High Performance Computing - Assignment 3

Aryan Pawar SE22UARI195

1. Introduction

Concurrent data structures are fundamental building blocks for multi-threaded applications. As modern processors increasingly rely on multiple cores for performance, the ability to efficiently manage shared data across threads becomes critical. This project presents a comprehensive implementation and performance analysis of five concurrent data structures: Binary Search Tree (BST), AVL Tree, Treap, Striped HashSet, and Refinable HashSet.

Each data structure employs different synchronization strategies ranging from coarse-grained locking to fine-grained hand-over-hand locking and lock striping. The goal is to understand how these different approaches impact throughput and scalability under various workload conditions.

2. Objectives

The primary objectives of this project are:

- Implement five concurrent data structures with thread-safe operations (contains, insert, remove)
- Evaluate performance with 1 million nodes and 50% prefill
- Test scalability by varying thread count from 1 to 16
- Analyze behavior under different workload distributions (read-heavy to write-heavy)
- Compare tree-based structures (BST, AVL, Treap) with hash-based structures (Striped, Refinable)
- Identify optimal use cases for each data structure

3. Data Structures Implemented

3.1 Concurrent Binary Search Tree

The Concurrent BST implements fine-grained hand-over-hand (lock coupling) locking. Each node contains its own ReentrantLock, allowing multiple threads to traverse and modify different parts of the tree simultaneously. During traversal, a thread holds locks on at most two adjacent nodes at any time, releasing the parent lock before acquiring the grandchild lock.

Synchronization: Fine-grained hand-over-hand locking
Lock per node: Yes (ReentrantLock)
Sentinel node: Used to simplify edge cases

3.2 Concurrent AVL Tree

The AVL Tree maintains balance through rotations after insertions and deletions. The concurrent implementation uses optimistic reads for contains operations (lock-free traversal) and fine-grained locking for modifications. Balance factors are computed locally, and rotations are performed while holding locks on affected nodes.

Synchronization: Optimistic reads, fine-grained writes
Balancing: AVL rotations (LL, RR, LR, RL)
Height maintenance: Updated during rebalancing

3.3 Concurrent Treap

A Treap combines BST properties with heap properties using random priorities. This randomization provides probabilistic balancing without explicit rebalancing operations. The concurrent implementation assigns random priorities at insertion and performs rotations to maintain heap order.

Synchronization: Fine-grained locking
Balancing: Probabilistic via random priorities
Expected height: O(log n)

3.4 Striped HashSet

The Striped HashSet uses lock striping to reduce contention. Instead of a single lock, it maintains a fixed array of locks, each protecting a subset of buckets. The number of locks is typically set to the number of available processors multiplied by a factor.

Synchronization: Lock striping
Number of locks: Fixed (processors × 4)
Resizing: Not dynamic

3.5 Refinable HashSet

The Refinable HashSet extends the striped approach with dynamic resizing and lock refinement. When the load factor exceeds a threshold, the table doubles in size and the number of locks increases. A read-write lock coordinates resizing with normal operations.

Synchronization: Lock striping + ReadWriteLock for resize
Load factor: 0.75
Resizing: Dynamic (doubles capacity)

4. Implementation Details

4.1 Common Interface

All data structures implement a common ConcurrentSet interface:

```
public interface ConcurrentSet {
    boolean contains(int key);
    boolean insert(int key);
    boolean remove(int key);
}
```

4.2 Fine-Grained Locking (BST Example)

The hand-over-hand locking technique ensures thread safety while maximizing concurrency:

```
private static class Node {
    int key;
    volatile Node left, right;
    final ReentrantLock lock = new ReentrantLock();

    void lock() { lock.lock(); }
    void unlock() { lock.unlock(); }
}

// Hand-over-hand traversal pattern: // 1. Lock
parent // 2. Lock child // 3. Unlock parent // 4.
Move down (child becomes new parent)
```

4.3 Lock Striping (Refinable HashSet)

Lock striping distributes contention across multiple locks:

```
private volatile ReentrantLock[] locks;
private final ReentrantReadWriteLock resizeLock;

private ReentrantLock getLock(int bucketIndex) {
      return locks[bucketIndex % numLocks];
}

// Read operations acquire read lock
// Resize acquires write lock (exclusive)
```

5. Experimental Setup

5.1 Benchmark Configuration

Total Elements	1,000,000
Prefill Percentage	50%
Duration per Run	10 seconds
Runs per Configuration	5
Thread Counts	1, 2, 4, 6, 8, 10, 12, 14, 16

5.2 Workload Distributions

Workload	Contains %	Insert %	Delete %	Description
100C-0I-0D	100	0	0	Read-only
90C-9I-1D	90	9	1	Read-heavy
50C-25I-25D	50	25	25	Mixed
30C-35I-35D	30	35	35	Write-heavy
0C-50I-50D	0	50	50	Write-only

5.3 System Configuration

Platform: macOS (Apple Silicon / Intel)
Java Version: OpenJDK 17+

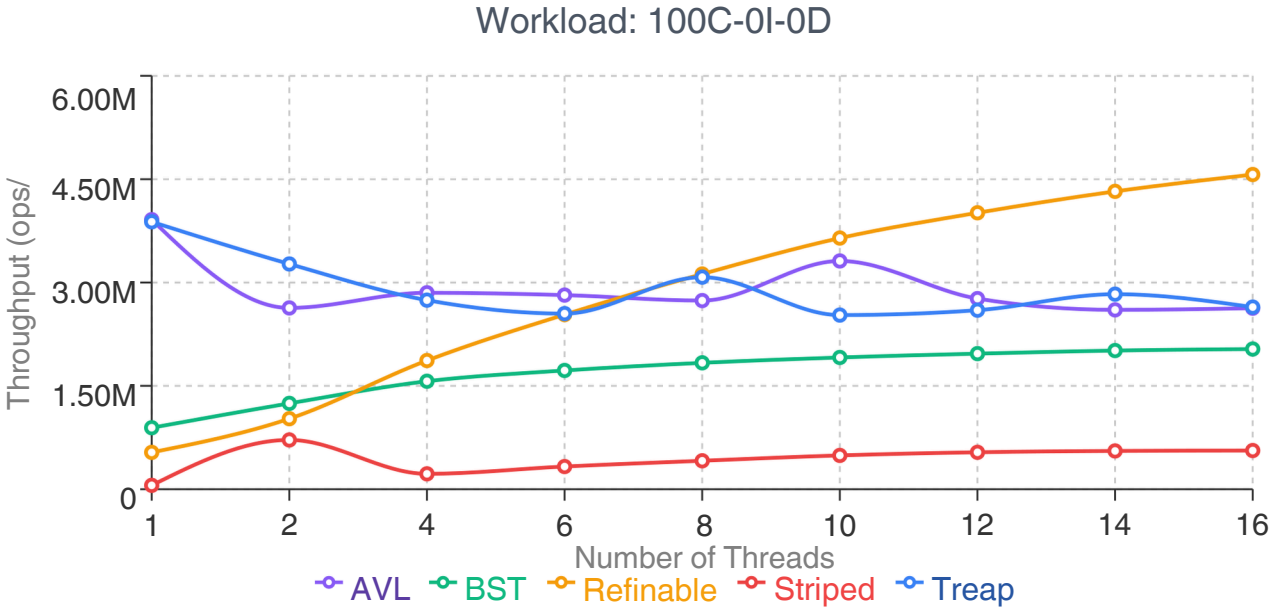
JVM Options: -Xmx8g -Xms4g -XX:+UseG1GC
Build Tool: Maven

6. Results and Analysis

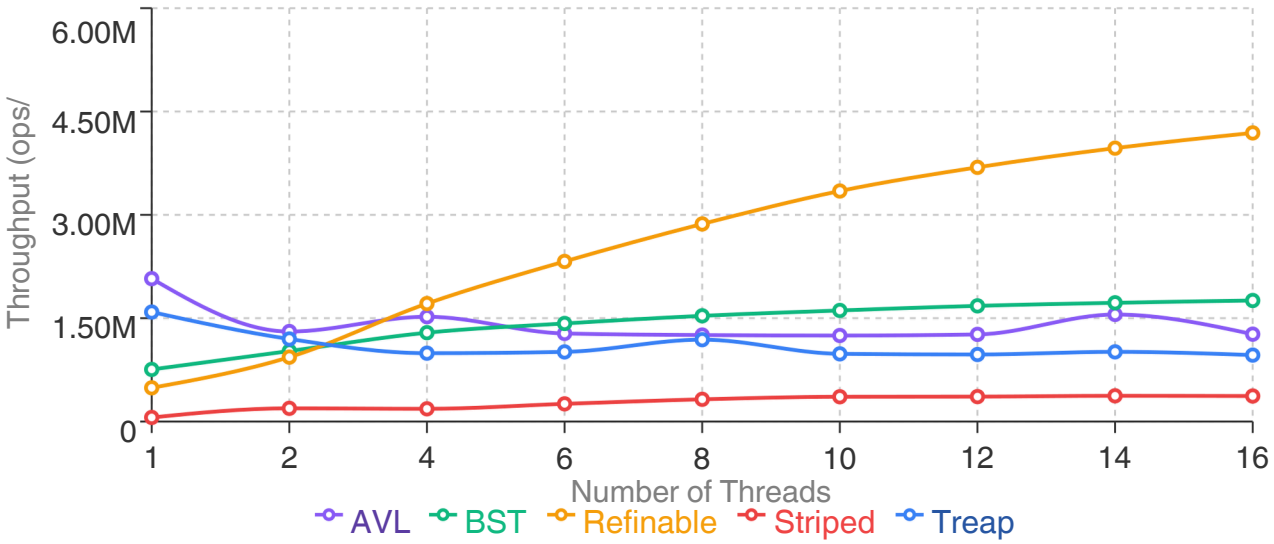
6.1 Peak Performance Summary

Rank	Data Structure	Peak Throughput	Best Workload	Threads
1	**Refinable**	4.57M ops/sec	100C-0I-0D	16
2	**AVL**	3.91M ops/sec	100C-0I-0D	1
3	**Treap**	3.88M ops/sec	100C-0I-0D	1
4	**BST**	2.03M ops/sec	100C-0I-0D	16
5	**Striped**	715K ops/sec	100C-0I-0D	2

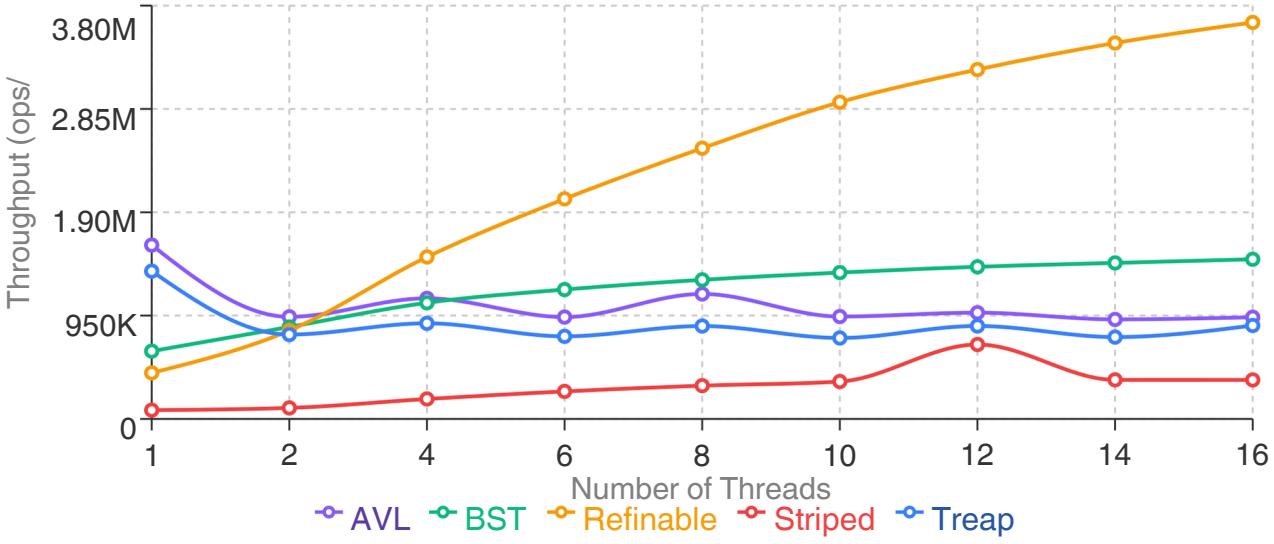
6.2 Throughput vs Threads - All Data Structures



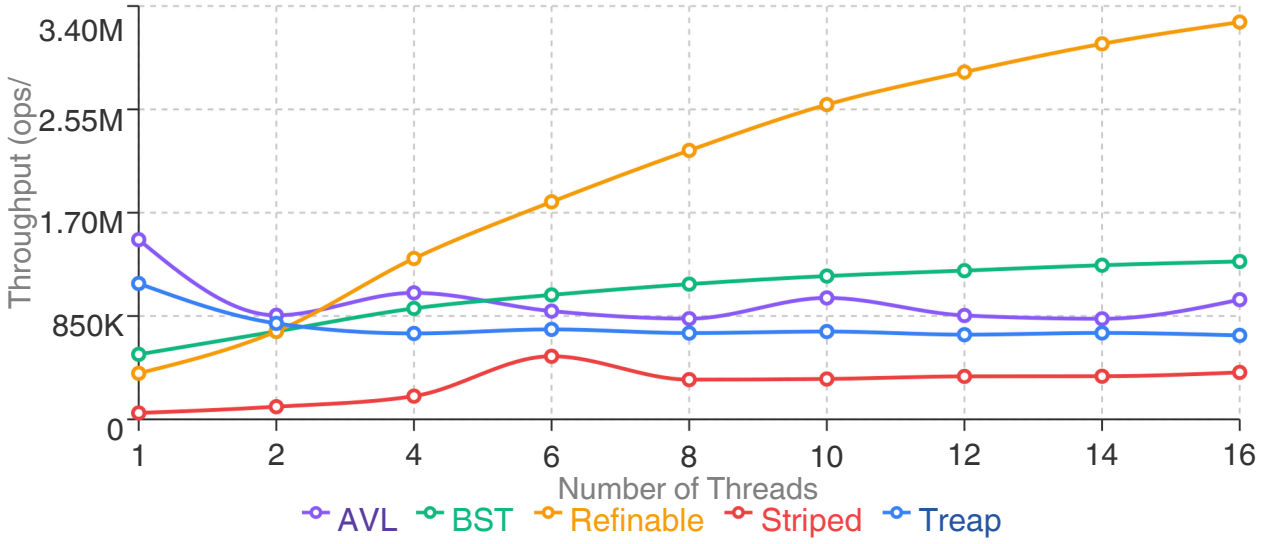
Workload: 90C-9I-1D



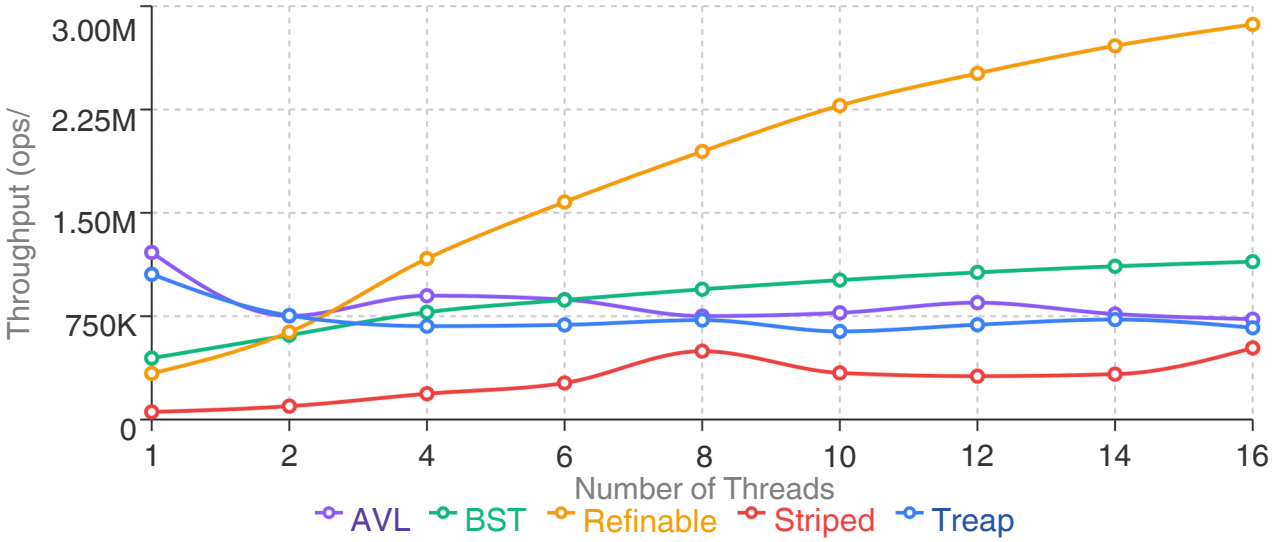
Workload: 50C-25I-25D



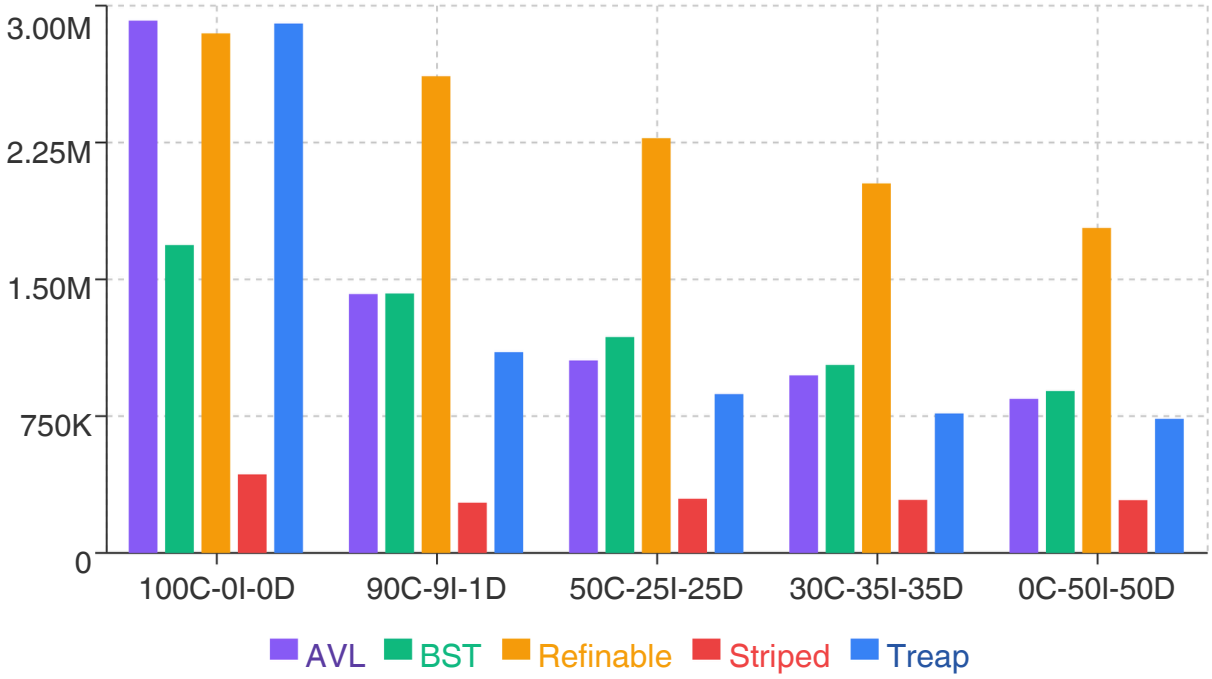
Workload: 30C-35I-35D



Workload: 0C-50I-50D

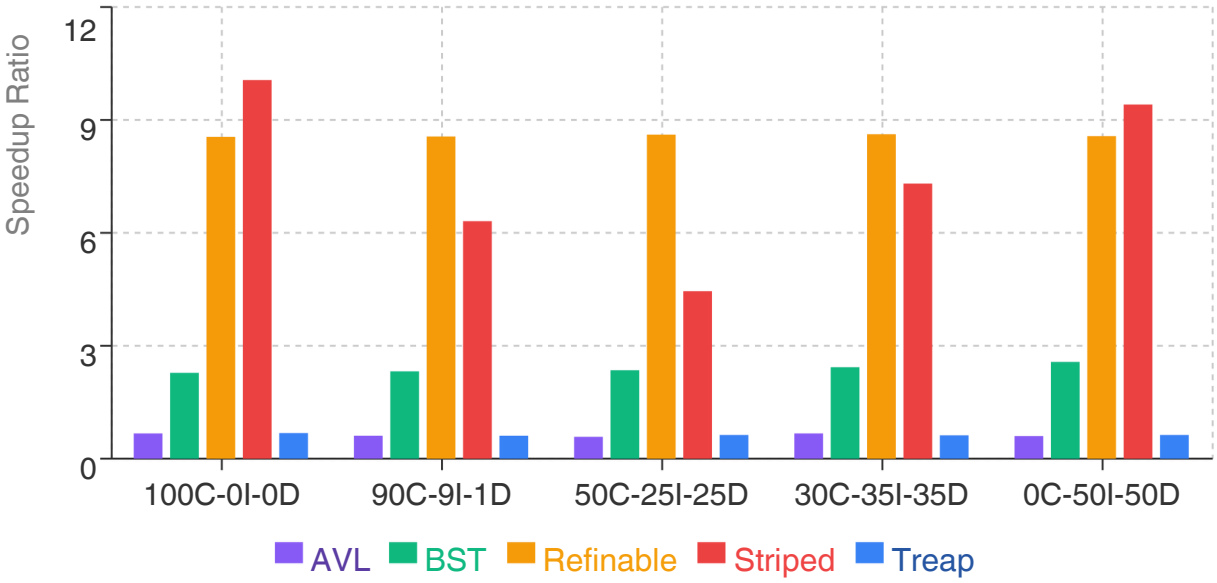


6.3 Average Throughput Comparison



6.4 Scalability Analysis

Scalability is measured as the ratio of throughput at 16 threads to throughput at 1 thread. Values greater than 1 indicate positive scaling.



6.5 Tree Structures Comparison

Comparing the three tree-based structures (AVL, BST, Treap) reveals distinct performance characteristics:

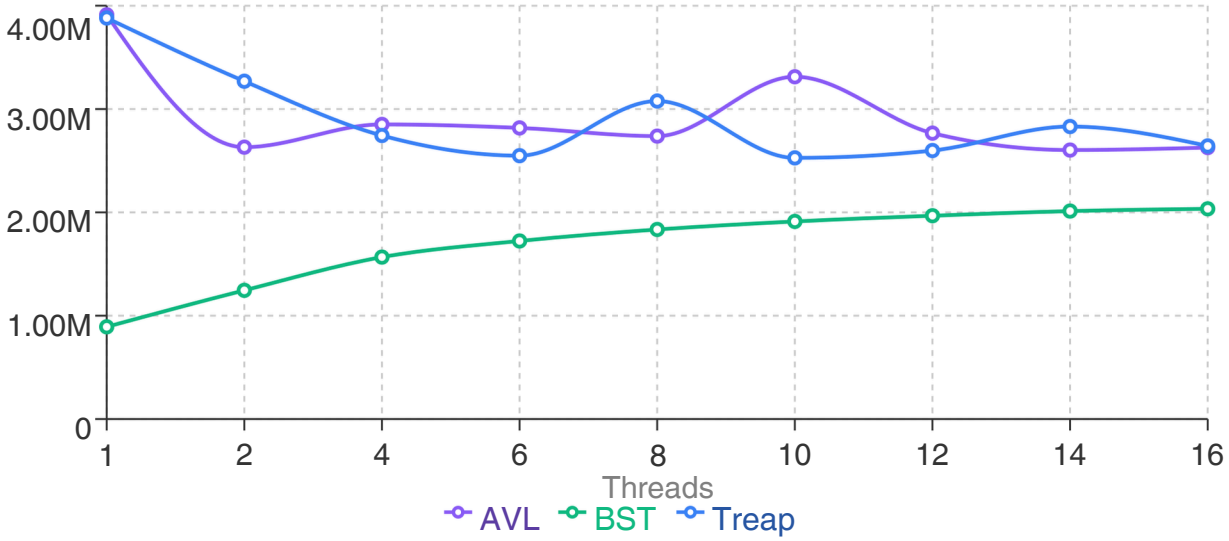


Figure: Tree structures under read-only workload (100C-0I-0D)

6.6 Hash-based Structures Comparison

The hash-based structures show superior scalability due to lock striping:

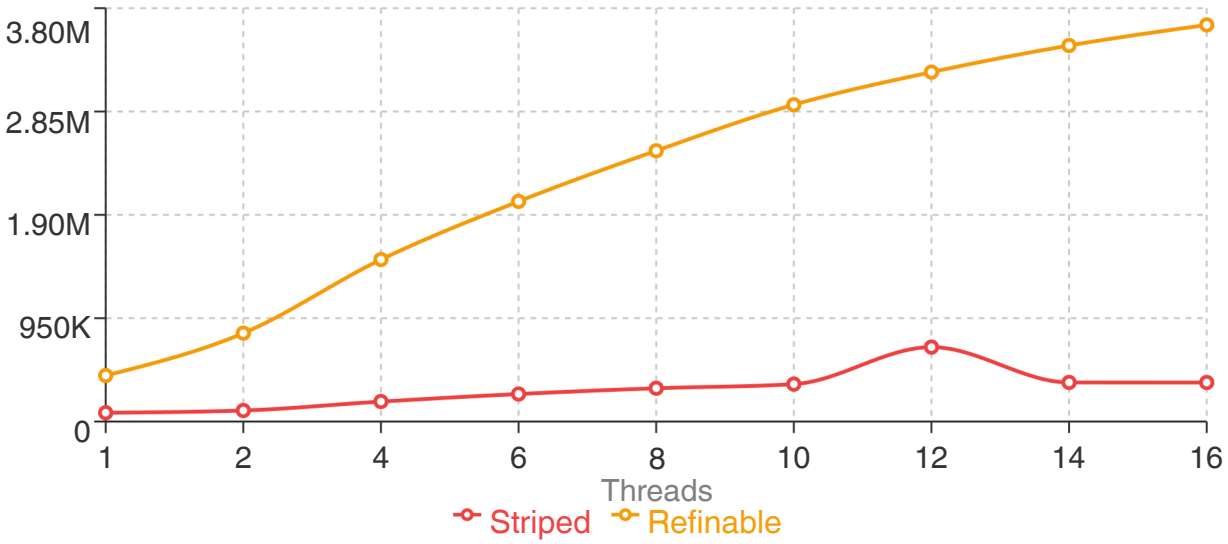


Figure: Hash structures under mixed workload (50C-25I-25D)

6.7 Key Observations

AVL & Treap Performance

Achieve highest single-thread throughput (~3.9M ops/sec) but show negative scaling with increased threads due to root contention in tree traversals.

BST with Fine-Grained Locking

Hand-over-hand locking enables 2.3x speedup at 16 threads. Performance improves consistently as threads increase.

Refinable HashSet Scalability

Best scalability among all structures (8.5x at 16 threads). Dynamic resizing and lock refinement effectively reduce contention.

Workload Impact

Read-heavy workloads (100C) consistently outperform write-heavy workloads (0C-50I-50D) due to reduced lock contention on reads.

7. Conclusions

7.1 Summary of Findings

This project successfully implemented and benchmarked five concurrent data structures with varying synchronization strategies. The results demonstrate that no single data structure is

optimal for all scenarios—the choice depends heavily on workload characteristics and scalability requirements.

7.2 Recommendations

Use Case	Recommended Structure	Reason
Single-threaded, read-heavy	**AVL Tree**	Highest throughput (3.9M ops/sec)
Multi-threaded, high scalability needed	**Refinable HashSet**	Best scaling (8.5x at 16 threads)
Ordered data with concurrency	**BST (fine-grained)**	Maintains order, scales well (2.3x)
Simple implementation needed	**Striped HashSet**	Good balance of simplicity and performance
Probabilistic balancing	**Treap**	No explicit rebalancing, good average case

7.3 Lessons Learned

- **Lock granularity matters:** Fine-grained locking significantly outperforms coarse-grained approaches in multi-threaded scenarios.
- **Root contention is a bottleneck:** Tree structures suffer from contention at the root, limiting scalability.
- **Lock striping is effective:** Distributing locks across buckets dramatically improves hash-based structure performance.
- **Workload characteristics are crucial:** Read-heavy workloads benefit from optimistic/lock-free reads.
- **Memory overhead trade-off:** Fine-grained locking requires additional memory for per-node locks.