

Concurrent Sorted Linked List: Synchronization Techniques in Parallel Computing

Mayas Alkhateeb

November 2025

1 Introduction

This report documents the implementation of the "Concurrent Sorted Linked List" assignment, a parallel computing exercise designed to illustrate synchronization techniques in multi-threaded Java applications. The project involves creating a thread-safe sorted linked list that supports add, remove, and contain operations without duplicates, using sentinel nodes for boundaries. Multiple synchronization methods are implemented and benchmarked under concurrent workloads to compare performance.

The assignment, provided in the course materials from the Higher Institute for Applied Sciences and Technology (HIAST), focuses on key concurrency principles such as coarse-grained locking, read-write locks, and fine-grained object-based isolation. The implementation includes bug fixes, size computation, and a testing framework to measure execution times and correctness.

The source code is available on GitHub: <https://github.com/k3rnel-paN1c5/ConcurrentSortedList>

2 Assignment Requirements

The assignment requires implementing the following components:

- **SortList Abstract Class:** Defines the interface for add, remove, contain, and getSize operations, with sentinel nodes (MIN_VALUE and MAX_VALUE).
- **SyncList:** Uses synchronized methods for coarse-grained synchronization.
- **LockList:** Uses a single ReentrantLock for coarse-grained synchronization.
- **RWLockList:** Initially uses a global ReentrantReadWriteLock (coarse-grained), then modified to per-node ReentrantReadWriteLock with hand-over-hand locking for fine-grained object-based isolation.
- **Testing Framework:** Benchmark with 40..501 random operations (seed=0, range=80,000), using 7 threads for add, 8 for contain, and 8 for remove. Measure times, list lengths, and success/failure counts.

Expected behavior:

- The list remains sorted and duplicate-free under concurrent access.
- Benchmarks produce consistent metrics across runs, with varying times due to synchronization overhead.
- Fine-grained RWLock improves concurrency for read-heavy operations.

3 Implementation Details

The project builds on the provided base code, adding features like thread-safe size computation and operation success counters.

3.1 Code Structure

The project is organized as follows:

- **src/**: Contains core classes for the list implementations and entry node.
- **test/**: Contains thread classes for testing and the benchmark runner.
- **SyncListTest.java**: Entry point for running benchmarks.

Key classes:

- **Entry.java**: Node class with object, next, and (for fine-grained RWLock) a per-node ReentrantReadWriteLock.
- **SortList.java**: Abstract base with sentinels and method declarations.
- **SyncList.java**: Synchronized methods for add/remove/contain/getSize.
- **LockList.java**: Uses a global ReentrantLock.
- **RWLockList.java**: Fine-grained version with hand-over-hand locking (read locks for contain/getSize, write locks for add/remove).
- **TestThread.java**: Abstract base for add/contain/remove threads, with success counters.
- **AddThread.java, ContainThread.java, RemoveThread.java**: Operation-specific threads.
- **RandomSeq.java**: Shared random number generator.
- **SyncListTest.java**: Benchmark runner with uneven partitioning for threads.
- **BenchmarkDriver.java**: Benchmark runner with different number of threads and list sizes.

3.2 Threading Concepts Demonstrated

- **Coarse-Grained Synchronization**: In SyncList and LockList, a single lock protects the entire list, simple but limits concurrency.
- **Read-Write Locking**: In coarse RWLockList, allows multiple contains simultaneously but exclusive adds/removes.
- **Fine-Grained Locking**: Per-node locks in modified RWLockList enable overlapping traversals, reducing contention.
- **Hand-Over-Hand Locking**: Prevents deadlocks by acquiring locks in order during traversal.
- **Shared Resources**: Threads share the list and random sequence, with counters for successes.

3.3 Extensions and Enhancements

- Added getSize() with thread-safety.
- Implemented success counters in test threads.
- Used uneven partitioning for non-divisible thread counts (e.g., 50,000 / 7).
- Ran two benchmarks: one with coarse RWLock, one with fine-grained.

4 How to Run

1. Clone the repository: `git clone https://github.com/k3rn3l-paN1c5/ConcurrentSortedList`
2. Compile: `javac *.java` (or use IDE like IntelliJ).
3. Run: `java SyncListTest` and invoke `testRun()` (or run via JUnit/IDE).

The benchmark outputs times, lengths, and counts to console, matching the assignment's expected format.

5 Observations and Learnings

Two benchmark runs were conducted:

- First: RWLockList as coarse-grained.
- Second: RWLockList as fine-grained.

Metrics (lengths, successes) are consistent, confirming thread-safety. Times vary due to hardware and JVM.

Table 1: Execution Times (First Run - Coarse-Grained RWLock)

Implementation	Add (ms)	Contain (ms)	Remove (ms)
Synchronization	4036	8064	6268
RWLock (Coarse)	4712	1112	1624
Lock	4256	8302	7206

Table 2: Execution Times (Second Run - Fine-Grained RWLock)

Implementation	Add (ms)	Contain (ms)	Remove (ms)
Synchronization	5710	11336	8245
RWLock (Fine)	2370	3427	3293
Lock	5431	11480	8269

List length after add: 31851. Successes found: 16282 (failures: 24219). Length after remove: 19212. Successes removed: 12639 (failures: 27862).

Fine-grained RWLock shows improved add/remove times in the second run due to reduced contention, though contain may have overhead from per-node locks.

This exercise reinforced Java concurrency tools (locks, RW locks), importance of fine-grained synchronization for scalability, and benchmarking techniques.

5.1 Scalability Analysis (List Size: 50,000)

To analyze how the implementations handle increasing contention, a stress test was conducted with a list size of 50,000. Table 3 details the execution times across varying thread counts, ranging from sequential execution (1 thread per type) to high contention (16 threads per type).

Table 3: Performance Comparison under Increasing Contention (Size = 50,000)

Implementation	Threads (Add/Cont/Rem)	Add (ms)	Contain (ms)	Remove (ms)
<i>Low Contention</i>				
Synchronization	1 / 1 / 1	6,980	13,256	9,777
Lock (Reentrant)	1 / 1 / 1	7,010	15,738	11,815
RWLock (Fine-Grained)	1 / 1 / 1	10,246	21,809	12,609
<i>Medium Contention</i>				
Synchronization	4 / 4 / 4	7,180	16,325	12,407
Lock (Reentrant)	4 / 4 / 4	7,230	13,689	10,019
RWLock (Fine-Grained)	4 / 4 / 4	4,586	10,861	7,438
<i>Medium-High Contention</i>				
Synchronization	7 / 8 / 8	7,187	14,891	10,085
Lock (Reentrant)	7 / 8 / 8	7,153	14,178	10,066
RWLock (Fine-Grained)	7 / 8 / 8	5,140	4,740	5,133
<i>High Contention</i>				
Synchronization	16 / 16 / 16	7,345	14,989	10,454
Lock (Reentrant)	16 / 16 / 16	7,587	15,627	10,895
RWLock (Fine-Grained)	16 / 16 / 16	2,176	3,272	1,936

Discussion: The Crossover Point Table 3 clearly illustrates the trade-off between synchronization overhead and concurrency benefits (Shown in Figure 1).

At **Low Contention (1/1/1)**, the Fine-Grained RWLock is the slowest implementation (Add: 10,246 ms) due to the overhead of acquiring and releasing locks for every node traversal. In contrast, the coarse-grained methods are faster because they only acquire a single lock per operation.

However, a **crossover point** occurs around the **4/4/4 thread configuration**. Here, the fine-grained RWLock begins to outperform the coarse-grained approaches (Add: 4,586 ms vs. $\approx 7,200$ ms). The coarse-grained locks saturate the CPU with thread contention, showing almost no performance gain as threads increase.

As the load increases to **7/8/8** and **16/16/16**, the benefits of fine-grained locking become exponential, particularly for read-heavy operations. For example, the RWLock *Contain* time drops from 10,861 ms (at 4 threads) to 3,272 ms (at 16 threads), proving that fine-grained Read-Write locks successfully enable parallel readers and non-overlapping writers.

6 Discussion

Some points are to be mentioned:

1. Compare performance of synchronization methods.

Coarse-grained (Sync/Lock) have similar times, with bottlenecks in high contention. Coarse RWLock excels in contains due to concurrent reads. Fine-grained RWLock further improves writes by localizing locks.

2. Why use hand-over-hand in fine-grained locking?

It ensures consistent views during traversal, preventing concurrent modifications from causing inconsistencies or skips, while allowing progress without a global lock.

3. What happens with high contention (e.g., many adds)?

Coarse methods serialize all operations, leading to longer waits. Fine-grained allows non-overlapping operations to proceed, but hand-over-hand may cause lock convoys if many threads traverse simultaneously.

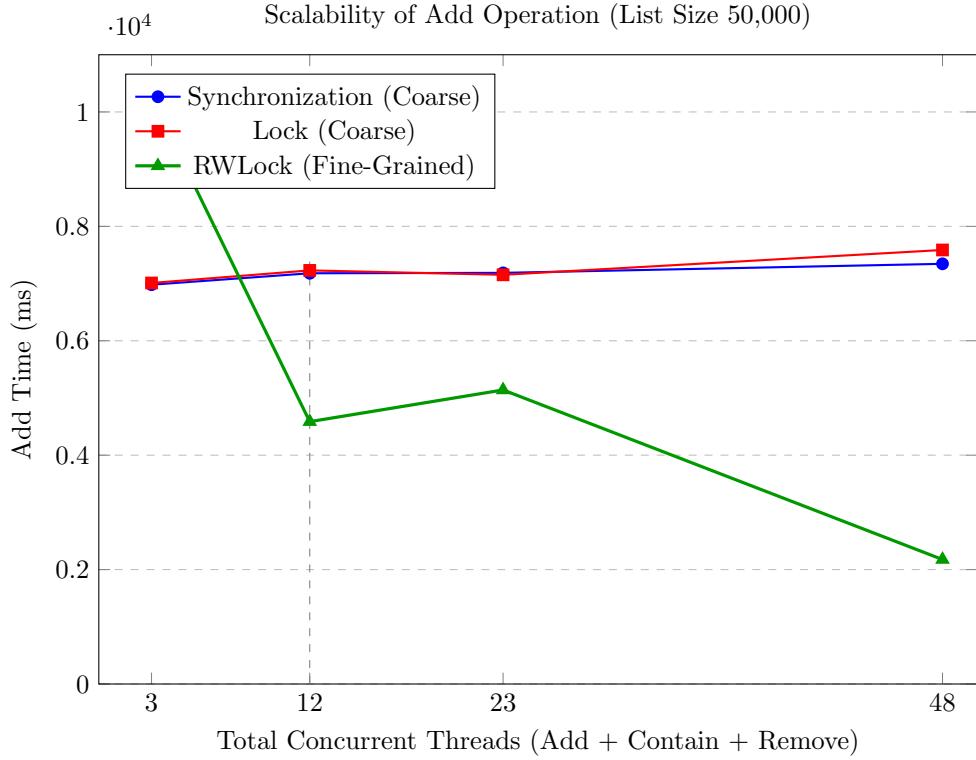


Figure 1: Performance Scalability of List Add Operations. The **crossover point** is evident between 3 and 12 total threads, where the synchronization overhead of the Fine-Grained RWLock is overcome by its superior parallel processing capability under high contention.

4. Security/robustness risks in concurrent lists?

Without proper locking, races can cause lost updates, duplicates, or unsorted states. Fine-grained reduces risks but requires careful deadlock avoidance.

5. Does fine-grained guarantee better performance?

No—overhead from multiple lock acquires can worsen low-contention scenarios. It's OS/JVM-dependent; benchmarks show variability between runs.

7 Conclusion

The "Concurrent Sorted Linked List" assignment provides a practical exploration of synchronization in parallel computing. By implementing and benchmarking different techniques, it highlights trade-offs in simplicity, performance, and scalability. Future work could extend to lock-free implementations or distributed systems.