Description:

I have attempted an embarrassingly parallel task of web scraping and tokenizing.

The program is designed to fetch, parse, tokenize, and store transcripts of my favorite podcast, The Seen and the Unseen by Amit Varma. Someone has already transcribed and is hosting it in their server.

The program retrieves HTML pages from URLs, extracts textual content from specific HTML tags, tokenizes the text into individual words, and writes the results to disk. The problem involves significant network I/O, disk I/O (saving .txt files) and computational tasks (parsing and tokenizing content). The goal is to improve performance by parallelizing these tasks.

To Run:

Go to the parse directory

```
go run parse.go <number of links to parse> <number of threads> <work stealing (true/false)>
```

Note: the number of links has to be less than 400

Examples:

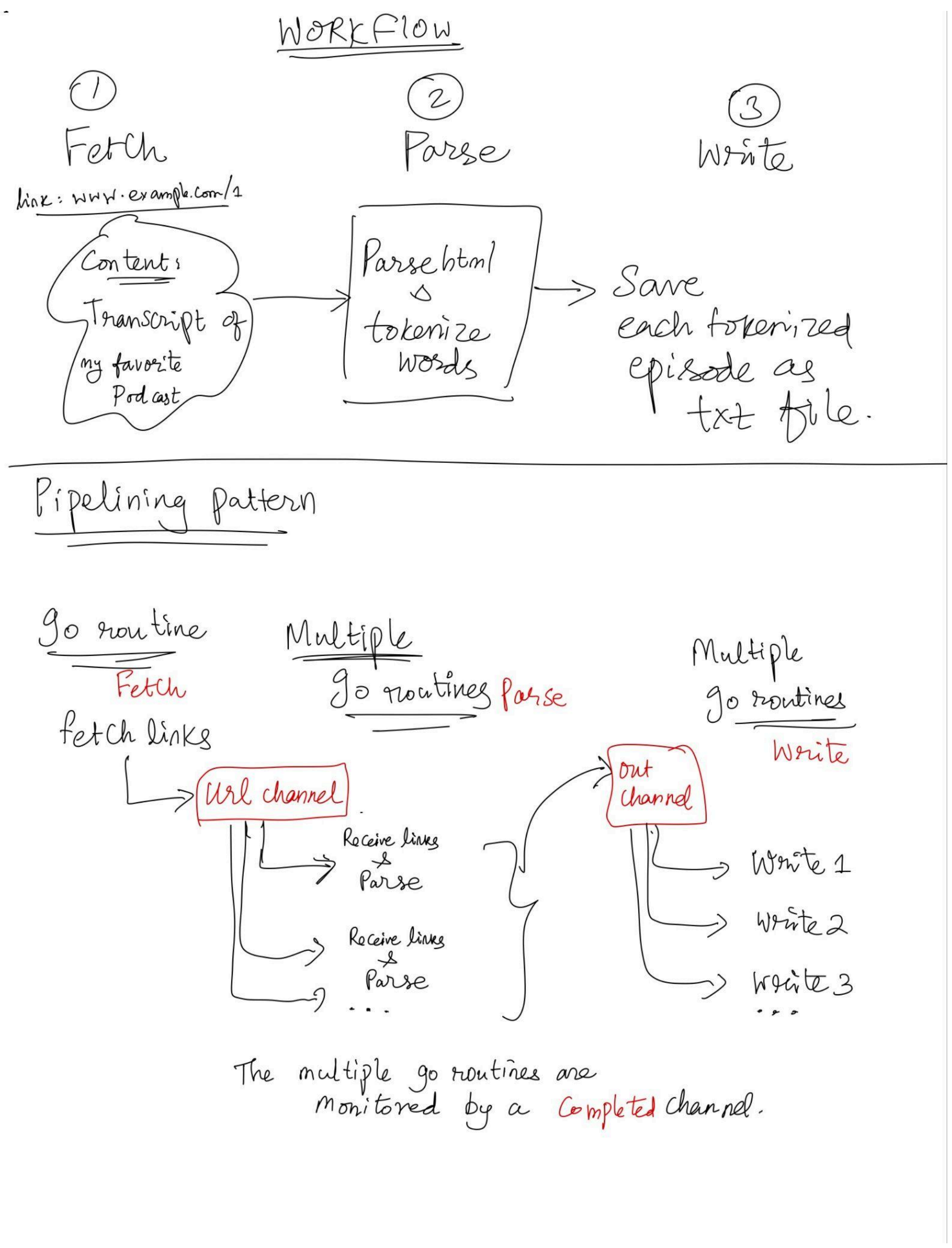
"go run parse.go 100 2 true" - 100 links, 2 threads and workstealing
"go run parse.go 100" - 100 links sequential

Approach picked for parallelization:

I picked the pipeline approach, as it lends itself well to this problem. We have independent individual chunks of tasks, which each have their bottleneck in the form of i/o operations. Hence this “asynchronous” sort of approach that the pipeline method offers is very suitable for this. Additionally, the results are not order dependent. Each task has a separate modular pathway. And tasks are comparable in their input to output step loads.

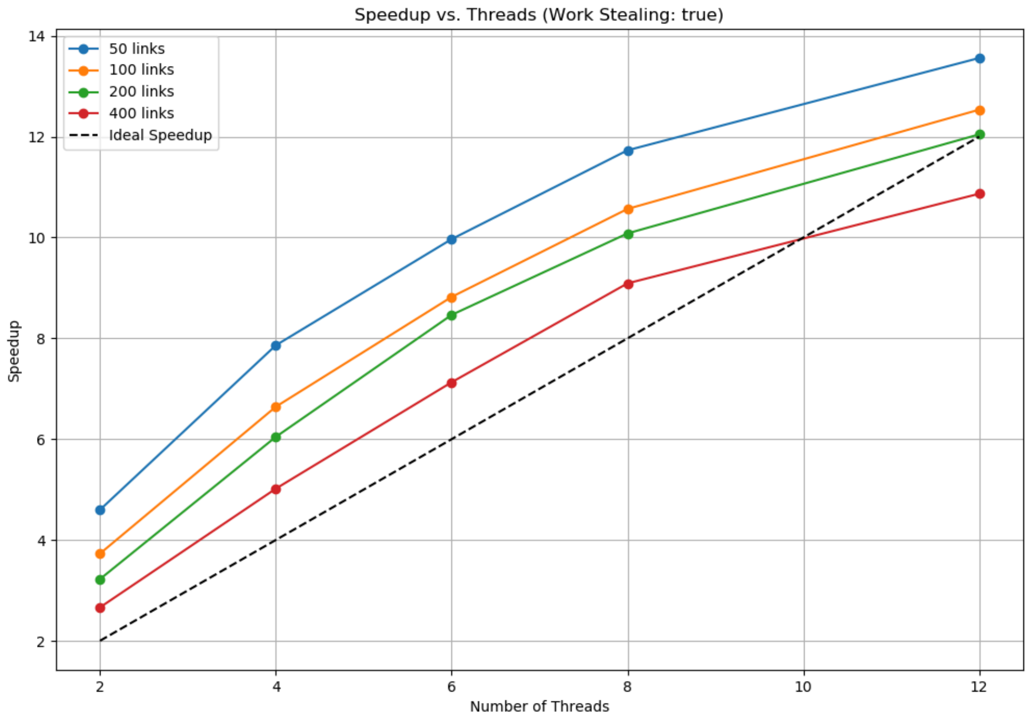
The big challenge i faced was handling the channels when multiple go routines are spawned. A single go routine performing the tasks is the most frequently used example online. Hence it was difficult to find examples of synchronizing multiple go routines without using a waitgroup.

Second, it was difficult to handle queues with channels when spawning multiple go routines because a thread could not get started on the tasks. I settled on a round robin way of distributing tasks to queues, which causes this issue. I will do queue by queue populating and then send the queues to the workers so that they can get started.

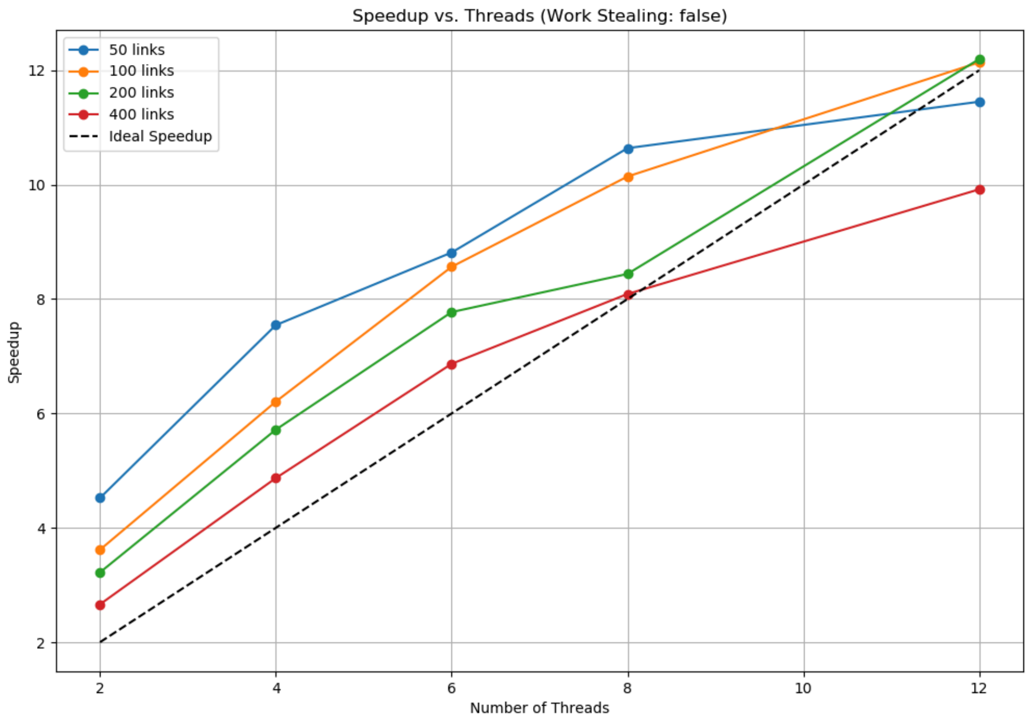


Workstealing did improve performance. This is because there was sufficient variability in the lengths of the tasks, that there was a difference in timings. Initially, I was not randomizing the stealing pattern, but then I looked at the lecture, where the professor alluded that randomizing stealing leads to good results. Workstealing works better when there are more tasks and more threads, which can steal from others. This is what I see in practice as well.
Find the below plots to explain this (read ideal speedup as theoretical linear speed up line).

The two parallel runs have different speedups for a larger number of tasks at a greater number of threads.



Speedup vs. Threads (Work Stealing: true)
50 links
100 links
200 links
400 links
Ideal Speedup
Speedup
Number of Threads
2
4
6
8
10
12
14



Speedup vs. Threads (Work Stealing: false)
50 links
100 links
200 links
400 links
Ideal Speedup
Speedup
Number of Threads
2
4
6
8
10
12

Hotspots that I parallelized:

Fetching URLs: Network requests are independent and can be parallelized.

Parsing and Tokenizing: Each fetched content can be processed independently.

Writing Results: Writing results to disk can be parallelized, as each result is independent.

Network Latency: Dominates sequential execution time. Parallelizing network requests significantly reduces total waiting time. This is why I get super-linear speed-ups.

Bottlenecks still in parallel :

Task Generation: Generating the initial list of URLs is sequential but negligible overhead.

Task pushbottoms to the queues. Currently, I perform a round robin addition of tasks to ALL QUEUES before sending it to the workers, this is a bottleneck.

Random number generation is a bottleneck but it's a quirk of this program to help generate variable tasks.