Parallel IO withbject store

Robert G. Clapp

ABSTRACT

Cloud-based object-store offers large bandwidth but significantly increased latency compared to block and file store systems. To achieve performance on cloud-based object-store systems datasets can be broken into multiple objects, and IO operations are done in parallel. Performance on Google's Cloud Processing system using this technique is described. To scale to hundreds to thousands of instances datasets structure must be redesigbed to allow the concatenation of thousands datasets with minimal additional booking. A modification to SEP's SEP-3D grid based processing system is described.

INTRODUCTION

For performance and scale reasons object-store has become the dominant approach to cloud computing. Object-store based systems have an order of magnitude larger latency but offer significantly larger total bandwidth than conventional parallel file or block storage systems.

To achieve performance on an object-store system IO must be done in parallel with large sized blocks to hide its inherent latency. In this paper I build on the library described in Clapp (2018), which broke datasets into blocks, to implement parallel IO on Google's Cloud Processing (GCP) system for Regular Sampled Functions (RSF) datasets.

In most cases seismic datasets are at some level irregular. Twenty years ago SEP designed SEP3D (Biondi et al., 1996) which relied on placing a regular grid on irregular traces to allow quick transversal of datasets. In this paper I suggest changes to the SEP3D approach that improve performance and allow it to scale to be decomposed/composed into hundreds/thousands of instances.

PARALLEL OBJECT-STORE

While converting an application to run on the cloud can be a relatively painless process, converting it to run efficiently on the cloud can be much more challenging. When it comes to IO, you can choose to build your own parallel file system in the cloud, which is costly in both time and money, or use the cloud providers' preferred method converting your IO to an object-store based methodology.

The reason cloud providers have embraced object-store has to do with scalability. The book keeping associated with file storage is limited to millions rather than billions or trillions of files. In addition, things like file locking, guaranteeing a single instance is modifying a file at a given time, also does not scale well. As a result object-store was conceived. Object-store consists of a flat name space where each object has a unique key used to access the object. In addition, in most object-store systems, objects are read-only once created. As a result, file locking becomes unnecessary.

A naive conversion to cloud-based object-store leads to poor performance due to latency. In the cloud, objects are usually in the same data center but are unlikely to be even in the same rack. As a result latency that is in the tenths of milliseconds for SSD drives, low 10s of milliseconds for HDD, becomes hundreds of milliseconds for objects. In addition, the bandwidth to SSD (500+ MB/s) and HDD (200MB/s) are closer to (120 MB/s) for object-store. An IO pattern with many small reads is going to have disastrous performance on object-store systems.

Achieving good performance with object-store requires a different approach. The `buffers` library described in Clapp (2018) breaks up regular sampled data, data that can be described by hypercubes, into multi-dimensional blocks. Each block is written to a different file or object. When the user requests to read a multi-dimensional window of the dataset, the library figures out what blocks contain the window requested. The library reads all of the blocks containing the requested data, and outputs the portion corresponding to the requested window parameters.

Describing a dataset as multiple blocks can be very beneficial in an object-store framework. As stated earlier, one of the reason for object-store is it scalability. If we have several buffers to read, we can spawn multiple threads, each reading a different buffer in parallel. We still pay the initial latency for reading the first byte, but by doing large reads and launching many reads simultaneously we can start to hide latency in a manner similar to how GPUs hide global memory latency.

GCP implementation

Google provides a C++ library to interface for its object-store library (Google, 2019). The basic procedure is to create a communicator, called a `gcs::Client`. The library requires the environmental variable `projectID` to be set with the user's GCP project ID. Objects are grouped in buckets belong to a project. In my implementation, the user provides the project ID (think billing mechanism) through the environmental variable `projectID`. If creating a bucket, the user also must specify the region (what data center) to create the bucket through the environmental variable `region`. The library writes datasets into pseudo-directories, basically prepending a name followed by a '/' to all objects associated with a given dataset.

To test the library I created a 10GB 4-dimensional dataset. For the first test I read the entire dataset on a 32-core instance, modifying the size of the blocks. The following table shows the performance as a function of block size.

Block size (MB)	Speed (MB/s)
.64	130
1.28	230
3	500
6	1000
12	1700
25	2400
50	2600

Using the block size of 25MB I then read on a series of different sized instances. As you can see performance scaled with number of cores.

Cores	Speed (MB/s)
4	1100
8	1900
16	2400
32	2400

One of the largest benefits of a cloud-based solution is that this approach also scales over instances. I can use multiple instances each reading a portion of the dataset and get nearly linear increases in bandwidth. These performance on a single node are possible to achieve with a parallel file system; when spanning over many nodes, the aggregate bandwidth of an object-store approach can not be matched.

Using compression, these speeds can be improved even further. By using the optimal block size and ZFP compression embedded in the `buffer` library we can achieve read speeds of up to 5000 MB/s.

DATA FORMAT

The above section indicates the advantage of using cloud-based object-store for IO. A fundamental limitation of the above approach is that it is designed for regular sampled data. Seismic data is irregular in most of the processing flow.

There are two fundamental concepts in SEP3D. The first is to separate the headers from data. The rational is that many early operations in the processing flow only require accessing the headers. By separating the headers from the data, significant IO can be saved. The second major concept is that to tranverse an irregular dataset, it is still useful to impose a regular grid upon based on properties of the headers. This grid will basically bin the data into different buckets that share similar header information.

Figure 1 shows the layout of a SEP3D dataset. It consists of six files:

History file which stores both the processing history, the number of traces, and the sampling in time. It also contains pointers to the location of the data, history format file, and grid format file.

Data A binary data file which contains all of the traces.

Header format file Which contains the list of header keys, the number of headers, and a pointer to the headers in binary format.

Headers A binary representation of the headers.

Grid format file Which contains a description of the regular grid imposed on the irregular data. It also contains a pointer to a binary representation of the grid.

Grid Though the standard does not define the format of the file, it has always been represented as an integer list of either the header number specifying to a given grid location, or a `-1` specifying a location that doesn't exist.

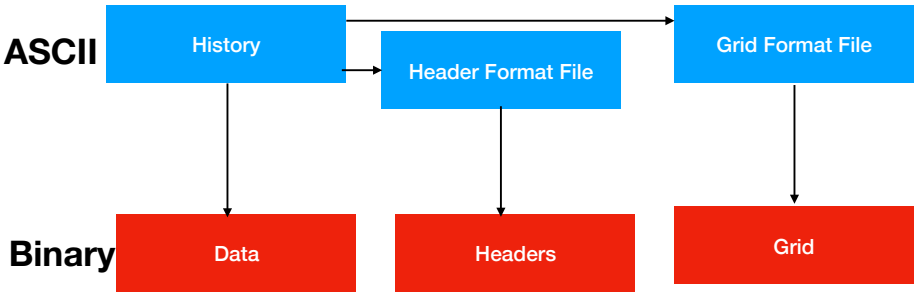

Figure 1: The six files representing a SEP3D dataset. The arrows represent which files point to other files.

To read traces from a specific grid cell meant first reading in a portion of the grid. From the grid you could get a pointer to a specific header. You could then read that header, looking for a key entitled `data_record_number` (DRN). If the DRN existed you would then know the trace you wanted to access. If the DRN did not exist the header number would be the same as the trace number. The pattern to read a trace is shown graphically in Figure 2.

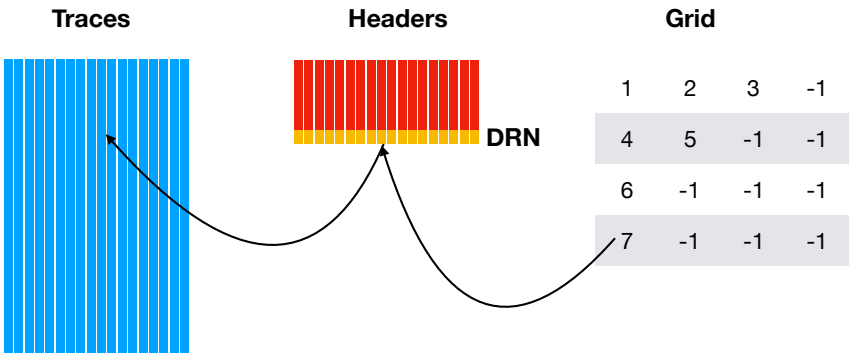

Figure 2: How to read a trace from a SEP3D dataset. First read the grid to get a pointer into the headers. Look in the headers for DRN key, which provides the location within in the traces.

There were several deficiencies of this approach:

- Too many files, all text based information should exist in a single file. The three used by SEP3D just added confusion.
- Using the relation between header location and trace location as a header key. The trace location should be independent.
- Limited to irregularity only at the trace level.

The first change is to allow the regular portion of the dataset to be any dimensionality. For example, the RTM volume,sorted by shot, would be an example of an irregular dataset where the 3-D single shot image is the regular portion. To improve parallel performance, both the headers and data can be broken into blocks. Next, the relation between headers and data is stored as part of the history file. What use to be the header key as a `data_record_number` is now appended to the history file. The grid is potentially broken into multiple files and consists of only 1s and 0s. The library converts these 1s and 0s into header locations. The advantage of 1s and 0s is that it allows the grid to be highly compressed.

An advantage of modifying how positioning is handled in the grid and in the history file is that it lends itself well to running massively parallel jobs on a cloud-

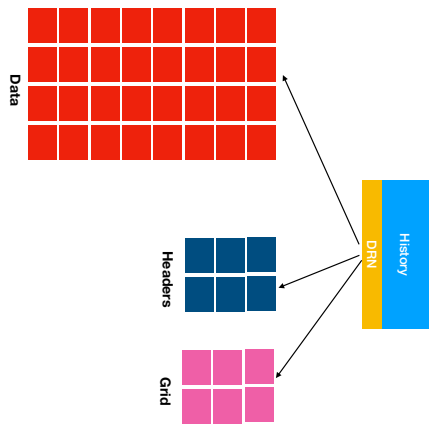

Figure 3: The diagram of an updated data format. The history file also contains the grid, header key description, and the trace order. The data and headers are broken into many files.

based system. Many nodes can all read from a large initial dataset, each acting on part of the file. These instances can all create smaller datasets. With this format description, the datasets can be cheaply and quickly recomposed back into a large dataset. On the cloud, concatenation of objects is a cheap exercise. If we tried to concatenate a normal SEP3D dataset, the grid and headers would have to be processed before concatenation in order to update positioning information. By storing the grids as simply ones and zeros, and removing the DRN from the headers, all of the large objects - grids, headers, and data - can simply be concatenated. Only the history files, the smallest objects, need to be processed.

CONCLUSIONS

In this paper, modifications to SEP's IO to better fit cloud environments is described. By breaking regular cubes into sub-cubes, and using many threads to do large reads, performance can significantly exceed conventional parallel file based systems. By keeping the general concept of grids from SEP-3D, but modifying the file formats to enable better performance and greater level of parallelism, a strategy to handle irregular data is proposed.

REFERENCES

Biondi, B., R. Clapp, and S. Crawley, 1996, Seplib90: Seplib for 3-D prestack data.

Clapp, R., 2018, Buffers: A library for fast parallel io and compression data.

Google, 2019, Google cloud client library for c++: `https://github.com/googleapis/google-cloud-cpp`.