I/O Mini-apps, Compression, and I/O Libraries for Physics-based Simulations

Sean Ziegeler
DoD HPCMP PETTT / Engility
sean.ziegeler@engilitycorp.com

Scot Breitenfeld
The HDF Group
brtnfld@hdfgroup.org

Jose Renteria
DoD HPCMP PETTT / Engility
jose.renteria@engilitycorp.com

Jordan Henderson
The HDF Group
jhenderson@hdfgroup.org

Compression is attractive because it is often the least invasive data reduction method, but this advantage often comes at the cost of both compression ratio and I/O peformance. Here, we focus on performance. Presented is a mini-app designed, in part, to explore the effectiveness of compression for physics-based simulations that use I/O libraries. The DoD High Performance Computing Modernization Program (HPCMP) recently completed a project to create parallel I/O mini-apps: *MiniIO*. It consists of four open source mini-apps that represent major simulation types. The dedicated mini-app approach is desirable, compared to those that extract emulation kernels from codes, because many sets of changes and parameter settings can be applied to the mini-app based on physics concerns and their effects directly observed without re-extracting many kernels.

While all of the mini-apps are interesting for compression, one of the apps, *struct*, is designed in part to specifically address compression. It simulates an application with a static, structured 3D grid, 2D tiling of that grid across parallel ranks, time-variant data values, blanked values and load balancing. A real world example is an ocean model, where data points that are located on land would be "blanked-out" because no computations occur. For an equally-distributed parallel decomposition, this creates load imbalance, but struct includes the option to load balance. The problem for I/O in a structured grid is that blanked data points are marked as invalid but still remain in the arrays. Thus, the I/O *becomes load-imbalanced*; yet, compression could be a solution. Since blanked points in an array can be set to a constant value, these areas of the arrays can compress trivially, resulting in similar-sized compressed data tiles.

With compression, one must take care how the data structures are filled. Filling valid data points with a constant value results in trivial compression. Filling with pseudo-random data is also unfair, as the high apparent entropy results in poor compression. The mini-apps include a 4D coherent noise variable known as simplectic noise, related to the venerable Perlin noise. The simplectic noise is also used to generate the blanked regions and can generate Earth-like data sets, e.g., with large continents, by adjusting the frequency of the noise.

As part of the project, parallel compression was added to the HDF5 library. Collective operations now coordinate chunk operations to process through parallel filters and, for writes, collectively re-insert modified chunks as needed. HDF5 had only tested the following compression filters in parallel: zlib, szip, and shuffle+zlib (a bit shuffling filter inserted before zlib). Each of these filters with HDF5 have been benchmarked and included in the results so far.

The ADaptive I/O System (ADIOS) provides a single interface in MPI to multiple output methods, aka., *transport layers*. In this study, we examined four transport layers: POSIX, MPI, MPI-Lustre, and MPI-Aggregate. ADIOS implements compression through data "transformations." As with HDF5, zlib and szip tranformations were utilized. ADIOS does not offer a shuffle transformation, but *zfp* was utilized, which has error-bounded lossy compression.

The study explored the performance of HDF5 and ADIOS with the struct MiniIO mini-app on a Cray XC50 system with both Intel Broadwell and Knights Landing (KNL) cores on a Lustre file system. Computationally unbalanced and balanced loads were run with each combination of library output option. Struct was set to generate approximately 65% valid data points, based on typical global ocean models and some regional models. Core counts ranged from 512 - 21912. Zfp compression was set to an accuracy of 0.0001 and produced a 9x compression ratio on average. A subset of the results so far are included in Fig. 1.

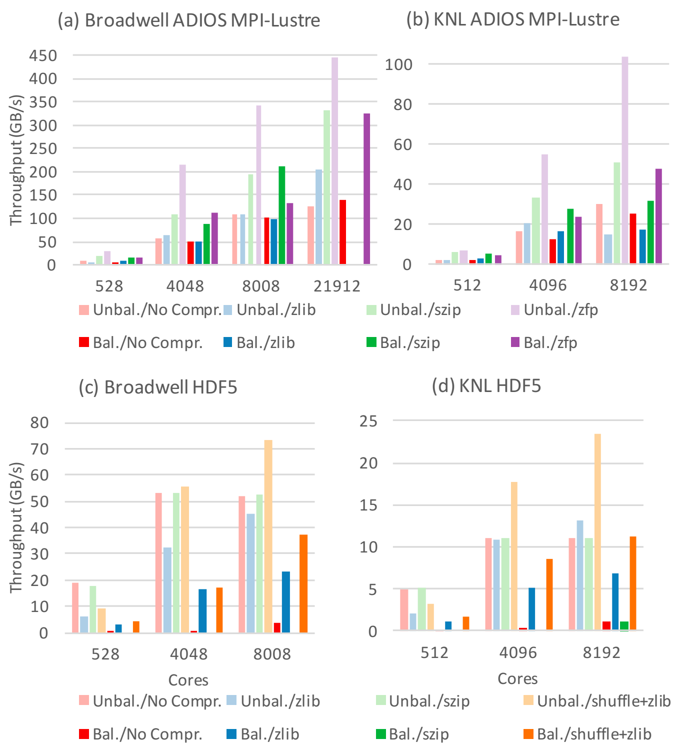

Figure 1. A sampling of compression-assisted throughput results with two output methods on two processor types.

Perhaps the most noticeable and promising trend across all results is compression scalability with core count. Nearly every output type on both processors, with/without load balancing, and compression method shows speed up with increasing cores. Zfp compression is so scalable that it results in throughputs that are faster than the bandwidth of the file system (about 200 GB/s). One sees overall performance degradation on KNL, as expected given the core counts, weak scaling being utilized, and weaker integer computation on KNL. Computational load-balancing indeed has a significant effect on performance. Note that for all, the unbalanced load is faster, and in several cases significantly faster. Compression does indeed partially fix the issue of performance loss due to load balancing (i.e., an imbalanced I/O load). In most cases, it outperforms an uncompressed, unbalanced load. Future work includes further scalability testing on Broadwell and Google Compute Engine to test the scalability of compression on Intel Skylake cores.