Buffers: A library for fast parallel IO and compression

Robert G. Clapp

ABSTRACT

Data transfer speed improvements from different memory levels has continually lagged behind floating point operations per second (FLOPS) improvements. For seismic applications these differing growth curves have made more and more applications IO bound. We created a library that breaks a dataset into blocks. These blocks can be read/written to disk in parallel, significantly reducing IO time when using parallel file system or object store. Each block can further be compressed, using multi-dimension compression schemes, reducing the amount of data that needs to be transferred between memory levels.

INTRODUCTION

Until the early 2000s algorithms were judged primarily by the operation count. Today, due to the drastically different growth rates of Central Processing Unit (CPU) speed and main memory (McCalpin, 1995, 2007) almost all applications are judged based on either their memory bandwidth or memory latency behavior. These bottlenecks can come from many different memory levels: from disk, from main memory on the CPU, or even global memory on a General Purpose Graphical Processing Unit (GPGPU). To address the latency issue, it is important to read in large chunks. To address the bandwidth issue, doing parallel reads can often prove performance particularly on parallel file and object store (Factor et al., 2005) system. Further performance improvements can be achieved through compression. Seismic data have been shown to be compressible when using multi-dimensional schemes(Villasenor et al., 1996).

In this paper we describe a library, called `buffers`, that breaks a multi-dimensional hypercube into blocks. These blocks can be compressed using public domain compression packages designed for scientific computing such as ZFP(Lindstrom and Isenburg, 2006). The library implements its own crude caching system that can be used to minimize memory usage. The library `genericIO`(Clapp, 2017) has been extended to use the `buffers` library. As a result, any program using this library for IO can automatically taking advantage of the speedup it offers.

We will begin by describing the interfaces to the library. We will then present some preliminary results showing both the speedup in IO and the error using various compression schemes.

LIBRARY

The `buffers` library functions are controlled by three objects (classes): how/whether to compress the data, the scheme used to block the hypercube, and how to handle memory usage. By default it does not compress the data, it holds everything in memory, and it is a relatively smart blocking scheme for conventional CPU architectures.

Compression

Currently two different compression schemes are offered. The first is `noCompression`. The second option uses the ZFP library to compress the data. ZFP is basically a very fast 1, 2, or 3-D JPEG compresser. For seismic data it can achieve compression rates of 6-10x with acceptable loss for most applications. Its big advantage is its speed. It can compress and decompress at nearly memory bandwidth. It is ideally suited for applications where data access speed is essential, such as visualization or compressing wavefields. Currently the `buffers` library supports the tolerance, precision, and rate methods ZFP uses.

Blocking

Blocking describes how to break up an N-D cube. By default it tries to makes blocks in the low Megabytes in size, maxing out in 3-D blocks (4 and 5-D blocks are better for the SZ library). The user can create his own block object specifying a target blocksize along each axis and desired chunk, the minimum unit to use along each axis (think cache line).

Memory usage

As mentioned above the library defaults to keeping every block in memory(`memoryAll`), often in an uncompressed state. For many applications this is an undesired behavior. As a result a second memory option, `simpleMemoryLimit`, exists. This class allows the user to set a target memory for buffers library. Every time a `buffers` function is called the `simpleMemoryLimit` module will attempt to first compress then store to disk blocks that have been unused for the longest period. Further, more sophisticated caching schemes might be added later.

Buffer functions

A `buffers` object can be created by either initializing with a hypercube (description of the N-D cube) and potentially the `compression`, `blocking`, and `memory` objects or by a JSON file that describes these choices. Most of the access to the actual data is through the `getWindow` and `putWindow` functions. As their names imply, they are used to get and put data to/from the `buffers` class. The user also has the ability to `changeState` which will change between the three possible states of a buffer: on disk, compressed, and decompressed. The final function `setDirectory` can be used to specify where to store the dataset.

GenericIO

A new IO object type, `BUFFERS` has been created within the `GenericIO` library. The `BUFFERS` IO class uses JSON for parameter handling and file descriptions. Rather than specifying a file, the user specifies the directory to read/write a `buffers` object.

RESULTS

To test the speed of library, I compared using standard SEPlib IO routine calls such as `sreed`, `sreed_window` with all code compiled with the same optimization level (3). The test used a 2.5 GB file of migrated seismic data from the Gulf of Mexico. The file compressed to 670 MB using the ZFP technique using constant rate compression method. I tested the read speed on a Mac, a local disk on a Linux server, and off of a parallel file system. On all systems the blocked ZFP dataset proved to be faster with speedups varying from 3.2 (Mac) to 12 (on an active parallel file system). Speedup on the cloud should be even more significant.

As a second test, I used a smaller dataset with lower wave number features. When compressed the dataset shrunk in size from 216 MB to 14 MB. Figure 1 shows an example of the errors introduced by using the ZFP compression method. The top left shows a slice from a migrated cube from the North Sea, the top right shows the result of reconstructing the cube. The bottom left shows the difference. The bottom right show the difference scaled by 100. Note how the error is small and generally random.

DISCUSSION AND CONCLUSIONS

In this paper, we described a new library `buffers`. The library is designed to help people reduce memory bandwidth bottlenecks. It allows the user to seamlessly read and write in parallel to improve disk IO. It also can be used in applications such as visualization and storing/compressing wavefields to greatly reduce the memory and disk bandwidth requirements.

REFERENCES

Clapp, R. G., 2017, Facilitating code distribution: Docker and Generic IO.

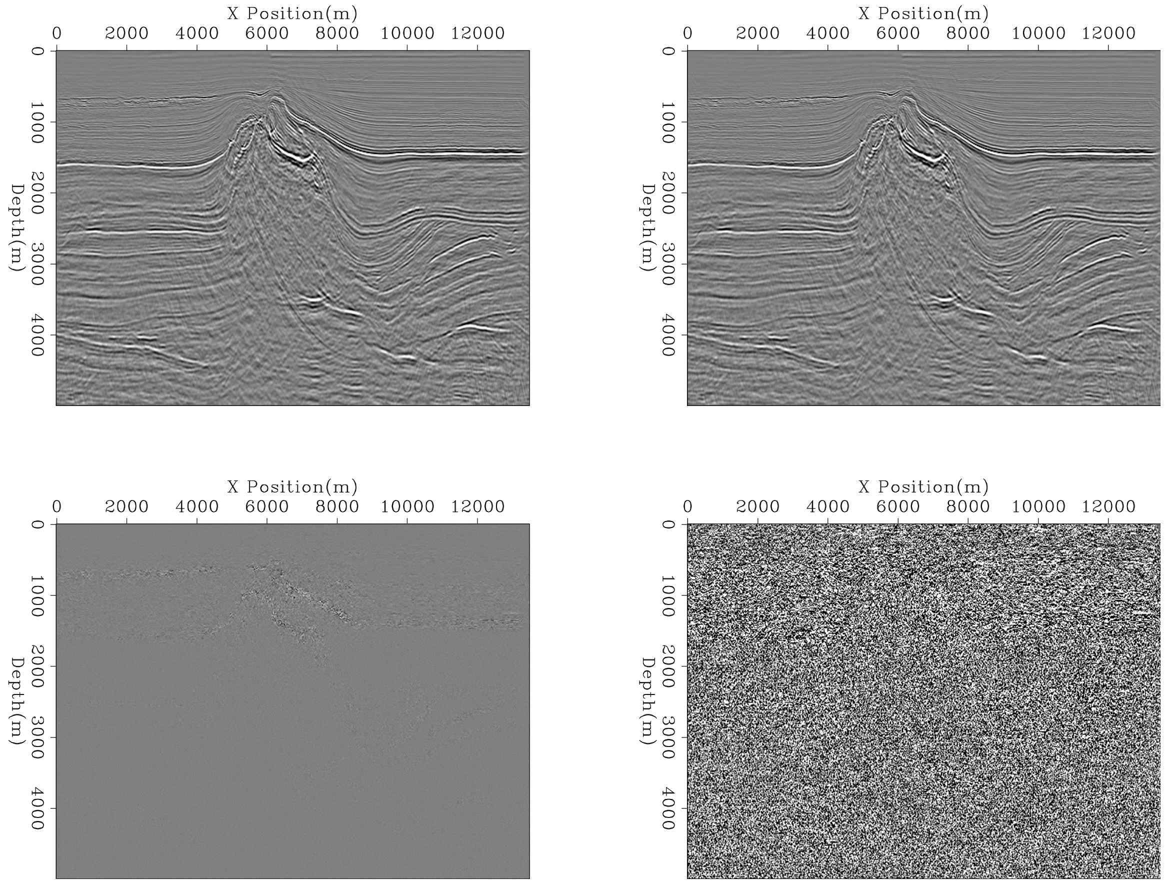

Figure 1: The top left shows a slice from a migrated cube from the North Sea, the top right shows the result of reconstructing the cube. The bottom left shows the difference. The bottom right show the difference scaled by 100. [**ER**]

Factor, M., K. Meth, D. Naor, O. Rodeh, and J. Satran, 2005, Object storage: The future building block for storage systems: Local to Global Data Interoperability-Challenges and Technologies, 2005, IEEE, 119–123.

Lindstrom, P., and M. Isenburg, 2006, Fast and efficient compression of floating-point data: **12**, 1245–50.

McCalpin, J. D., 1991-2007, Stream: Sustainable memory bandwidth in high performance computers: Technical report, University of Virginia, Charlottesville, Virginia. (A continually updated technical report. http://www.cs.virginia.edu/stream/).

———, 1995, Memory bandwidth and machine balance in current high performance computers: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 19–25.

Villasenor, J. P., R. A. Ergas, and P. L. Donoho, 1996, Seismic data compression using high-dimensional wavelet transforms: Snowbird, UT, USA, Proceedings of the Data Compression Conference, IEEE Computer Society Press, 396–405.