Project 2: An Image Processing System

Sonia Sharapova

November 2024

1 Project Overview

This project explores the design and implementation of a thread-safe, parallelized Twitter-like server capable of handling concurrent feed operations such as adding posts, removing posts, checking post existence, and retrieving the entire feed. The primary goal was to implement a parallel data structure using low-level concurrency primitives, such as locks, condition variables, and atomic operations. This project also focused on benchmarking the performance of both sequential and parallel implementations across varying workloads and thread counts, analyzing speedups to assess scalability and efficiency.

Part 1: Twitter Feed

Task: Implement Add, Remove, and Contains methods for the feed (in feed.go).

- Add: Insert a post in chronological order (most recent first).
- Remove: Remove a post by timestamp.
- Contains: Check if a post with a specific timestamp exists in the feed.

Usage

Within the /proj2/feed directory, run the following command for all sequential tests: TestSimpleSeq, TestAdd, TestContains, TestRemove

```
$ go test -v -run ŝeq_test$
```

Results (All Tests Pass)

Execution Time for Sequential Tests (in seconds)			
TestSimpleSeq	TestAdd	TestContains	TestRemove
0.244s	0.203s	0.212s	0.575s

Part 2: Thread Safety using a Read-Write Lock

Goal: Implement a read/write lock with the required methods - Lock, Unlock, RLock, RUnlock.

- **Lock**: Writers have exclusive access when rw.writer is set to true and the this Lock method ensures that no readers (rw.readers ¿ 0) or writers (rw.writer) are active before granting the lock to a writer.
- **Unlock**: Releases the write lock and signals all waiting threads.
- **RLock**: Acquires a read lock, allowing multiple readers but blocking if a writer is present or waiting, or the maximum reader count is reached. It limits the number of concurrent readers to 32 by checking rw.readers ¿= maxReaders before allowing for a new reader. Multiple readers are allowed simultaneously unless a writer is waiting (rw.waitingWriters ¿ 0) or active (rw.writer).
- **RUnlock**: Signals waiting threads if no readers remain.

A single mutex and a single condition variable are used for synchronization across all operations, and to manage the signaling between readers and writers.

Coarse Grain Feed

Goal: Implement thread-safety in feed.go using custom implementation.

The linked list implementation is shown in the ll_feed.txt file.

Implementation: The feed library calls Lock/Unlock for write operations and RLock/RUnlock for read operations, ensuring thread-safe access at a high level and making it coarse-grained.

- Added an RWLock instance to the feed struct.
- Used Lock/Unlock for write operations (Add and Remove).
- Used RLock/RUnlock for read operations (Contains).
- Ensured the lock is properly acquired and released using defer statements for safety.

Usage

All tests pass :) For testing, run the following commend in the /proj2/feed directory:

$ go test -v

Passes all tests in 15.599s.

Part 3: A Twitter Server

Implementing the Server

Task: Implement a server that processes requests, or tasks, which perform actions on a single feed. Depending on the specified mode, the program should run in sequential or parallel execution.

For Sequential: The program processes each request on the main goroutine, and uses the feed library for feed operations.

For Parallel: Implements a producer-consumer model for the parallel version:
- Producer reads tasks from the Decoder and adds them to a task queue.
- Consumers fetch tasks from the queue and process them in parallel.

A condition variable manages synchronization between producers and consumers.

Queue Implementation

The queue (located in proj2/queue) is implemented using a linked list with a head and tail pointer for unbounded growth. Atomic operations (non-blocking) are used for synchronized enqueueing and dequeueing.

Part 4: The Twitter Client

Task: Implement a Twitter-client to analyze tasks.

Test cases:
- Added to 'proj2/twitter/tasks.txt' and run with '$ go run twitter.go 2 < tasks.txt'

Usage

All tests pass :) For testing, run the following commend in the /proj2/twitter directory:

$ go test -v

Results:
PASS ok proj2/twitter 115.755s

2 Part 5: Benchmarking Performance

Task: Test execution time on the CS cluster.

Averaging the elapsed time of the twitter tests.

All tests pass running:

$ go run proj2/grader proj2

$ go test -v in proj2/twitter

Mine:

PASS

ok proj2/twitter 112.512s

Success! Below 150s!

3 Part 6: Performance Measurement

Provided benchmark.go: benchmark performance of the sequential and parallel implementations of the Twitter server across various test sizes (xsmall, small, medium, large, xlarge). To benchmark the results, I implemented a python script to generate the benchmark results and graph them.

Benchmarking

Benchmarking was done across all test sizes by running 'brnchmarking.go' and taking the average across 5 runs.

For both sequential and parallel executions, benchmark all test sizes across the thread counts 2, 4, 6, 8, 12.

To calculate the speedups:

$Speedup = \frac{sequentialexecutiontime}{parallelexecutiontime}$

To execute benchmarking:

$ sbatch run_benchmark.slurm

The results from this run are written to 'benchmark_results.csv' with the following indices:

- test_size: The test size (e.g., xsmall, small, etc.).
- threads: The number of threads used in the parallel version.
- sequential_time: The average wall-clock time for sequential execution.
- parallel_time: The average wall-clock time for parallel execution.

Generating speedup plots

The speedup plots were generated using a python script (plotting.py) that was run manually after the execution of the SBATCH script that created the 'benchmark_results.csv' file containing all timestamps.

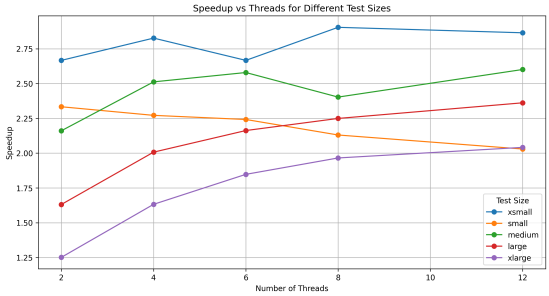

Figure 1: Speedup graph showing execution times across multiple file sizes for 2, 4, 6, 8, and 12 threads

Analysis

Small Workloads (xsmall and small):
Speedup increased up to 4 threads but plateaued or slightly declined after that. The performance bottleneck was likely due to parallel overhead (e.g., thread creation and synchronization), which outweighed the benefits of parallelism for smaller workloads.

Medium Workloads (medium and large):
Medium workloads showed consistent speedup improvement with thread count, up to around 8-12 threads. The implementation scaled well, with diminishing returns only appearing after higher thread counts.

Large Workloads (xlarge):
Speedup was modest compared to medium and large workloads. This suggests contention for shared resources, memory bandwidth issues, or inefficiencies in task decomposition for very large workloads.

Across all test sizes, speedup gains decreased with increasing thread counts, highlighting the limits of parallelism due to overhead, synchronization, and hardware constraints.

4 Conclusion

The implementation utilized a lock-free queue for task distribution among threads, ensuring efficient producer-consumer interactions while maintaining thread safety. Both sequential and parallel versions of the server were benchmarked against workloads of different sizes (xsmall to xlarge) and thread counts (2–12). The results demonstrated significant speedups for larger workloads in the parallel version, with diminishing returns beyond 8 threads due to overhead and contention. Smaller workloads benefited less from parallelism, indicating the need for adaptive threading to balance efficiency and overhead. These findings underscore the trade-offs in designing scalable concurrent systems and provide a foundation for future research in optimizing thread-based workload management. This paper discusses the design, implementation, and benchmarking of the system, offering insights into practical applications of parallelism in scalable server architectures.