Mercury: A Lock-Free Approach to a Merkle Tree

Archit Somani, Rachit Anand, Tarun Jain
Department of Computer Science and Engineering
SNIoE, India

***Abstract*—Merkle trees are foundational data structures that are widely used to verify data integrity in distributed systems. This paper introduces *Mercury*, a lock-free and parallel approach to the construction of Merkle trees. Using atomic flag-based synchronization and dynamic worker reallocation, Mercury ensures efficient, contention-free construction of the tree. This method is particularly well-suited for high-throughput environments where performance and scalability are critical.**

***Index Terms*—Merkle Tree, Hashing, Lock-Free Algorithm, Parallel Computing, Distributed Systems**

I. Introduction

The exponential growth of data and the rise of decentralized technologies have renewed attention to the challenge of ensuring data integrity and authenticity. Merkle trees, also known as hash trees, offer a compact and cryptographically secure method of verifying data in distributed environments. First introduced by Ralph Merkle in 1979, these structures enable efficient verification without requiring access to the entire dataset.

In systems such as blockchains, distributed version control systems, and cloud storage platforms, Merkle trees allow for scalable integrity checks. Their hierarchical structure ensures that even the slightest modification, such as a single bit flip in a data block, alters the hash of that block, which in turn changes the hashes of all parent nodes up to the Merkle root. This cascading effect makes tampering easily detectable, as the final root hash serves as a fingerprint of the entire data set.

This paper proposes *Mercury*, a lock-free approach to Merkle tree construction that facilitates parallelism and minimizes synchronization overhead, enhancing performance in concurrent settings.

II. Structure of a Merkle Tree

A Merkle tree is a binary tree in which each leaf node represents the hash of a data block, while each internal node is the hash of the concatenation of its two child nodes. This recursive construction culminates in a single root hash — the Merkle root — which serves as a cryptographic fingerprint of the entire dataset.

One of the key features of Merkle trees is their support for efficient proofs of data inclusion. A Merkle proof enables the verification of the membership of a specific data block using only a logarithmic number of hashes relative to the number of leaf nodes. Additionally, any change to a single data block will propagate upward, modifying the Merkle root, and thereby exposing tampering or corruption.

III. Proposed Algorithm: Mercury

We propose *Mercury*, a lock-free and parallel algorithm for constructing Merkle trees using atomic flags and a worker reallocation mechanism. Each node in the tree maintains two state flags: `done` and `inProgress`. These flags help coordinate parallel threads and avoid race conditions.

A. Initialization Phase

All leaf nodes are initialized in parallel. Each node's hash is computed from its data block and marked as completed.

```
For each leaf node L:
    L.hash = computeHash(L.data)
    L.done = true
    L.inProgress = false
```

B. Node Processing Logic

Worker threads traverse the tree to compute parent hashes, moving upward when both child nodes are marked as done.

```
function processNode(node):
    if (node == null or
    node.done or node.inProgress):
        return

    node.inProgress = true
    sibling = node.getSibling()
    parent = node.getParent()

    if sibling != null and sibling.done:
        parent.hash = computeHash(
            node.hash + sibling.hash)
        parent.done = true
        parent.inProgress = false
        processNode(parent)
    else:
        node.inProgress = false
        reallocateWorker()
```

C. Worker Reallocation Strategy

Idle workers are reassigned to the bottom-most available node that has not been completed or claimed.

```
function reallocateWorker():
    node = getBottomMostFreeNode()
    if node != null:
        processNode(node)
    else:
        terminateWorker()
```

D. Finding Free Nodes

```
function getBottomMostFreeNode():
    For node in tree from leaves to root:
        if not node.done and not node.inProgress:
            return node
    return null
```

E. Execution Loop

The main loop launches workers which keep processing nodes until the Merkle root is computed.

```
While root not done:
    node = getBottomMostFreeNode()
    if node != null:
        processNode(node)
    else:
        wait or terminate
```

IV. Results

A. Methodology

To compare the performance of *Mercury*, we compared it 3 other approaches for Merkle tree construction.

- **Sequential Approach**: Computing each node one by one, layer by layer.
- **Coarse-Grained Approach**: Using a single global lock for the entire tree.
- **Fine-Grained Approach**: Using multiple locks to lock each layer before adding leaf nodes

B. Observations

The time of each approach was measured in milliseconds, and the performance was plotted on a graph with the x-axis showing nodes in the range $[2^{15}, 2^{20}]$.

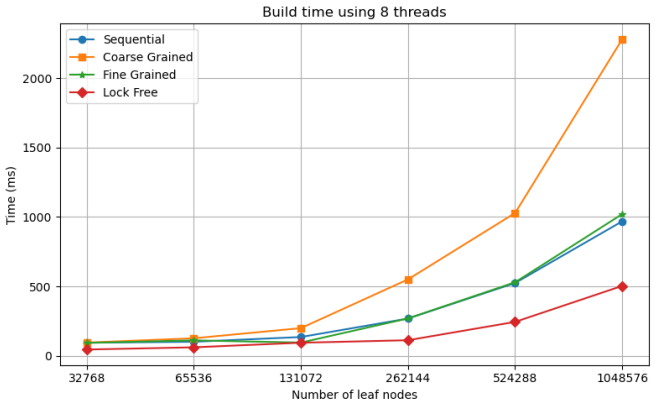

Fig. 1. Performance comparison of approaches

C. Analysis

- *Mercury* gave a significant increase in performance compared to other approaches
- Slowest approach was the coarse-grained approach as it was locking the entire tree, which caused additional overhead in large number of leaf nodes.
- The fine grained approach and sequential approach took almost the same time.

V. Applications

A. General Applications of Merkle Trees

Merkle trees are a core component in numerous technologies that require verifiable and tamper-evident data structures.

In blockchain systems such as Bitcoin and Ethereum, they enable transaction inclusion proofs without requiring full data access, supporting efficient light clients and Simplified Payment Verification (SPV). In distributed version control systems like Git and decentralized storage platforms like IPFS, Merkle trees help track changes, ensure data consistency, and detect divergence across copies.

They are also used in secure communication protocols and certificate transparency frameworks, where they offer cryptographic guarantees over log integrity and event histories. Their logarithmic proof sizes and low computational overhead make them highly suitable for systems with limited bandwidth, storage, or processing power.

B. Applications of the Mercury Approach

The proposed *Mercury* algorithm enhances traditional Merkle tree construction by enabling lock-free, parallel computation. This makes it particularly valuable in environments where concurrency, performance, and scalability are critical.

Potential applications include:

- **High-throughput blockchain nodes:** Full nodes that validate blocks can use Mercury to construct or verify Merkle roots faster by utilizing multi-core architectures.
- **Distributed storage verification:** In systems where data integrity checks must scale with size—such as in cloud backup systems or content delivery networks—Mercury can expedite Merkle tree computation during verification and deduplication tasks.
- **Real-time logging systems:** Systems like tamper-evident audit logs, where Merkle roots are recomputed frequently, can benefit from Mercury's low-latency computation.
- **Parallel cryptographic protocols:** Mercury can be integrated into zero-knowledge proof systems or trusted execution environments where data structures must be built and validated concurrently.

By minimizing synchronization overhead while maintaining correctness, Mercury enables Merkle tree integration into latency-sensitive, high-concurrency systems — areas where traditional locking-based approaches would degrade performance.

VI. Conclusion

This paper presented *Mercury*, a lock-free and parallel approach for constructing Merkle trees. By leveraging atomic flag-based synchronization and a dynamic worker reallocation strategy, Mercury eliminates the need for traditional locking mechanisms while ensuring correctness and consistency in tree construction.

The algorithm allows for efficient utilization of parallel resources, making it well-suited for high-performance systems that require frequent data verification. Through its scalable and

contention-free design, Mercury serves as a practical enhancement over conventional Merkle tree construction methods, especially in decentralized and distributed environments.

VII. Future Work

There are several directions in which this work can be extended:

- **Performance Evaluation:** A comprehensive benchmarking study comparing Mercury with traditional lock-based approaches across various hardware configurations and tree sizes would help quantify its scalability.
- **Memory Optimization:** Investigating more efficient memory management strategies, particularly for large-scale trees, could further improve performance and reduce resource consumption.
- **Integration with Real-world Systems:** Applying Mercury within actual blockchain nodes, distributed storage engines, or real-time log verification systems would provide insights into deployment challenges and optimization opportunities.
- **Fault Tolerance and Recovery:** Exploring how the algorithm handles partial failures, such as worker crashes or hardware faults, can enhance its robustness in critical systems.
- **Formal Verification:** Proving the correctness and safety of Mercury using formal methods or model checking can ensure reliability in safety-critical applications.

Future enhancements along these lines will help refine Mercury into a production-ready component for a wide range of secure and high-performance systems.

References

[1] Janakirama Kalidhindi, Alex Kazorian, Aneesh Khera, Cibi Pari
Angela: A Sparse, Distributed, and Highly Concurrent Merkle Tree.

[2] Haojun Liu, Hongrui Liu, Xinbo Luo, Xubo Xia
Merkle Tree: A Fundamental Component of Blockchains.
2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)

[3] ZIHENG WANG, XIAOSHE DONG, YAN KANG, HENG CHEN, QIANG WANG
An Example of Parallel Merkle Tree Traversal: Post-Quantum Leighton-Micali Signature on the GPU.

[4] Anoushk Kharangate
Asynchronous Merkle Trees.