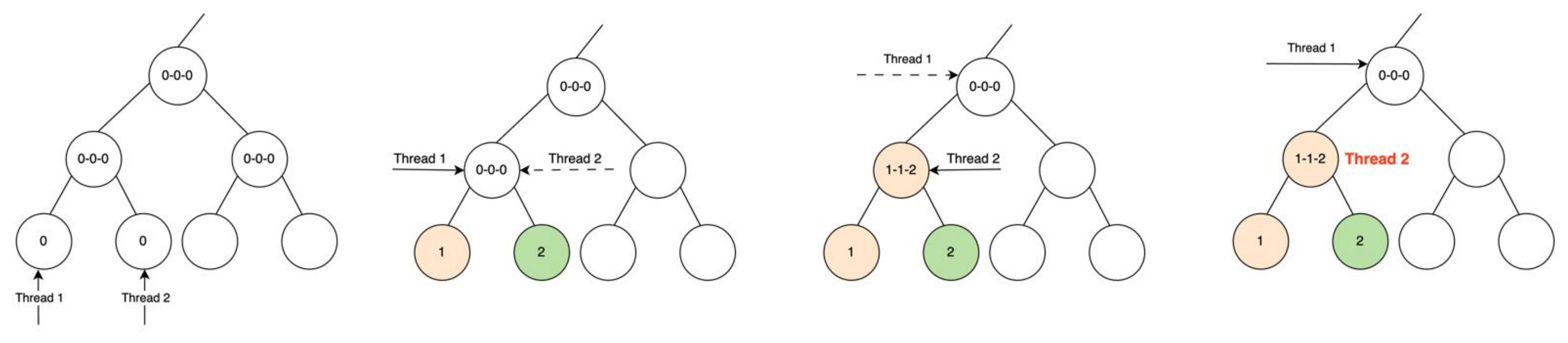


Figure 4. Stopping Redundant update

Concurrency Model

- Every update is given a unique id(thread index and thread-local update count).
- Child update IDs are stored in the parent to detect redundant updates and terminate corresponding threads.
- A global vector is used to signal threads to halt if other threads update nodes along the path they already updated.
- To guarantee consistency, **reads** are serviced only after all **prior updates** have been committed to the root, establishing a stable state for **snapshot** capture.

Correctness of the algorithm is verified by comparing parallel and serial execution root hashes.

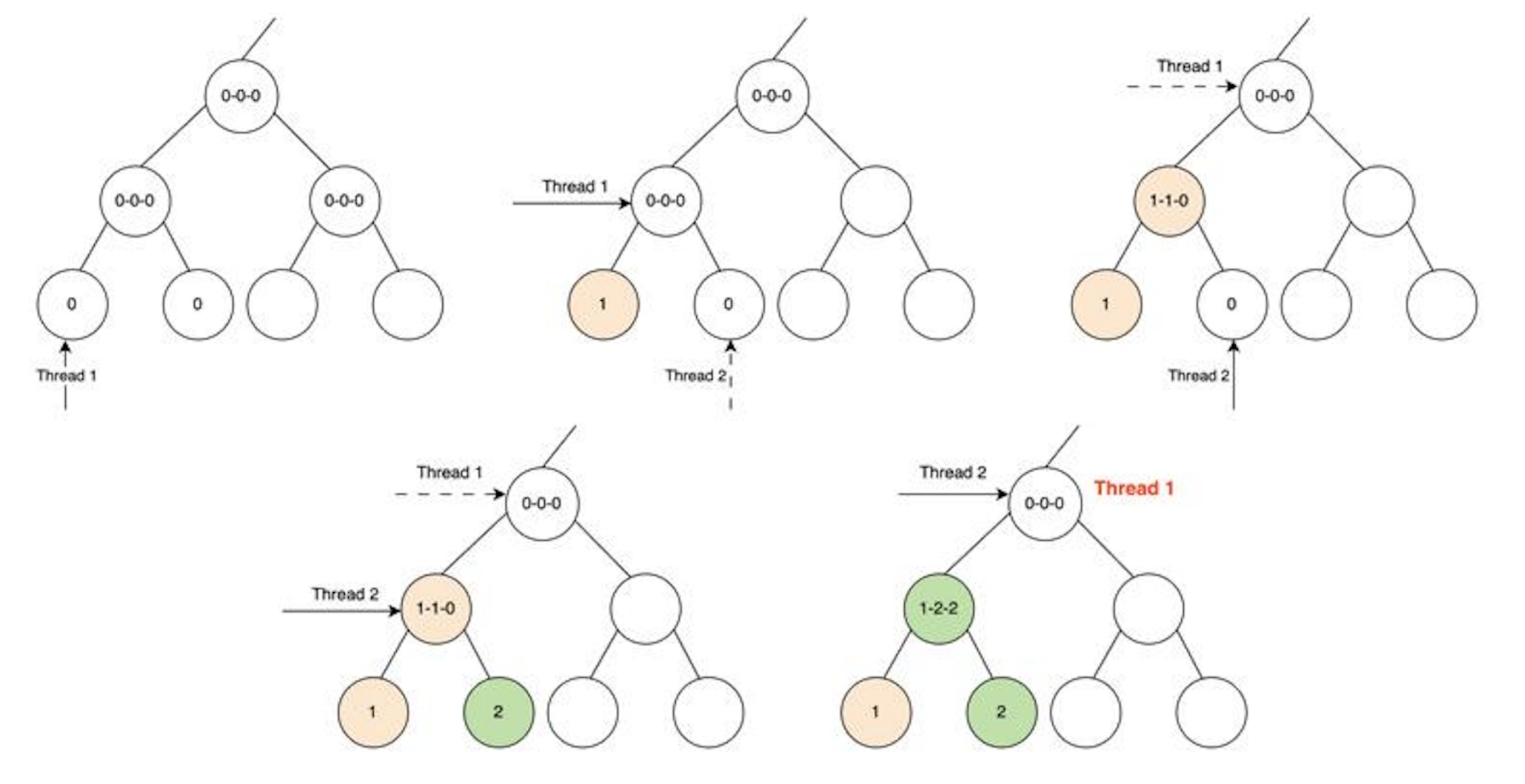
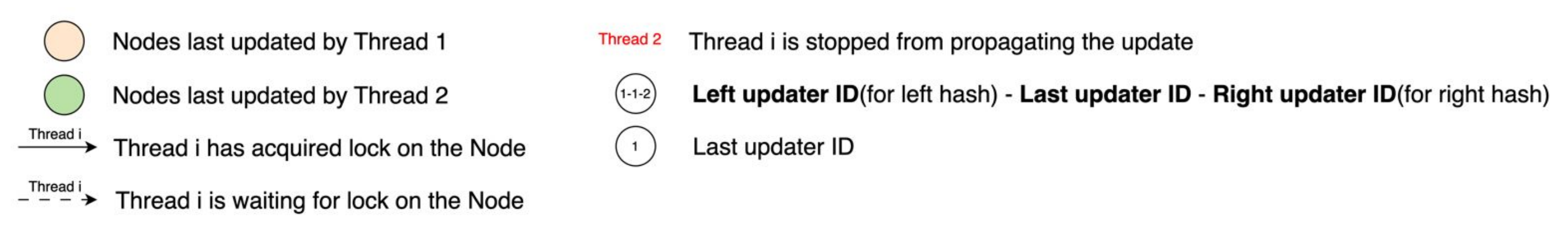


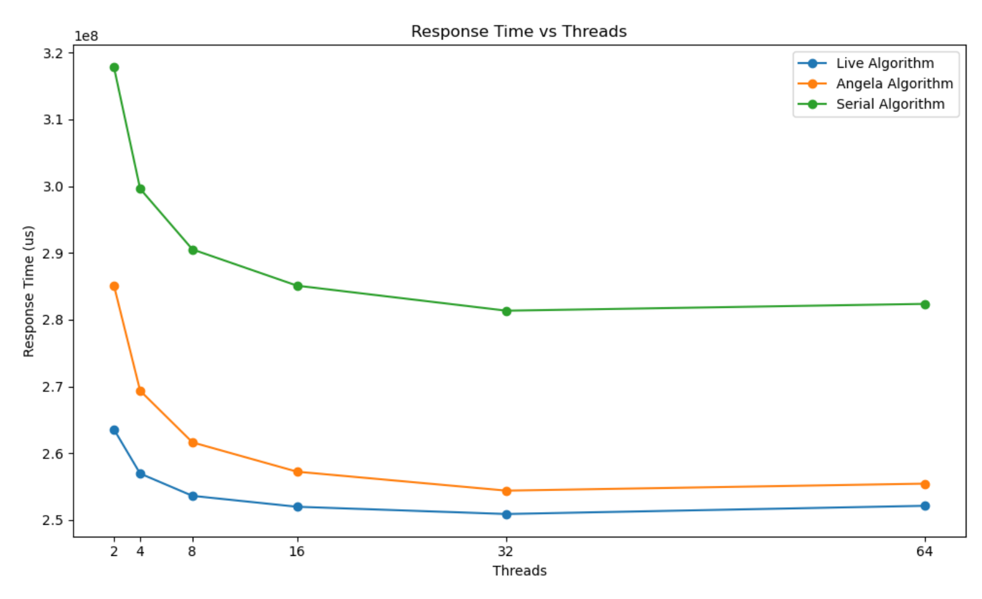
Figure 5. Stopping Stale update

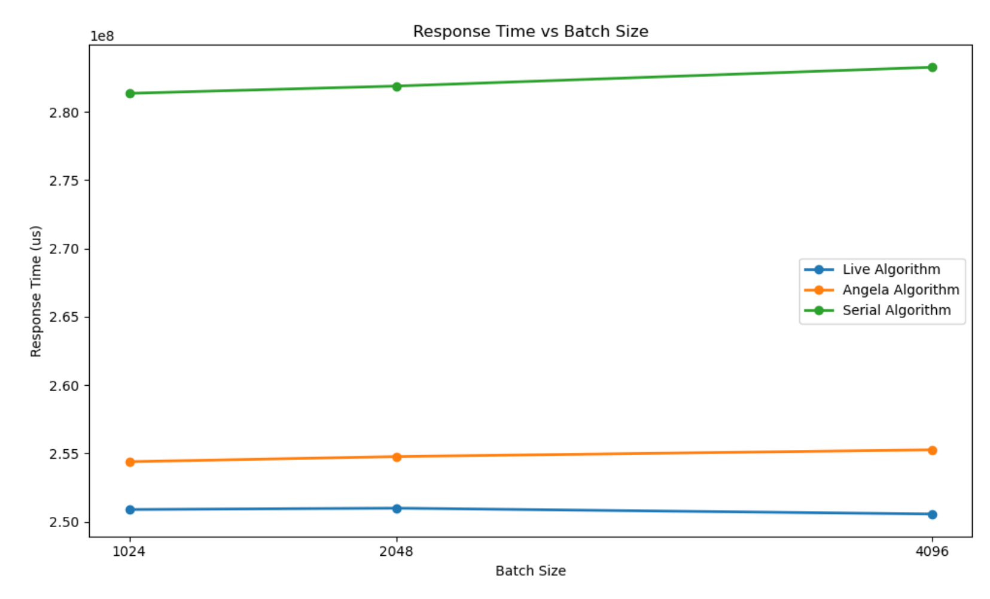

Nodes last updated by Thread 1
Nodes last updated by Thread 2
Thread i has acquired lock on the Node
Thread i is waiting for lock on the Node
Thread 2
Thread i is stopped from propagating the update
1-1-2
Left updater ID(for left hash) - **Last updater ID** - **Right updater ID**(for right hash)
1
Last updater ID

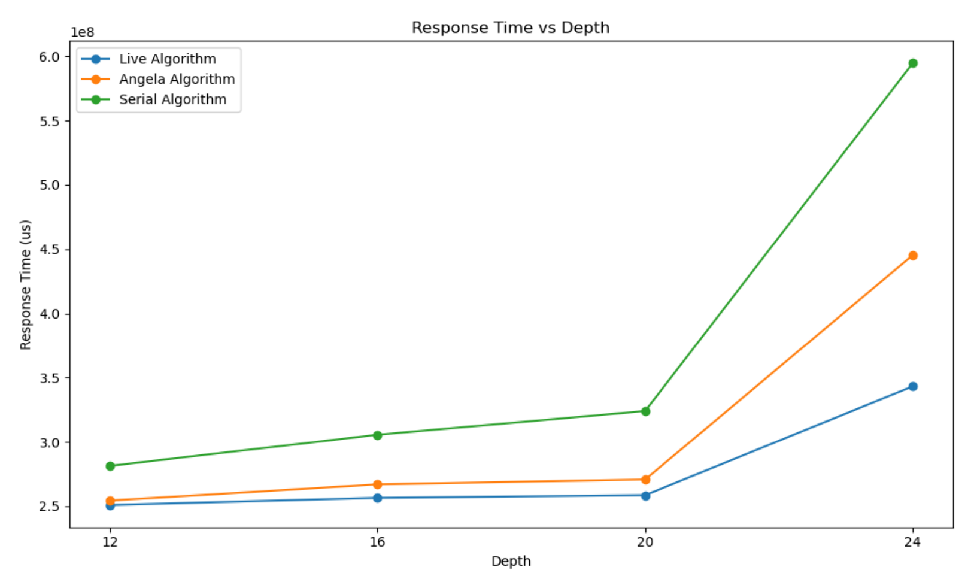
Experimental Setup

- Tested on 28 physical cores system. Test parameters : Tree depth (12 to 24), thread count (2 to 64), batch size (1024 to 4096), total of 1 million operations.
- Due to restrictions on RAM experiments were conducted with a maximum of $2^{25}-1$ nodes (depth of merkle tree = 24) and usually in block chains an average of 3000-5000 leaves are present in a merkle tree.
- Employed SHA-256 for cryptographic hashing.

Results







For experiment 1, Depth of Merkle Tree = 12, Batch Size = 1024
For experiment 2, Depth of Merkle Tree = 12, Number of Threads = 32
For experiment 3, Number of threads = 32, Batch Size = 1024
The **Live algorithm** achieves the **lowest response time** across all workloads, outperforming Angela and Serial by a significant margin.

Future Improvements

- Verify the working of the solution on very large Tree depths.
- Realising Merkle Tree on a distributed setting.

References

[1] J. Kalidhindi, A. Kazorian, A. Khera, and C. Pari.
Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree.