CS4532 Concurrent Programming

Take-Home Lab 1

Fernando T.H.L (210167E)
Gamage M.S (210176G)

September 4, 2025

System Information

CPU

- Model: Intel(R) Xeon(R) Processor @ 2.30GHz
- Vendor/Arch: GenuineIntel / x86-64
- Physical cores: 4
- Threads per core: 1
- Caches:
  - L1i: 128 KiB (4 instances)
  - L1d: 128 KiB (4 instances)
  - L2: 1 MiB (4 instances)
  - L3: 45 MiB (1 instance)

Memory / NUMA

- Total (kB): 8150140
- NUMA nodes: 1
- THP: madvise
- Swap total (kB): 0

Operating System

- Distro: Ubuntu 24.04.2 LTS
- Kernel: 6.8.0
- Logical CPUs: 4

Toolchain

- Compiler: gcc (Ubuntu 13.3.0-6ubuntu2 24.04) 13.3.0
- make: GNU Make 4.3
- glibc: glibc 2.39
- libpthread (NPTL): NPTL 2.39
- Python: 3.12.11
- pandas: 2.3.2
- matplotlib: 3.10.6

Approach

We implemented a singly linked list supporting:

- `Member`
- `Insert` (unique keys only)
- `Delete`

Three variants were tested:

- Serial (no locks)
- Pthreads + single mutex
- Pthreads + single read–write lock

Initialization: $n = 1000$ unique keys in $[0, 2^{16} - 1]$. Workloads: $m = 10000$ operations with given fractions, distributed across $T \in \{1, 2, 4, 8\}$ threads. Timing measures only the m-operations region, not initialization.

Experiment Report (Overview Tables)

Case 1: n=1000, m=10000, m_member=0.99, m_insert=0.005, m_delete=0.005

Threads	Serial (s)	Mutex (s)	RW-lock (s)
1	0.0096 ± 0.0003	0.0109 ± 0.0006	0.0109 ± 0.0008
2	0.0100 ± 0.0003	0.0324 ± 0.0058	0.0132 ± 0.0008
4	0.0095 ± 0.0004	0.0304 ± 0.0007	0.0164 ± 0.0021
8	0.0100 ± 0.0005	0.0345 ± 0.0012	0.0193 ± 0.0011

Table 1: Summary of results for Case 1.

Case 2: n=1000, m=10000, m_member=0.90, m_insert=0.05, m_delete=0.05

Threads	Serial (s)	Mutex (s)	RW-lock (s)
1	0.0197 ± 0.0005	0.0207 ± 0.0006	0.0211 ± 0.0006
2	0.0199 ± 0.0004	0.0420 ± 0.0050	0.0812 ± 0.0075
4	0.0206 ± 0.0007	0.0470 ± 0.0020	0.0925 ± 0.0100
8	0.0212 ± 0.0013	0.0510 ± 0.0028	0.0879 ± 0.0117

Table 2: Summary of results for Case 2.

Case 3: n=1000, m=10000, m_member=0.50, m_insert=0.25, m_delete=0.25

Threads	Serial (s)	Mutex (s)	RW-lock (s)
1	0.0676 ± 0.0018	0.0664 ± 0.0022	0.0674 ± 0.0018
2	0.0644 ± 0.0018	0.0992 ± 0.0073	0.1673 ± 0.0047
4	0.0651 ± 0.0017	0.1144 ± 0.0084	0.1835 ± 0.0104
8	0.0653 ± 0.0023	0.1301 ± 0.0025	0.2136 ± 0.0107

Table 3: Summary of results for Case 3.

Sampling/Confidence For Case 1, the worst relative CI was 15.54For Case 2, the worst relative CI was 11.67For Case 3, the worst relative CI was 6.44The target of a 5

Case Analyses with Plots

Case 1: Read-Heavy Workload

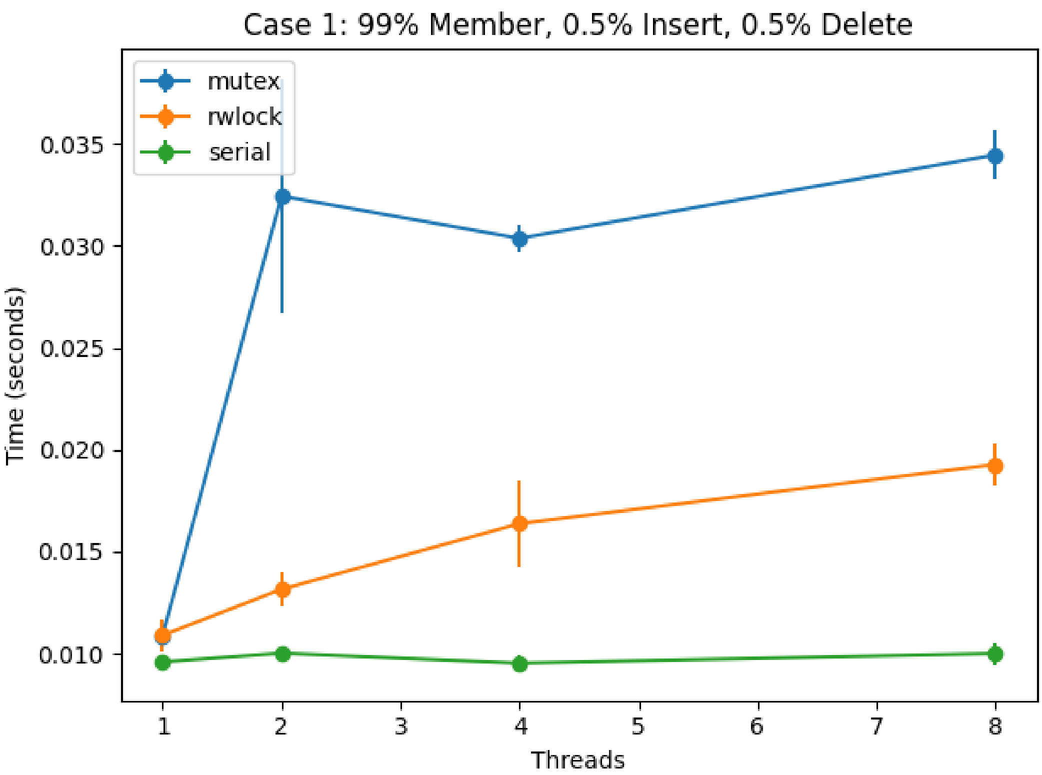

Figure 1: Average time vs. threads for Case 1.

Analysis As shown in Table 1 and Figure 1, at 1 thread, serial is fastest (0.0096s) vs mutex (0.0109s) and rw-lock (0.0109s). From 1 to 8 threads, mutex changes by 217.12At 8 threads, rw-lock is 1.79x faster than mutex. This workload is read-heavy (99

Case 2: Balanced Workload

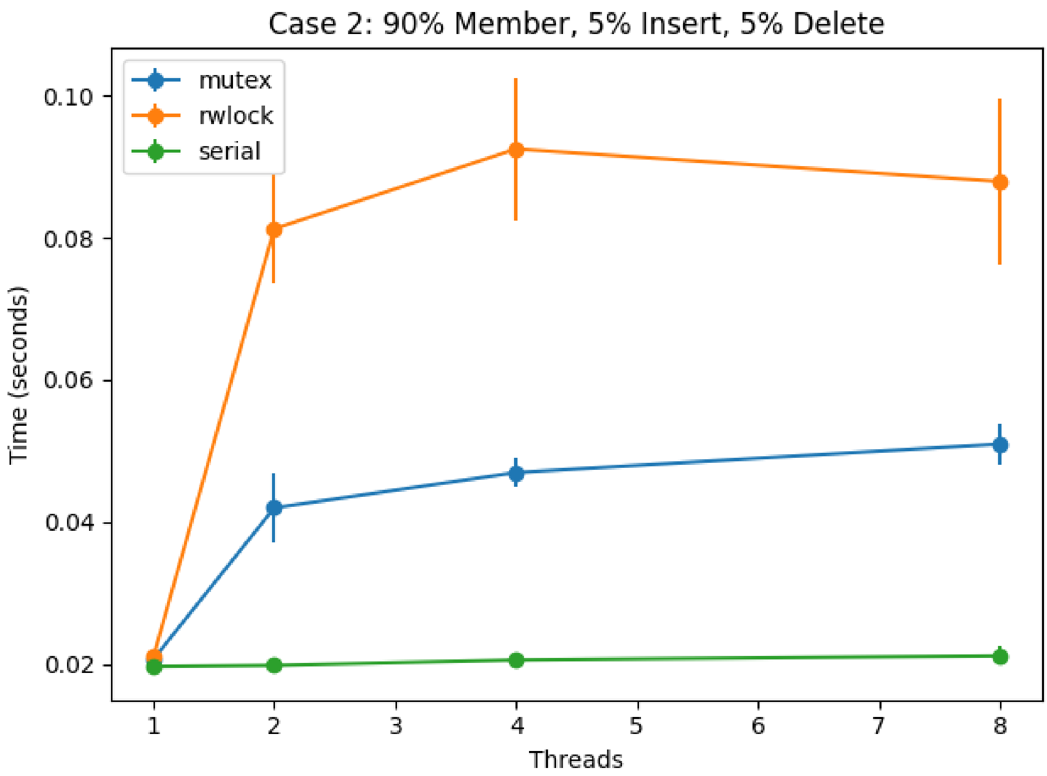

Figure 2: Average time vs. threads for Case 2.

Analysis As shown in Table 2 and Figure 2, at 1 thread, serial is fastest (0.0197s) vs mutex (0.0207s) and rw-lock (0.0211s). From 1 to 8 threads, mutex changes by 146.05At 8 threads, rw-lock is 0.58x faster than mutex. With a higher write fraction (10

Case 3: Write-Heavy Workload

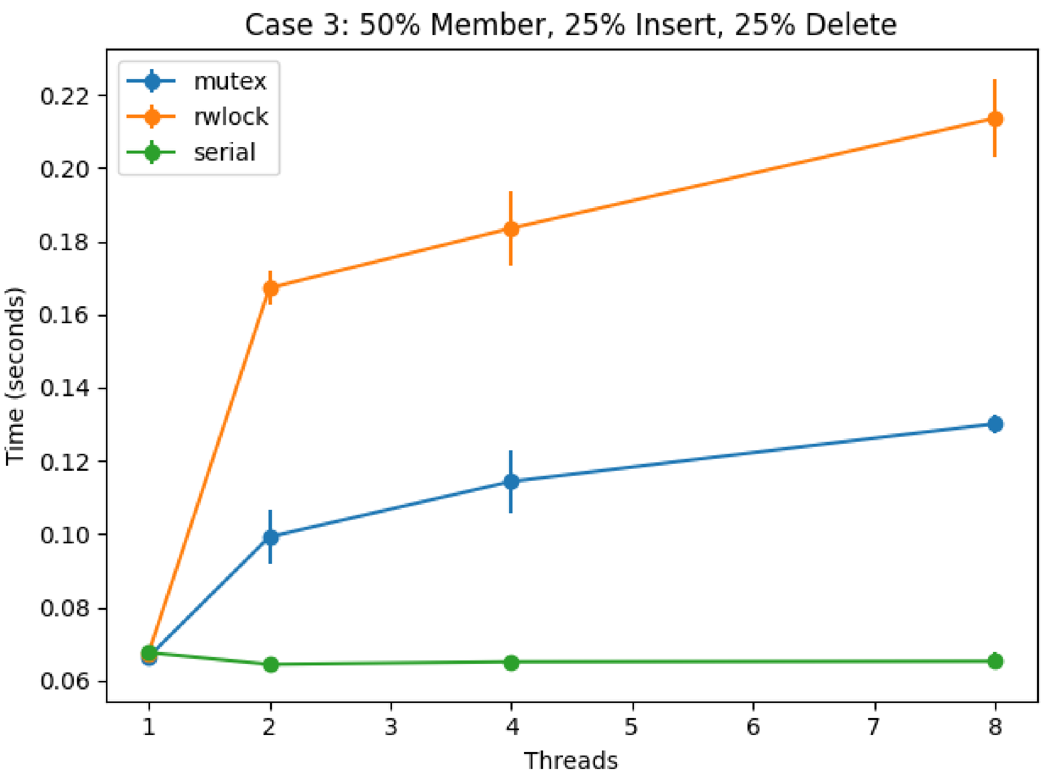

Figure 3: Average time vs. threads for Case 3.

Analysis As shown in Table 3 and Figure 3, at 1 thread, serial is fastest (0.0676s) vs mutex (0.0664s) and rw-lock (0.0674s). From 1 to 8 threads, mutex changes by 95.90At 8 threads, rw-lock is 0.61x faster than mutex. In this write-heavy scenario (50

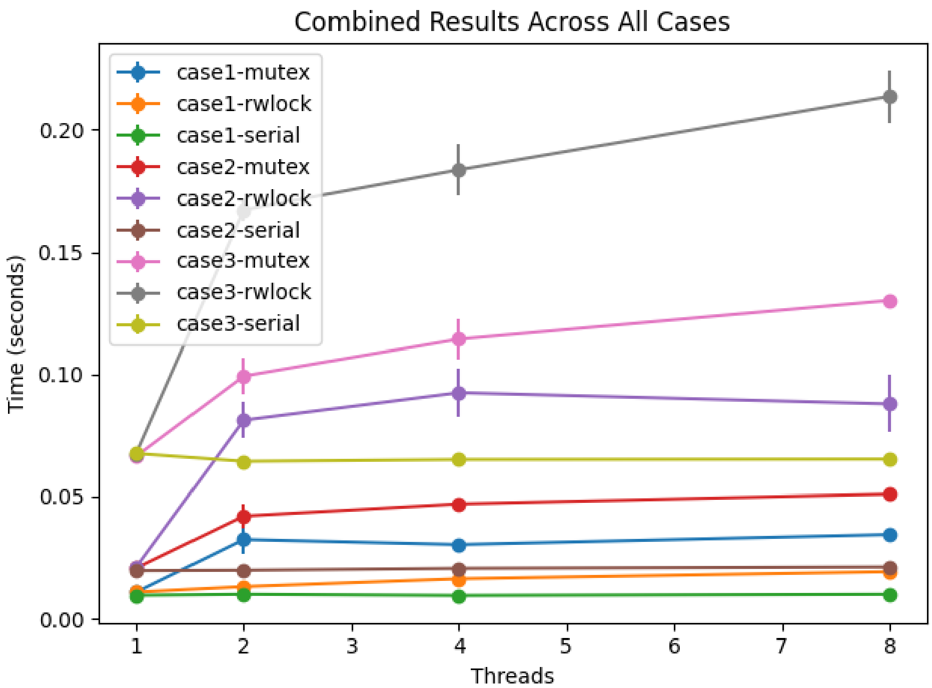

Figure 4: Combined view across all cases and implementations.

Conclusion

Results align with expectations: the serial baseline dominates at T=1 (no lock overhead). Read-heavy workloads: rwlock outperforms mutex via concurrent readers. Write-heavier workloads: rwlock advantage shrinks; both converge due to writer serialization; parallel versions can underperform serial when contention dominates. Scaling saturates near core count due to contention and scheduling overhead. The ± 5