

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 is one of the most powerful tools for gaining insight into large datasets. Many mainstream algorithms are inherently data-parallel and can be scaled to large process counts, providing more interactive methods to handle scientists' growing data demands. The study of how visualization approaches interact with parallel storage devices, however, has largely been neglected. While recent reports have shown the dominant role of I/O when scaling procedures like volume rendering [1], [2], [3], there is still no clear consensus on the best techniques to use or an effort to create more generalized I/O solutions for scaling other visualization approaches.

Our viewpoint is that parallel I/O should be better integrated into the community with more generalized designs and accepted practices. To justify our viewpoint, we discuss common I/O challenges and describe how limitations of current technologies can make it difficult to achieve best performance. We illustrate how the use of a simple design pattern can alleviate many difficult I/O scenarios and leverage current parallel I/O libraries in more beneficial manners for parallel visualization applications. As we will show, solutions like this can mean the difference between minutes and seconds of I/O time, and we believe they will be necessary as more applications are scaled on HPC architectures. While our design is only for a subset of all problems, we want to bring this research issue to the attention of the field and instill an effort for more comprehensive solutions.

II. THE BURDEN OF I/O ON VISUALIZATION

Visualization as a field is burdened by I/O. The interdisciplinary nature of visualization forces developers to deal with a large number of file formats. In fact, production applications like Visit¹ and ParaView² have over one hundred different file readers in use. The amount of domain-specific tools that depend on the rich meta-information of these formats gives them good cause to persist in their respective fields with less chance of a universal format replacing them.

¹<http://wci.llnl.gov/codes/visit>

²<http://www.paraview.org>

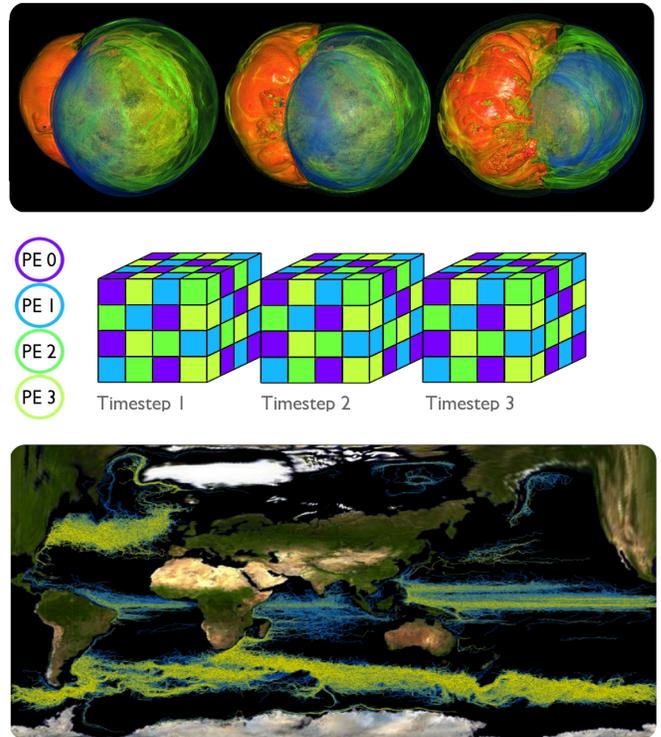


Figure 1. Examples of parallel visualization approaches that often leverage block-based domain partitioning strategies. The top shows time-varying volume rendering of a supernova core collapse. The bottom shows pathlines traced through a global ocean simulation to reveal major ocean currents. An illustration of a block-based partitioning strategy using four processing elements on a uniform time-varying grid is shown in the middle.

The amount of formats, however, is only part of the real issue that occurs when developing parallel applications. The physical layout of the data on disk often does not match the data partitioning strategy of the application, which can cause serious I/O bandwidth deficiencies if advanced measures are not taken. This becomes even more apparent when trying to partition data that spans many files, whether these are different timesteps, variables, or parameter sweeps.

Let us consider volume rendering and particle tracing on uniform grids as two major examples of how these issues come in effect. As depicted in Figure 1, these in-

dispensable algorithms give scientists the ability to examine complex time-varying phenomena like the core collapse of a supernova or the behavior of global ocean currents. Many of the standard approaches that parallelize these algorithms partition the domain into blocks and assign them to processing elements (PEs). More advanced approaches, as illustrated in Figure 1, assign multiple blocks to PEs to more effectively balance the workload. The individual blocks, however, translate into noncontiguous regions of the underlying stored data. If each PE uses traditional POSIX I/O routines, the numerous disk seeks and reads will likely result in dismal I/O bandwidth. Using only one PE to read and distribute the dataset can also lead to poor bandwidth results since it does not sufficiently utilize the network links in a parallel file system.

Parallel I/O libraries can alleviate many serial I/O access issues by providing the capability to take distributed requests and aggregate them into requests that more closely match how the data is stored on disk. The standardization of I/O in the Message Passing Interface 2 (MPI2)³ has allowed popular formats with strong community support like the network Common Data Form (netCDF)⁴ and the Hierarchical Data Format 5 (HDF5)⁵ to have parallel interfaces [4]. Understanding low-level details of the interfaces, however, is often necessary in order to correctly use them. For example, accessing the pattern in Figure 1 from only one timestep of raw data involves formulating the request into an MPI indexed datatype and using collective I/O. For classic netCDF datasets, the newer non-blocking I/O routines in Parallel netCDF [5] must be used for the same pattern. Performing this across all files at once, a common need in time-varying visualization like our pathline tracing example, is not natively supported by these libraries. These limitations create challenges in efficiently harnessing the throughput available from parallel file systems. There is a need for more higher-level interfaces that are portable across file formats and can mask complexity of parallel I/O.

III. PROPERLY UTILIZING PARALLEL FILE SYSTEMS

In order to perform parallel I/O properly, it is necessary to understand common designs and architectures of parallel file systems. Figure 2 shows a typical design. A parallel file system is often 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 actual compute nodes. One or more metadata servers that are responsible for handling information about the file, such as permissions and storage location, are often included in the design.

When a file is stored on a parallel file system, it is striped across storage servers. Each of these storage servers obtain

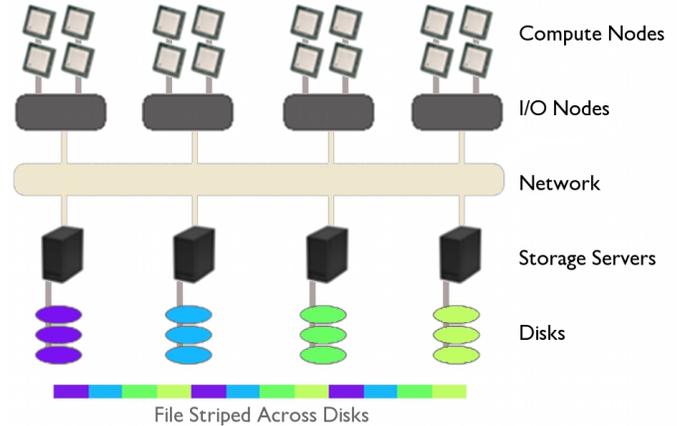


Figure 2. A typical parallel file system design. The bottom of the illustration represents how a file may be distributed across the disks of a parallel file system.

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

Taking advantage of large contiguous access optimizations can be difficult given distributed noncontiguous patterns such as shown in Figure 1. The classic method to solve this is with a technique known as 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, PEs will all communicate and aggregate their requests, perform I/O on more contiguous regions, and then exchange the data back to the requesting PEs. This technique is known as two-phase collective I/O since it involves an additional phase of data exchange.

IV. A MORE GENERALIZED APPROACH

One motivating approach that can generalize I/O is the usage of a block-based design pattern. Consider the notion of a block, which can span any dimensionality, contain any amount of variables, and span multiple files as well. This representation of data can be used for the access patterns of the visualization procedures we have discussed along with other problems that typically use block-cyclic distribution, such as matrix analysis (e.g. Principle Component Analysis).

By restricting the description of access patterns, we can mask parallel I/O complexity with a greatly simplified interface. To illustrate this ability, we have designed and implemented a prototype, known as the Block I/O Layer (BIL), with the following interface:

³<http://www.mpi-forum.org>

⁴<http://www.unidata.ucar.edu/software/netcdf/>

⁵<http://www.hdfgroup.org/HDF5>

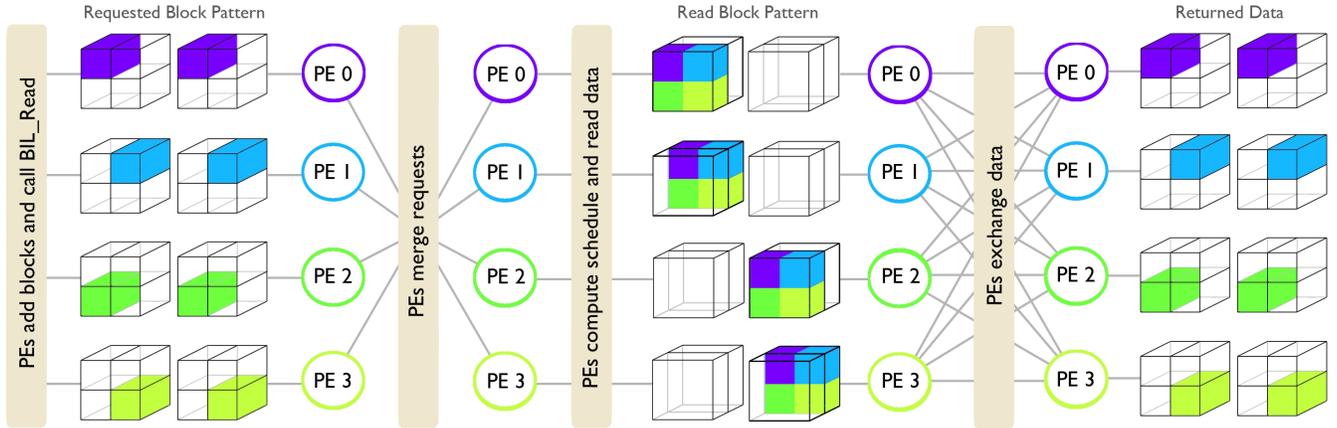


Figure 3. An example of how our I/O implementation performs reading of requested blocks. This illustration uses four PEs that each request two blocks that are in separate files. The procedure uses a two-phase I/O technique to aggregate requests, schedule and perform large contiguous reads, and then exchange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

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

The interface is similar to HDF’s hyperslab functionality and Parallel netCDF’s non-blocking procedures. The main difference is that it also provides an additional flexibility to specify multi-file access patterns, which is leveraged in our underlying implementation. To illustrate this, Figure 3 shows a simple example of four PEs reading a block-based pattern that spans two files. The PEs first add the necessary blocks that are needed and then call *BIL_Read*. The requested blocks, which start out as noncontiguous 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.

When aggregating block requests, we use a tree-based reduction algorithm. At each stage of the reduction, the PEs aggregate the requests by unioning the blocks that have the same file and variable names. For non-power-of-two PE counts, the PEs aggregate and send their requests to the highest power-of-two PE count. The PEs then operate as leaves on a binary tree and transmit their requests to neighboring leaves until only the root PE remains. The root PE broadcasts the final global I/O request to all PEs, which can individually compute the portions they should read.

When reading, PEs have access to identification numbers (IDs) for each file and variable. The IDs can be used to group PEs and intelligently use advanced features of underlying libraries. For example, we can detect the variable layout used in netCDF files (record or non-record layout) and use

the non-blocking I/O interface accordingly. We can also use collective I/O when the individual I/O requests are smaller than the file system’s stripe size. Furthermore, groups can also perform I/O in stages to avoid potential bottlenecks that might arise when using too many PEs for I/O.

The last step of exchanging data is formulated into one *MPI_Alltoallv* call. This allows us to take advantage of the underlying MPI implementation for network communication, which is able to efficiently utilize certain network topologies and architectures. Since communication bandwidths are often orders of magnitude larger than I/O bandwidths, this step is usually a small portion (< 10%) of the overall time.

V. A DRIVING APPLICATION - PARALLEL PATHLINE TRACING

Particle tracing is one of the most ubiquitous methods for visualizing flow fields. Seeds are placed within a vector field and are 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 the usage of pathlines to visualize major ocean currents.

We have integrated BIL into OSUFlow, a particle tracing library originally developed by the Ohio State University in 2005 and recently parallelized. In summary, the application partitions the domain into four-dimensional blocks and assigns them round-robin to each of the PEs (similar to the illustration in Figure 1). The time dimension of each block is equal to the number of timesteps, allowing the application to only load slices of the time domain during advection if memory limitations exist. For an extensive explanation, we refer the reader to [6].

OSUFlow has the ability to load blocks that span multiple files, primarily because collaborators would often store

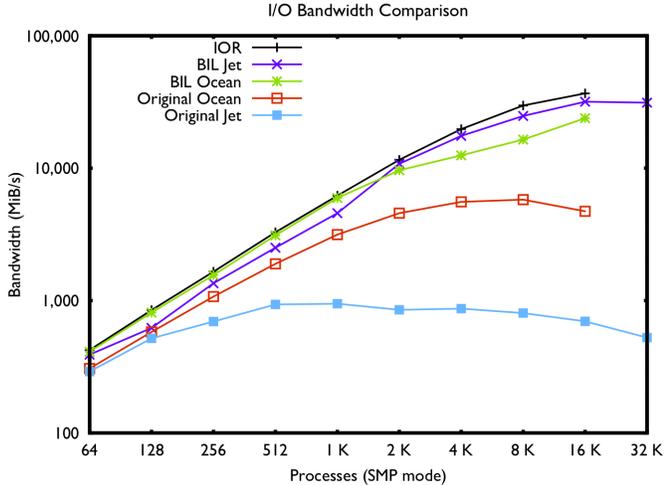


Figure 4. Bandwidth results (log-log scale) of our parallel I/O method versus the original parallel I/O method in OSUFlow. Tests were conducted in SMP mode (one core per node) on Intrepid with two different datasets. The top line represents the IOR benchmark. The original method was using the newer non-blocking 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.

separate files for each timestep. The original implementation used parallel I/O libraries to open and collectively read one file at a time until blocks were completely read. This method, which is a proper usage of the parallel I/O libraries, can still leave a significant amount of bandwidth unused.

To show the effects of this, we have compared BIL with the original implementation on *Intrepid*, an IBM BlueGene/P system at Argonne National Laboratory that consists of 40,960 quad-core 850MHz PowerPC processors and a GPFS parallel file system. We used two test datasets in the comparison. The first is generated from the Parallel Ocean Program (POP), an eddy-resolving global ocean simulation [7]. 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 timesteps that are saved in separate netCDF files (82 GiB). The second dataset is a Navier-Stokes jet propulsion simulation that has u , v , and w floating point variables saved in tuples on a $256 \times 256 \times 256$ grid across 2,000 timesteps in separate raw binary files (375 GiB).

Bandwidth results are shown in Figure 4. The top line represents IOR⁶, a popular benchmark for parallel I/O systems, while the others represent the total bandwidths of the original method and BIL on the test datasets. The differences are significant at large scale. For the ocean dataset, we observed a factor of 5 improvement at 16 K PEs. For the jet dataset, BIL obtained over 30 GiB/s at 32 K PEs and a factor of 60 improvement – a difference between 12 minutes and 12 seconds of I/O time. The simplest reason for these

⁶http://www.cs.sandia.gov/Scalable_IO/ior.html

differences is that the individual files do not warrant the use of such high parallelism at once, especially in the case of the jet dataset. The advantage of scheduling and reading more files concurrently, however, can more efficiently leverage the available bandwidth of the file system.

Although we have primarily discussed better utilizations of parallel I/O libraries, it is important to note that the initial usage of a parallel I/O library was a necessity to begin scaling an application like OSUFlow with irregular access patterns. For example, at 64 PEs on the jet dataset, the usage of POSIX I/O resulted in ≈ 30 MiB/s bandwidth. This number is only estimated, however, since we could only read a fraction of the dataset before one hour time limits on the test would expire.

VI. CLOSING REMARKS

If used properly, parallel file systems can greatly enhance the interactivity that is so important to visualization applications. This is especially true for applications that perform post-analysis directly after simulations, in our case, parallel pathline tracing. As we have shown, there are methods that can utilize current parallel I/O libraries in more beneficial manners for visualization applications. Although these methods are quite advanced, there are promising ways to simplify them with design patterns that can be portably used across many popular scientific data formats. There is an urgent need for more solutions like these, as they will be critical for others that develop and research parallel applications and algorithms. These types of solutions will also provide the field with solid and accepted practices of performing parallel I/O – a necessity as more and more applications are scaled on HPC architectures. As a step forward in this direction, we have released BIL under the LGPL, available at <http://seelab.eecs.utk.edu/bil>.

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 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. Yu and K.-L. Ma, “A study of I/O techniques for parallel visualization,” *Parallel Computing*, vol. 31, no. 2, pp. 167–183, 2005.
- [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 *ICPP '09: Proceedings of the International Conference on Parallel Processing*, 2009.

- [3] 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, pp. 22–31, 2010.
- [4] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *SC '03: Proceedings of IEEE/ACM Supercomputing*, Nov. 2003.
- [5] K. Gao, W.-K. Liao, A. Choudhary, R. Ross, and R. Latham, "Combining I/O operations for multiple array variables in parallel netCDF," in *IASDS '09: Proceedings of the IEEE Cluster Workshop on Interfaces and Architectures for Scientific Data Storage*, 2009.
- [6] 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 *IPDPS '11: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, 2011.
- [7] 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.