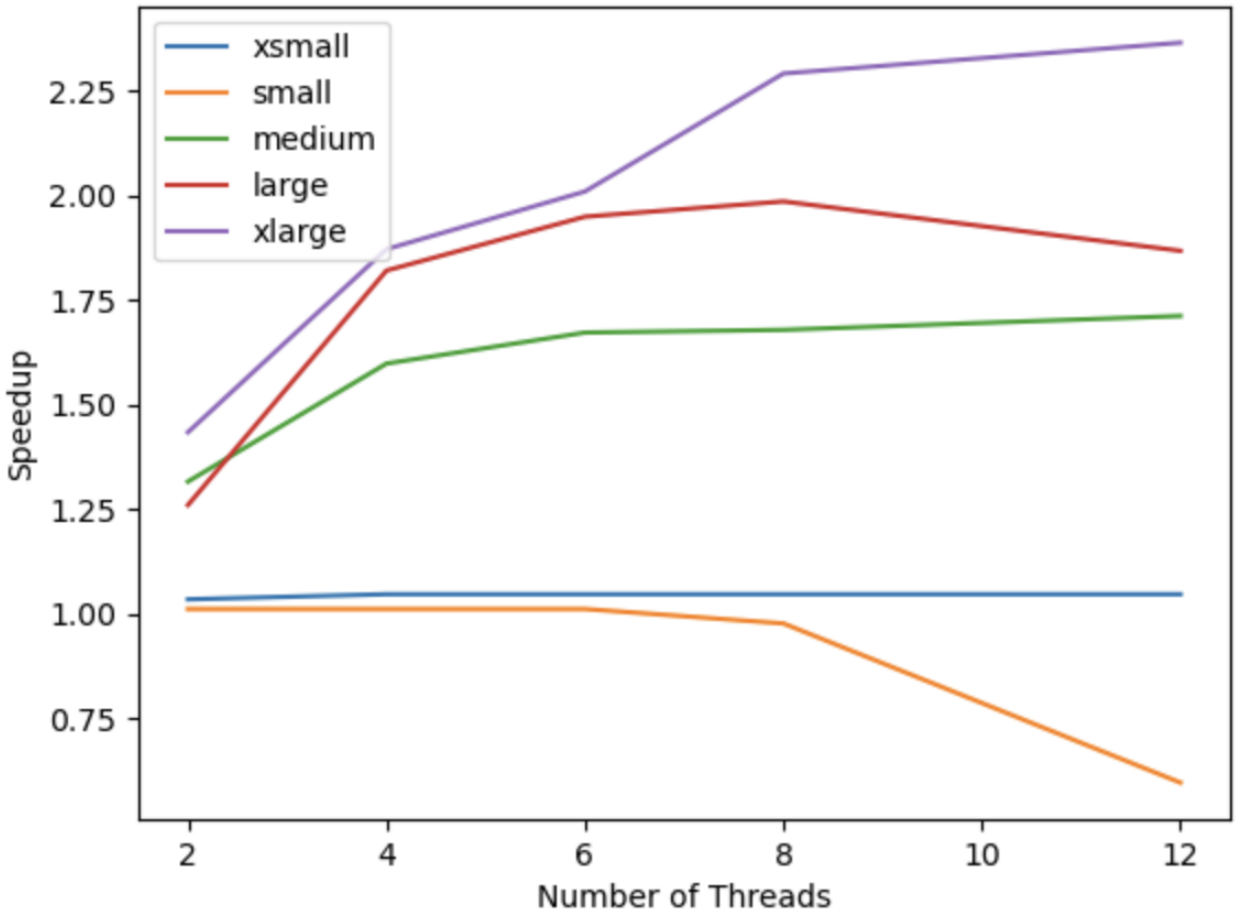
Project Report

Description

This project implements a user Twitter feed in a producer-consumer environment. In `feed.go`, we implements a `feed` package that mimics a Twitter feed and supports the user's add post, remove post, and contain post, operations. Each post is uniquely identified by its timestamp. In `server.go`, a `server` package is used to creat a produser-consumer environment. The producer extracts feed requests, e.g., add a post, remove a post, etc., from a given input source and enqueues the requests to a lock-free queue. The consumers are goroutines that dequeue the requests from the lock-free queue and process them to the user's Twitter feed, and output the result, e.g., success or not, to an output source. In `twitter.go`, we specify the input and output sources, the number of goroutines we'd like to spawn (no specification if we use the sequential version), and call the server in the `server` package to start running.

Speedup graph analysis

We would like to analyze the program performance by analyzing the speedup graph created as follows: for each input test requests size of extra small, small, medium, large, and extra large, we ran the server with 2, 4, 6, 8, and 12 threads. We also run the server using the sequential version. For each number of threads, the speedup is calculated by $\frac{\text{sequential serving time}}{\text{serving time using a certain number of threads}}$. For accuracy reasons, the serving time here all represents the average serving time of 5 same executions.



As we can see from the speedup graph, small test sizes, including the extra small and small tests, do not have a clear speedup. Instead, for 12 threads, the small test tends to lag behind the sequential version. That might be caused by threads competetion when dequeueing tasks from the lock-free queue, which outweighs the parallelization advantages since the test size is relatively small. Apart from that, all large test sizes have speedups, and we observed that the amount of speedup is positively correlated to the test size, with the most speedup takes place at the extra large test.

The bottleneck of the program might be the `feed` package, where we used a self-implemented read-write lock to guarantee thread safety. When one thread holding the lock is put to sleep by the operating system, other threads waiting for the lock will significantly slow down the program. An improvement is seen in the lock-free `queue` package, where instead of a lock, we used compare-and-swap (CAS) operations to make the queue thread safe. However, CAS should be used carefully; otherwise, data race and other stubborn bugs might occur, which can be hard to catch.

Besides, hardware plays a role in determining the program performance. That is related to thread utilization/effectiveness in our case.

How to reproduce the speedup graph

To preproduce the above speedup graph, navigate to `proj2/benchmark/` directory and run `sbatch plot.sh` on the UChicago Linux cluster. After about 35 minutes, see the output graph `speedup-image.png` in the same directory.