Parallel vs. Serial Performance Analysis of twitter.go

Eddie Shim
Parallel Programming
eddieshim@uchicago.edu

05/15/2020

Overview

In this assignment, I will be testing the serial and parallel versions of `twitter.go`. This program is a basic Go implementation of Twitter's feed infrastructure, where each item on the feed represents a post with a unique id, timestamp, and message body. `feed.go` is a thread-safe linked list of tasks ordered by timestamp that uses coarse-grained synchronization of read-write locks (implemented in `rwlock.go`) to operate on the linked list using commands `Add, Remove, Contains`. The max number of readers is limited to 32. `twitter.go` is an implementation of Twitter's task queue, using a parallelized producer-consumer ("client-server") model. The user inputs requests (ie tasks sent from a client) through `os.Stdin` in the following JSON data format:

```
{
"command: string,"
"id": integer,
... data key-value pairings...
}
```

This input data is then enqueued into a thread-safe (coarse-grained synchronization) task queue by a producer. The consumer (ie "server") will then dequeue tasks off the queue and execute the command onto our thread-safe feed. The user will define the program's n number of threads and b block size, with which

spawns n consumers each performing chunks of b tasks.

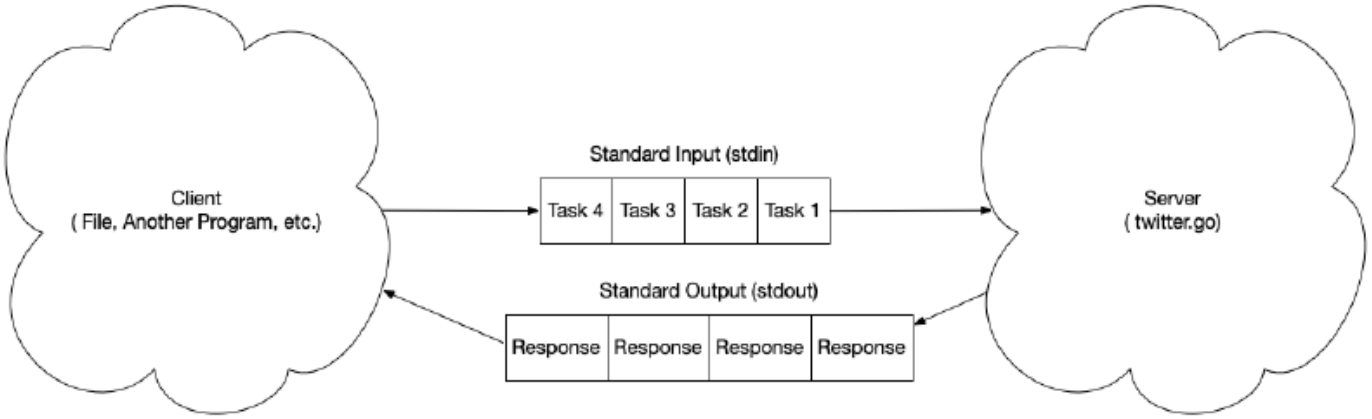

Figure 1: Twitter client-server diagram

Testing Machine

Execution performance was measured on a Lenovo Thinkpad laptop with the following specifications:

- Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz
- Number of Cores: 4
- Operating System: Windows 10 64-bit
- RAM: 16.0 GB

Performance Analysis

Using the Python notebook `twitter_performancer_eval.ipynb`, I tested the performance of serial and parallel versions of twitter.go by measuring execution times with the following parameters:

$$P = [50000, 100000, 500000, 1000000]$$

$$N = [1, 2, 4, 6, 8]$$

$$B = \lceil P/N \rceil$$

Where P is the total number of operations being processed under one program execution, N is the number of threads, and B is the block size per thread.

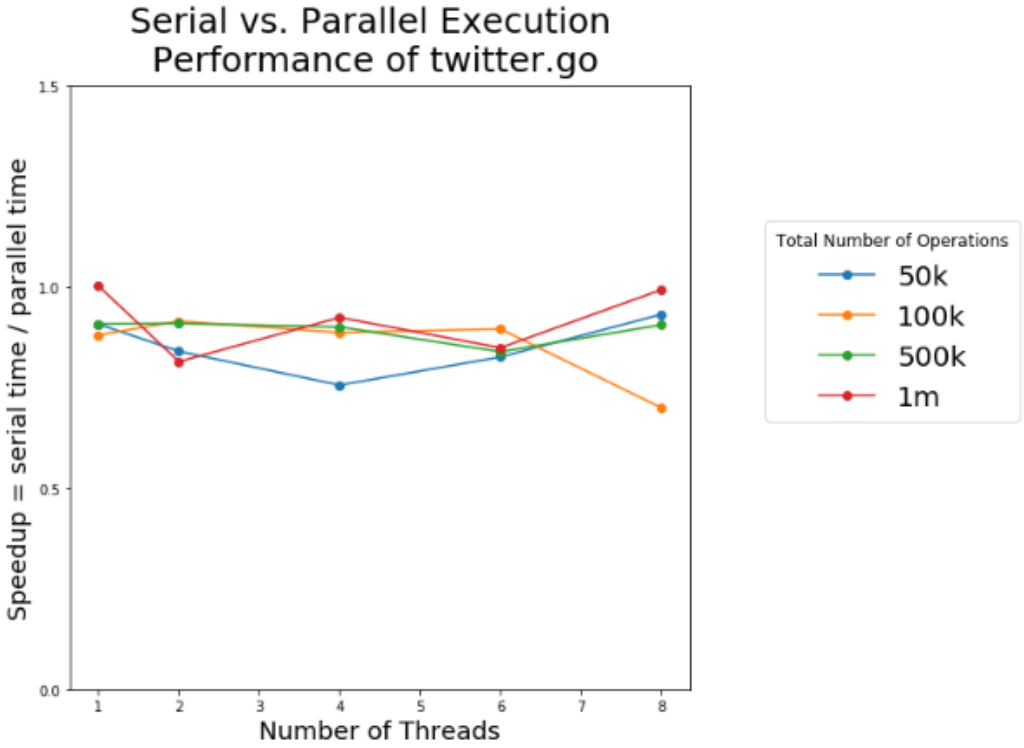

Figure 2: Performance graph of 99% contains command

As we can see in the performance graph below, there isn't much of a performance boost due to the fact that twitter.go uses a coarse-grained lock synchronization. Because we are locking the linked list each time an operation is performed, it's essentially running in parallel and adding more threads do not improve throughput as seen by how horizontal the lines are, and how they mostly hover around speedup = 1.

Identify the areas in your implementation that could hypothetically see increases in performance if you were to use a synchronization technique or improved queuing techniques. Explain why you would see those increases.

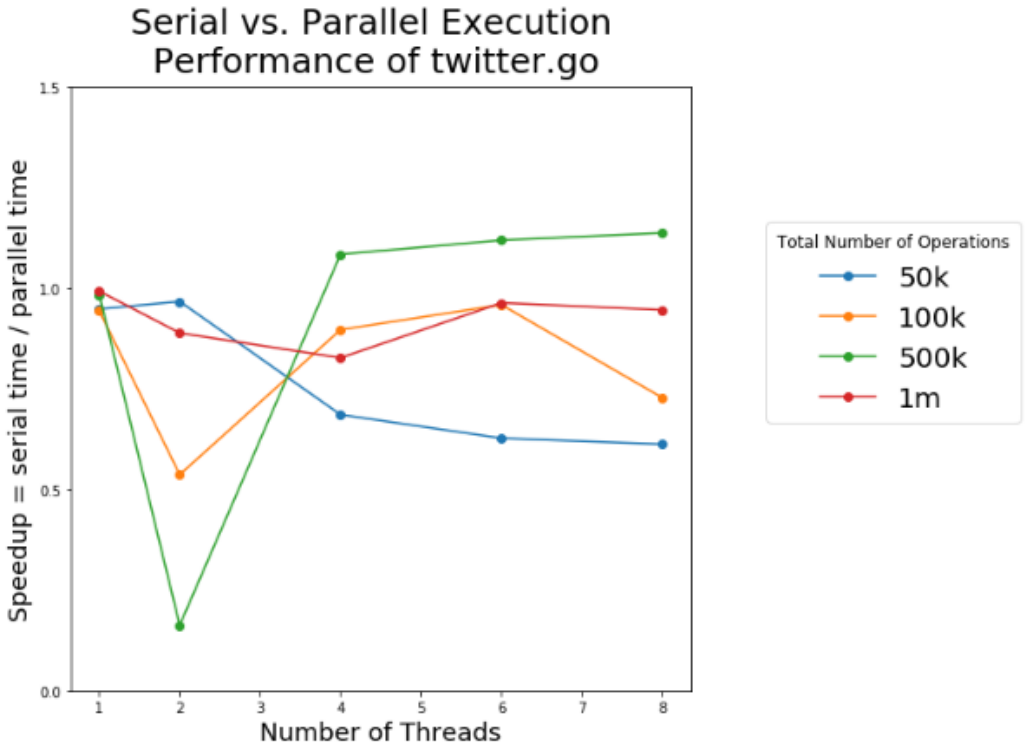

Figure 3: Performance graph of 99% add/remove commands

The differences in these two graphs show there is some effect on performance due to the type of operations performed. When 99% of the input commands are add or remove, there's a lot more variance in performance especially at lower thread counts. This is due to the fact that the contains command simply uses a read lock whereas the add or remove commands use the write lock. However, overall improvement still remains at or below speedup = 1. Reimplementing the synchronization method using a more fine-grained lock approach, such as optimistic synchronization or lazy synchronization would largely improve performance and allow us to take advantage of using a higher thread count. In either of these approaches, each object in the data structure would have their own locks, and instead of having each operation lock the entire data structure, we would target our locks only on the nodes we would be performing the operation onto. Therefore, instead of having the entire data structure wait for the lock to release, we could in parallel perform other operations on other objects in the data structure.