Abstract—In situ visualization promises to offer one solution to the problem of stagnant I/O bandwidths relative to computing capacity. Yet, it has drawbacks, including a lack of exploratory results. This can be addressed by producing derived or extracted results instead of just images. However, the I/O for these results is not guaranteed to be sufficiently scalable if not designed properly. We present an I/O mini app that measures derived quantity I/O performance. It produces isosurfaces with a data generation function allowing fine control of the volume, load imbalance, and other aspects of the isosurface geometry output data. We also provide ongoing results of scalability benchmarks with various output methodologies. (Abstract)

Keywords—data systems; visualization; high performance computing (key words)

I. INTRODUCTION

The Department of Defense (DoD) High Performance Computing Modernization Program (HPCMP) User Productivity Enhancement, Technology Transfer, and Training (PETTT) program initiated a project to create a series of I/O mini-apps. We refer to the collection as “MiniIO.” These mini-apps are intended to provide performance feedback information on various I/O methods and libraries in the context of common physics-based simulations. The HPCMP PETTT MiniIO mini-apps are available on GitHub under the Berkeley Software Distribution (BSD) license.

Tools do already exist to benchmark and measure parallel I/O. Some are pure I/O benchmarks, like IOR and midtest [1], which only measure some form of throughput or latency without simulation-based data structures. Others are benchmarks or tests dedicated to an I/O method or library, such as the MPI-I/O benchmarks in the Intel MPI Benchmarks (IMB) suite [2] or the NetCDF performance tests in the Parallel I/O library (PIO) [3]. Still others are I/O kernels from actual simulation codes, for example, the Hardware Accelerated Cosmology Code (HACC) I/O Benchmark [4] or the HYbrid Coordinate Ocean Model (HYCOM) I/O kernel [5]. Similarly, tools exist to extract I/O kernels from a larger code; an example is ADIOS skel [6]. Finally, MACSio attempts to model a variety of potential simulation data structure designs and provides a modular system to add I/O routines [7]. MACSio appears to have been conceived at around the same time or just before the MiniIO collection. There are several close similarities between the two tools, including support for physics-based simulation data structures, a modular, extensible I/O subsystem, and even a very similar method for generating synthetic data [8]. One important difference is that MACSio is designed as a single, more sophisticated application rather than a collection of smaller dedicated applications. In the context of this venue, the primary difference is that MACSio does not appear to plan to support derived in situ visualization data [7]. That is an important capability, since in situ visualization is slated to be a critical solution to the increasing performance gap between computation and I/O [9].

This work presents one of the PETTT I/O mini-apps, MiniIO/cartiso, or just “cartiso,” which is dedicated to measuring the I/O performance of example derived or extracted output as might be produced by an in situ visualization system. An early version was presented at [10]. The mini-app consists of three tightly integrated components: (1) lightweight data generation intended to produce simulation-like data fields, (2) a simple isosurface extractor that operates on the aforementioned data field to produce geometry data, and (3) a straightforward, modular system to add I/O methods and libraries for both the original data fields and geometry data. Timings can be collected to compare the performances of writing the “full,” original data to writing only the isosurface geometry. The variation in geometry data sparseness and potentially extreme load imbalance can introduce significant performance challenges; so, cartiso seeks to enable I/O library developers to ensure performant support for this data type.

Prior art exists in the measurement of parallel I/O performance for derived in situ output. Isosurface output from ParaView Catalyst [11] with a newly introduced ADIOS [6] writer was benchmarked and improved based upon those benchmarks [9]. The improvements were contributed back to the ADIOS community. A similar study was performed with VisIt Libsim [12] to benchmark and improve the XDB format [13]. In both cases, the studies refer to this type of output as “extraction.” Based on terminology from the In Situ Terminology Project, this type of in situ output is referred to as derived with a sub-type of proportional, because the size of the output scales with the input data [14].

II. DESIGN

The cartiso mini-app is named as such because it generates a Cartesian volumetric grid as the basis for computing an isosurface. The isosurface extraction algorithm is marching cubes, which parallelizes well. The mini-app utilizes only MPI for distributed parallelism with a 3D block parallel decomposition of the volume. No on-node parallelism scheme, e.g., threading, is currently employed; however, hybrid MPI-threaded codes can be emulated for the purposes of I/O measurement by simply reducing the number of MPI tasks per
node. Data generation and isosurface computation is lightweight enough that the computational cost of leaving idle compute cores is not currently a concern in this scenario.

A. Data Generation for Isosurface Placement

Cartiso is designed to generate data such that the resultant isosurface properties are quantitatively controlled. Because of this level of control, we argue that cartiso serves as a proxy for many potential derived output types. It achieves this using a combination of a Gaussian and sinusoid function, inspired by the VTK “wavelet” source [15], but modified as follows:

\[ G_{\text{sin}}(x) = e^{-\alpha(x-x_0)^2} \left[ \frac{\sin(2\pi fx) + 2}{\alpha A_f - \alpha + 1} - 1 \right] \]

The first major factor is the common Gaussian function, \( G(x) \) but without the usual scaling coefficient and with the addition of the “\( \alpha \)” parameter. The entire function produces a sinusoid within the envelope of the Gaussian. The function and parameters are illustrated in 1D in Fig. 1. The red line shows a 1D example of an isosurface; the red circles indicate the crossings and are conceptually equivalent to 3D isosurface points. The Gaussian standard deviation, \( \sigma \), controls the width. Setting the isosurface threshold greater than \( G(1\sigma) = e^{-1/2} \approx 0.61 \) guarantees that the isosurface width is \( \leq 2\sigma \).

\( x_0 \) sets the center of the Gaussian and therefore the isosurface. The sinusoid amplitude, as measured from the top, \( A_t \), can simply be fixed at \( \leq 1-G(1\sigma) \) to ensure the isosurface area is not too small. The sinusoid frequency, \( f \), controls the number of isosurface facets and therefore the amount of isosurface geometry data.

The final parameter, \( \alpha \), creates a linear morphing between a traditional sinusoid (\( \alpha = 0 \)) and the combined Gaussian*sinusoid (\( \alpha = 1 \)) to govern the load balancing of the isosurfaces. Fig. 2 shows the contrast between \( \alpha = 0 \) and \( \alpha = 1 \) both in 1D and 3D. In 3D, one can see more intuitively that at \( \alpha = 0 \), the isosurfaces are perfectly load balanced across the domain (assuming that \( f \) is at least equal to the number of parallel tasks in each dimension). At \( \alpha = 1 \), the maximum load imbalance (as set by \( \sigma \)) is reached.

B. Modes of Operation

Like many physics based simulations, cartiso iterates through a series of time steps. The number of time steps is configurable by the user. There are 3 possible modes of time step iteration: \textit{sin2gauss}, \textit{gaussmove}, and \textit{gaussresize}. The \textit{sin2gauss} mode linearly iterates through time from \( \alpha = 0 \) to \( \alpha = 1 \) to progress from load-balanced data to load imbalanced. The \textit{gaussmove} mode moves the center of the Gaussian, and isosurface, through the spatial domain of all ranks, using a serpentine space-filling curve to determine the order of movement. This helps the user determine if shifting the data across various parts of the domain has an effect on performance. The \textit{gaussresize} mode linearly iterates from a starting \( \sigma \) to a final \( \sigma \). The purpose is to help measure the performance difference between two intermediate amounts of load imbalance.
Several of the parameters, taken together, affect the total amount of isosurface geometry generated. This can be seen qualitatively in both Fig. 2 and 3. The factors that most affect this are sinusoid frequency, Gaussian σ, α, and also the size of the grid. Table 1 shows several examples of these settings.

All of the modes of operation are shown in Fig. 3. The top row of the figure shows three time steps of sin^2gauss. The second row shows three time steps of gaussmove, moving the isosurface about the domain. The left time step of the bottom row is the result of a gaussresize. The final two time steps on the bottom are another gaussmove, but in reverse to maintain continuity.

### C. Synthetic Mapped Data Generation

Notice in Fig. 3 that the isosurfaces are color-mapped with data values. Cartiso provides secondary data fields that can be mapped to the isosurface points. The purpose is to replicate the mapping of simulation variables to an isosurface, e.g., an isosurface of pressure with temperature mapped to it. This could represent a significant amount of data, depending upon the number of additional data fields. This secondary data field is generated by a derivation of Perlin noise [16]. Ken Perlin computationally improved the noise significantly in simplex noise [17], especially in dimensions higher than 2, but also patented that method. Since we generate the noise in 4D, we could not ignore this improvement, so we chose to use an open variant known as simplectic noise made available via the public domain OpenSimplexNoise C library [18].

The simplectic noise will eventually prove critical for compression efforts planned for later in this project. Data fields filled with a constant compress trivially and would provide an unfair advantage to output methods using compression. Conversely, pseudorandom noise compresses very poorly. True simulation data will be somewhere in between, with some amount of spatial coherence, depending upon the type of simulation. Simplectic noise is composed of a set of basis points with pseudorandom vectors and a non-linear falloff function. As such, sampling at a smaller period creates data with less apparent entropy, which would compress better. We also generate the noise in 4D in order to simulate reasonable time step changes in the data field. To provide initial evidence that simplectic noise would be effective, we ran compression tools on files of simplectic noise created at various sampling periods. As shown in Fig. 4, increasing the period reduces the compressibility. The compressibility reaches its apparent minimum as the period increases and matches the results of compressing pure pseudorandom noise (last point). Compression with gzip and bzip2 was not useful because the data is IEEE floating point; the HDF library overcomes this issue with a bit-shifting transform. We chose to work around it by using XZip (which is LZMA-based) and the PAQ [19] algorithm, which detects the entropy of floating-point numbers more effectively.

### D. Controls and Options

Cartiso currently provides command-line options to control all of the aforementioned mini-app design components. These are as follows: (1) number of parallel tasks along each dimension (in 3D), (2) size of the grid in 3D, (3) Gaussian σ in 3D (initial in gaussresize mode), (4) final σ in 3D (gaussresize mode), (5) Gaussian center in 3D manually specified, (6) Gaussian center automatically computed from domain, (7) sinusoid amplitude from the top, (8) sinusoid frequency in 3D, (9) spatial sampling of noise in 3D, (10) temporal sampling of noise, (11) number of time steps, (12) starting time step, (13) mode (sin^2gauss, gaussmove, gaussresize), (14) a flag to reverse the mode order, and (15) additional flags to enable different output modules and associated options. Future simplified options are planned and some are partially implemented. First, smarter defaults for the Gaussian σ values and sinusoid frequency that scale with task count are implemented in a script and will be moved directly into the application. Then, options will be added to allow such task scaling in conjunction with a coefficient for each argument. Finally, an option will be added to allow the user to select a target load imbalance and will automatically select a Gaussian σ to achieve that target.

### E. Output Design and Methods

Cartiso is two I/O mini-apps in one: full Cartesian data field output and isosurface geometry output. As the name suggests, the MiniIO project plans to address read performance as well as write. However, write performance is our current development priority, since this impacts simulation run time.

The code and build system is set up to allow simple modular additions of output methods for both output types. The process to add an output method starts with the Makefile: add include, library, and other flags with the option to disable it and
a definition to drive conditional compilation. In the main source file, there are blocks of code identified where various phases of the output module may be added, including header files, usage string documentation, flag variables, command line options, initialization, output per time step, and final cleanup. Each portion must be enclosed in the conditional compilation directives as defined in the Makefile.

The sections where one adds full output and isosurface output per time step are instrumented with timers. The timers currently do not distinguish operations within an output method, such as file opens, writes, and closes. Such information can indeed be important if one is trying to optimize the output method or just understand why one method is slower than another. However, some I/O libraries like ADIOS can produce this information upon request or an I/O profiling tool could be used.

Added output methods should focus on optimal output performance over compatibility with post-processing or visualization tools, unless compatibility is not expected to conflict. We want to compare I/O methodologies and libraries, not necessarily formats. Similarly, the number of files produced per time step is up to each output module, but should be clearly documented, as it can significantly affect performance. Since the purpose of cartiso is to measure I/O performance, we do not currently attempt to overlap data generation with writing, though output modules wishing to do this easily could. This may need an artificial delay to simulate more computation, since the data generation may not represent the computation time that the user may wish to overlap.

Cartiso currently includes three output modules each for full and isosurface output type. The first are the VTK formats: Parallel VTK Image (PVTI) for full output and Parallel VTK Polygonal (PVTP) for isosurfaces. Both utilize the MPI-IO individual approach; that is, MPI-IO is used, but each task operates independently, writing one file per task. This was initially meant to be a test to verify correct operation of the mini-app, but ended up having better performance than expected and was left in as a viable option and included in the results section below. The second available approach is ADIOS. The ADIOS transport layer can be selected on the command line. One file minimum per time step is produced, though multiple files may be produced, depending upon the transport layer chosen. For example, the MPI transport layer produces one file only per time step, whereas the POSIX transport layer produces one file per MPI task per time step. File append is avoided to work around known append delays in Lustre. The third output type is parallel HDF5 with a straightforward, flat layout. It was implemented by The HDF Group. As with ADIOS, one file per time step is produced. Note, however, that the HDF5 implementation is not yet entirely tuned within the code.

III. RESULTS

Initial benchmarks have been performed (and are currently on-going) on a small variety of systems. These systems include Garnet (Cray XE6 with an older Lustre file system), Excalibur (Cray XC40 with Lustre), and Kilrain (IBM iDataPlex with GPFS). Kilrain is a smaller system, so only jobs of 512 and 4096 cores were run. Excalibur ran jobs of those sizes and 8000 and 21952 cores. Garnet ran all of those sizes and 46656 core jobs. The sizes of all jobs, including core count, grid size, and full output size are provided in Table 2.

The core counts (P) were chosen such that the tasks per dimension (p) are the cube root (P=p3), e.g., P=512 cores is p=8, so an 8x8x8 task decomposition. The grid scales up with the task size, at 256² points per task, so an 8x8x8 task layout would have 2048² grid points. The initial Gaussian σ is set to 0.96/p, so it scales down with core count to keep the triangles per task relatively constant. Similarly the final σ, for the gaussrsize operation, was set to 1.92/p. The center of the Gaussian was automatically computed from the domain size and the task count. The sinusoid frequency was set to p, allowing one period per task. A maximum of 104 time steps were executed, moving through all three cartiso modes, sin2gauss, gaussmove, and gaussrsize, with the gaussmove mode executing twice: one each for the smaller and larger Gaussian with the second running spatially in reverse. The sin2gauss mode used the first 11 time steps, and for larger runs with full output (i.e., not isosurface output) only those were used. All three output types, MPI-IO individual (VTK), HDF5 and ADIOS were used. For ADIOS, the following transport layers were attempted: POSIX, MPI, MPI-Lustre (except on GPFS), MPI-Aggregate, and PHDF5. All other cartiso settings were left at their defaults. The results are plotted in Fig. 5-7.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Input Grid Size (points)</th>
<th>Output Per Time (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>2048³</td>
<td>64</td>
</tr>
<tr>
<td>4096</td>
<td>4096³</td>
<td>512</td>
</tr>
<tr>
<td>8000</td>
<td>5120³</td>
<td>1000</td>
</tr>
<tr>
<td>21952</td>
<td>7168³</td>
<td>2744</td>
</tr>
<tr>
<td>46656</td>
<td>9216³</td>
<td>5832</td>
</tr>
</tbody>
</table>

Table 2. Grid size and Gaussian* sinusoid parameters effect upon number of triangles

Fig. 5. Full output throughput versus core count. (figure caption)
ADIOS MPI-Aggregate works reasonably well for nearly any case, though its actual isosurface scalability (Fig. 7) begins to suffer at higher core counts. We have not yet employed further optimizations discovered by Kitware, i.e., reducing the number of aggregators to significantly less than the default [9]. ADIOS’ PHDF5 transport layer was often too slow to complete a single time step within 1 hour. Manual Lustre tuning provided ~2x improvement, though it is still unusable. The pure HDF5 implementation is usable though mediocre. This is not surprising given the lack of tuning so far and Lustre stripe limitations on a single file. Minor hand tuning of Lustre parameters did produce a 2-10x improvement in most cases. This manual tuning included setting the Lustre stripe count to the maximum available for a single file (e.g., 165 stripes on Garnet), setting the stripe size to at least 8MB and setting the number of MPI-IO collective buffering nodes equal to the number of compute nodes used for the job. These settings work well for full output data and larger isosurface sizes; it would be even better to adjust the settings per time step based on isosurface size. An HDF5 autotuner module is under development and will be applied in the near future to assist in finding better settings for various scenarios. Finally, other ADIOS transport methods do not perform well for isosurface output. This includes the MPI-Lustre method, which likely indicates that it is not tuned for small, load imbalanced data.

IV. DISCUSSION AND CONCLUSION

The MPI-IO individual (VTK) method was consistently superior in nearly every case (except in some cases on GPFS). This was especially surprising for the full output. One possible explanation is that utilizing a vendor-tuned MPI (versus POSIX) allowed for optimization of metadata throughput despite the large file count. Another possible explanation is recent improvements to Lustre and its handling of metadata. The excellent performance of MPI-individual for isosurface output is also slightly surprising, but our thought is that the elimination of “zero triangle count” files significantly offsets the effects of load imbalance (most apparent in Fig. 7). These results are critical, since they demonstrate that derived in situ data output can scale with the simulation itself, if both the visualization algorithm is scalable and the I/O method used to write results is efficient.

Fig. 6 is very similar to the previous figure, but shows the “virtual” throughput of the isosurfacing and output. That is, we consider the in situ isosurface extraction and its output together to be a black box that directly replaces traditional I/O. As such, the total time to compute the isosurface and write the geometry data is treated as a measured output time and divided by the original data size to produce a rate in GB/s. The purpose is to directly compare to the full output performance. Since the isosurface extraction likely significantly reduces the amount of data to write, and that reduction happens on compute resources, it is expected that the throughput will be significantly greater than traditional I/O, and if done well, should scale with the computation itself. It is here where one sees if in situ truly benefits the application performance, and for the purposes of this paper, where one sees that the output method of the derived results has a significant impact upon its overall performance.

Finally, Fig. 7 shows the “actual” throughput – the geometry size in GB per time to write it. Some data points are missing because not all queued runs had completed, some output types were too slow to complete, and the POSIX method is currently failing at large core counts. This is possibly due to a memory leak in ADIOS that may be fixed in the ADIOS repository or due to an improper setting on our part.
REFERENCES


