Parallel implementation of zlib compression & decompression

Authors: Pengyun Zhao, Zian Ke
Andrew ID: pengyunz, ziank
April 11, 2020

1 Summary

We are going to implement a parallelized version of zlib, a library used for data compression and decompression, and benchmark its performance against the sequential version.

2 Background

zlib is a software library used for data compression and is an abstraction of the DEFLATE and INFLATE algorithms. It is an important library used in major programming languages and operating systems.

The DEFLATE algorithm, used by zlib and will be our targer for parallelism, is a lossless data compression algorithm that uses a combination of two compression strategies – Huffman coding and LZ77 compression. The algorithm first breaks data into blocks, after which LZ77 is first used for duplicate string elimination (duplicate strings are replaced with pointers) followed by applying Huffman coding for bit reduction to replace commonly used symbols with shorter representations and less commonly used symbols with longer representations. The decompression algorithm, INFLATE, is a simple reversal of the process.

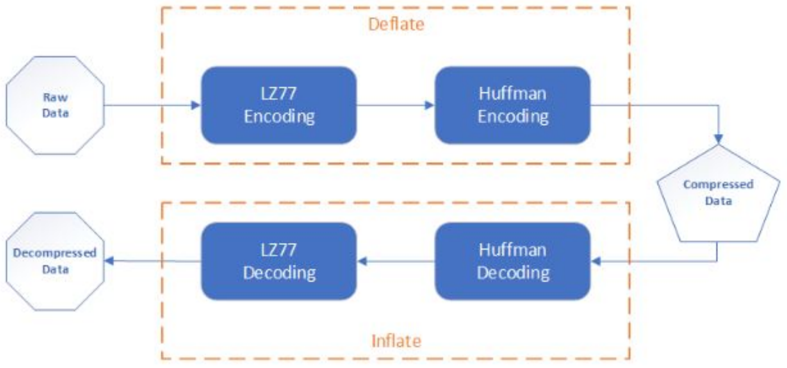

Figure 1: Diagram for the DEFLATE and INFLATE algorithm

3 The Challenge

Obviously, the zlib algorithm will have very long running time when encountering data of large scale, but we can see from the characteristic of the algorithm breaking data stream into blocks that it naturally allows for parallelism. The current bottleneck is its time spent on data I/O, duplicate search and elimination (across the whole data stream) and encoding data with Huffman coding. By distributing data chunks among processes, we may be able to achieve a great speedup of the performance.

The difficulty of parallelism lies in the dependence between consecutive chunks: the latter bytes of a deflated stream depend on the earlier bytes of that stream. A solution to this problem is to supply a certain amount of uncompressed data proceeding a particular chunk. We may also need to find a way to share the same dictionary used in encoding among multiple threads.

4 Resources

- To have a comprehensive understanding of the zlib algorithms, we can read the article "An Explanation of the Deflate Algorithm" by Antaeus Feldspar and the implementation of zlib by the Go authors.
- Mark Adler, one of the authors of zlib, has completed a parallel implementation of gzip, named pigz. This implementation will give us the basic ideas of making the DEFLATE and INFLATE algorithms parallelized.
- We'll also take pgzip, a Go implementation of parallel gzip compression & decompression, as reference, as we are going to implement the parallel zlib with Go and gzip shares similar underlying algorithms with zlib.
- Our benchmarks will be run on multi-core servers, such as the GHC and Latedays machines.

5 Goals & Deliverables

Goals that we plan to achieve:

- 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.

To ensure the correctness of our implementation, we need to write a test suite with high code coverage. We can take the unit tests of the standard Go zlib package as reference and reuse the same data as input. It's also important to write tests with large number of Goroutines and have the Go Race Detector turned on, in order to check whether any race condition exists.

To test the performance of our immplementation, we need to run benchmarks against the standard Go zlib package, and study the relationship between speedup and number of processors. The benchmarks will be run on multi-core servers, such as the GHC and Latedays machines, or cloud instances with more processors such as Amazon EC2 c5.9xlarge if necessary.

At the final poster session, we expect to present graphs of our benchmark results, as well as give a demonstration of data compression and decompression using our parallel zlib and show its compatibility with the standard zlib.

6 Platform Choice

The language we are going to use is Go. Go is proved to perform very well for concurrency. As the parallelized version of zlib is mainly based on multi-threading, we think Goroutine can be a powerful tool to achieve both high performance and concise implementation. We also plan to run benchmarks on our implementation against pigz, to gain more insights on the performance comparison between Go and C.

7 Schedule

	Task
4/16-4/18	Reviewing sequential zlib algorithms
4/19-4/23	Implementation of parallel DEFLATE
4/24	Checkpoint report
4/25-4/27	Implementation of parallel INFLATE
4/28-4/30	Correctness testing & Debugging
5/1-5/2	Datasets collection & Benchmarks
5/3-5/4	Final report
5/5	Presentation