Performance Analysis of Decompressor

Peng Wei
MPCS 52060, University of Chicago

Problem

Dealing with multiple compressed files before any actual data science job is frustrating, especially every csv file is compressed into different format. It's a tedious job for human beings. Decompressor is designed to deal with this situation.

System

input/output

The system reads in a text file from stdin. Parse it line by line. Each line is a repo's location. It walk all the files in the repo recursively. And decompress all the compressed files with format of .bz2, .gz, .zlib. Store the results in its original location.

Parallel design and Advance features

The system using fanIn-fanOut and Pipeline pattern to do the job. One goroutine acts as generator walk the path from the root, it pump the file path to a path channel.If there is some error happend, pump it to the error channel. Another goroutine distributes the path from the path channel based on the format of the file. So the compressed file paths are fanOut into 3 channels. Then sprawn one goroutine for each file to decompress the file. Each has a error channel to hold the errors from the processing. At last, Merge all 4 error channel into 1 error channel, then put it in stdout.

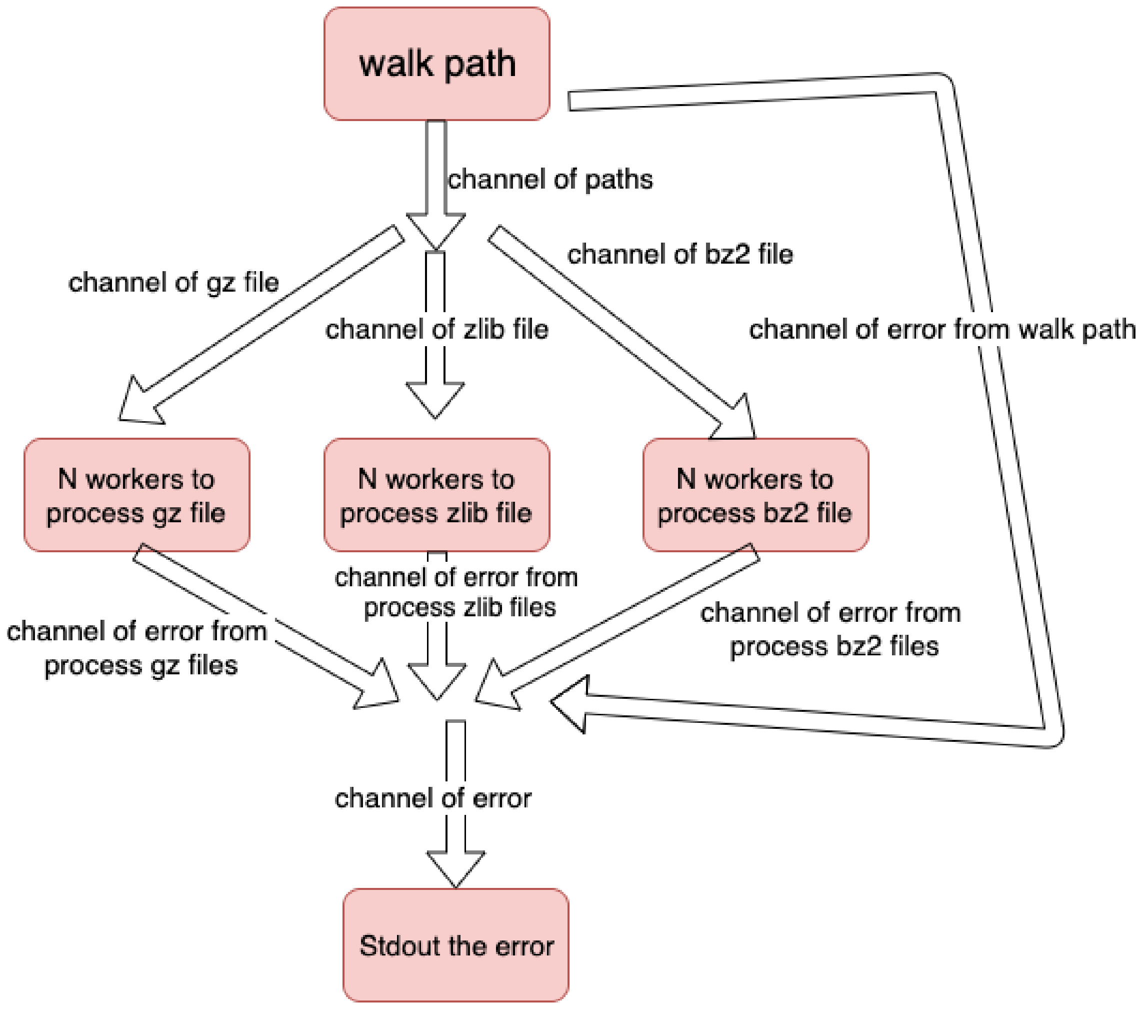
walk path
channel of paths
channel of gz file
channel of zlib file
channel of bz2 file
channel of error from walk path
N workers to process gz file
N workers to process zlib file
N workers to process bz2 file
channel of error from process gz files
channel of error from process zlib files
channel of error from process bz2 files
channel of error
Stdout the error

Figure 1: the design of the system

Data decomposition: Distribute all the compressed data into three channels, and for each file give a goroutine to process it.

Function decomposition: Split the whole procedure into 4 layer.

1.Generator layer walk the path.

2.Distribute layer distributes the job.

3.Process layer process the job.

4.Merge layer merge all the errors together.

Experiment of the performance

OS

Ubuntu 16.04.6 LTS

CPUs and RAM

the following is the profile of the CPUs' profile.

Architecture:	$x86_64$
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	12
On-line CPU(s) list:	0-11
Thread(s) per core:	1
NUMA node(s):	1
Model:	158
Model name:	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
Stepping:	10
CPU MHz:	2208.000
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	9216K

There are $16G$ ram in this machine.

Results and Analysis

hotspots and bottlenecks

Since decompressing the file require a lot of IO operation, which will make the core idle in the sequential version of the system. The bottlenecks are reading from disk and writing it back to the disk. The hotspot is the part that processing the compressed files.

Since all the algorithm are mature to perform the work, I didn't modify the code, just use it from the standard library. The hotspot remain the same in the parallel version.

However, in the parallel version, the system removed the bottleneck by spawning unbounded number of goroutines. That means if one thread is waiting for the IO operation. We can assign it to a new goroutine to process the file that is already in the memory.

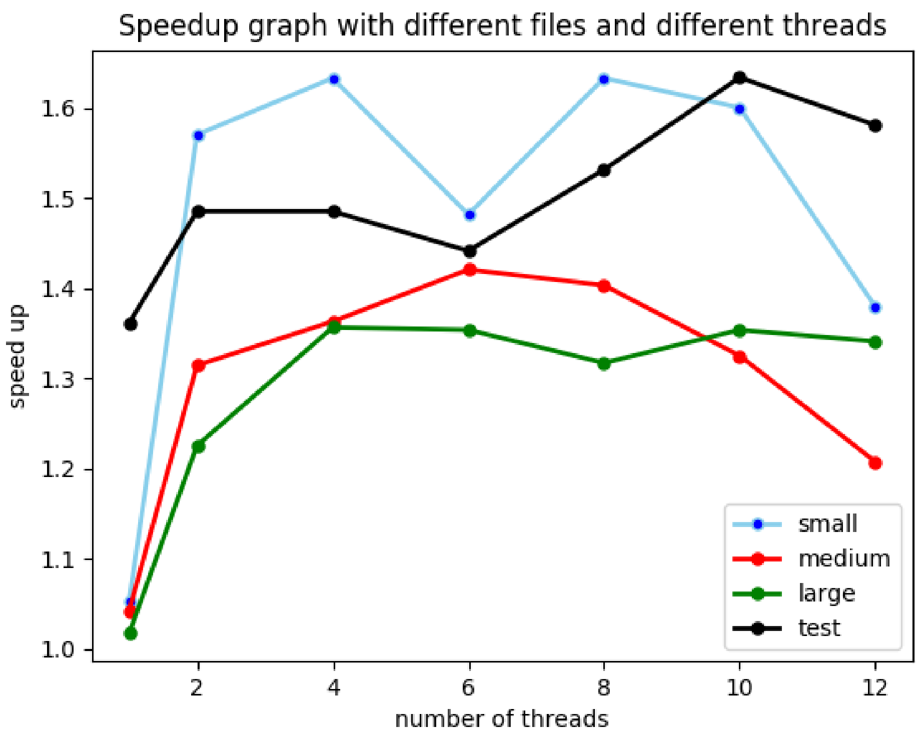

Figure 2: the performance of the system

Limitation and problems

we have 4 different kind of test data.

size	number of gz file	number of bz2 file	number of zlib
Small(86.6 kb)	3	3	3
Medium (2.3 mb)	4	4	4
Large (223 mb)	5	5	5
Test (1.6 kb)	21	21	21

The number of cores that can be used at run time are $1, 2, 4, 6, 8, 10, 12$.

what limited the speedup?

As discussed before, the system removed the bottleneck by allowing the idle core to perform the other jobs. Since the IO operation is the most time consuming part of the system, and the decompressing algorithm is relatively fast. IO operations are the deterministic factor in the system. From the performance graph, we can tell that the test with relative small size data over-performs the test with medium and large size data. It means that if the number of files that are similar, the larger file will require more IO operations than the smaller file. And the speedup we can get from the larger file is smaller than that from the small files.

On the other hand, if we compare the performance of the test data and the small data. There are 63 files in the test data but 9 files in the small data. Although the size of the files are similar. The system prefer the small number of files. It requires less IO operation than that from the test data.

To sum up, it's the IO operation limited the speedup.

Is it a lack of parallelism?

Yes. if we can split the larger file into smaller chunks, we can only process each chunk instead of the whole large file. It seems a reasonable approach.

However, since every compressed file has some header part to store the information that can be used to decompressing. For example, Huffman tree is stored in the header of the compressed file if we use Huffman coding to compress the file. To split the compressed file into chunk, we must make sure each chunk can access the shared header information. It will require a lot of job overhead. The result may not be appealing.

Communication or synchronization overhead?

The system is using pure messing-passing model instead of the shared-memory model.

There is very few place that needs to communicate or synchronization overhead. The communication happened between each layer of the system. For example, when we distribute the file path based on the type of the file into three channels. The message passed via channel from the distribution layer to the processing layer. Between the processing layer and merging layer, all 4 smaller error channel push error to one last error channel. With the help of the channel, the communication and synchronization is relatively easy than the shared-memory model.

However, as we mentioned above, if we want to split the larger file into chunks, it will require communication or synchronization overhead. Because, all the header information are shared between chunks.