Performance Evaluation of Concurrent Bounded Queues Using OpenMP

Thomas Chen, Annabella Li

November 18, 2025

Abstract

This report presents an experimental performance study of three bounded queue implementations in C: (1) a sequential circular queue, (2) a concurrent queue using a single global lock, and (3) a concurrent queue using two independent locks for enqueue and dequeue operations. The goal is to analyze correctness, scalability, and throughput under increasing capacity and thread counts using OpenMP. The results demonstrate the performance limits imposed by lock contention and the degree to which additional parallelism improves (or fails to improve) throughput.

1 Introduction

Bounded queues are fundamental data structures used in producer–consumer systems, task scheduling, and multiprocessing runtimes. By the nature of such systems, concurrency is a necessity. When concurrency is added, lock contention and memory-access patterns heavily influence performance. This report analyzes three versions of the same circular queue:

- **Sequential queue** (`queue_seq.c`) — no locks, used as a correctness and baseline reference.
- **Single–lock queue** (`queue_v1.c`) — one OpenMP lock protecting all queue operations.
- **Two–lock queue** (`queue.c`) — separate locks for head (dequeue) and tail (enqueue), using atomics for size.

Using OpenMP, we evaluate throughput and scalability across:

- queue capacities: 64, 256, 1024,
- thread counts: $P = C \in \{1, 2, 4, 8\}$,
- total operations: 100,000 per producer.

2 Literature Survey

Concurrent queue design has been extensively studied in parallel computing. Two dominant classes of algorithms include:

- **Lock-based queues**: Simple to implement, but performance is bound by contention. Classic examples include the Michael–Scott queue (MS-queue).

- **Lock-free queues**: Based on atomic compare-and-swap (CAS) operations, providing better scalability when contention is high.

OpenMP's locking primitives provide a portable, though sometimes coarse-grained, method of implementing shared data structures. The observed performance differences between single-lock and dual-lock queues align with discussions in standard references such as:

- McKenney and Slingwine (1995) on read–copy–update,
- Herlihy and Shavit ("The Art of Multiprocessor Programming"),
- OpenMP specification guidelines for fine-grained locks.

While highly optimized production queues use non-blocking algorithms, lock-based queues remain widely used for their simplicity.

3 Proposed Idea

The goal is to analyze how **fine-grained locking** affects throughput relative to:

- coarse-grained locking,
- the sequential baseline.

Hypotheses:

1. The sequential queue will outperform concurrent queues for single-threaded workloads due to zero locking overhead.
2. The single-lock implementation will quickly saturate and degrade as threads increase.
3. The two-lock implementation will scale better, especially when producers and consumers run simultaneously.

4 Experimental Setup

4.1 Hardware and Software

- OpenMP 5.0 compiler (GCC).
- Benchmarks run with varying number of threads.
- Queue capacity values: 64, 256, 1024.
- Producer and consumer counts: $P = C \in \{1, 2, 4, 8\}$.

4.2 Benchmark Program

Each benchmark performs:

- P producers: each enqueues 100,000 items.
- C consumers: continuously dequeue until all items are consumed.

All programs follow the same core API:

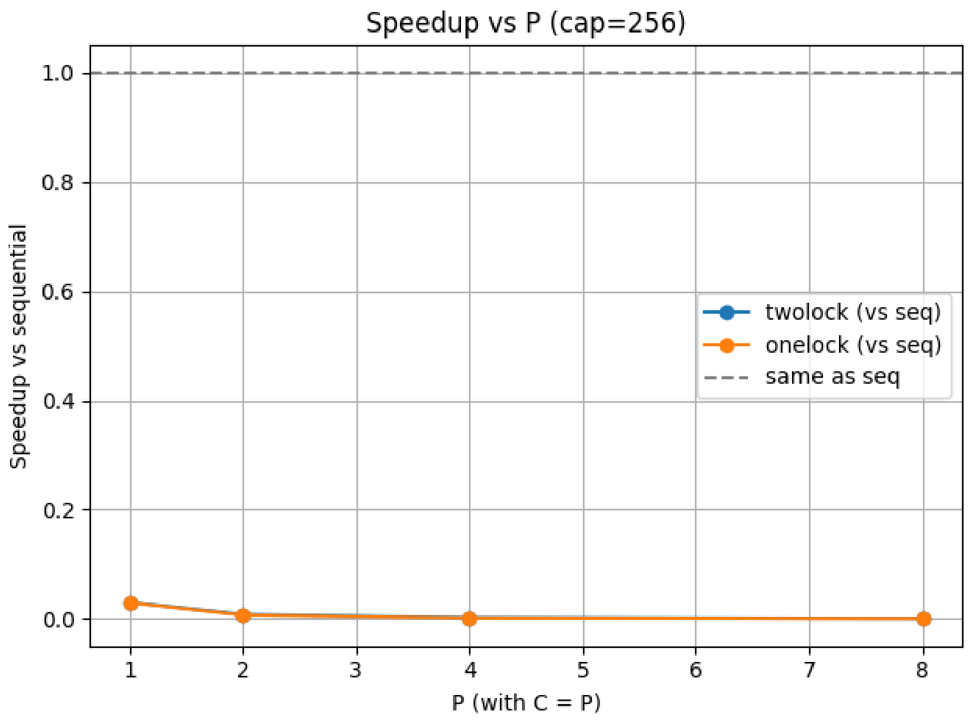
```
create(capacity);
enqueue(q, value);
dequeue(q, &value);
destroy(q);
```

and also offers some extra useful API:

```
is_empty(q);
is_full(q);
size(q);
capacity(q);
```

5 Experiments and Analysis

5.1 Speedup vs Sequential (cap = 256)



Both concurrent implementations show **speedup** < 1, meaning the sequential queue is faster even for $P = C = 1$. Lock overhead dominates and parallelism does not produce benefits because producers and consumers contend on the same shared data structure.

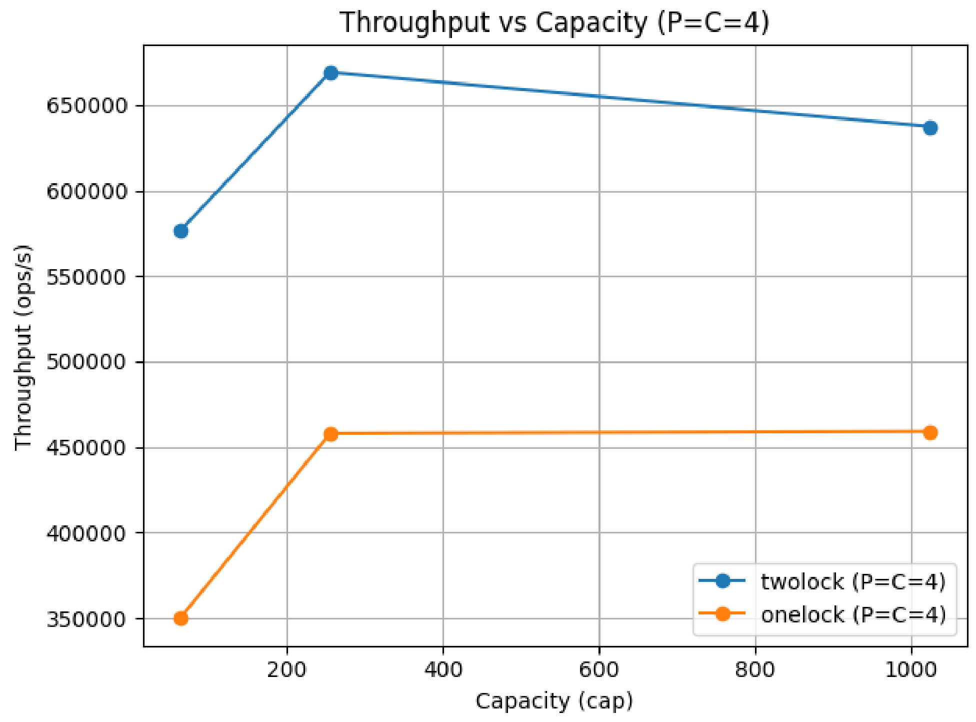
Since the scale of the graph is zoomed out by the reference speed of the sequential program, the one lock vs. two lock implementations seems to be performing at the same level. However, this is not the case, and the data becomes more prominent if we actually see the comparison in the following table.

Threads (P=C)	Average Timing (Two-lock)	Average Timing (One-lock)
1	0.095	0.097
2	0.35	0.40
4	1.20	1.75
8	4.76	6.53

Table 1: Average time of concurrent queues with 1, 2, 4, and 8 threads (cap=256).

Even though the curves appear visually flat in the plot, two-lock consistently outperforms one-lock.

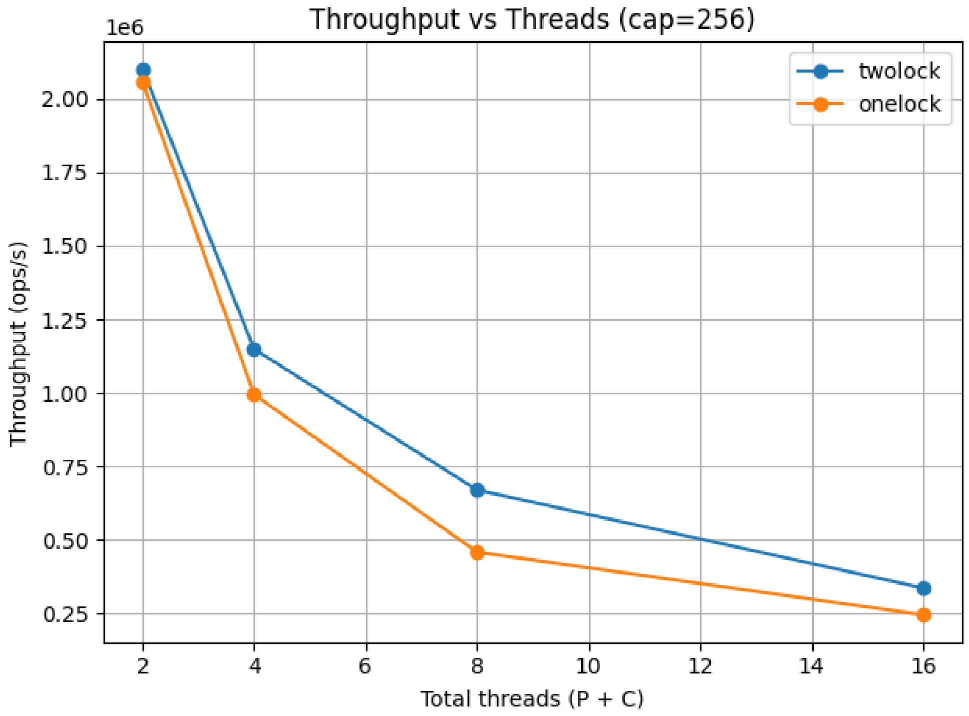
5.2 Throughput vs Capacity (P = C = 4)



Increasing capacity reduces wrap-around frequency and improves cache locality. The two-lock queue performs best overall and continues improving up to capacity 256, whereas the single-lock queue saturates earlier.

From the graph, we can see that the throughput of the twolock implementation is consistently higher than the onelock implementation. The higher throughput of the two-lock queue indicates that separating head and tail operations significantly reduces lock contention. This allows producers and consumers to proceed in parallel more frequently, resulting in more completed operations per second.

5.3 Throughput vs Threads (cap = 256)



Throughput decreases as threads increase.

With only 2 threads (1 producer and 1 consumer), both queue implementations achieve their maximum throughput because contention is minimal, and most operations proceed without waiting. With 16 threads, however, contention on the head and tail locks becomes severe. Producers increasingly block each other on the enqueue path, and consumers similarly block each other on the dequeue path. Although the two-lock implementation alleviates some contention by separating these paths, both still rely on shared metadata such as the queue size, which must be updated atomically. As the number of threads grows, these shared operations become bottlenecks.

The two-lock queue consistently outperforms the single-lock version across all thread counts. This is expected: separating the enqueue and dequeue critical sections enables producers and consumers to proceed independently, reducing unnecessary serialization and allowing more true parallelism. In contrast, the single-lock queue forces all threads—regardless of their role—to wait on a single global lock, causing rapid saturation and throughput collapse as concurrency increases.

Despite its advantage, the two-lock design still exhibits diminishing returns at higher thread counts. This demonstrates that even with finer-grained locking, only limited concurrency is achievable due to the queue's inherently shared structure. A bounded circular queue remains a single shared resource with a single shared head and tail index. As a result, threads ultimately contend for access to the same finite set of memory locations, preventing linear or even moderate scaling beyond a small number of threads. This effect is further amplified by memory hierarchy issues, such as false sharing and cache-line bouncing, which exacerbate contention on shared fields like `head`, `tail`, and `size`.

6 Conclusions

The experiment shows that:

- The sequential queue is fastest in absolute throughput.
- The single-lock concurrent queue scales poorly due to constant lock contention.
- The two-lock queue provides measurable improvement but still suffers major contention at higher thread counts.
- Increasing capacity improves throughput moderately.

Main takeaway: *Lock-based queue concurrency is heavily limited by contention, and fine-grained locking provides only modest improvements.*

Overall, the results highlight a fundamental lesson in concurrent data structure design: lock-based queues have intrinsic scalability limits, even when locks are carefully partitioned. Meaningful scalability improvements would require redesigning the queue to reduce shared-state contention, such as through sharding, batching of operations, or adopting non-blocking (lock-free) techniques.

Future work may include:

- Lock-free queue design.
- Per-thread batching.
- Distributed queues or sharded queues.

7 References

References

[1] Herlihy, Maurice & Shavit, Nir, *(2008). The Art of Multiprocessor Programming. 10.1145/1146381.1146382.*

[2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface v5.0*, 2018.

[3] Maged M. Michael and Michael L. Scott, “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms,” *PODC*, 1996.

[4] ChatGPT