Parallel Project 2 -Performance Analysis Report for Twitter Feed Implementation

Ya-Wei Tsai

November, 10, 2024

Project Overview

In this project, we redeveloped the data structure representing a user's Twitter feed after it was deleted from the servers. The feed was implemented using a singly linked list, and the following functionalities were provided:

- **Feed Implementation (`feed.go`)**: The feed consists of posts organized in chronological order using a singly linked list. We implemented the methods `Add`, `Remove`, and `Contains` for managing posts within this feed.
- **Server Implementation (`server.go`)**: We built a server that processes requests sequentially or in parallel, depending on configuration. The server can handle multiple types of requests, including adding, removing, checking (contain function), and retrieving posts (feed function).
- **Lock-Free Queue (`queue.go`)**: We implemented a lock-free queue to manage tasks in a producer-consumer model for parallel execution. This lock-free approach improved performance by avoiding unnecessary blocking.
- **Custom RW Lock (`lock.go`)**: A read-write lock mechanism was developed to provide thread-safe access to the feed while allowing concurrent read operations when no write is in progress.

Running the Benchmark Script

To run the benchmark tests and generate speedup graphs, navigate to the `benchmark` directory and execute the following command:

```
sbatch graph.sh
```

This script runs the benchmark tests for different test sizes (`xsmall`, `small`, `medium`, `large`, `xlarge`) and varying thread counts (2, 4, 6, 8, 12). The speedup graph is then generated based on the benchmark results.

Analysis of Speedup Graph Results

Below is the speedup graph for different test sizes:

Graph Overview

The graph presents speedup results for different feed operations (of varying sizes) by increasing the number of threads used for processing. The x-axis represents the number of threads, while the y-axis shows the speedup relative to the sequential version.

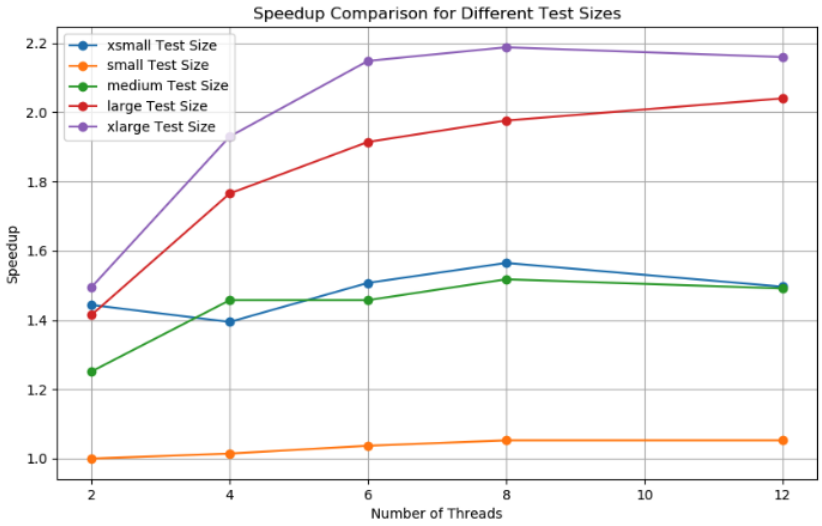

Figure 1: Speedup Comparison for Different Test Sizes

Impact of the Linked List Implementation

The use of a singly linked list impacts performance due to its linear nature. Operations like `Add`, `Remove`, and `Contains` require traversing the list, which becomes inefficient as the number of posts increases. In parallel scenarios, the linked list's linear structure means contention may arise when multiple threads are trying to modify or traverse the list. The speedup is limited for smaller feed sizes (`xsmall`, `small`), as the computational workload is insufficient to benefit significantly from additional threads.

Effect of Lock-Free Algorithm

Using a lock-free algorithm in the task queue for the producer-consumer model improved performance considerably. Lock-free approaches reduce the contention between threads compared to traditional locks, resulting in better scalability, particularly in larger test sizes (`xlarge` and `large`). The lock-free queue allows concurrent tasks to be managed more efficiently, especially when there are frequent task additions and removals.

Graph Observations

- **Smaller and Medium Test Sizes (`xsmall`, `small`, `medium`)**: The speedup for smaller feed sizes remained close to 1, with limited improvement when increasing the number of threads. This is expected because the workload is too small for parallelism to offer significant gains, and the overhead of thread management may outweigh the benefits.
- **Large Test Sizes (`large`, `xlarge`)**: For larger workloads, speedup increases as the number of threads increases, up to a certain point. The `xlarge` test size achieved a speedup of approximately 2.2 with 8 threads. However, speedup gains diminish or even decrease beyond 8 threads, as overheads such as contention, cache invalidation, and resource saturation start to take effect.

Potential Performance Improvements

- **Fine-Grained Locking**: The current implementation employs a coarse-grained read-write lock, which locks the entire feed during write operations. This leads to contention when multiple threads need access. Switching to a fine-grained locking mechanism could improve concurrency by allowing multiple threads to read/write different parts of the feed simultaneously, thus reducing contention and increasing throughput.
- **Optimized Queue and Synchronization Mechanism**: The lock-free queue currently employed for task management offers significant advantages over traditional locking queues. However,

advanced queuing techniques, such as using bounded lock-free queues, could further improve performance by balancing workload among threads more effectively.

- **Hardware Considerations**: There is a significant difference in execution speed when running benchmarks on a local computer versus the cluster. The local setup was considerably faster compared to the cluster. This difference is likely due to differences in hardware configuration and system optimizations, such as CPU clock speed, memory access times, and overall system load.

Conclusion

The implementation of the Twitter feed as a singly linked list, along with a parallel server for request handling, demonstrated the trade-offs between different data structures and synchronization mechanisms in terms of scalability and efficiency. The lock-free queue implementation showed significant speedup potential, particularly in scenarios with larger workloads. However, the linear nature of the linked list and the use of coarse-grained locks were limiting factors that affected the scalability, particularly at higher thread counts. Moving to finer-grained locking, using more efficient queuing mechanisms, and enhancing the underlying data structure are key steps for improvement.

The results of the benchmarks and speedup graph demonstrate that while the implementation can handle moderate levels of concurrency, optimizations are necessary to achieve further scalability. Specifically, using finer-grained synchronization mechanisms and enhancing the underlying data structure are key steps for improvement.