Project A4: Concurrent Data Structures and Memory Coherence
ECSE 4320 Advanced Computer Systems

Jaithra Pagadala

1 Experimental Setup and Methodology

1.1 System Configuration

All experiments were conducted on a Lenovo laptop running Ubuntu Linux

- CPU model: 13th Gen Intel Core i9-13905H
- OS: Ubuntu 22.04.6
- Cores: 14
- GCC: 11.04
- Perf : 5.11
- Pinned Frequency: 1.5 GHz

1.2 Motivation

Modern multicore processors rely on cache coherence to maintain correctness, but scalability is often limited by synchronization design. This project explores how lock granularity affects throughput, contention, and effective parallelism in shared in-memory data structures.

The primary focus is to compare:

- a coarse-grained synchronization strategy using a single global mutex, and
- a finer-grained synchronization strategy using per-bucket locks.

1.3 Data Structure

The chosen data structure is a hash table with separate chaining. Keys and values are integers, and each bucket is implemented as a singly linked list. The hash table supports:

- `insert(key, value)`
- `find(key)`
- `erase(key)`

1.4 Correctness Invariants

Correctness under concurrency is maintained by enforcing the following invariants:

(1) Each bucket pointer references a valid linked list or is `NULL`.

(2) Insert and erase operations preserve list structure without losing nodes.

(3) A key maps deterministically to a single bucket.

(4) All modifications to shared state occur while holding the appropriate lock.

1.5 Synchronization Strategies

1.5.1 Coarse-Grained Locking

In the baseline implementation, a single global mutex protects the entire hash table. Every operation acquires this lock before accessing or modifying any bucket.

This approach simplifies correctness reasoning but serializes all operations, making it vulnerable to contention as thread count increases.

1.5.2 Fine-Grained Locking

In the fine-grained implementation, each bucket has its own mutex. Operations only lock the bucket corresponding to the accessed key.

This design reduces contention by allowing concurrent operations on different buckets and avoids deadlock because each operation acquires at most one lock.

2 Workloads

Three workload mixes were evaluated:

(1) **Lookup-only:** all operations are reads.

(2) **Insert-only:** all operations are writes.

(3) **Mixed (70/30):** 70% lookups and 30% inserts.

These workloads represent read-dominated, write-dominated, and realistic mixed-access scenarios.

3 Measurement Methodology

3.1 Timing and Throughput

Execution time was measured using a high-resolution monotonic clock. Throughput was computed as:

$$\text{Throughput} = \frac{\text{operations}}{\text{time (seconds)}}$$

Each configuration was executed multiple times, and results were averaged to reduce noise.

3.2 Derived Cycle Counts

Hardware cycle counters were not used due to limited support on the experimental platform. Instead, cycle counts were derived analytically using the fixed CPU frequency:

$$\text{Estimated cycles} = T_{\text{seconds}} \times 1.5 \times 10^9$$

This allows computation of:

$$\text{cycles/op} = \frac{T_{\text{seconds}} \times 1.5 \times 10^9}{\text{operations}}$$

Because the CPU frequency was pinned, this estimate is consistent across runs and sufficient for relative comparisons.

4 Experimental Procedure

1. Install dependencies and compiler toolchain.
2. Pin CPU frequency to 1.5 GHz.
3. Compile benchmark and hash table implementation.
4. Run automated sweep across:
   - key sizes: 10^4, 10^5, 10^6
   - thread counts: 1, 2, 4, 8, 16
   - workloads: lookup, insert, mixed
5. Record execution time and throughput.
6. Post-process results to compute speedup and estimated cycles per operation.

5 Results and Analysis

The primary behavior of the system is revealed by two core metrics:

1. throughput as a function of thread count, and
2. estimated cycles per operation derived from execution time and fixed CPU frequency.

Additional plots are included to confirm trends across workloads and data sizes, but the majority of insight is captured by these two views.

5.1 Throughput vs Threads (Primary Result)

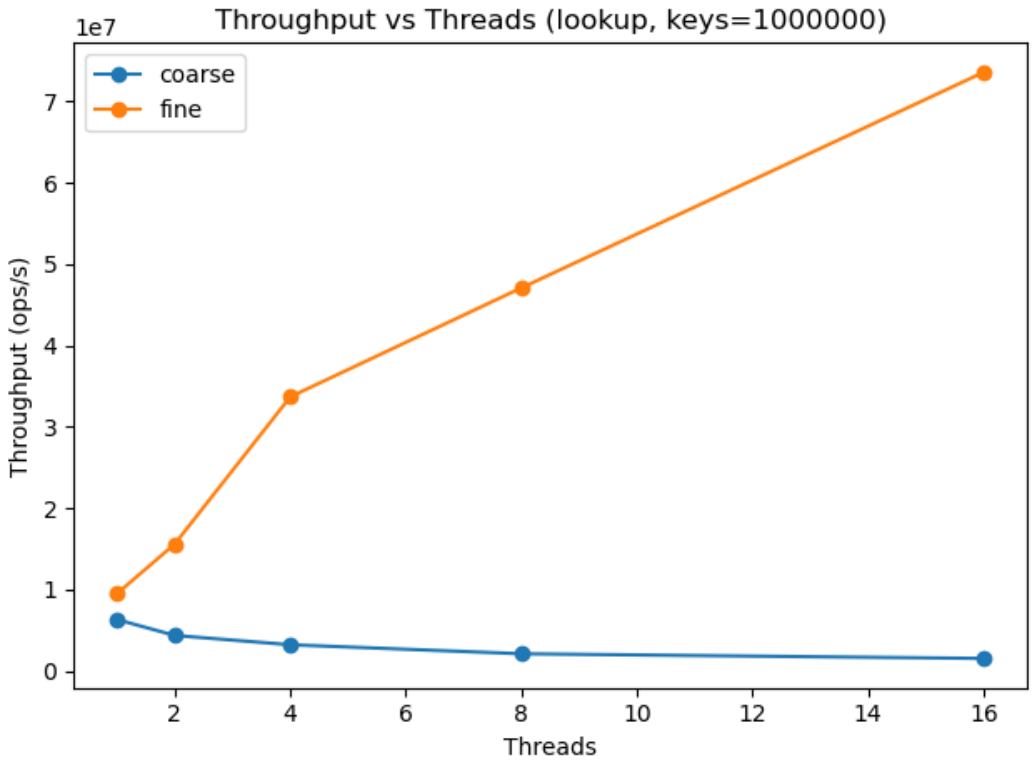

Figure 1: Lookup-only throughput vs threads for coarse-grained and fine-grained locking.

Throughput scaling behavior clearly differentiates the two synchronization strategies. The coarse-grained implementation shows limited improvement beyond one to two threads. As thread count increases, throughput plateaus and eventually degrades slightly. This behavior is caused by contention on the single global mutex, which serializes access and forces threads to wait even when operating on independent keys.

In contrast, the fine-grained implementation continues to scale to higher thread counts. By locking individual buckets rather than the entire table, multiple threads can perform operations concurrently as long as they target different buckets. This significantly reduces contention and allows throughput to increase with available parallelism, especially in lookup-heavy workloads.

Similar trends are observed for insert-only and mixed workloads, although peak throughput is lower due to the additional overhead of memory allocation and writes.

5.2 Insert and Mixed Workloads (Supporting Evidence)

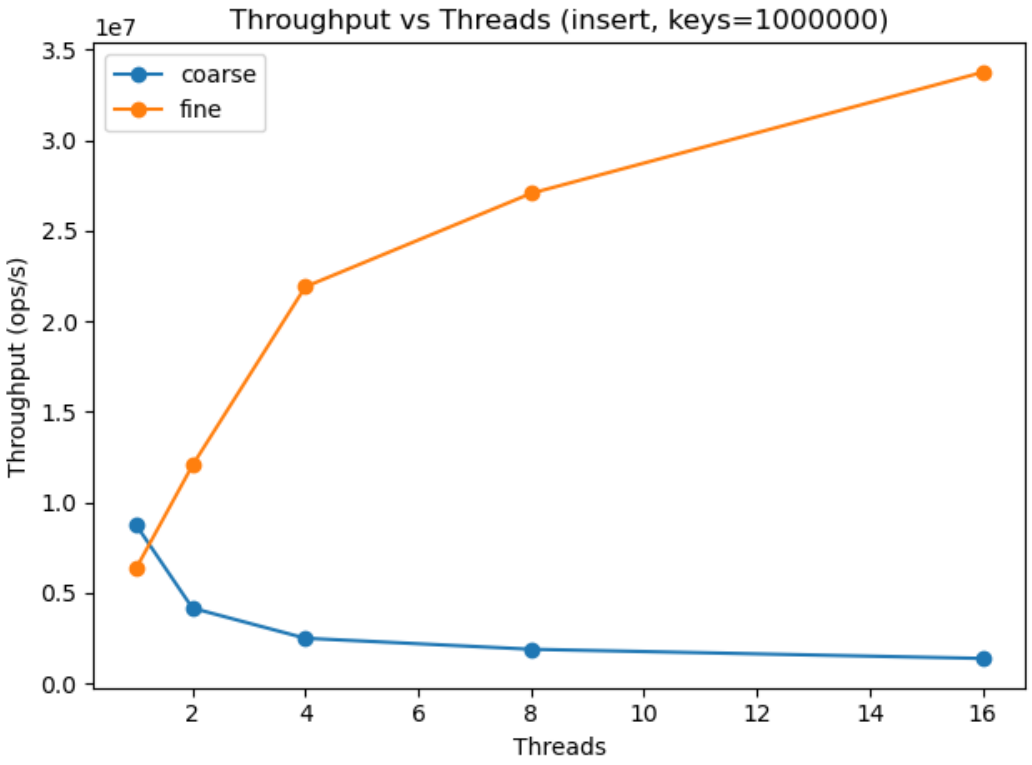

Figure 2: Insert-only throughput vs threads.

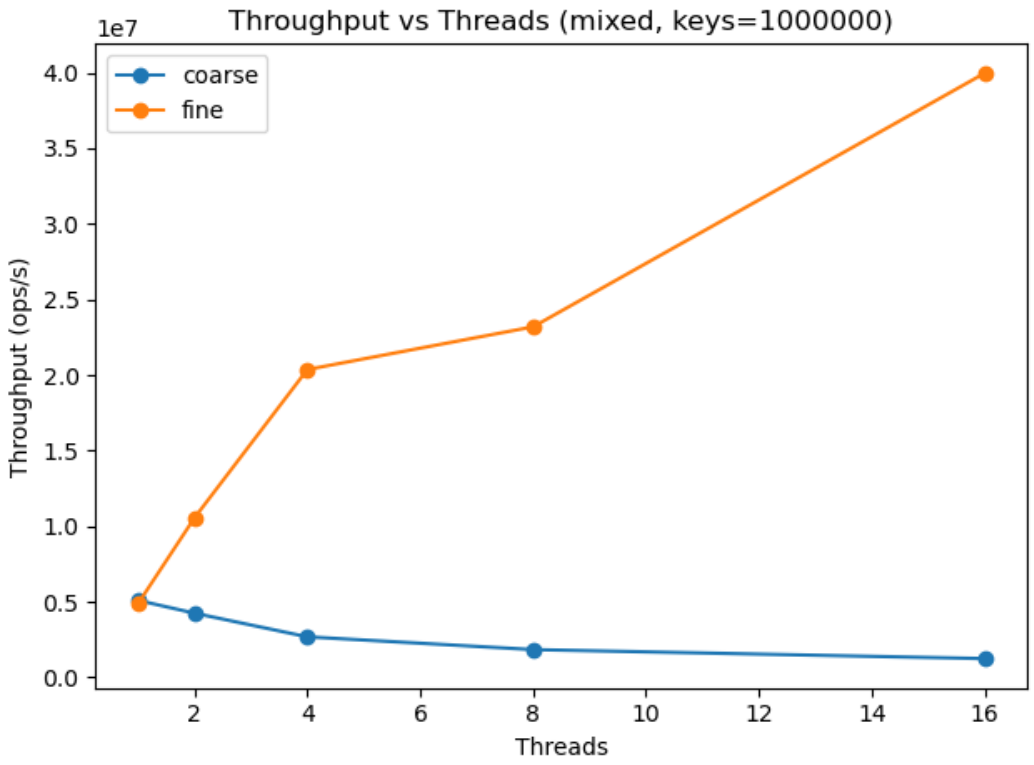

Figure 3: Mixed workload throughput vs threads.

Insert-heavy workloads exhibit weaker scaling for both implementations. Writes introduce additional overhead from dynamic memory allocation and cache-line invalidations, which increase coherence traffic. While fine-grained locking still outperforms coarse locking, the performance gap narrows compared to lookup-only workloads. This highlights that synchronization granularity is necessary but not sufficient to achieve ideal scaling; memory behavior also plays a critical role.

5.3 Speedup Analysis (Confirmation of Scaling Limits)

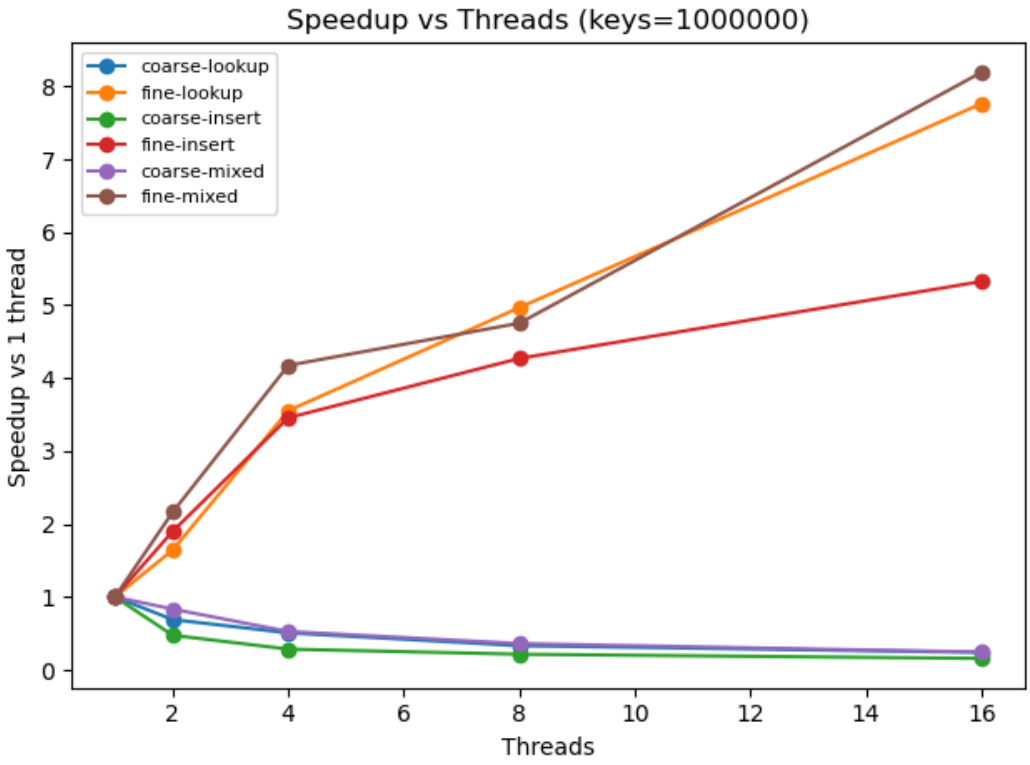

Figure 4: Speedup relative to one thread.

Speedup curves flatten as thread count increases, even for the fine-grained implementation. This behavior is consistent with Amdahl's Law, which states that the maximum achievable speedup is limited by the serial portion of the workload. In this case, serial components include lock acquisition, memory allocation, and unavoidable coherence operations.

The coarse-grained implementation reaches its scaling limit quickly because the global lock dominates execution time as contention increases.

5.4 Estimated Cycles per Operation (Primary Diagnostic)

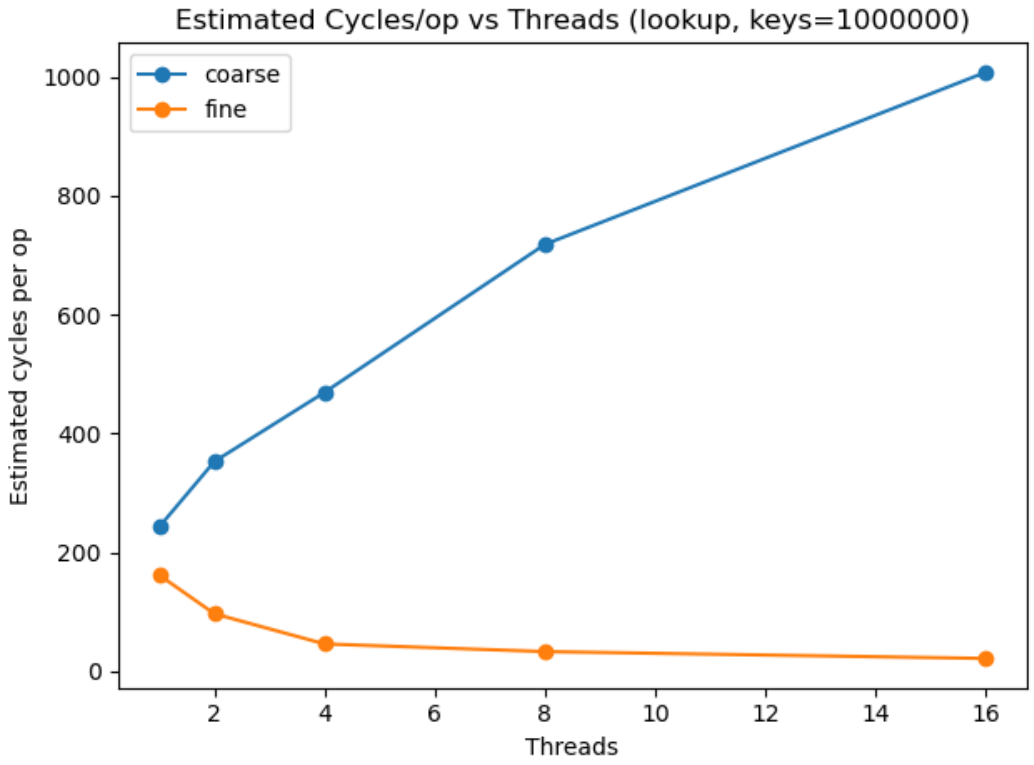

Figure 5: Estimated cycles per operation vs threads for lookup workload.

Estimated cycles per operation provide a clear explanation for the throughput trends. As thread count increases, the coarse-grained implementation shows a steep increase in cycles per operation. This reflects time spent waiting for the global lock rather than performing useful work.

The fine-grained implementation exhibits a slower increase in cycles per operation. Although coherence overhead still grows with thread count, the cost is distributed across multiple locks, reducing the frequency with which threads contend for the same cache line.

Similar trends are observed for insert and mixed workloads, confirming that increased contention and coherence traffic directly translate into higher per-operation cost.

6 Discussion

6.1 Cache Coherence Effects

Locks are shared variables that are frequently written by multiple threads. Each lock acquisition requires exclusive ownership of the cache line containing the lock, triggering coherence traffic between cores. In the coarse-grained design, all threads contend for the same cache line, causing repeated invalidations and ownership transfers.

Fine-grained locking mitigates this effect by spreading lock variables across many cache lines. While coherence traffic still exists, it is distributed, allowing greater parallelism before contention dominates.

6.2 Connection to Amdahl's Law

Even with fine-grained locking, speedup eventually saturates. This reflects the presence of unavoidable serial components, including critical sections, memory allocation, and coherence-induced stalls. As thread count increases, these serial portions dominate execution time, limiting scalability regardless of lock granularity.

7 Conclusion

This project demonstrates that synchronization granularity strongly influences the scalability of concurrent data structures. Throughput and derived cycles-per-operation plots show that coarse-grained locking quickly becomes a bottleneck due to contention and cache-coherence overhead. Fine-grained locking significantly improves scalability by reducing contention and distributing coherence traffic across multiple locks.

Although fine-grained locking does not eliminate all scaling limits, it substantially lowers per-operation cost and delays saturation. By fixing CPU frequency and deriving cycle counts from execution time, the experiments provide clear, reproducible insight into how synchronization design interacts with memory coherence and parallel execution.

A Appendix: Referenced Source Files

A.1 benchmark.c

A.2 hashtable.c

A.3 run_sweep.sh