ACS Final Track A Report

Jaden Tompkins

December 2025

1 Introduction

1.1 Motivation and Problem Statement

This project asks a focused question: for a concurrent hash table on a modern multicore CPU, how much does synchronization granularity (one global lock versus per-bucket locks) affect throughput, speedup, and sensitivity to the cache hierarchy? Previewing the main result, we find that a simple fine-grained lock-striping design delivers up to roughly 49× higher throughput than the coarse-grained baseline at 16 threads.

1.2 Learning Goals and Contributions

This project was designed to: implement a correct thread-safe index, reason about its invariants under concurrency, and explain scaling behavior using contention and cache-coherence principles. Concretely, this report contributes:

- Two hash-table implementations: a simple coarse-grained baseline with a single global mutex, and a fine-grained lock-striping variant with one mutex per bucket.
- A configurable benchmark harness that runs lookup-only, insert-only, and mixed 70/30 workloads over dataset sizes from 10^4 to 10^6 keys and thread counts from 1 to 16.
- A full analysis pipeline that aggregates 5 repetitions per configuration, computes throughput, speedup, and efficiency, and generates publication-quality plots.
- Empirical evidence that fine-grained locking achieves up to roughly 49× higher throughput than coarse-grained locking at 16 threads, and that performance is strongly shaped by cache capacity and memory latency.

2 Background

2.1 Concurrent Hash Tables and Synchronization

The data structure studied here is a separate-chaining hash table that stores $(\text{key}, \text{value})$ pairs in an array of buckets, where each bucket is a singly linked list of nodes. In a shared-memory setting, multiple threads may attempt to insert, find, or erase keys concurrently, so access to shared state must be coordinated. The coarse-grained design enforces mutual exclusion with a `std::mutex` protecting the entire table, while the fine-grained design associates a distinct mutex with each bucket so that independent operations can proceed in parallel.

2.2 Amdahl's Law and Cache Coherence

Amdahl's Law states that the speedup from parallelization is fundamentally limited by the fraction of execution time that remains serial: even a small serial component or synchronization overhead can cap scaling as thread counts grow. On real hardware, additional overhead comes from cache coherence, where shared cache lines move between cores and may be invalidated on writes according to a protocol such as MESI. In the analysis section, these ideas are used to interpret why coarse-grained locking exhibits negative scaling and why fine-grained locking eventually becomes coherence-bound rather than CPU-bound.

3 Design and Implementation

3.1 Data Structure Overview

Both implementations share a common underlying hash table layout. The table contains a fixed-size array of 1024 buckets, each of which points to a singly linked list of nodes storing a 64-bit key, a 64-bit value, and a pointer to the next node. Keys are mapped to buckets using a simple modulo-based hash function, which is sufficient here because the benchmark uses uniformly distributed keys. The table does not resize, so the focus of the study is purely on synchronization and memory behavior rather than on dynamic growth policies.

3.2 Coarse-Grained Locking Baseline

The coarse-grained baseline protects the entire table with a single global `std::mutex`. Every operation (`insert`, `find`, and `erase`) acquires this mutex for the duration of its work, traverses or mutates the appropriate bucket, and then releases the lock. This design is easy to implement and reason about, since all shared state is guarded by one lock and no deadlock is possible. However, it also serializes all operations under heavy load, so we expect intense lock contention, long queues of waiting threads, and poor scalability as the number of threads increases.

3.3 Fine-Grained Lock Striping

The fine-grained variant assigns a separate `std::mutex` to each bucket and acquires only the lock associated with the target bucket for a given key. Because the hash function spreads keys roughly uniformly across buckets, most operations touch different buckets and can proceed fully in parallel. This lock-striping approach dramatically reduces lock contention and expected waiting time while still preserving correctness. The trade-off is a larger number of locks in the table and more complex reasoning about how cache lines containing locks and data move between cores.

3.4 Correctness and Invariants

Both implementations maintain the same logical invariants. For any key, there is at most one node in the table with that key; inserts either add a new node or fail if the key already exists, finds return the most recently inserted value, and erases remove exactly one matching node. Using the shared `HashTable` interface, a suite of unit tests exercises basic operations, duplicate inserts, large bulk insertions, and re-insert after erase. Additional multi-threaded stress tests run mixed workloads at high thread counts to detect deadlocks or data races. Together, these tests provide evidence that both designs are functionally correct before we study their performance.

4 Experimental Methodology

4.1 Hardware and Software Environment

All experiments were run on a single modern laptop-class CPU (AMD Ryzen 9 5900HS, 8 physical cores and 16 hardware threads) using a Linux environment under WSL. The processor has private L1 and L2 caches per core and a shared last-level cache, with DRAM access roughly one to two orders of magnitude slower than L2. Code was compiled with a recent GCC supporting C++17 and optimization flags such as `-O3` and `-pthread`. To reduce noise, background applications were minimized and the CPU frequency governor was configured for consistent performance where possible.

4.2 Workloads and Parameters

The benchmark suite exercises three representative workloads over a shared hash table instance. The *lookup-only* workload performs 100% `find` operations against a pre-populated table, modeling read-heavy query patterns. The *insert-only* workload performs 100% `insert` operations starting from an empty or partially filled table, stressing write paths and allocation. The *mixed 70/30* workload chooses `find` 70% of the time and `insert` 30% of the time. For each workload, we vary the dataset size from 10^4 to 10^6 keys (with additional points at 50K and 500K) and the thread count from 1 to 16, running 5 repetitions per configuration and recording median throughput in operations per second.

In addition to timing-based throughput measurements, a wrapper script optionally invokes the Linux `perf stat` tool to record hardware performance counters including cycles, retired instructions, and last-level-cache load and store misses for each benchmark run. While the main analysis in this report focuses on throughput, speedup, and efficiency, these counters corroborate the qualitative story: under coarse-grained locking, high-thread-count runs exhibit substantially more LLC misses and cycles per successful operation than the fine-grained design, consistent with lock contention and cache-line bouncing on the global mutex.

4.3 Benchmark Harness

The benchmark driver constructs either the coarse or fine-grained table via a common factory interface and configures a `WorkloadConfig` describing the dataset size, number of operations, thread count, and workload type. A pool of worker threads is then launched, each repeatedly performing operations over a shared vector of random keys. Timing starts immediately before the workers begin and stops after all threads have joined, so each run captures end-to-end throughput for that configuration. Helper shell and Python scripts enumerate the full grid of strategies, workloads, thread counts, and dataset sizes, and write per-run results to CSV files in a consistent format.

4.4 Analysis and Plotting Pipeline

Raw CSV output from the benchmark harness is merged into a single "long" dataset containing strategy, workload, thread count, dataset size, repetition number, and measured throughput. Analysis scripts then compute per-configuration statistics such as the median, mean, standard deviation, and speedup relative to the one-thread baseline, writing the aggregates to `statistics.csv` and `speedup.csv`. Finally, the plotting script reads these aggregate files and generates the figures included in this report, so all plots can be reproduced automatically from the raw measurements.

Note: Because each individual run has a timeout, large configurations can expire before completion. To mitigate this, I first swept with a shorter timeout, saved all successful runs, and then selectively re-ran only the missing configurations with a longer timeout. Some coarse-grained runs with the largest dataset sizes were projected to take on the order of ten hours and were therefore omitted; instead, I focused analysis on smaller dataset sizes where measurements were practical.

5 Results

5.1 Throughput vs Threads (Fine-Grained)

The first set of results looks at absolute throughput of the fine-grained implementation as we increase the number of threads for a large dataset that does not fit in cache. This highlights both the benefit of adding cores and the point at which coherence and memory bandwidth begin to limit performance. This shows performance on full dataset size, with performance increasing as parallelization increases, and will later be compared to course-grained tables on a smaller dataset.

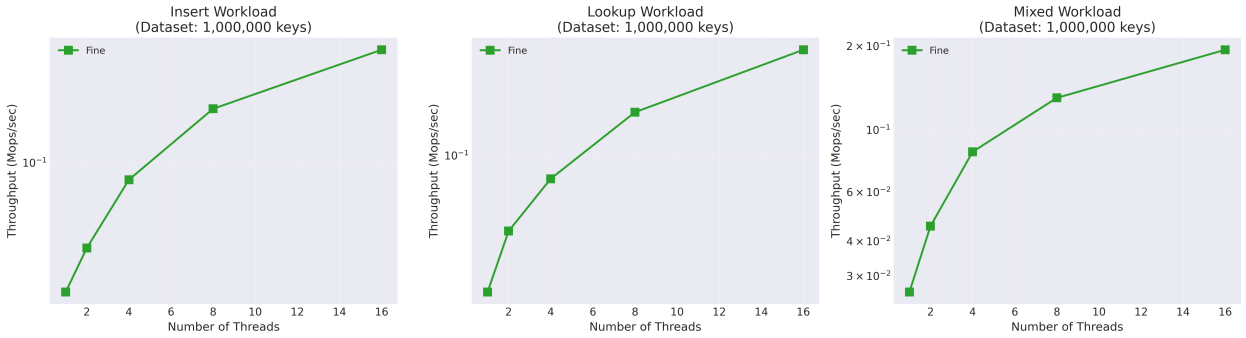

Figure 1: Throughput of the fine-grained hash table as a function of thread count for each workload at 1,000,000 keys.

5.2 Speedup and Amdahl's Law

Measured speedup relative to a single thread makes it easier to compare behavior to the ideal linear baseline predicted by Amdahl's Law. In particular, we can estimate the effective serial fraction of the fine-grained implementation and observe where additional threads provide diminishing returns. For example, for the fine-grained lookup workload at 100,000 keys, the measured 16-thread speedup of about 6.4 implies an effective serial fraction of roughly 10% via Amdahl's formula

$$S(N) = \frac{1}{s + \frac{1-s}{N}},$$

where $S(N)$ is speedup at N threads and s is the serial fraction. Solving this equation for s using the measured $S(16)$ yields $s \approx 0.10$, which in turn predicts a whole family of speedups $S(N)$ that closely track the measured values across 2, 4, 8, and 16 threads.

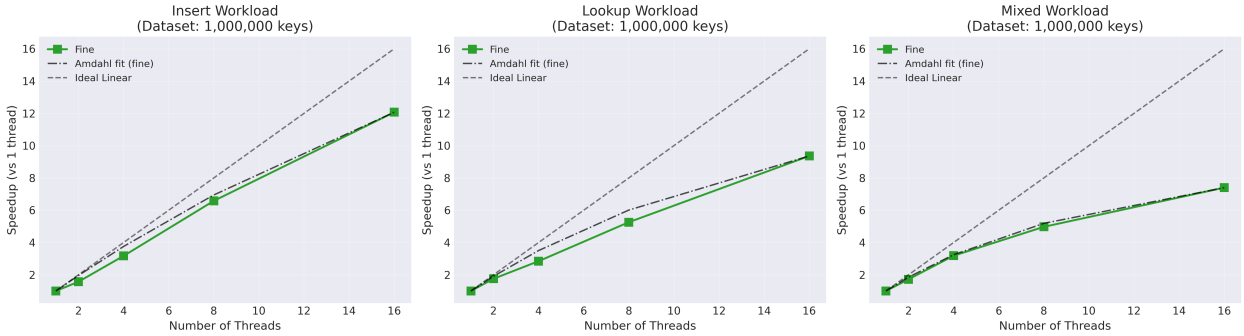

Figure 2: Speedup of the coarse- and fine-grained implementations relative to a single thread at 1,000,000 keys, with both an ideal linear speedup line and an Amdahl-law fit for the fine-grained curves for reference.

Table 1 summarizes this comparison for the fine-grained lookup workload at 100,000 keys. The measured speedup and efficiency columns are derived directly from the benchmark data, while the Amdahl-fit column shows the theoretical speedup obtained by plugging the fitted serial fraction ($s \approx 0.10$) back into the equation above. The close agreement supports the interpretation that the remaining gap to ideal linear scaling is largely due to an irreducible serial component together with hardware limits such as cache coherence and memory bandwidth.

Table 1: Measured speedup and efficiency for the fine-grained lookup workload at 100,000 keys, compared to an Amdahl-law fit with an effective serial fraction of $s \approx 0.10$.

Threads	Measured Speedup	Efficiency (%)	Amdahl Fit $S(N)$
1	1.00	100.00	1.00
2	1.62	81.00	1.82
4	3.27	82.00	3.08
8	5.18	65.00	4.71
16	6.41	40.00	6.40

5.3 Workload Comparison

To understand how read-heavy, write-heavy, and mixed workloads behave under the same hardware and table implementation, we compare throughput for each workload at a fixed thread count and dataset size. This helps explain which microarchitectural costs dominate for different mixes of operations. As seen in the figure, insert-only leads to the highest throughput, followed by lookup, with mixed as the lowest throughput.

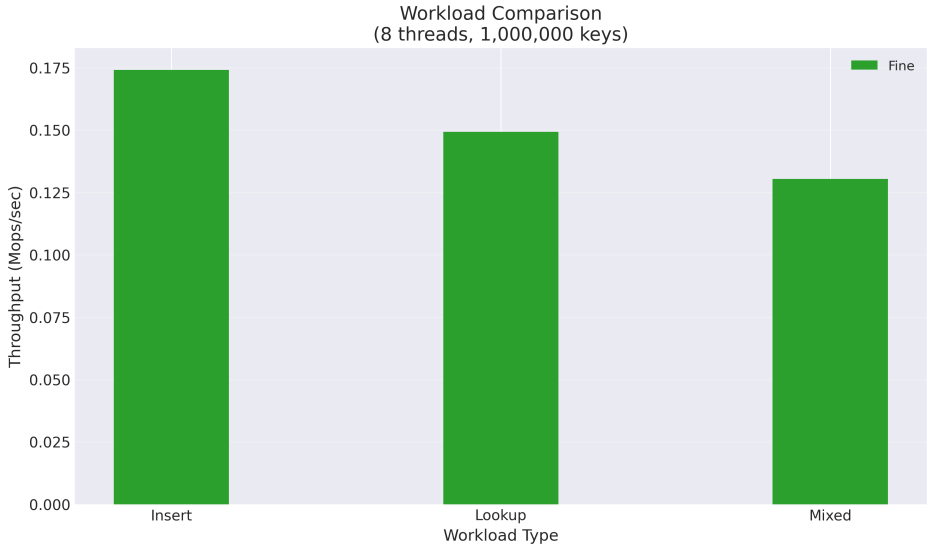

Figure 3: Throughput comparison across workloads for the fine-grained implementation at 8 threads and 1,000,000 keys.

To see how these workload differences evolve as the dataset size grows, we also sweep the number of keys on the x-axis and plot throughput for each combination of strategy and workload on a single set of axes. This produces six curves (coarse/fine for lookup, insert, and mixed), making it easy to see both the impact of the cache hierarchy and the persistent performance gap between coarse- and fine-grained locking. Note the difference in y-axis scaling.

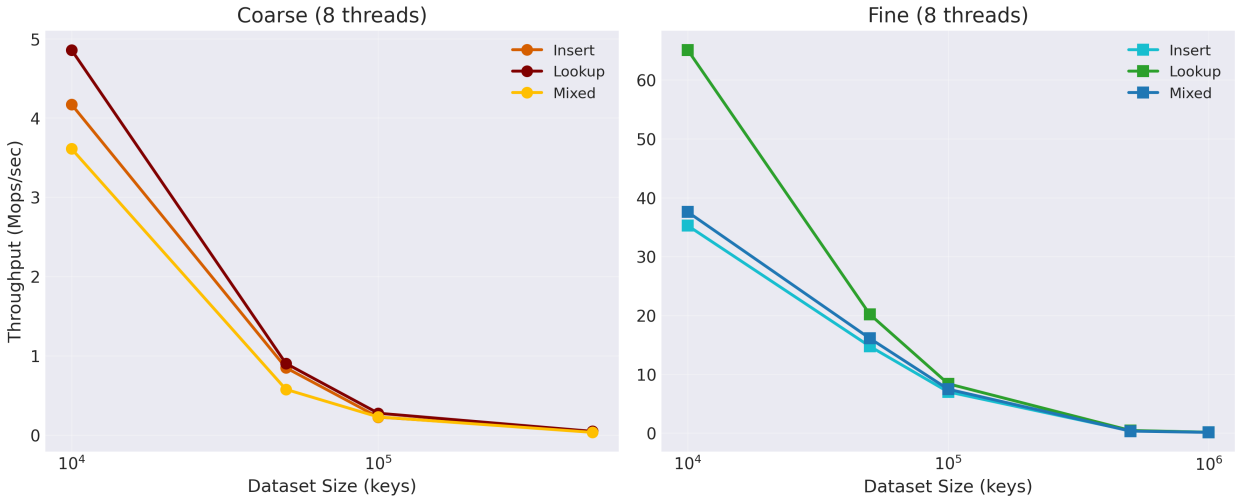

Figure 4: Throughput vs. dataset size (10K–1M keys, log-scale x-axis) at 8 threads, split into separate panels for coarse (left) and fine (right). Within each panel, three colored curves distinguish lookup, insert, and mixed workloads.

.

5.4 Dataset Size Sensitivity and Cache Hierarchy

Here we fix the number of threads and vary the dataset size to see how performance degrades as the working set grows beyond private caches into the shared last-level cache and finally into DRAM. Plotting both strategies on the same axes makes it clear that the memory hierarchy affects them similarly in absolute terms even though fine-grained locking maintains a large relative advantage.

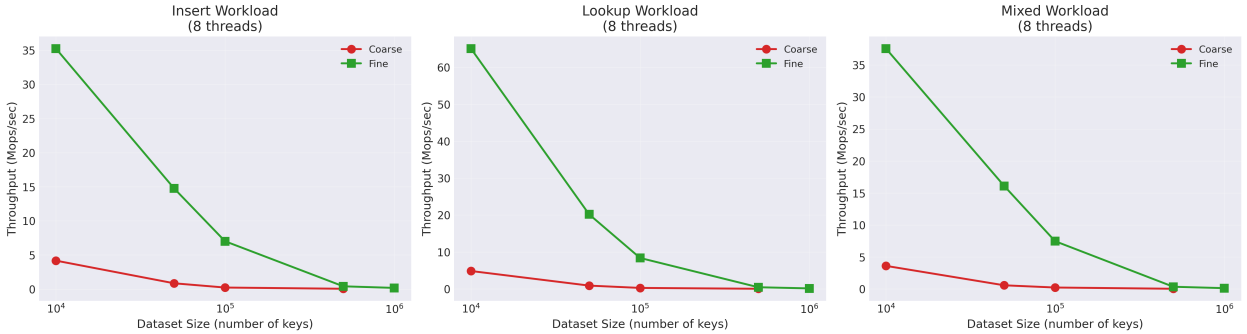

Figure 5: Throughput vs. dataset size (10K–1M keys, log-scale x-axis) for coarse- and fine-grained locking at 8 threads.

5.5 Parallel Efficiency and SMT Effects

Parallel efficiency normalizes speedup by the number of threads and answers the question "how much useful work do we get per thread?" This view is particularly helpful for understanding when the system is core-limited versus coherence- or memory-limited, and how much benefit simultaneous multithreading (SMT) provides beyond 8 hardware cores. The decreased efficiency of the mixed workload supports the results seen earlier on the workload comparison at 8 threads, and it can be reasoned that the workload comparison would further differentiate as thread count increases due to this inability to take full advantage of the parallelization.

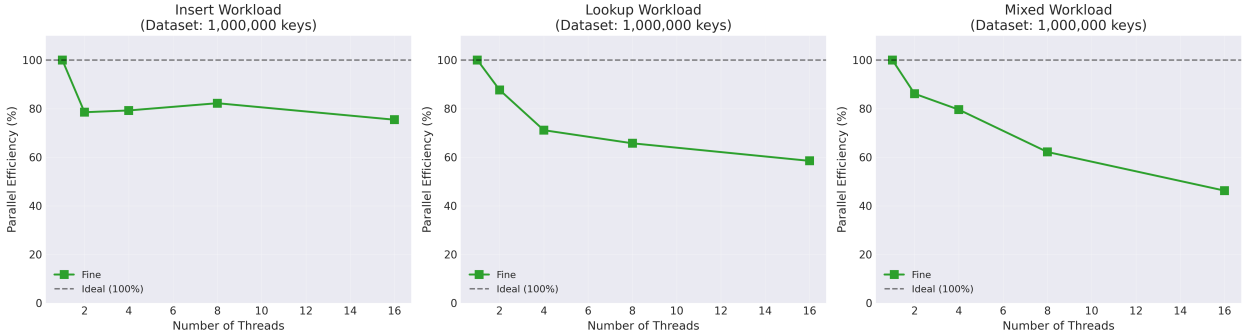

Figure 6: Parallel efficiency of the fine-grained implementation vs. thread count at 1,000,000 keys for each workload. This figure should be used to argue where the system is core-limited vs coherence/memory-limited and to interpret SMT gains.

5.6 Direct Coarse vs Fine Scaling at 100K Keys

To directly compare the two synchronization strategies, we fix the dataset size to a moderate value (100,000 keys) where both complete successfully and plot throughput versus thread count for

each workload. This visualization makes the negative scaling of the coarse-grained design and the healthy scaling of the fine-grained design immediately apparent, and showcases the clear benefit of a fine-grained approach in parallel use cases.

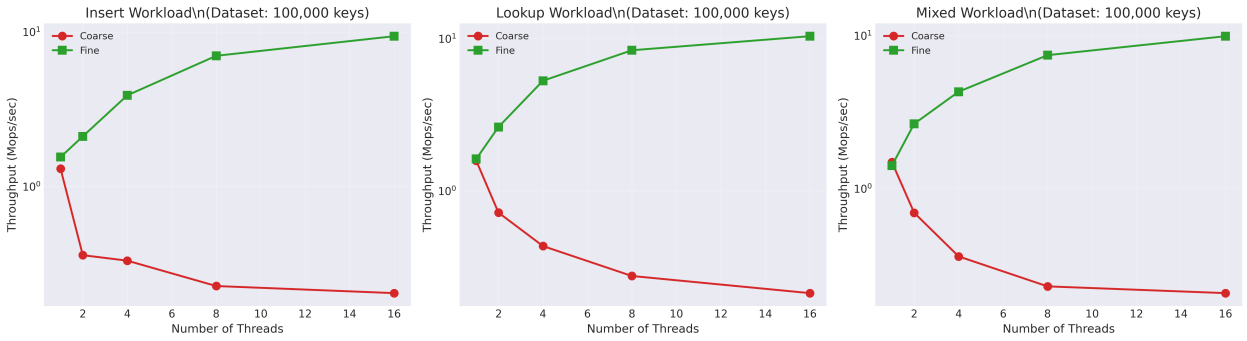

Figure 7: Throughput vs. thread count for coarse- and fine-grained locking at 100,000 keys across all three workloads.

5.7 Direct Coarse vs Fine Scaling at 500K Keys

While 100,000 keys primarily exercises the shared last-level cache, a dataset of 500,000 keys pushes the working set closer to the LLC/DRAM boundary. Repeating the coarse vs fine comparison at this larger size shows that the coarse-grained design slows even further, while the fine-grained design continues to deliver substantial speedup: for example, in the mixed 70/30 workload at 16 threads, fine-grained locking reaches roughly 0.5 Mops/sec compared to about 0.03 Mops/sec for coarse-grained locking, a 17× advantage. This figure complements the 100K comparison by demonstrating that the qualitative gap between the two strategies persists even as memory bandwidth and cache capacity become the dominant bottlenecks. It also shows how, on a large enough dataset, when working without parallelization, the additional overhead of tracking the invariants causes a noticeable decrease in throughput for the fine-grained approach.

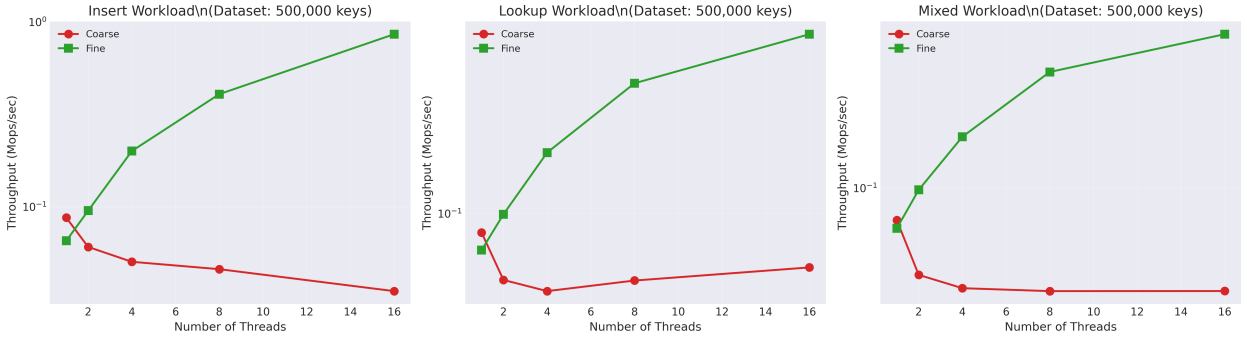

Figure 8: Throughput vs. thread count for coarse- and fine-grained locking at 500,000 keys across all three workloads.

5.8 Coarse vs Fine Summary at 100K Keys

Table 2 summarizes the headline comparison for 100,000 keys and 16 threads, highlighting just how different the two designs behave under heavy contention. These numbers are drawn from the median throughput across five repetitions per configuration.

Table 2: Throughput (Mops/sec) of coarse- vs. fine-grained locking at 100,000 keys and 16 threads.

Workload	Coarse (Mops/s)	Fine (Mops/s)	Fine / Coarse
Lookup	0.21	10.38	48.90
Insert	0.20	9.39	46.20
Mixed 70/30	0.21	9.94	48.20

For a larger dataset that lies closer to the LLC/DRAM boundary, Table 3 provides an analogous summary at 500,000 keys and 16 threads. Even though absolute throughput is much lower in this DRAM-heavy regime, fine-grained locking still outperforms coarse-grained locking by roughly one to two orders of magnitude.

Table 3: Throughput (Mops/sec) of coarse- vs. fine-grained locking at 500,000 keys and 16 threads. This table shows that fine-grained locking maintains a large throughput advantage even when the working set size approaches or exceeds the last-level cache.

Workload	Coarse (Mops/s)	Fine (Mops/s)	Fine / Coarse
Lookup	0.05	0.81	15.30
Insert	0.04	0.86	24.40
Mixed 70/30	0.03	0.52	15.60

6 Analysis and Discussion

6.1 Why Coarse-Grained Locking Fails to Scale

The coarse-grained implementation forces all operations to acquire a single global mutex, so at most one thread can make progress at a time. As the number of threads grows, more time is spent waiting on the lock, the operating system performs more context switches, and the cache line containing the mutex bounces rapidly between cores. The measurements in Figure 7 and Table 2 show that these overheads more than offset any potential parallelism, leading to speedup well below 1 and even making the 16-thread configuration slower than a single thread. Hardware performance counters collected with `perf stat` reinforce this picture: at high thread counts, coarse-grained runs consume many more last-level-cache load and store misses and cycles per successful operation than their fine-grained counterparts, indicating that much of the extra work goes into servicing coherence traffic and repeatedly acquiring the global lock rather than performing useful hash-table operations.

Table 4 provides concrete snapshots of this gap for the lookup workload at 50,000 and 500,000 keys (8 threads). Even though the WSL environment used here does not expose last-level-cache miss events, the available counters already show a large disparity in work per operation: at 50K keys the coarse-grained run uses roughly 4.6 times as many cycles and 3.5 times as many instructions per successful lookup as the fine-grained run, while at 500K keys both strategies become much more expensive in absolute terms but fine-grained locking still cuts cycles and instructions per operation by about a factor of two and maintains an order-of-magnitude throughput advantage.

Table 4: Perf-stat snapshots for the lookup workload at 50,000 and 500,000 keys with 8 threads (WSL2 environment). Throughput is reported by the benchmark, while cycles/op and instructions/op are derived from `perf stat` counters. LLC miss events are not supported by this kernel and are therefore omitted.

Dataset	Strategy	Throughput (Mops/s)	Cycles/op	Instructions/op
50K	Coarse	1.26	5800.00	1167.00
50K	Fine	16.70	1260.00	337.00
500K	Coarse	0.04	69 760.00	3858.00
500K	Fine	0.56	35 400.00	1738.00

6.2 Why Fine-Grained Locking Succeeds (and Its Limits)

In contrast, the fine-grained design distributes synchronization across 1024 independent bucket locks. With uniformly hashed keys and up to 16 threads, the probability that two threads need the same lock at the same time is small, so most operations proceed without blocking. This yields strong speedup and respectable efficiency up to 8 threads, as seen in the throughput and speedup plots. Beyond that point, however, additional threads primarily increase cache-coherence traffic and contention for memory bandwidth, so performance gains taper off and efficiency falls to around 40–60% at 16 threads. The probability that two threads contend for the same bucket lock also increases, although this effect remains relatively small at this scale.

6.3 Cache Hierarchy, Dataset Size, and the Memory Wall

The dataset-size sensitivity plot reveals sharp drops in throughput as the working set moves from L2 into the shared last-level cache and eventually into DRAM. A simple size estimate for the table shows that 10K keys fit comfortably in L2, 50K–100K keys primarily occupy the LLC, and 500K–1M keys exceed the LLC, forcing most accesses to go to DRAM. The roughly 10–60$\times$ slowdowns observed at these transition points, together with the additional 500K comparison in Figure 8, are consistent with the known latency gaps between cache levels and main memory, and they affect both strategies similarly in absolute terms even though fine-grained locking maintains a large relative advantage.

6.4 Workload Characteristics and Surprising Trends

Different workloads stress different parts of the system. Lookup-only runs tend to be dominated by read latency and cache locality, while insert-heavy runs add write traffic, allocation, and potential cache line invalidations. The results show that, somewhat surprisingly, insert workloads can achieve comparable or even better speedup than lookups at the largest dataset sizes, likely because the cost of individual operations is dominated by DRAM latency and synchronization overhead becomes a smaller fraction of total time. The mixed 70/30 workload usually achieves throughput slightly below lookup-only and comparable to insert-only, but still benefits substantially from fine-grained locking.

6.5 Threats to Validity

Several factors limit the generality of these results. All experiments were run on a single CPU microarchitecture and memory subsystem, so systems with different cache hierarchies or NUMA

layouts may behave differently. The hash table uses a fixed number of buckets, a simple modulo hash, and uniformly distributed keys, which avoids skew and hot spots that might arise in real workloads. The table also does not resize, so we do not study rehashing costs. Finally, some coarse-grained configurations with the largest dataset sizes time out or take impractically long to complete, so those points are excluded from the final plots; this biases comparisons against coarse-grained locking but reflects that such configurations are not viable in practice.

7 Conclusion and Future Work

In summary, the experiments confirm that synchronization strategy is the dominant factor in the scalability of this concurrent hash table. A coarse-grained global lock leads to negative scaling and throughput that decreases as threads are added, while a fine-grained lock-striping design achieves robust speedup up to 8 threads and still delivers substantial gains at 16 threads despite coherence and memory limitations. The cache hierarchy imposes large absolute slowdowns as the dataset size grows, but does not change the qualitative ordering between strategies. Looking forward, the same benchmarking and analysis framework could be used to evaluate reader–writer locks for read-heavy workloads, explore lock-free or wait-free designs that further reduce coherence overhead, or apply similar techniques to other data structures such as trees or queues.