Towards a General I/O Layer for Parallel Visualization Applications

Wesley Kendall and Jian Huang
Department of Electrical Engineering and Computer Science
The University of Tennessee
Email: {kendall, huangj}@eecs.utk.edu

Tom Peterka, Rob Latham, and Robert Ross
Mathematics and Computer Science Division
Argonne National Laboratory
Email: {tpeterka, robl, rross}@mcs.anl.gov

I. Introduction

Parallel visualization as a tool is regularly needed for handling scientists' growing data demands. For many mainstream visualization algorithms, the computation parts are inherently data-parallel and amenable for efficient scaling on even the largest of today's parallel architectures [1]. Accompanying architectural shifts, however, the primary limiting factor in scalability of large scale visualization applications has shifted from computation to I/O [1], [2].

Our viewpoint is that, with some urgency, the visualization community calls for general designs to efficiently perform parallel I/O in large scale visualization applications. In particular, more generalized I/O designs in parallel visualization should center around a partitioning strategy as opposed to a file format. We motivate our viewpoint by discussing current limitations of parallel I/O APIs with respect to the needs of the field, and show that the use of a simple design pattern can greatly alleviate I/O burdens without needing to "reinvent the wheels." We also show greatly accelerated performance with an implementation of this design in the context of a large-scale particle tracing application – an otherwise very challenging use case. We hope our work will instill further research efforts to address the I/O bandwidth challenge in the large data visualization domain.

II. The Burden of I/O on Parallel Visualization

I/O can place an expensive burden on parallel visualization practitioners. The example in Figure 1 is one illustration of this dilemma. In this scenario, a simulation generates data on an IBM BlueGene/P architecture. The simulation computes the physics of a time-varying phenomenon and saves a three-dimensional rectilinear volume at each time step. Many parallel I/O libraries are available to the simulation for storage, with some examples including PnetCDF (for the network Common Data Form), HDF5 (for the Hierarchical Data Format), and MPI-I/O for other custom formats.

While there are many visualization options, such as pathline tracing for vector fields or volume rendering for scalar fields, a parallel visualization approach must first partition the domain across processing elements (PEs). Block-based partitioning is one of the most popular choices for rectilinear grids. In our example, PEs are each assigned multiple blocks for more efficient workload balancing.

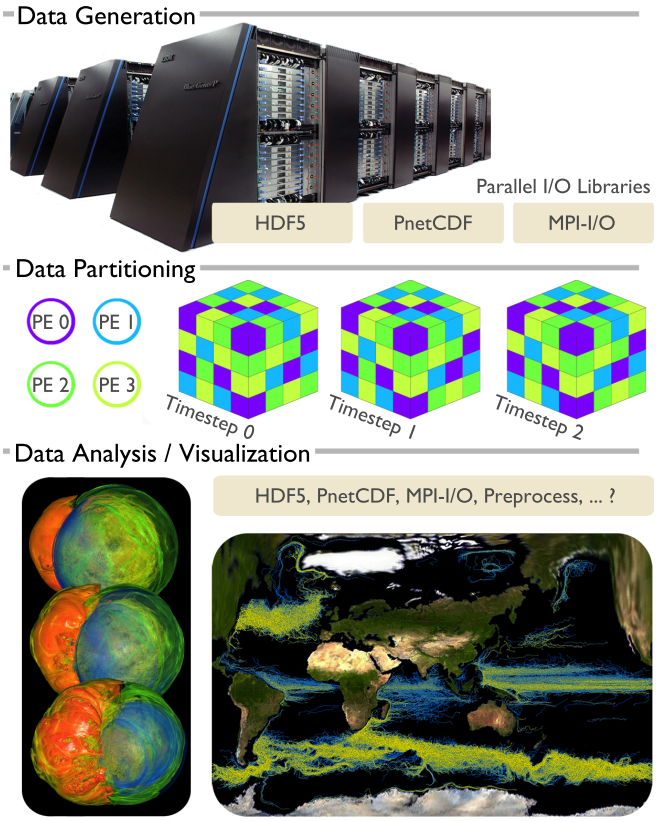

Figure 1. Example of a typical simulation and visualization scenario that illustrates some of the primary steps, including data generation, data partitioning, and the resulting visualization. In this example, blocks are distributed and colored by their assignment to four processing elements. The two visualizations show common time-varying techniques, which are pathline tracing and volume rendering.

The partitioning strategy, which is important for scaling computation, conflicts with physical data storage. If PEs were to issue separate I/O requests for blocks, the many disk seeks and reads will likely result in poor performance. Reading and distributing the dataset from one PE is often the first step taken to avoid this consequence, however, this does not effectively utilize standard parallel file system architectures. Transforming the dataset into a more I/O-efficient format is also another common step. While there has been success

in using multi-resolution or compressed out-of-core [3] formats, many of these techniques are optimized for serial file systems. Only very recently has parallel access been studied for multi-resolution formats [4]. Furthermore, metadata from higher-level formats which is needed for scientific analysis can easily be lost during this transformation.

The most practical approach uses the same parallel I/O library that the simulation used to write out the data. This approach, however, is still not ideal because of the many possible simulation formats and the difficulties in tuning and understanding low-level details about the parallel I/O APIs. For example, efficiently reading the block pattern in Figure 1 requires significant knowledge about MPI Datatypes, the newer non-blocking interface in PnetCDF, or the hyperslab functionality in HDF5. Furthermore, the semantics of these APIs restrict I/O operations to a single file at a time. As we will show in Section V, this can lead to a major underutilization of the available I/O bandwidth for multi-file datasets.

These complexities prompt many challenges for parallel visualization practitioners. Do all researchers and developers have to be parallel I/O experts to create applications that are scalable and portable across scientific formats? Production applications like Visit and ParaView have over one hundred different file readers in use. Will others that desire the same level of ubiquity in their parallel applications also have to pay this much attention to I/O? We believe that there is a need for more generalized parallel I/O solutions that scale across scientific data formats and storage conventions.

III. Properly Utilizing Parallel File Systems

It is necessary to understand standard parallel file system architectures in order to efficiently perform parallel I/O. Figure 2 shows a typical design. A parallel file system is usually a separate entity that is accessed through storage servers via high-speed networks. Some machines have dedicated I/O nodes that communicate with storage servers while others may use the compute nodes. Systems also often include one or more metadata servers that are responsible for handling information about the file, such as permissions and storage location.

When a file is stored on a parallel file system, it is striped across storage servers. Each of these storage servers obtain pieces of the entire file and may split them into finer grained portions across multiple underlying disks. Data is obtained in parallel among the disks and forwarded to the I/O nodes from the storage servers when requested. Large contiguous accesses aid in amortizing disk latency, allow more efficient prefetching of data, and also help obtain more total concurrency during retrieval.

Since large contiguous accesses provide the highest performance from parallel file systems, distributed and noncontiguous patterns such as those shown in Figure 1 must be transformed prior to file system access. The standard method

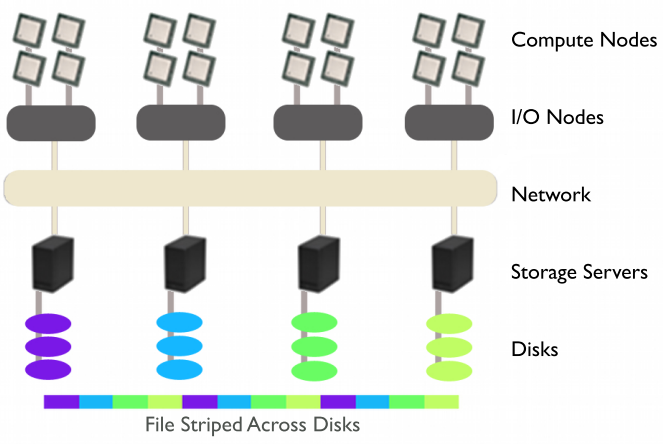

Figure 2. A typical parallel file system design. The bottom of the illustration represents how a file may be distributed across the disks of a parallel file system.

of enabling these transformations is via *collective I/O*. This technique aggregates distributed requests into larger more contiguous requests. It can be implemented on the disk, server, or client level. When performed on the client level, clients will communicate and aggregate their requests, perform I/O on more contiguous regions, and then exchange the data back to the requesting clients. This technique is known as *two-phase collective I/O* since it involves an additional phase of data exchange.

IV. A More Generalized Approach

We believe that a more generalized I/O design should center around a partitioning strategy instead of a file format. Rather than having to deal with many formats and API complexities, applications should have access to a simple I/O layer optimized for their partitioning strategy that abstracts file formats and even other intricacies like multi-file dataset storage.

A block-based I/O layer is a motivating example for our viewpoint. Block-based partitioning, such as the example shown in Figure 1, is not only popular in many parallel visualization strategies, but also prevalent in other applications like parallel matrix analysis. To illustrate how such a layer would operate, we have designed and implemented a prototype software, known as the Block I/O Layer (BIL). In the BIL interface, PEs specify a collection of blocks that they individually intend to access, then they collectively operate on the global collection. The interface is designed to simply have two functions:

- *BIL_Add_block_{file format}* – Takes the starts and sizes of a block along with the variable and file name. PEs call it for as many blocks as they need, whether they span multiple files or variables. Currently it operates on raw, netCDF, and HDF formats.

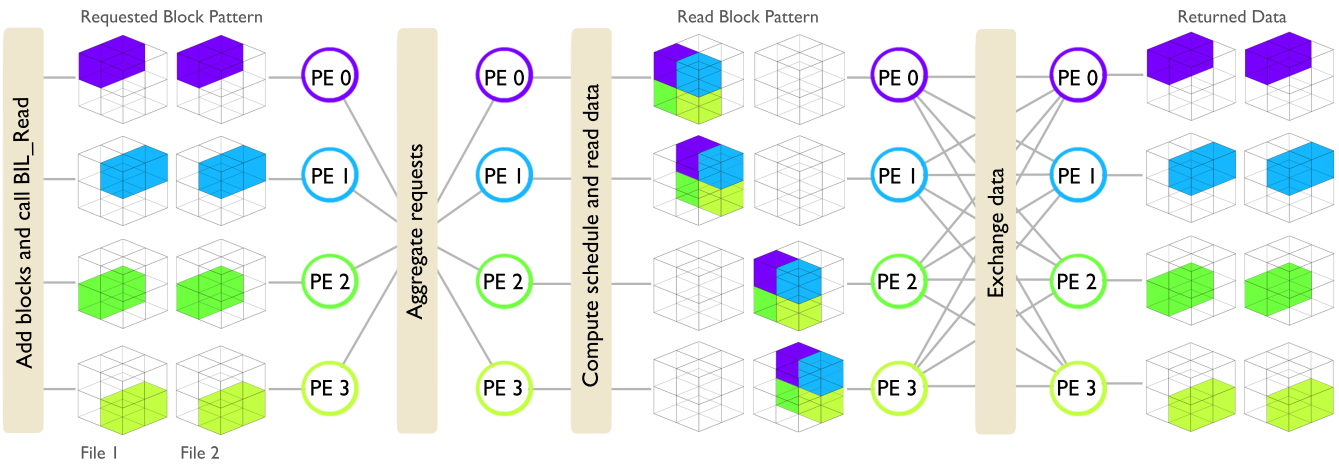

Figure 3. An example of how our I/O implementation performs reading of requested blocks. This illustration uses four PEs that each request two blocks that are in separate files. The procedure uses a two-phase I/O technique to aggregate requests, schedule and perform large contiguous reads, and then exchange the data back to the requesting PEs.

- *BIL_{Read, Write}* – Takes no arguments. The blocks that were added are either read in or written from the user-supplied buffers.

The implementation is illustrated in Figure 3, which shows a simple example of four PEs reading a block-based pattern spanning two files. The PEs first add the desired blocks and then call *BIL_Read*. The requested blocks, which start out as noncontiguous storage accesses for each PE, are aggregated and scheduled into large contiguous accesses. Reading then occurs in parallel and data are exchanged back to the original requesting PEs.

Although the semantics of the underlying parallel I/O APIs would normally restrict users to operate on single files at a time, this design allows the implementation to collectively perform I/O across multiple files. Furthermore, the implementation can use advanced features of I/O libraries when necessary and can be configured for different file systems. For example, we are able to detect when the individual reads of each PE are less than the file system's striping size. When this occurs, we have found that it is generally best to use collective I/O strategies or simply perform I/O from a smaller subset of PEs.

BIL's communication is also built upon advanced MPI mechanisms. For exchanging of data, we use collective communication routines to take advantage of the underlying MPI implementation, which is able to efficiently utilize certain network topologies and architectures. Exchanging data usually takes less than 10% of the overall time, as communication bandwidths are typically orders of magnitude larger than storage bandwidths.

V. A Driving Application - Parallel Pathline Tracing

Particle tracing is one of the most pervasive methods for flow visualization, and also one of the hardest to parallelize in a scalable manner. Seeds are placed within a vector field and advected over a period of time. The traces that the particles follow, streamlines in the case of steady-state flow and pathlines in the case of time-varying flow, can be used to gain insight into flow features. For example, Figure 1 shows a visualization of major ocean currents with pathlines.

We have integrated BIL into OSUFlow, a particle tracing library originally developed by the Ohio State University in 2005 and recently parallelized. The application partitions the domain into four-dimensional blocks (time blocks) and assigns them round-robin to each of the PEs (similar to the illustration in Figure 1). For an extensive explanation, we refer the reader to [5].

OSUFlow has the ability to load time blocks that span multiple files, primarily because scientists often store one file per time step. Its original implementation used parallel I/O libraries to collectively read one file at a time until blocks were completely read. Although this implementation used the I/O libraries in their intended manners, it would still often lead to mediocre performance results.

We compared the original I/O methods with BIL on *Intrepid*, an IBM BlueGene/P system at Argonne National Laboratory that consists of 40,960 quad-core 850 MHz PowerPC processors and a GPFS parallel file system. The comparison used two test datasets. The first is generated from the Parallel Ocean Program (POP), an eddy-resolving global ocean simulation [6]. Our version of the dataset consists of u and v floating point variables on a $3{,}600 \times 2{,}400 \times 40$ grid spanning 32 time steps that are saved in separate netCDF files (82 GB total). The second dataset is a Navier-Stokes jet propulsion simulation that has u, v, and w floating point tuples on a $256 \times 256 \times 256$ grid across 2,000 time steps in separate raw binary files (375 GB total).

Bandwidth results appear in Figure 4. The top line rep-

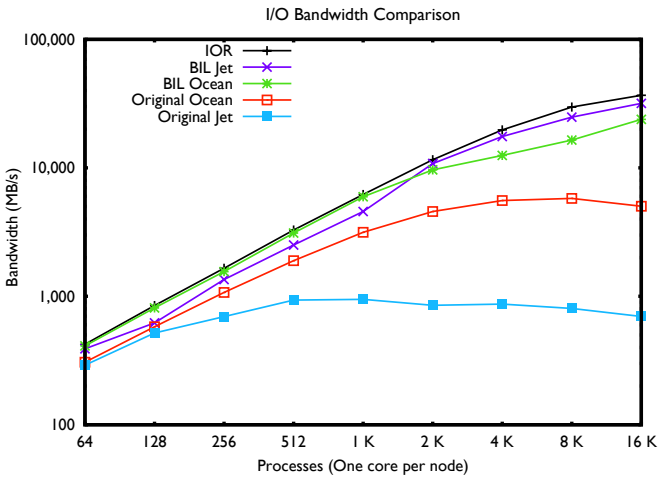

Figure 4. Bandwidth results (log-log scale) of our parallel I/O method versus the original parallel I/O method in OSUFlow. All tests were conducted using one core per node (to maximize the amount of I/O nodes used) on Intrepid with two different datasets. The top line represents the IOR benchmark. The original method was using the newer non-blocking Parallel netCDF routines for the ocean dataset and collective MPI-I/O for the jet dataset. The original procedure, however, was restricted to collectively reading one file at a time, leaving much of the available bandwidth unused for these multi-file datasets.

resents IOR [1], a popular bandwidth benchmark for parallel I/O systems, while the others represent the total bandwidths achieved by the original method vs. BIL. The differences are significant at large scale. At 16 K PEs, we observed a factor of 5 improvement for the ocean dataset and a factor of 45 improvement for the jet dataset. Both BIL results were able to maintain bandwidth rates that were very close to the peak IOR rates. For the jet dataset, BIL obtained roughly 30 GB/s at 16 K PEs and reduced I/O time from 9 minutes to 12 seconds. At such large PE counts, the amount of data accessed by any given PE when accessing one file at a time is too small to attain any substantial bandwidth; the capability in BIL to concurrently schedule reads to multiple files makes a difference.

For scaling an application like OSUFlow that has irregular access patterns, using parallel I/O is required, not an option. This is true even when the use of parallel I/O may not be optimal. POSIX I/O is impractical, because reading the jet dataset on 64 PEs through POSIX I/O led to a ≈30 MB/s bandwidth. In fact, this number could only be estimated since the one-hour time limit on our tests expired before the data could be read.

VI. Closing Remarks

When used properly, parallel file systems can greatly enhance the interactivity that is crucial to visualization applications, especially those that perform post-analysis after simulations. We have shown one way to integrate advanced parallel I/O methods under a simple and robust design that applies to a broad spectrum of scientific data formats. More solutions of this kind are needed as more and more data analysis applications are to be scaled to HPC architectures. This need is urgent and will require community effort to tackle the broad spectrum of parallel visualization applications and I/O needs.

Also addressing the "bandwidth challenge" are many advanced cases that pose even bigger I/O challenges. Out-of-core, data compression, and multi-resolution are just a few common examples. Due to space limit, herein we can only mention that other researchers have already started working in those tough areas with initial success [3], [4]. Future study in those areas will be crucial for progressing towards more accepted and standardized practices of parallel I/O in the visualization community. For dissemination and also for verification by the community, we have released BIL under LGPL at http://seelab.eecs.utk.edu/bil.

[1] http://www.cs.sandia.gov/Scalable_IO/ior.html

Acknowledgment

We would like to acknowledge Han-Wei Shen, as his collaboration has been pivotal to this work taking place. We would also like to thank Kwan-Liu Ma for providing the jet dataset and the Argonne Leadership Computing Facility for computing resources and support. This work is funded primarily through the Institute of Ultra-Scale Visualization (http://www.ultravis.org) under the auspices of the SciDAC program within the U.S. Department of Energy.

References

[1] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel, "Extreme scaling of production visualization software on diverse architectures," *IEEE Computer Graphics and Applications*, vol. 30, no. 3, pp. 22–31, 2010.

[2] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham, "End-to-end study of parallel volume rendering on the IBM Blue Gene/P," in *Proc. of the Intl. Conference on Parallel Processing*, 2009.

[3] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Trans. on Visualization and Computer Graphics*, vol. 12, pp. 1245–1250, Sept. 2006.

[4] S. Kumar, V. Pascucci, V. Vishwanath, P. Carns, M. Hereld, R. Latham, T. Peterka, M. Papka, and R. Ross, "Towards parallel access of multi-dimensional, multi-resolution scientific data," in *Petascale Data Storage Workshop*, 2010, pp. 1–5.

[5] T. Peterka, R. Ross, B. Nouanesengsey, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang, "A study of parallel particle tracing for steady-state and time-varying flow fields," in *Proc. of the IEEE Intl. Symp. on Parallel and Distributed Processing*, 2011.

[6] M. E. Maltrud and J. L. McClean, "An eddy resolving global 1/10 ocean simulation," *Ocean Modelling*, vol. 8, no. 1-2, pp. 31–54, 2005.