Parallel implementation of zlib compression & decompression

Authors: Pengyun Zhao, Zian Ke
Andrew ID: pengyunz, ziank
April 24, 2020

1 Current Progress and Possible Adjustments

The following things have been done (or in progress):

(1) According to schedule, we have read the source code for the Go implementation of zlib as well as the source code of pgzip, since they are written in Go, the language of our choice, and thus will provide us much more guidance on the project. Due to the difference between gzip and zlib specifications, we have also read RFC 1950 for zlib and RFC 1952 for gzip since zlib and gzip are both based on the DEFLATE algorithm and we need to understand the differences.

(2) Also according to schedule, we have implemented the compressor (Writer in Go) and have written some basic test cases to verify its correctness. One of the tests is to compress a string with our parallel zlib, then decompress it with the standard zlib decompressor. The output string is exactly the same as input. This ensures that our parallel zlib compressor is compatible with the standard zlib decompressor in Go.

Some changes we would like to introduce on our future schedule:

(1) We noticed that some descriptions on the project schedule may be incorrect. Instead of only implementing DEFLATE and INFLATE algorithms we are actually implementing the compressor (writer) that uses the DEFLATE algorithm and the decompressor (reader) that uses the INFLATE algorithm. The DEFLATE and INFLATE algorithms are encapsulated by the Go "compress/flate" package, though some extra methods, e.g., ResetDict(), will need to be exposed to our implementation. Therefore, we'd like to make corrections to some of our task descriptions.

(2) We noticed that putting correctness test and debugging after implementing the complete program may cause the debugging process to be more complex, so based on the principle of Incremental Development, we decided to split testing into blocks, with one after implementing the decompressor and one after implementing the compressor.

2 Future Schedule

	Task	By
4/25-4/28	Implementation of parallel decompressor using INFLATE	Zian Ke
4/25-4/27	Correctness testing & Debugging for compressor	Pengyun Zhao
4/28-4/29	Datasets collection	Zian Ke
4/28-4/30	Correctness testing & Debugging for decompressor	Pengyun Zhao
4/30-5/2	Benchmarks	Team
5/3-5/4	Final report	Team
5/5	Presentation	Team

A gantt chart of the timeline is as follows:

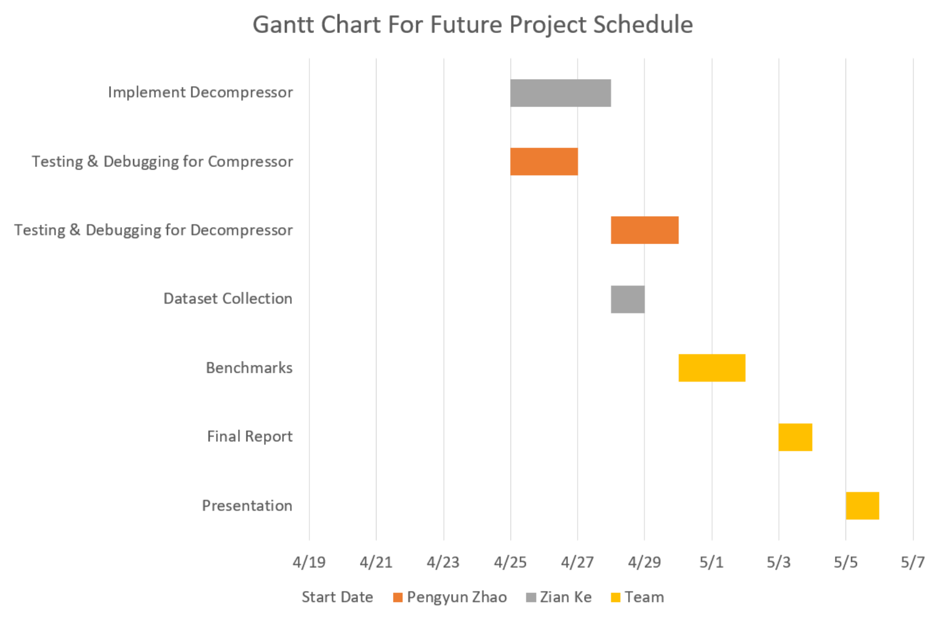

Figure 1: Gantt Chart for Future Project Schedule

3 Deliverables

Based on the fact that our current progress is going along quite well with our schedule, we think we will be able to achiveve the goals we plan to achieve stated in the project proposal. As for the "nice to haves", proving the efficiency and open source release are still challenging goals as they require a lot of work such as a diverse collection of datasets and use cases as well as detailed documentation and user support, but those goals are a good way to keep us motivated.

Also since now AWS is also offered as an option, we are interested in the program performance on different types of machines (GHC, Latedays, AWS) with different number and type of cores, and we want to include in the "nice to haves" in our goals.

So the new goals we plan to achieve are:

- Implement a parallelized version of zlib in Go that produces correct compression and decompression results and is fully compatible with the standard Go zlib package, i.e., the data compressed by our parallel zlib can be decompressed by the standard zlib, and vice versa.
- Achieve noticeable speedup compared to the original sequential implementation.
- Have a comprehensive analysis on the implementation for different kinds of workloads. What are the new bottlenecks? What kind(s) of workload is suited for the parallel implementation?

Goals that we hope to achieve:

- Achieve similar performance (or even better) with the currently released versions of parallel implementations.
- Prove the efficiency of parallel zlib in real-world applications such as large file compression and transport.
- Aim for open source release as a third-party Go package.
- **Analyze the program's performance on different machines. On which machines does the program have a higher performance, on which ones does it have a lower performance? What environment is it more suited for?**

4 Presentation

We plan to present a combination of demonstrations as well as charts and graphs.

Demonstrations are used to showcase the correctness of our program, i.e. it follows the zlib standard correctly and produces exactly same output (both decompression and compression).

Charts and graphs are used to represent the program's performance characteristics on different workloads and different machines and will help us present our analysis.

5 Preliminary Results

Based on our current implementation and testings, we have ensured that with a parallel zlib compressor that spawns multiple gorountines and compress multiple blocks concurrently, the correctness can still be guaranteed, i.e., the compressed result is fully compatible with a non-parallel zlib decompressor. We are also pretty sure that our parallel zlib compressor satisfies the requirement on RFC 1950, as we are strictly following the definitions of header, checksum, etc., by taking the implementations of the standard zlib package in Go as reference.

6 Issues

A minor issue we currently have is about writing test cases. We planned to reuse the test cases in the standard zlib package in Go, as our implementation is expected to be compatible with it. However, those test cases have a dependency on the "internal/testenv" package, while Go does not allow third party packages to import internal packages. Hence we'll need to fully understand the standard zlib test cases and attempt to reimplement them, and also add more test cases, e.g., to test concurrency.