Please submit a report (pdf document, text file, etc.) summarizing your results from the experiments and the conclusions you draw from them. Your report should also include the graph as specified above and an analysis of the graph. That is, somebody should be able to read the report alone and understand what code you developed, what experiments you ran and how the data supports the conclusions you draw. The report must also include the following:

Sequential Version:

The first and foremost step for a parallel programming project is to achieve all the required tasks in sequential order. This not only gives us a standard to compare our parallel programming approaches but also helps us in exploring various avenues of applying parallel programming techniques. This helps us to know the hotspots and bottlenecks present in the program.

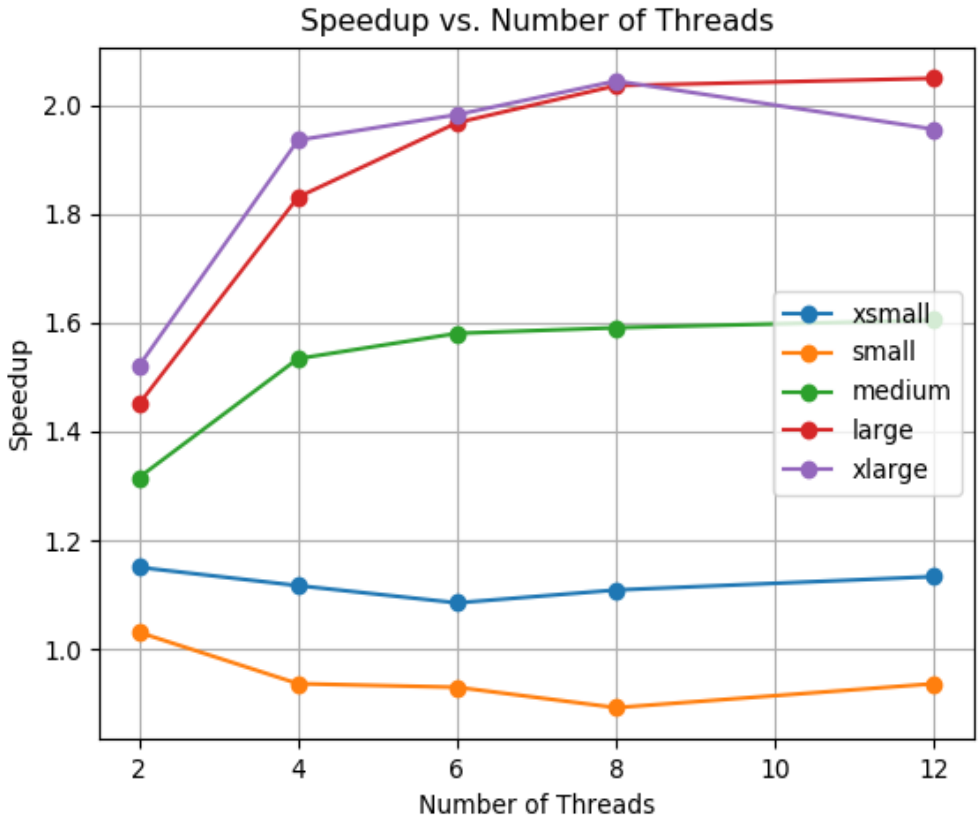
Parallel version:

In the parallel version we distribute the task of executing the tasks(add/remove/getfeed/done) and compare the speedup for each thread and each case of number of requests(from xsmall to xlarge).

The speed up for the whole project is not much and it does not cross 2 in any case. This is because of several reasons-

1) Majority of the project is sequential - json encoding, producer, json decoding
2) The part where parallelism is done a coarse grained lock has been utilized which is expensive and creates sequential bottlenecks majorly for the readers as we expect multiple readers to read simultaneously.

The following graph was obtained after running on the slurm cluster.



The plot has speedup on the y axis and number of threads on x axis. And legend depicts the corresponding lines.
In the speedup graphs, with speedup on the y-axis and the number of threads on the x-axis, a clear trend emerges. As the number of threads increases, speedup rises for medium, large, and xlarge request lines. However, intriguingly, small and xsmall lines exhibit almost stagnant speedup.
The graph also shows that there is almost a directly proportional relationship between speedup and size of the lines(requests).
Though with increasing thread count speed up increases however after 8 threads the rate of speed up decreases leading to a lower speedup with 12 threads and similar observation can be predicted for 10 threads.
The reason for this decline can be because of contention to get the lock for a high number of threads resulting in bottlenecks or load imbalancing.

Another interesting observation is that xsmall has higher speed up than small requests and this has been verified by plotting multiple speed up graphs on slurm batch and every time the same observation is observed. This might be attributed

to the fact that xsmall tasks might exhibit better cache locality, meaning the data accessed by each thread is likely to fit within the cache. On the other hand, small requests might have a higher likelihood of causing cache misses, where the data needed by one thread is not present in the cache, leading to longer memory access times.

A brief description of the project (i.e., an explanation what you implemented in feed.go, server.go, twitter.go. A paragraph or two recap will suffice.

The feed package implements a linked list of Feed interface where each feed can have multiple or no posts in it. The interface has few builtin methods of declaring/creating a new feed, adding a post, removing a post, contains to check whether a post is in it or not and getFeeddata to return all the posts in a feed. The feed has a read and write lock in it to maintain thread safety and avoid race conditions. The read-write lock is implemented using condition variables and mutex.

The Server package is the place where we implemented our first producer consumer model where producer in a parallel version is the main thread enqueuing tasks(add/remove/get posts in a feed) sent by the client (after reading using the json.Decoder) sequentially to the lockfree queue whereas consumers are several goroutines which dequeues a task from the implemented lockfree queue ~> execute it ~> send the response accordingly and all the sequence of tasks is done multiple threads in parallel.

Here we implement the sequential and parallel version:
As described above in the parallel version we implement the producer(main go routine) consumer(extra goroutines) model whereas in the sequential version the main go routine is allotted the task of producer and consumer both where it processes and executes all the requested tasks without spawning any goroutines.

The Twitter package houses a basic Twitter client with support for parallel execution. It accepts a command-line argument for the number of consumers (goroutines) and creates a server.Config based on the provided argument. It uses json.Decoder and json.Encoder to handle JSON data from os.Stdin and os.Stdout. The main function starts the server, and after the Run function completes, the program exits.

- **Instructions on how to run your testing script. We should be able to just run your script. However, if we need to do another step then please let us know in the report.**

The testing script for producing speedup graphs is located in the "benchmark" directory. To run the script, simply execute "sbatch benchmark.sh." The script generates speedup graphs for all available data sizes (e.g., xsmall, small, xlarge). For each data size, it begins by running the sequential version five times to calculate the average execution time. Then, it proceeds to run the program with varying thread counts (2, 4, 6, 8, 12) for five iterations, averaging the execution times for each thread count to plot the speedup graph.

In the "slurm/out" directory, you can find previous successful run outputs, confirming the script's functionality. Additionally, within the "grader" directory, both "grader-slurm.sh" and "benchmark-proj2.sh" scripts were executed. The grader script successfully passed all test cases, earning a 50/50 score within a brief timeframe (under 2 minutes). The "benchmark-proj2.sh" script was run five times to collect timing data for two implementations – one utilizing signaling after a writer is done and the other using broadcasting. The average time for the signaling implementation was approximately 105 seconds, while the broadcasting implementation took around 136 seconds, both comfortably under the 150-second threshold.

Please note that detailed timing information and grader outputs are available in the respective stdout files in the "grader/slurm/out" directory.

- As stated previously, you need to explain the results of your graph. Based on your implementation why are you getting those results? Answers the following questions:
- **What affect does the linked-list implementation have on performance? Does changing the implementation to lock-free or lazy-list algorithm size improve performance?**

  The linked list implementation with read - write lock has several advantages and disadvantages. It ensures thread safety in concurrent implementation and avoids race conditions at all costs. It also allows multiple readers to read simultaneously as well.
  However, if there are multiple writers then it creates a coarse grained implementation for both readers and writers leading to contention and performance bottlenecks.

Yes, changing the implementation to lock free queue or lazy list algorithm should improve the performance of our feed.

In lock free queue- First, as we don't have any locks and use only atomic load and compare and swap operations which are less expensive than creating locks. Moreover no locks means no contention and sequential bottlenecks.
Second, both read and write operations can have higher throughput and execution time will be reduced.

In the lazy list algorithm, we check first and then locks and hence the locking is more fine grained. It also delays the removal leading to less contention. Contains does not involve locking so it is less expensive in comparison to linked list feed implementation.

- **Based on the topics we discussed in class, identify the areas in your implementation that could hypothetically see increases in performance if you were to use a different synchronization technique or improved queuing techniques. Specifically, look at the RW lock, queue, and producer/consumer components and how they all might be affecting performance. Explain why you would see potential improvements in performance based on either keeping these components are substituting them out for better algorithms.**

  The use of signaling instead of broadcasting in the read-write lock reduces the total time, but it seems to result in lower speedup for medium-sized tests. This might be due to load imbalances or read-write contention causing bottlenecks in specific cases. However, for larger tests, we achieve a speedup of around 2.0 compared to approximately 1.5 with broadcasting.

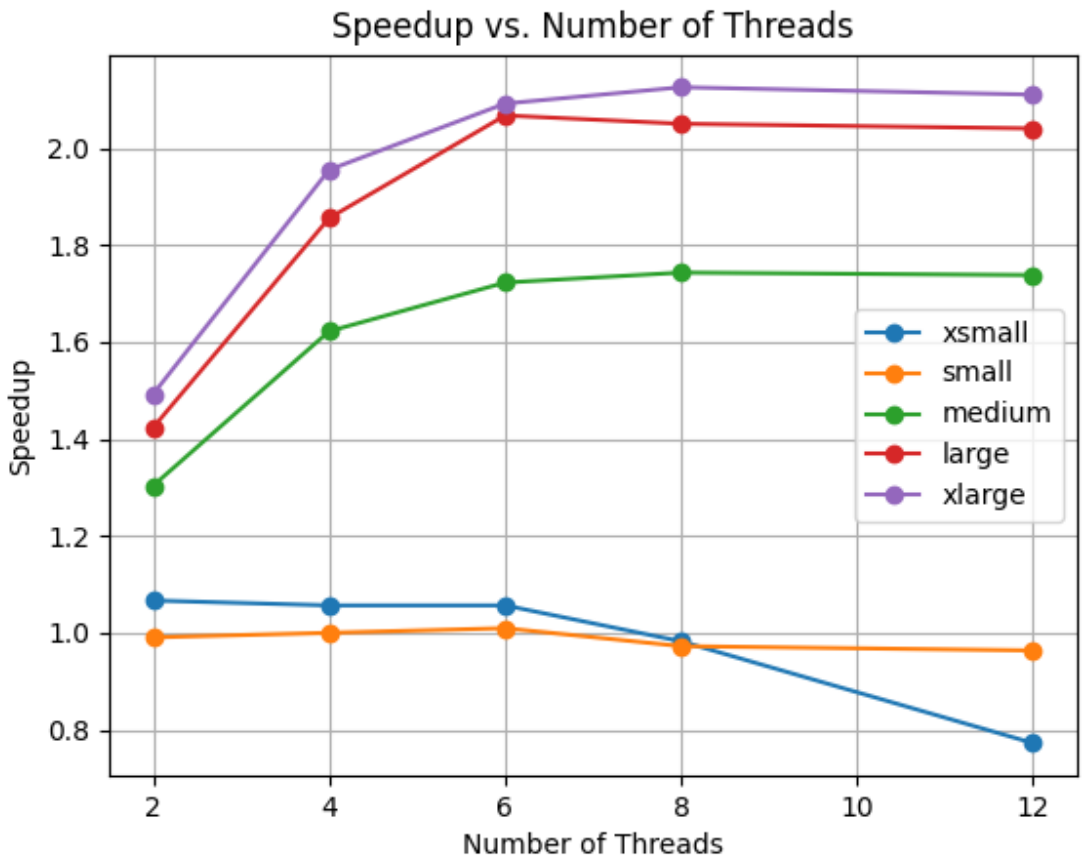
  In the case of the feed and read-write lock, implementing a lock-free queue or a lazy list synchronization approach could potentially yield better performance. Additionally, balancing the load by having an equal number of producers and consumers, or even using multiple producers, could help reduce the sequential portion of the code. The challenge with multiple producers is parallel JSON parsing and decoding, which may require careful synchronization.

  Given that the test cases indicate a high proportion of read operations, leveraging a map to store individual feeds or associating each feed with a map could significantly reduce the cost of read operations.

In the read write lock we could have used semaphores which might lead to increased thread safety and less contention.

- **Does the hardware have any affect on the performance of the benchmarks?** Yes, the performance varies with different hardware. The differences come first, when we run the test cases in the Twitter package- on a MAC M2 it completes within 80 seconds, on the CS 6 linux cluster it gets completed in around 145 seconds whereas on the slurm it completes in ~136 seconds which shows the hardware dependencies.

  Secondly, while plotting the speed up graphs, on my local machine - a MAC M2, the maximum speedup achieved for extra large lines went up to 2.2x. In case of running on slurm, the speedup achieved was somewhere in between (~2.0x), and that on the CS Linux 6 cluster was lowest (~1.5x).



On local computer