Towards a General I/O Layer for Parallel Visualization

Wesley Kendall, Jian Huang
The University of Tennessee, Knoxville
Tom Peterka, Robert Latham, Rob Ross
Argonne National Laboratory

(Large Scale) Parallel Visualization

Unless in-memory objects are used, parallel visualization has to get data from disk files, and ideally through a parallel file system and parallel I/O.

The solutions, or at least the implementations, are typically application-specific

Learning curve of parallel I/O is non-trivial

The gap between parallel I/O research and parallel visualization research needs to be addressed

Consider a few typical use cases for parallel visualization:

- Data partition on block granularity (2D, 3D, 4D, etc.)
- Many blocks per core (for various reasons: load balance …)
- Ghost regions added around each block (for various reasons: interpolation, …)
- Vis deal with concepts on dataset level, as opposed to file level

We developed BIL -- for block level parallel I/O in parallel visualization

- Sample performance (1): 410 GB, 5840 files on 16Kcores achieved 50% IOR in a real parallel particle tracing in an ensemble GEOS5 analysis
- Sample performance (2) : 375 GB, 2000 files on 16K cores, reduced I/O from 9 minutes down to 12 seconds for visualizing a turbulence simulation

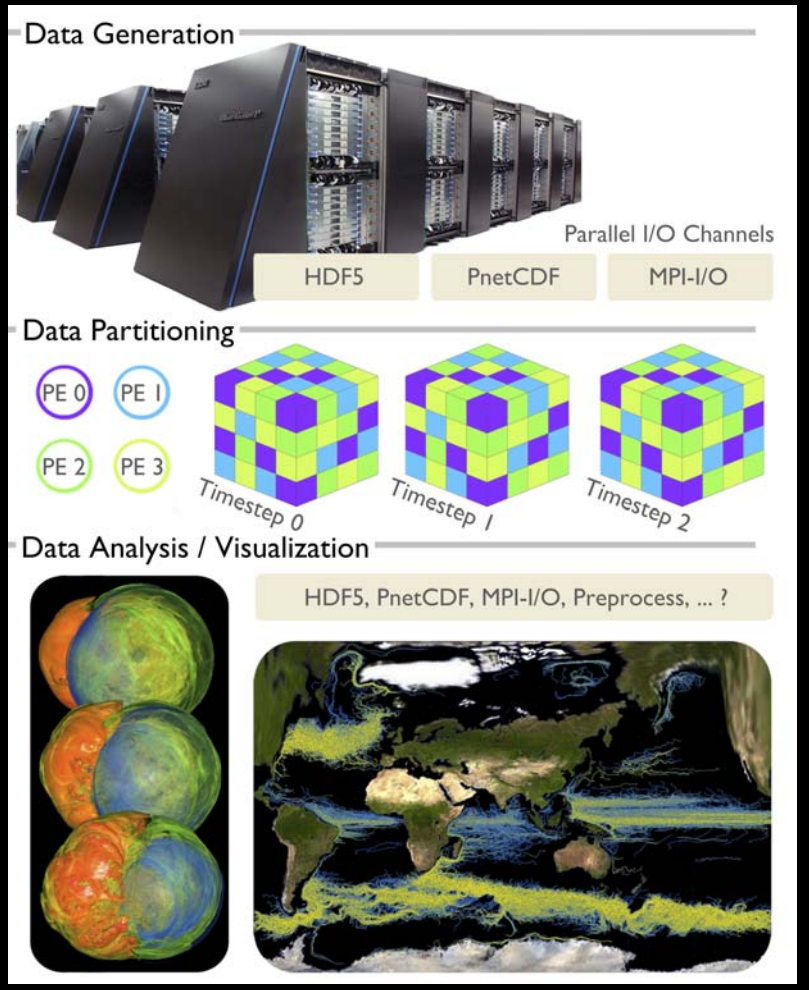
(Large Scale) Parallel Visualization

Vis and analysis applications typically have large input that is reduced to smaller output

- Volume Rendering – Renders large volumetric data and saves much smaller 2D image
- Flow Tracing – Integrates a large amount of time-varying volumetric data to arrive at tangential geometry

Analysis applications are also forced to deal with a variety of scientific outputs

- Output may span raw or high level formats such as netCDF and HDF
- Output also may span across many files, such as per timesteps



A typical simulation/visualization scenario. In this example, the visualization application creates a 4D partition and assigns multiple blocks per process for load balancing.

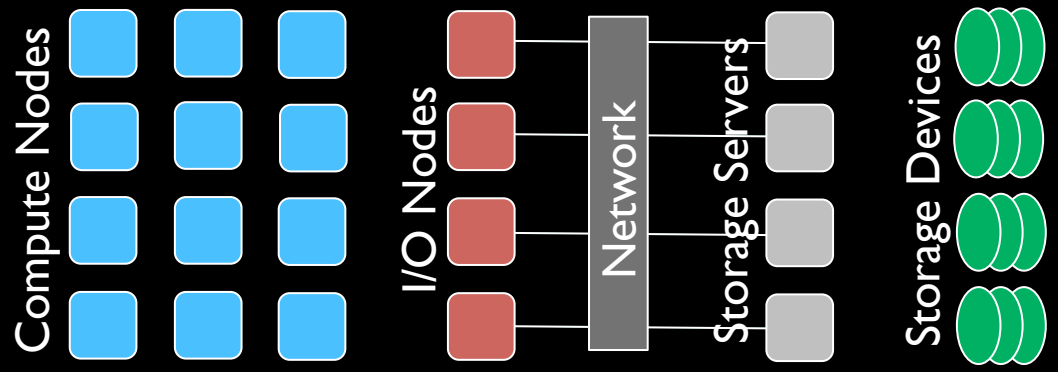
Parallel I/O

The goal of parallel I/O

- Scalable utilization of multiple storage devices
- Elegant interlacing with parallel applications

Parallel file systems

- Abstract the storage of data across multiple disks
- In supercomputing architectures, parallel file systems are often separate physical entities connected by a network



A common parallel file system / supercomputing architecture

"Simply adding disk drives to an I/O system without regards to software design will only minimally improve performance" - *John May, Parallel I/O for High Performance Computing*

Parallel I/O Pitfalls

Noncontiguous disk access

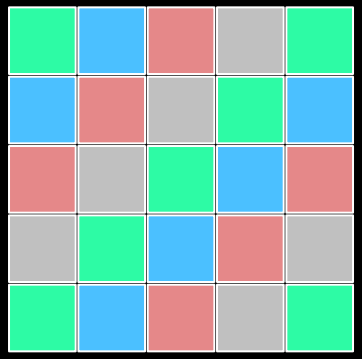
- Results in many disk seeks, which have a high cost
- Does not utilize prefetching often taken by the parallel file system

Underutilization of network links

- Shipping all I/O through one process does not utilize the network bandwidth to the parallel file system

Common noncontiguous access patterns

- Block-cyclic distribution



Disk access (first two matrix rows)

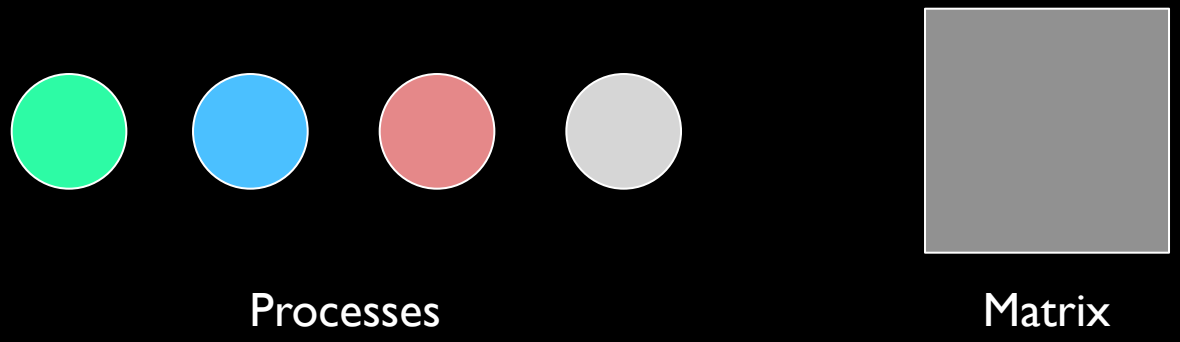
Block-cyclic distribution of a 2D matrix to four processes (colored by process). The right side of the figure shows the resulting disk access for the first two rows of the matrix.

Advanced Parallel I/O Techniques

"Collective I/O" is a common technique for avoiding noncontiguous storage access

- Can occur in a client- or server-side manner
- In server-side collective I/O, the storage servers aggregate incoming requests and perform larger, more contiguous accesses.
- In client-side collective I/O, processes communicate and aggregate requests, issue more contiguous requests to the file system, and the shuffle data back to requesting processes (Also known as two-phase I/O).
- Widely-studied problem in the I/O community [Rosario'93, Seamons'95, Thakur'96]

Collective I/O example – Four processes and a 2D matrix



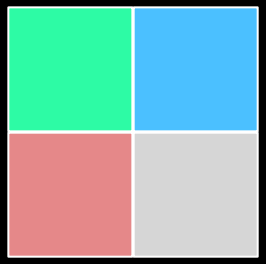
Advanced Parallel I/O Techniques

"Collective I/O" is a common technique for avoiding noncontiguous storage access

- Can occur in a client- or server-side manner
- In server-side collective I/O, the storage servers aggregate incoming requests and perform larger, more contiguous accesses.
- In client-side collective I/O, processes communicate and aggregate requests, issue more contiguous requests to the file system, and the shuffle data back to requesting processes (Also known as two-phase I/O).
- Widely-studied problem in the I/O community [Rosario'93, Seamons'95, Thakur'96]

Processes partition matrix into blocks

Processes



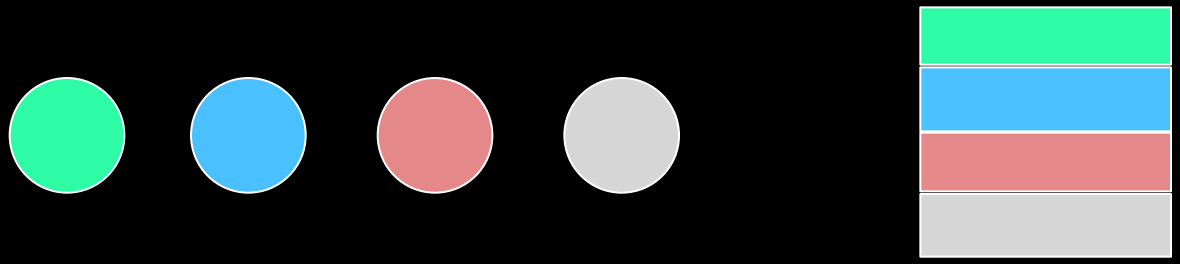
Matrix

Advanced Parallel I/O Techniques

"Collective I/O" is a common technique for avoiding noncontiguous storage access

- Can occur in a client- or server-side manner
- In server-side collective I/O, the storage servers aggregate incoming requests and perform larger, more contiguous accesses.
- In client-side collective I/O, processes communicate and aggregate requests, issue more contiguous requests to the file system, and the shuffle data back to requesting processes (Also known as two-phase I/O).
- Widely-studied problem in the I/O community [Rosario'93, Seamons'95, Thakur'96]

Processes communicate and translate request into contiguous regions



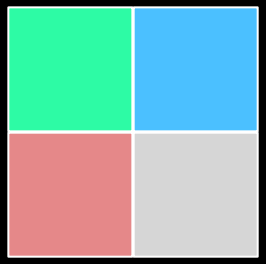
Advanced Parallel I/O Techniques

"Collective I/O" is a common technique for avoiding noncontiguous storage access

- Can occur in a client- or server-side manner
- In server-side collective I/O, the storage servers aggregate incoming requests and perform larger, more contiguous accesses.
- In client-side collective I/O, processes communicate and aggregate requests, issue more contiguous requests to the file system, and the shuffle data back to requesting processes (Also known as two-phase I/O).
- Widely-studied problem in the I/O community [Rosario'93, Seamons'95, Thakur'96]

After reading contiguous regions, processes shuffle data back to original request

Processes



Matrix

Current Parallel I/O APIs And Their Restrictions

Most popular parallel APIs in the scientific domain

- MPI-I/O
  - Used primarily for custom data output formats
- Parallel netCDF
  - Used mostly in Earth Sciences for structured gridded datasets
- HDF5
  - Currently one of the most sophisticated output formats
  - netCDF recently started using HDF as the underlying format

Semantics of I/O APIs are restricted to performing I/O requests at the file level

- For example, MPI-I/O and Parallel netCDF require passing a file handle to function calls
- This can potentially leave much bandwidth unused for multiple file datasets

Single variable requests

- Only recently, high-level parallel I/O APIs started to aggregate multiple variable requests [Gao'09]

Oftentimes significant low-level knowledge is needed

- For reading multiple blocks per process, the nonblocking interface in Parallel netCDF must be used. Also, advanced MPI Datatypes must be created for the request.

Our Solution – The Block I/O Layer (BIL)

Instead of focusing on a scientific storage format, build an API around a partitioning pattern of an application

- BIL is built on the most common type of partitioning scheme in analysis – block-based partitions.

API

- BIL_Add_block_{raw, nc, hdf} - Takes the block bounds, variable type or name, the file name, and a buffer for the data.
- BIL_{Read, Write} - Takes no arguments. All added blocks will be collectively read/written.

The design operates more on a dataset level, allowing processes to access data from as many files, variables, and file formats as needed

Usage Scenarios For BIL

BIL interlaces well with partitioning modules that output a list of blocks per process

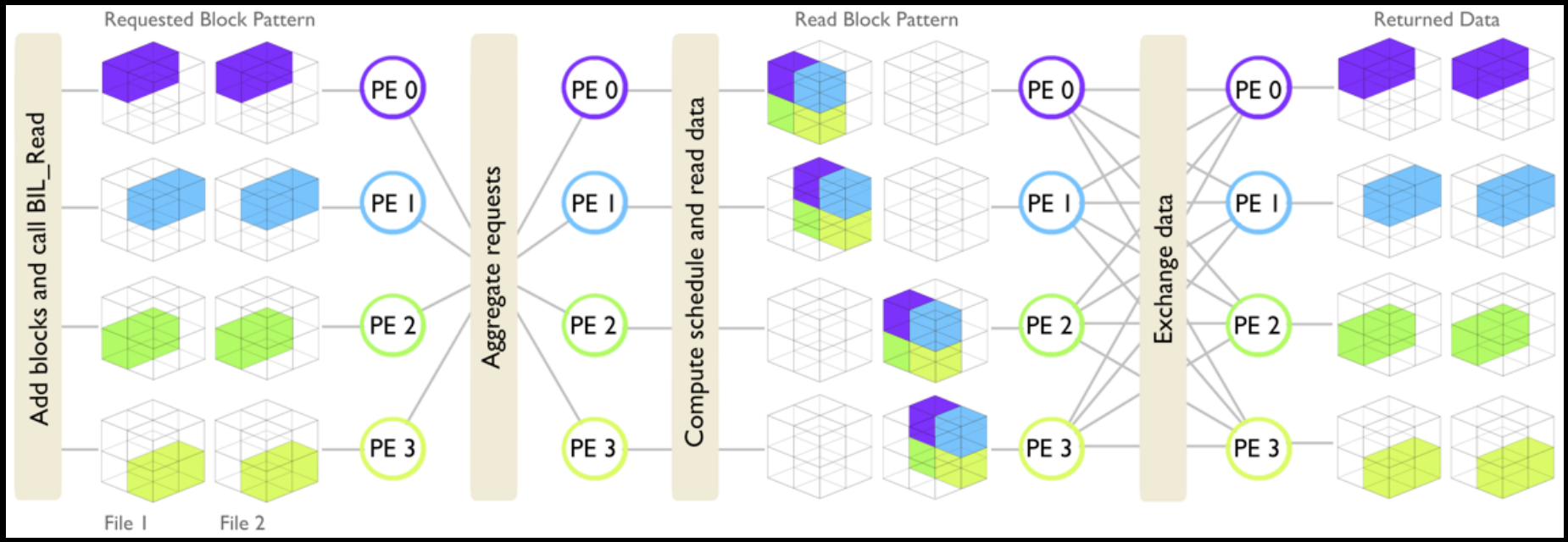
Example code for performing raw or netCDF parallel I/O

```
// Obtain a list of blocks from a global partition
Block blocks[] = create_partition();
foreach block in blocks {
  if (FileFormat == RAW) {
    BIL_Add_block_raw(block.num_dims, block.start,
                      block.size, block.file_name,
                      block.var_type, block.buffer);
  } else if (FileFormat == NETCDF) {
    foreach variable in block.variables {
      BIL_Add_block_nc(block.num_dims, block.start,
                       block.size, block.file_name,
                       variable, variable.buffer);
    }
  }
}
// Collectively read all of the blocks
BIL_Read();
```

Implementation

BIL performs a two-phase I/O over multiple files and variables

- Processes merge blocks and compute their contiguous portion of I/O
- BIL transparently uses advanced I/O techniques, for example, turning off collective I/O when the individual accesses are large and vice versa.
- Processes then shuffle data back to its original requested pattern.



Example of BIL with four processing elements (PEs) and two blocks per PE which span two 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.

Preliminary Results

BIL can offer significant performance increases over standard single file access

- We used BIL in OSUFlow, a parallel particle tracing library [Peterka'11].
- OSUFlow reads time-varying datasets stored across multiple files. The original parallel I/O implementation collectively read one file at a time, leaving much of the available bandwidth unused.

Ocean dataset

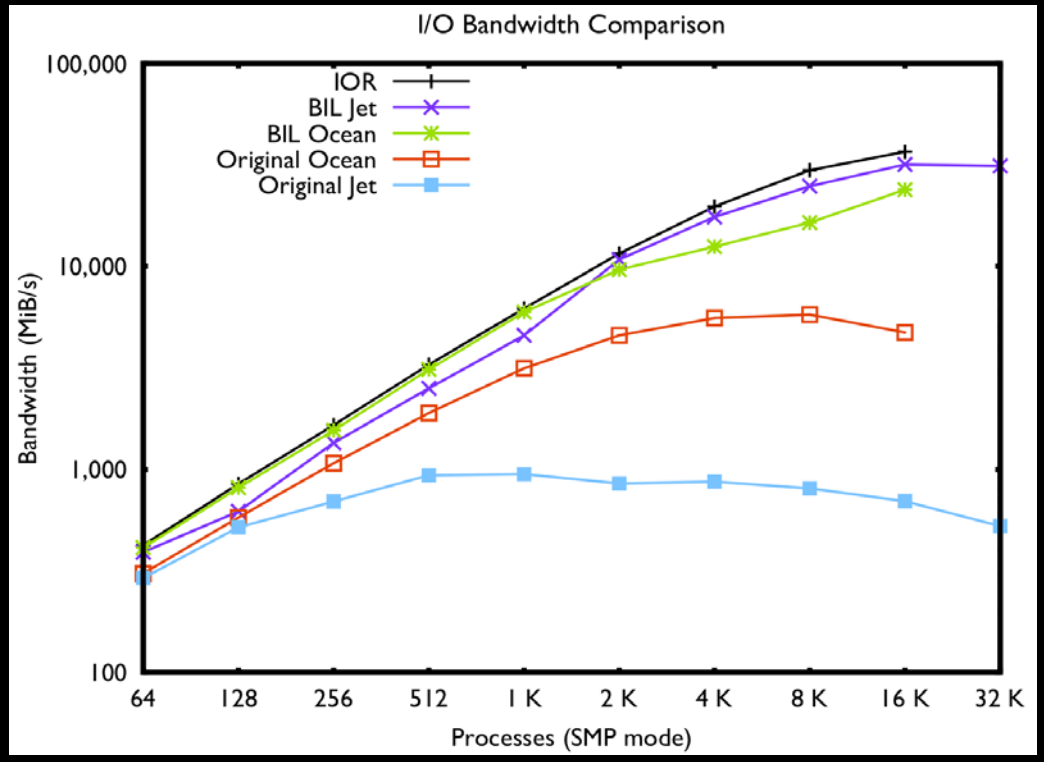
- 32 netCDF files, 3,600 X 2,400 X 40 grid, 2 variables, 82 GB

Jets dataset

- 2,000 raw files, 256 X 256 X 256 grid, 375 GB

Intrepid

- IBM BlueGene/P at Argonne National Lab
- 40,960 quad-core processors
- GPFS file system



Bandwidth results (log-log scale) of BIL versus standard single-file collective access in OSUFlow. The IOR benchmark is shown as the top line.

Closing Remarks

Simplified I/O solutions such as BIL will be necessary to foster more accepted practices for performing parallel I/O, especially in visualization and analysis

The ability to mask parallel I/O details under a simple design pattern, which in turn can provide superior scalability over other good I/O practices, provides a new capability for data-intensive analysis at scale

BIL does not solve everything

- We are bound by the storage choices of our scientists
- We are encouraging scientists to use better storage practices, such as HDF5's chunked layout
- Much more work is needed in multi-resolution I/O, and we believe better formats such as IDX [Kumar'10] are steps forward in this direction

BIL is available online at http://seelab.eecs.utk.edu/bil

BIL will appear in Visualization Viewpoint in IEEE CG&A, November 2011

References

[1] J. del Rosario, R. Bordawekar, and A. Choudhary. "Improved Parallel I/O via a Two-Phase Run-time Access Strategy," In *Proceedings of the Workshop on I/O in Parallel Computer Systems at IPPS `93*, pages 56–70, April 1993.

[2] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-Directed Collective I/O in Panda," In *Proceedings of Supercomputing `95*.

[3] R. Thakur and A. Choudhary. "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming*, 5(4):301–317, 1996.

[4] K. Gao, W. Liao, A. Choudhary, R. Ross, and R. Latham, "Combining I/O Operations for Multiple Array Variables in Parallel NetCDF." In the *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage*, held in conjunction with the IEEE Cluster Conference, 2009.

[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] 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.