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	0.1815 ± 0.0000	0.1919 ± 0.0000	0.2210 ± 0.0000
2	0.1849 ± 0.0000	0.2523 ± 0.0000	0.1579 ± 0.0000
4	0.1829 ± 0.0000	0.3796 ± 0.0000	0.1810 ± 0.0000
8	0.1780 ± 0.0000	0.4435 ± 0.0000	0.2054 ± 0.0000

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	1.0639 ± 0.0000	1.0372 ± 0.0000	1.0392 ± 0.0000
2	1.0582 ± 0.0000	1.2367 ± 0.0000	1.1793 ± 0.0000
4	1.0478 ± 0.0000	1.5626 ± 0.0000	1.2864 ± 0.0000
8	1.0130 ± 0.0000	1.7605 ± 0.0000	1.4382 ± 0.0000

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	4.4753 ± 0.0000	4.8215 ± 0.0000	4.5447 ± 0.0000
2	4.4604 ± 0.0000	5.8180 ± 0.0000	7.5133 ± 0.0000
4	4.4076 ± 0.0000	6.1148 ± 0.0000	8.8153 ± 0.0000
8	4.4076 ± 0.0000	6.3508 ± 0.0000	9.0376 ± 0.0000

Table 3: Summary of results for Case 3.

Sampling/Confidence The target of a 5

Case Analyses with Plots

Case 1: Read-Heavy Workload

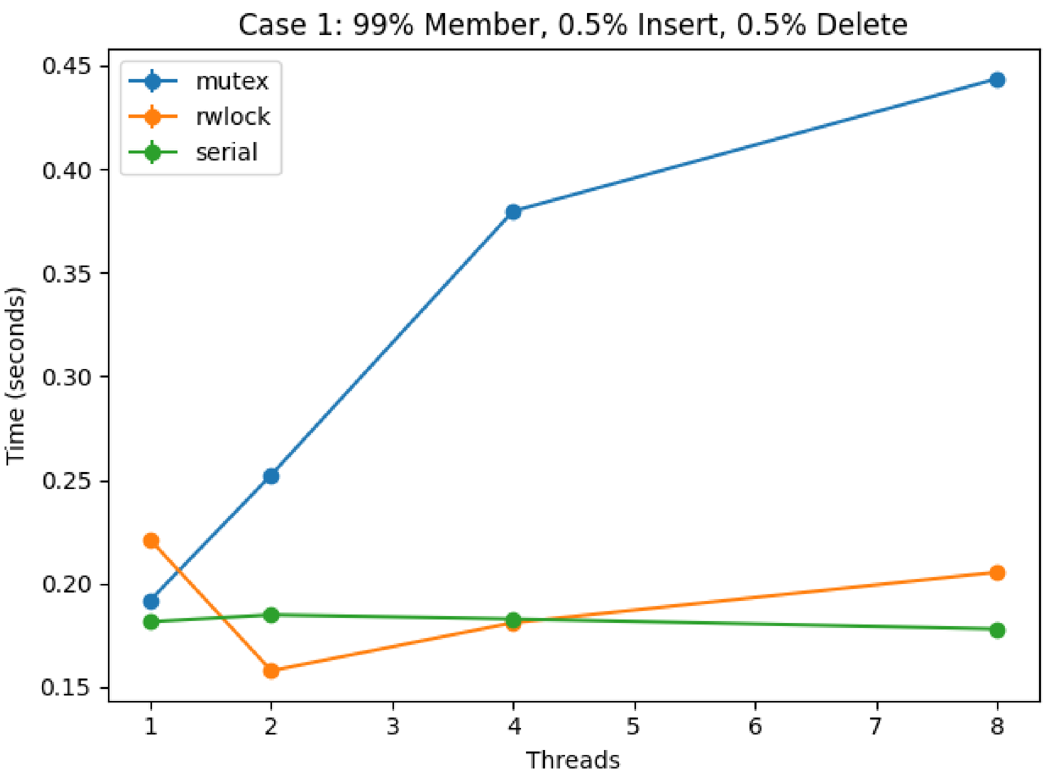

Figure 1: Average time vs. threads for Case 1.

Analysis As shown in Table 1 and Figure 1, at 1 thread, serial is fastest (0.1815s) vs mutex (0.1919s) and rw-lock (0.2210s). From 1 to 8 threads, mutex changes by 131.10At 8 threads, rw-lock is 2.16x faster than mutex. This workload is read-heavy (99

Case 2: Balanced Workload

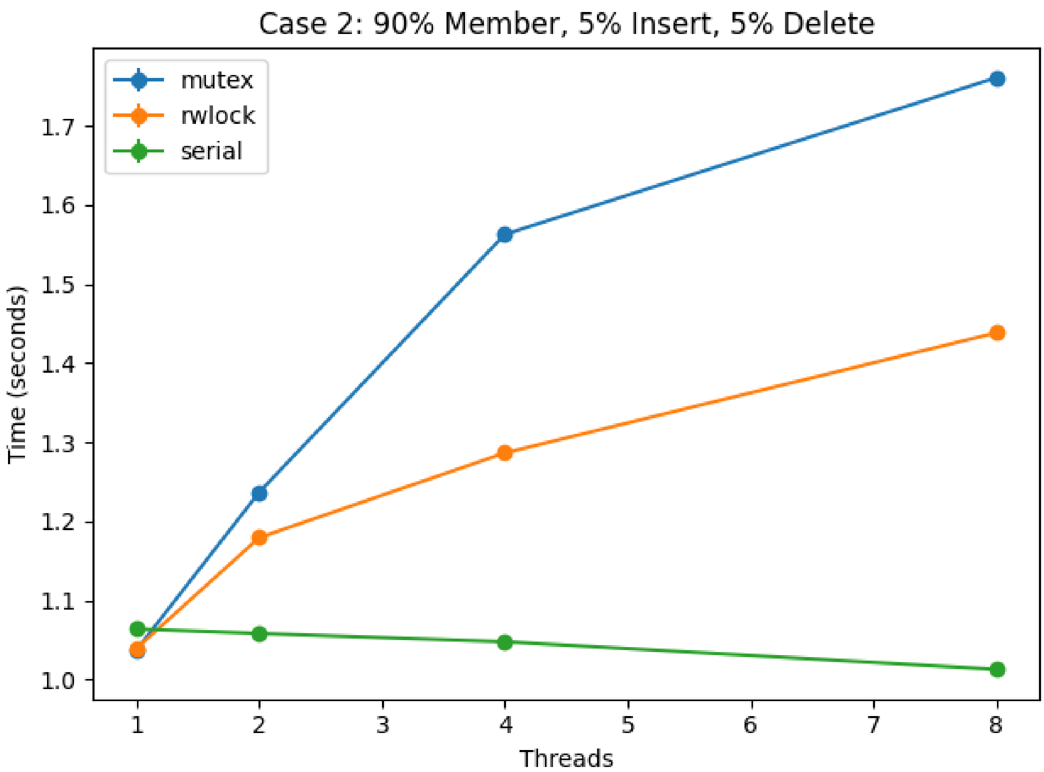

Figure 2: Average time vs. threads for Case 2.

Analysis As shown in Table 2 and Figure 2, at 1 thread, serial is fastest (1.0639s) vs mutex (1.0372s) and rw-lock (1.0392s). From 1 to 8 threads, mutex changes by 69.74At 8 threads, rw-lock is 1.22x faster than mutex. With a higher write fraction (10

Case 3: Write-Heavy Workload

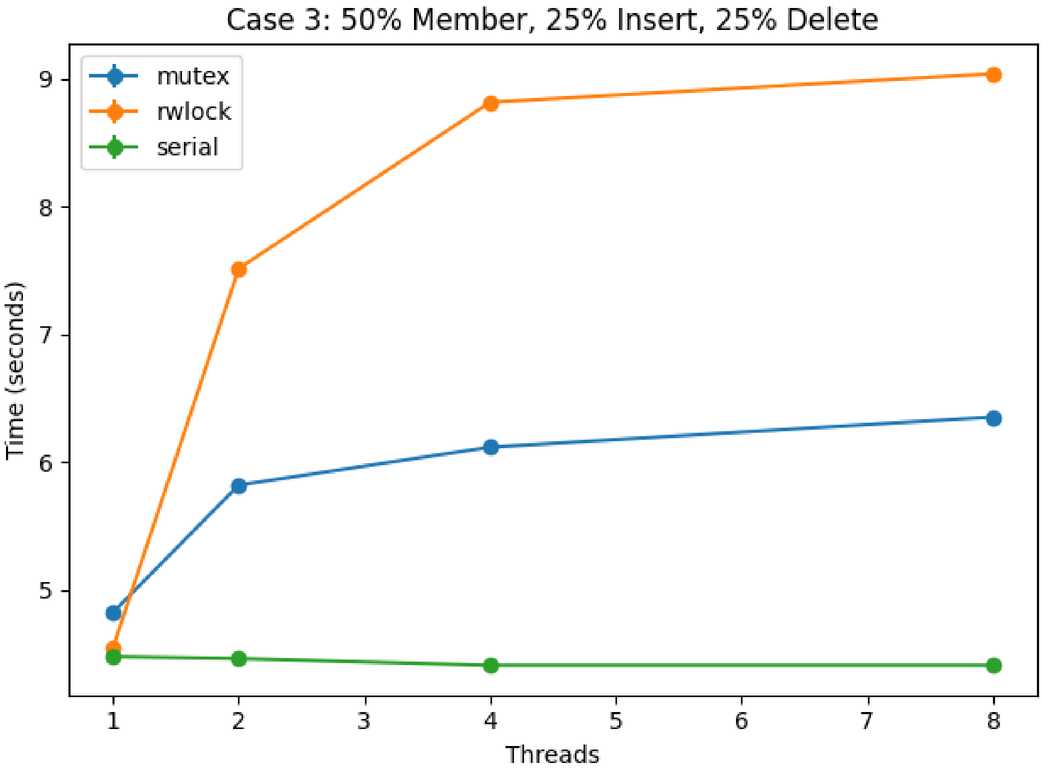

Figure 3: Average time vs. threads for Case 3.

Analysis As shown in Table 3 and Figure 3, at 1 thread, serial is fastest (4.4753s) vs mutex (4.8215s) and rw-lock (4.5447s). From 1 to 8 threads, mutex changes by 31.72At 8 threads, rw-lock is 0.70x faster than mutex. In this write-heavy scenario (50

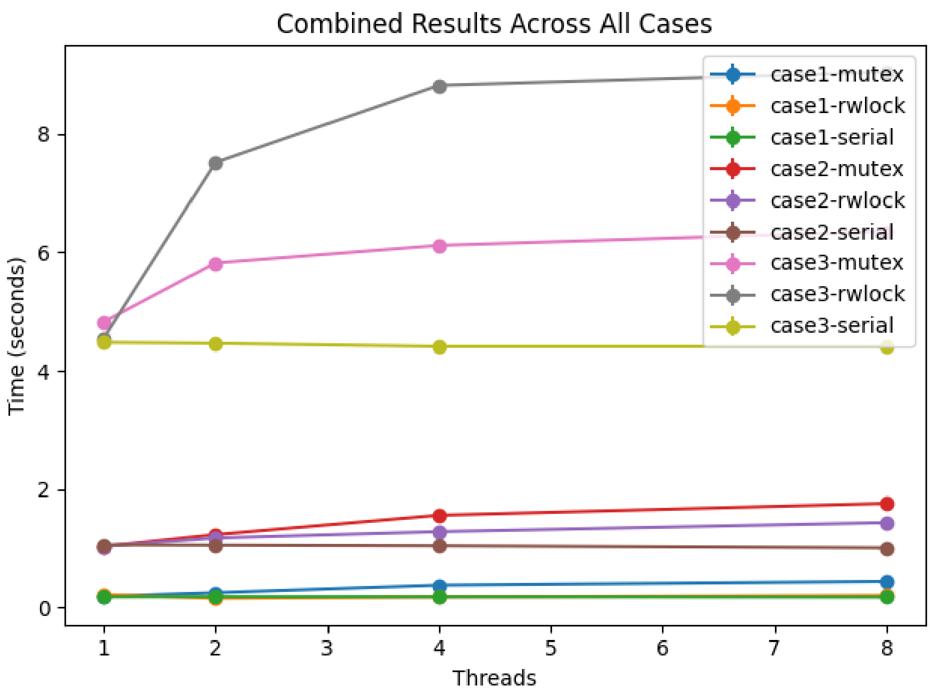

Figure 4: Combined view across all cases and implementations.

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