

# HPC I/O for Computational Scientists

## Rob Latham and Phil Carns

Math and Computer Science Division  
Argonne National Laboratory  
[robl@mcs.anl.gov](mailto:robl@mcs.anl.gov), [carns@mcs.anl.gov](mailto:carns@mcs.anl.gov)

## Scot Breitenfeld

The HDF Group  
[brtnfld@hdfgroup.org](mailto:brtnfld@hdfgroup.org)

## Rachana Ananthakrishnan

Computation Institute  
[ranantha@uchicago.edu](mailto:ranantha@uchicago.edu)

Many thanks for their contributions go to:

## Rob Ross

Argonne National Laboratory  
[ross@mcs.anl.gov](mailto:ross@mcs.anl.gov)

## William Gropp

University of Illinois  
[wgropp@illinois.edu](mailto:wgropp@illinois.edu)

## Rusty Lusk

Argonne National Laboratory  
[lusk@mcs.anl.gov](mailto:lusk@mcs.anl.gov)

## Brent Welch

Google  
[Welch.breng@gmail.com](mailto:Welch.breng@gmail.com)

## Katie Antypas

NERSC  
[kantypas@lbl.gov](mailto:kantypas@lbl.gov)

## Rajeev Thakur

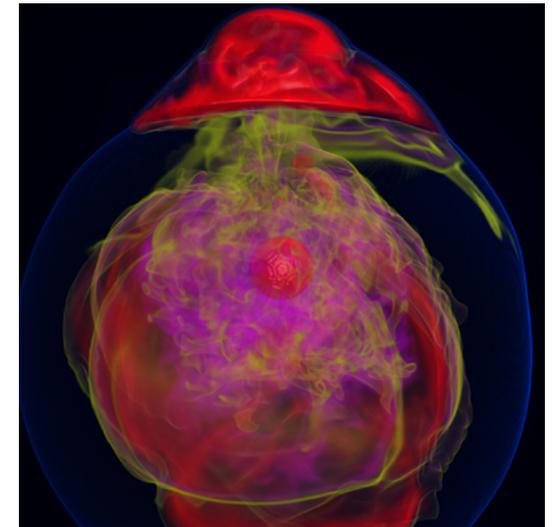
Argonne National Laboratory  
[thakur@mcs.anl.gov](mailto:thakur@mcs.anl.gov)

# Computational Science

- Use of computer simulation as a tool for greater understanding of the real world
  - Complements experimentation and theory
- Problems are increasingly computationally expensive
  - Large parallel machines needed to perform calculations
  - Critical to leverage parallelism in all phases
- **Data access is a huge challenge**
  - Using parallelism to obtain performance
  - Finding usable, efficient, and portable interfaces
  - Understanding and tuning I/O



IBM Blue Gene/Q system at Argonne National Laboratory.



Visualization of entropy in Terascale Supernova Initiative application. Image from Kwan-Liu Ma's visualization team at UC Davis.



# Goals and Outline

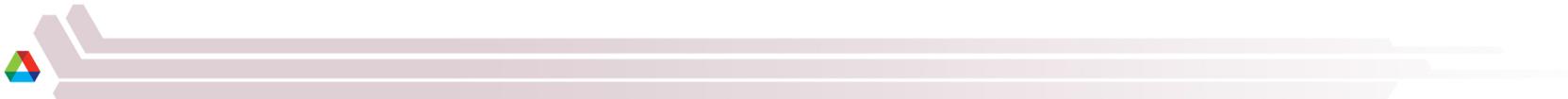
- Goals:
  - Share our view of HPC I/O hardware and software
  - Discuss interfaces that you can use to access I/O resources
  - Point to emerging and future trends in HPC I/O
- Outline (roughly)
  - Ways of thinking about I/O systems
  - How It Works: HPC I/O Systems
  - Using I/O systems
  - Emerging and future trends
- Notes
  - There will be slides that are hidden, don't be alarmed
  - After the morning break, we'll be looking through some of this code:  
<https://xgitlab.cels.anl.gov/ATPESC-IO/hands-on-2015>





## About Us (Before Lunch)

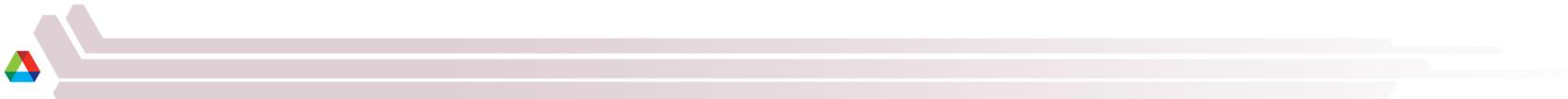
- Rob Latham
  - Principal Software Development Specialist, MCS Division  
Argonne National Laboratory
  - ROMIO MPI-IO implementation
  - Parallel netCDF high-level I/O library
  - Application outreach
- Phil Carns
  - Principal Software Development Specialist, MCS Division,  
Argonne National Laboratory
  - Darshan characterization tool
  - PVFS file system





## About Us (After Lunch)

- Scot Breitenfeld
  - HDF5 Team Lead
  - Applications Specialist
- Rachana Ananthakrishnan
  - Globus team, UChicago and Argonne
  - Distributed systems, recent focus on data management
  - Started on engineering team and now work on architecture and product management
  - Works with researchers across various domains from economists, physicists, radiologists, cancer biologists, to climate scientists



# Thinking about HPC I/O Systems

# HPC I/O Systems

**HPC I/O system is the hardware and software that assists in accessing data during simulations and analysis and retaining data between these activities.**

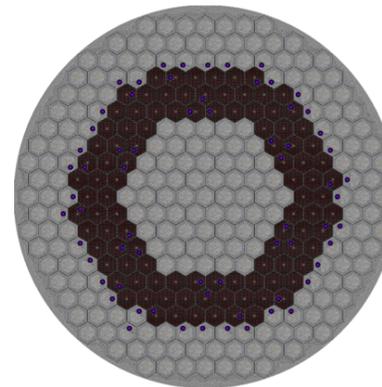
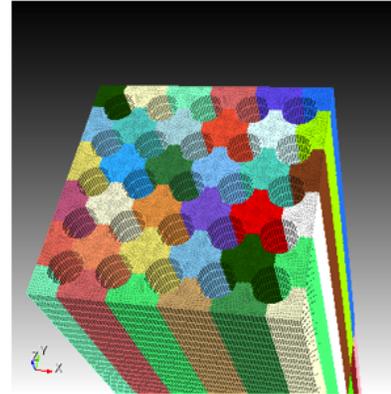
- Hardware: disks, disk enclosures, servers, networks, etc.
- Software: parallel file system, libraries, parts of the OS
  - More about these aspects of HPC I/O systems in upcoming slides
- Two “flavors” of I/O from applications:
  - **Defensive**: storing data to protect results from data loss due to system faults
  - **Productive**: storing/retrieving data as part of the scientific workflow
  - Note: Sometimes these are combined (i.e., data stored both protects from loss and is used in later analysis)
- “Flavor” influences priorities:
  - Defensive I/O: Spend as little time as possible
  - Productive I/O: Capture provenance, organize for analysis



# Data Complexity in Computational Science

- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, ...
  - Headers, attributes on data
- I/O systems have very simple data models
  - Tree-based hierarchy of containers
  - Some containers have streams of bytes (files)
  - Others hold collections of other containers (directories or folders)
- Mapping from one to the other is increasingly complex.

Images from T. Tautges (ANL) (upper left), M. Smith (ANL) (lower left), and K. Smith (MIT) (right).



**Model complexity:**  
Spectral element mesh (top) for thermal hydraulics computation coupled with finite element mesh (bottom) for neutronics calculation.



**Scale complexity:**  
Spatial range from the reactor core in meters to fuel pellets in millimeters.

# Data Volumes in Computational Science

Science teams are routinely working with tens and hundreds of terabytes (TBs) of data.

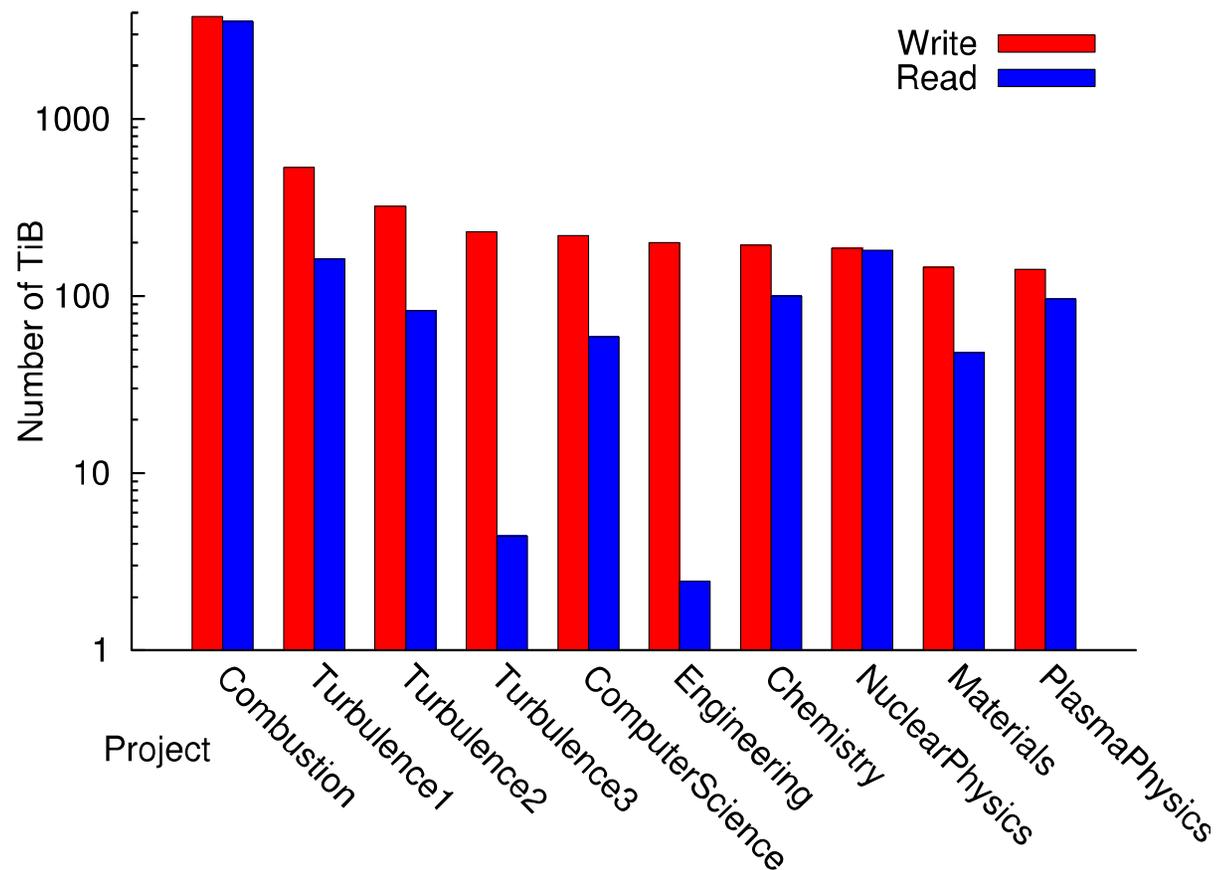
## Data requirements for select 2012 INCITE applications at ALCF (BG/P)

PI	Project	On-line Data (TBytes)	Off-line Data (TBytes)
Lamb	Supernovae Astrophysics	100	400
Khokhlov	Combustion in Reactive Gases	1	17
Lester	CO2 Absorption	5	15
Jordan	Seismic Hazard Analysis	600	100
Washington	Climate Science	200	750
Voth	Energy Storage Materials	10	10
Vashista	Stress Corrosion Cracking	12	72
Vary	Nuclear Structure and Reactions	6	30
Fischer	Reactor Thermal Hydraulic Modeling	100	100
Hinkel	Laser-Plasma Interactions	60	60
Elghobashi	Vaporizing Droplets in a Turbulent Flow	2	4



# Data Volumes in Computational Science

It's not just checkpoints – scientists are reading large volumes of data *into* HPC systems as part of their science.

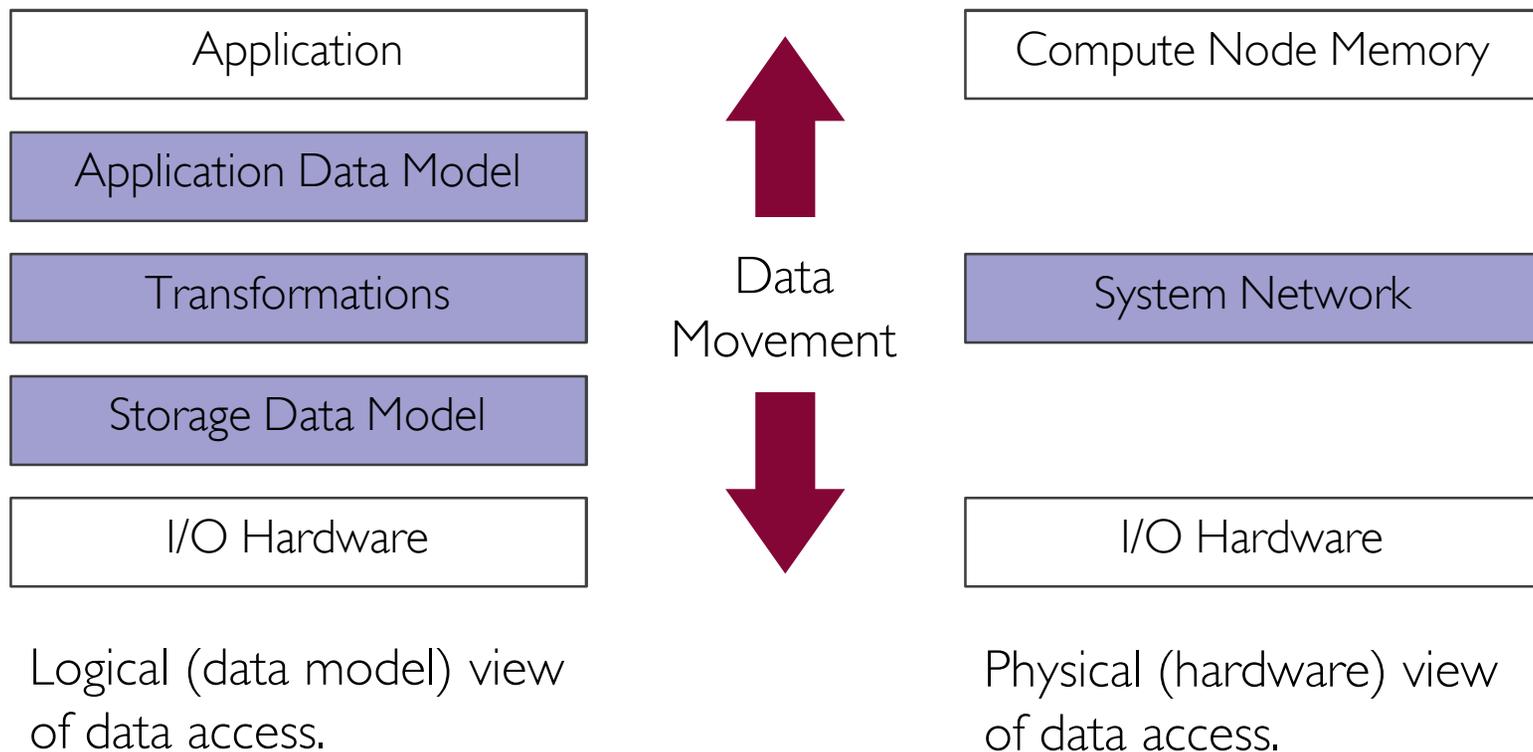


Top 10 data producer/consumers instrumented with Darshan from August 2014 to January 2015 (Mira).



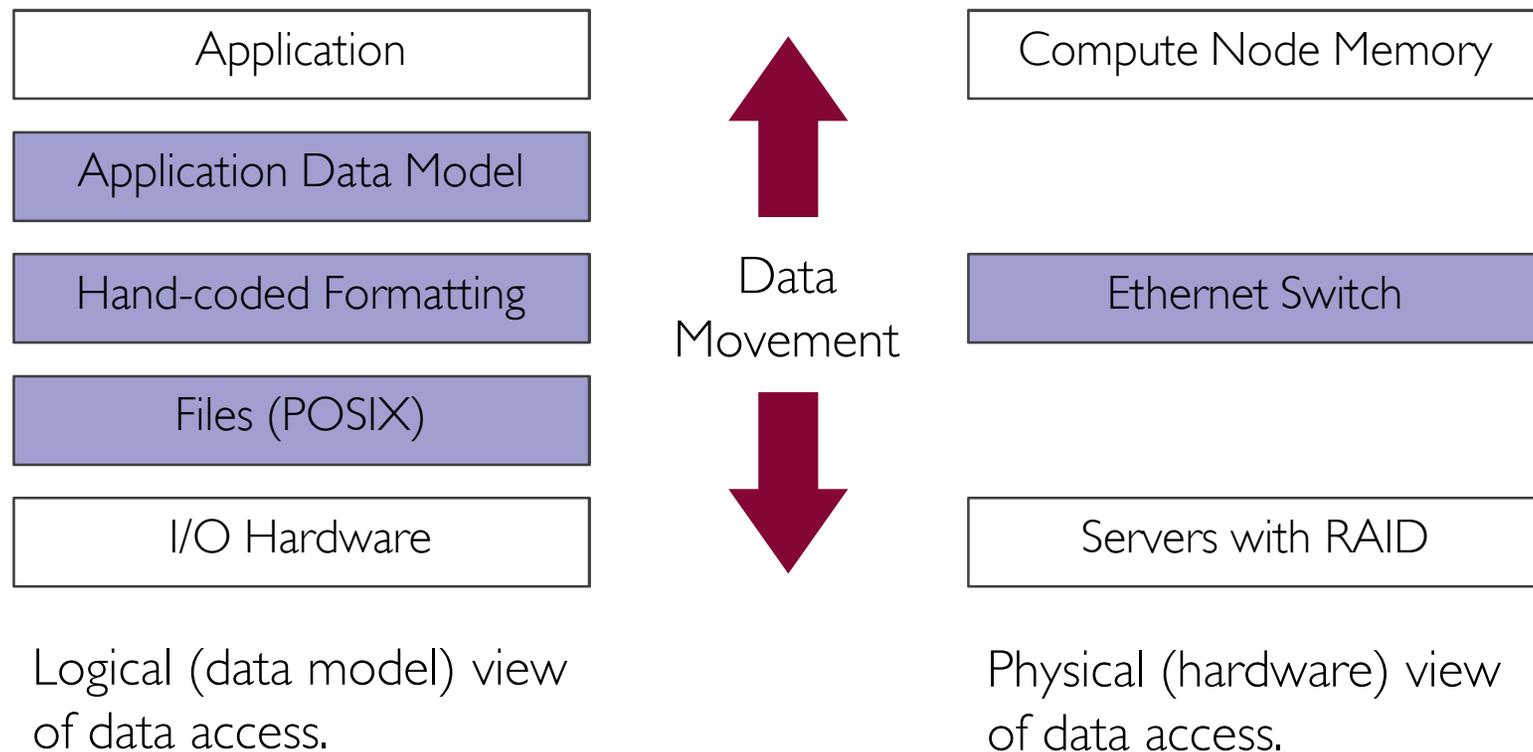
# Views of Data Access in HPC Systems

Two useful ways of thinking about data access are the “logical” view, considering data models in use, and the “physical” view, the components that data resides on and passes through.



## Data Access in Past HPC Systems\*

For many years, application teams wrote their own translations from their data models into files, and hardware model was relatively simple.

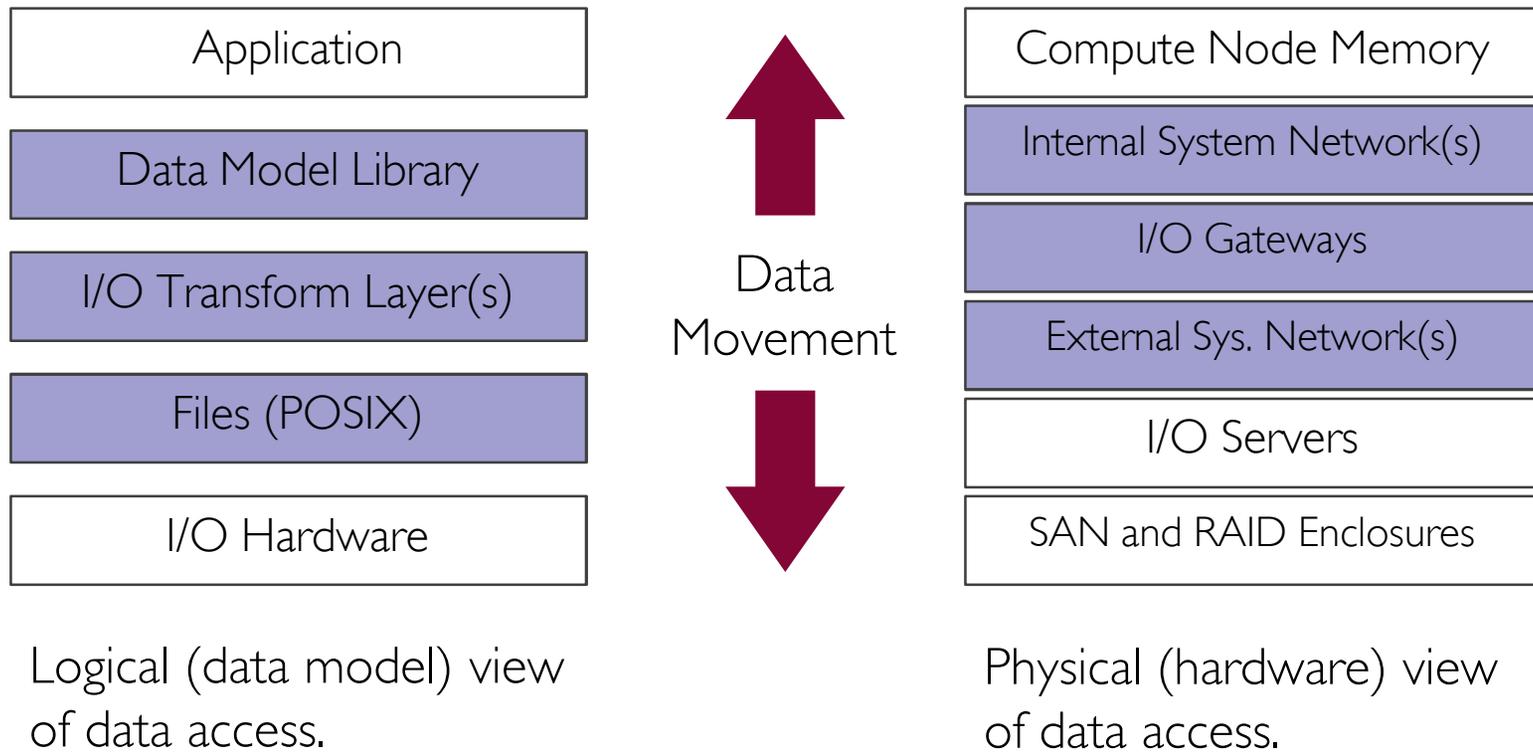


\* We're simplifying the story here somewhat ...



# Data Access in Current Large-scale Systems

**Current systems have greater support on the logical side, more complexity on the physical side.**

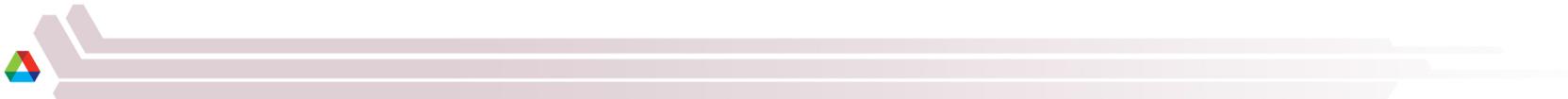


- Does this mean that applications must be more complex as well? *No!*
- More responsibility is assumed by system software and hardware in this model. It's just that there are more components to be aware of.



# Thinking about HPC I/O Systems

- Two (intertwined) challenges when thinking about data access:
  - Mapping application data model onto storage
  - Driving all the components so you don't have to wait too long for I/O
- Often these two can be at odds
  - “Richer” data models might require more I/O
  - Transformations that make writing fast might make reading slow (or vice versa)
- Lots of computer science R&D has gone into tackling these two problems
- Next we will dive down into some of the details of HPC I/O

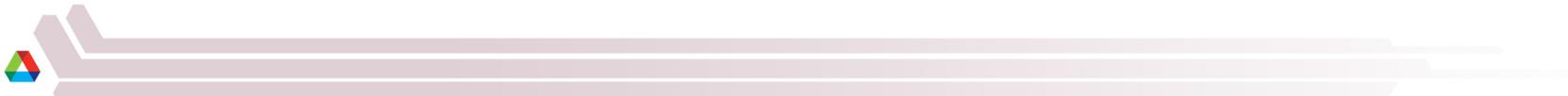


# What to expect from HPC I/O Systems

How do large-scale HPC I/O systems differ from conventional file systems (on servers, clusters, or even your laptop)?

- Vastly more bandwidth
  - You must read/write in parallel to exploit it
- ... but also higher latency
  - Multiple network/bus hops to get from application to disk

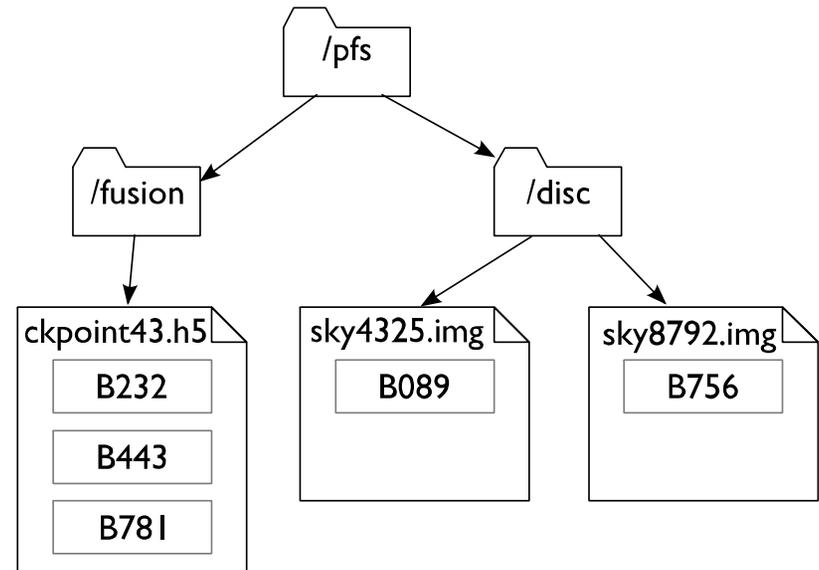
Most of the optimizations discussed in this presentation revolve around a central theme: organizing your data so that you can take advantage of the bandwidth while avoiding the latency.



# How It Works: HPC I/O Systems

# How It Works

- HPC I/O systems provide a *file system view* of stored data
  - File (i.e., POSIX) model of access
  - Shared view of data across the system
  - Access to same data from the outside (e.g., login nodes, data movers)
- Topics:
  - How is data stored and organized?
  - What support is there for application data models?
  - How does data move from clients to servers?
  - How is concurrent access managed?
  - What transformations are typically applied?

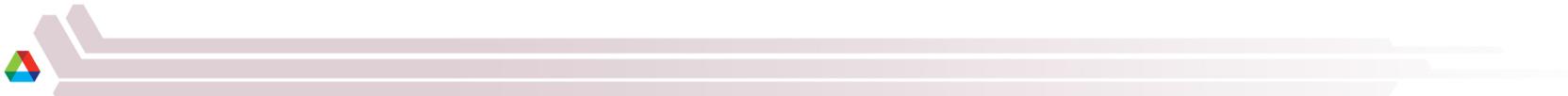


File system view consists of directories (a.k.a. folders) and files. Files are broken up into regions called extents or blocks.

## Storing and Organizing Data: Storage Model

HPC I/O systems are built around a *parallel file system* that organizes storage and manages access.

- Parallel file systems (PFSEs) are distributed systems that provide a file data model (i.e., files and directories) to users
- Multiple PFS servers manage access to storage, while PFS client systems run applications that access storage
- PFS clients can access storage resources in parallel!
  - This is critical for performance!
- On the surface it looks just like any other file system (home directory, laptop, etc.) but with different performance properties.



# Reading and Writing Data (etc.)

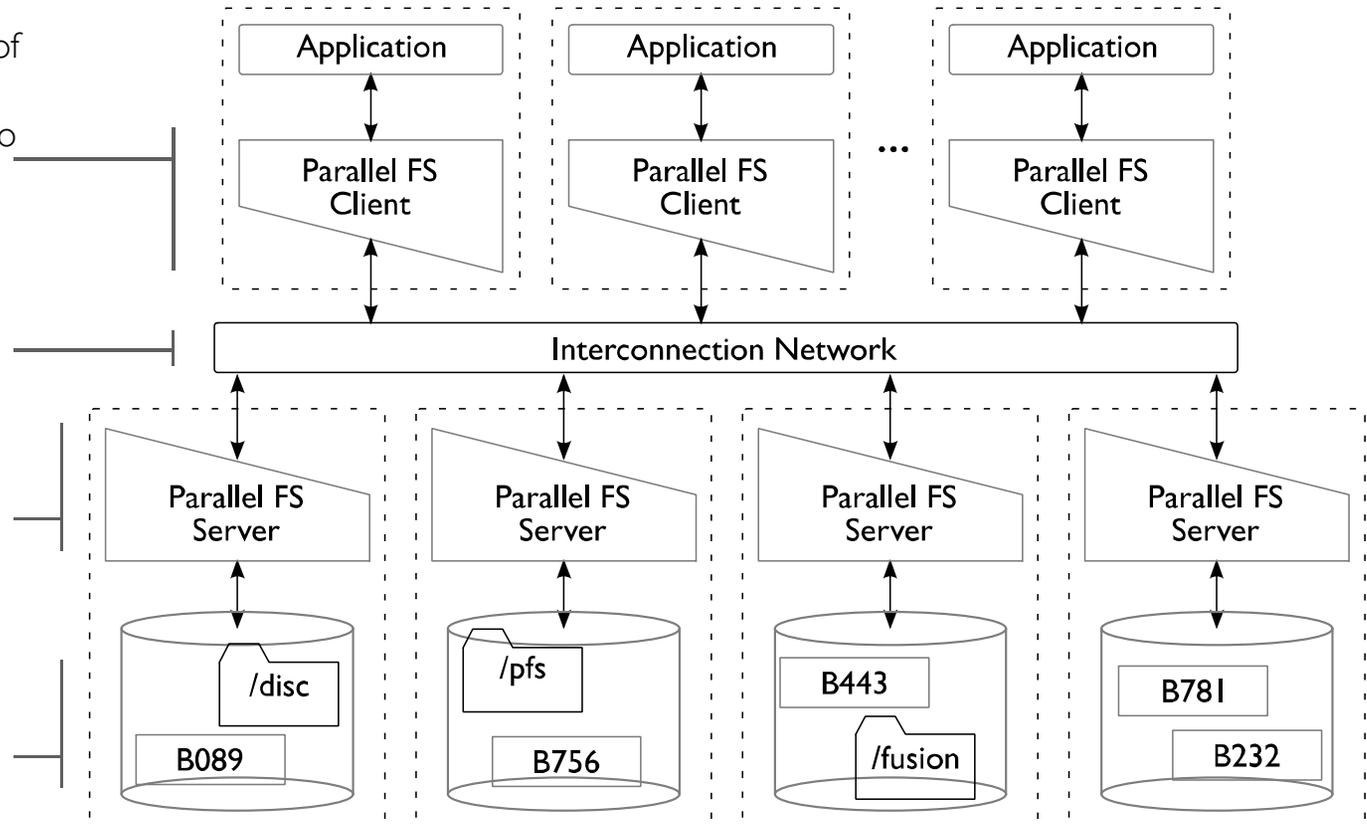
## PFS client software

requests operations on behalf of applications. Requests are sent as messages (RPC-like), often to multiple servers.

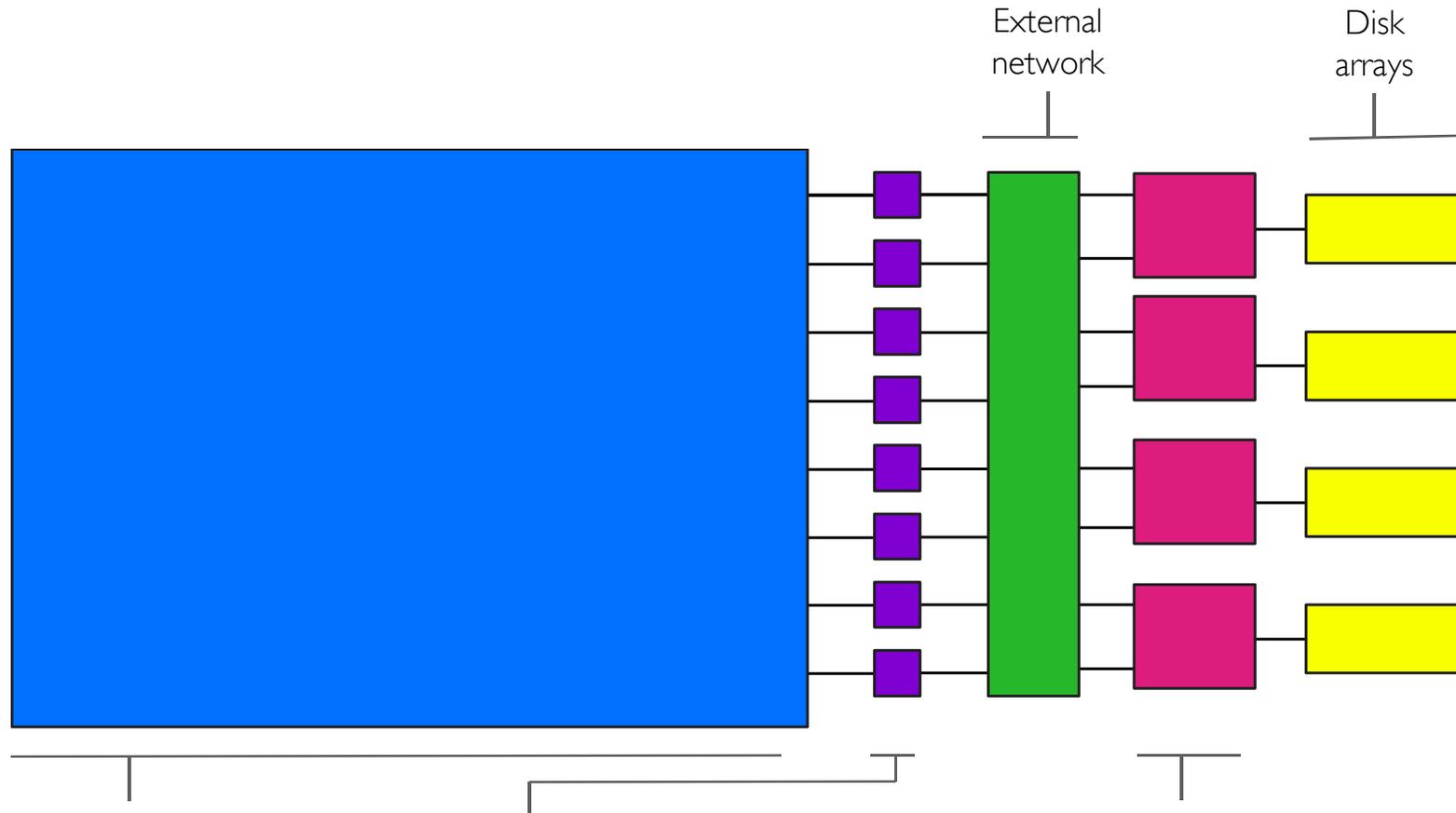
Requests pass over the interconnect, thus **each request incurs some latency**.

**PFS servers** manage local storage, services incoming requests from clients.

**RAID enclosures** protect against individual disk failures and map regions of data onto specific devices.



# Leadership Systems have an additional HW layer



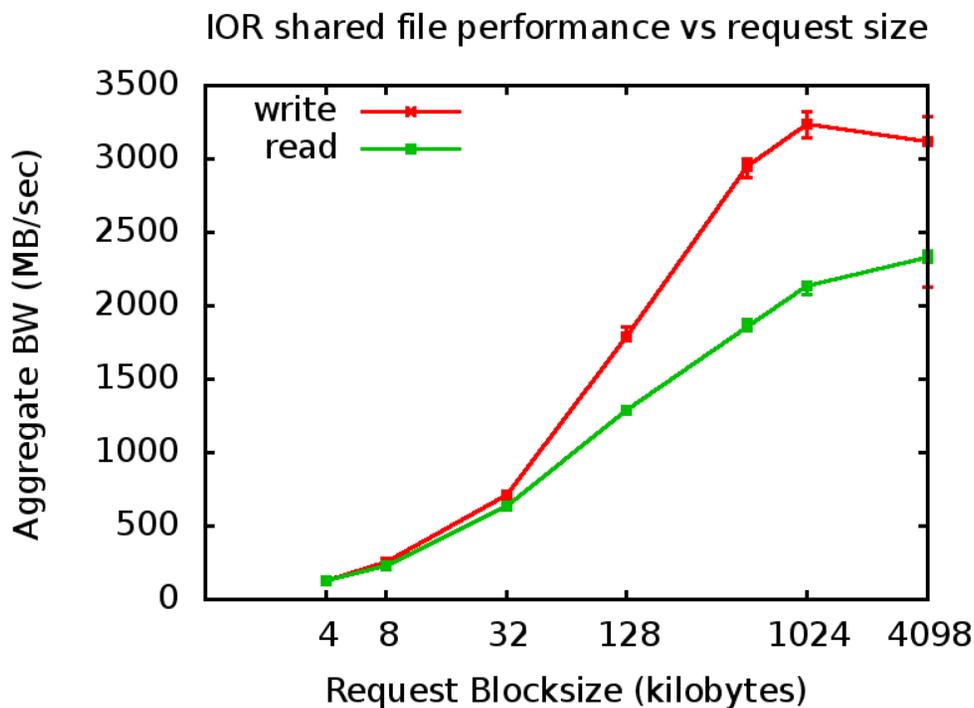
**Compute nodes** run application processes. Data model software also runs here, and some I/O transformations are performed here.

**I/O forwarding nodes** (or I/O gateways) shuffle data between compute nodes and external resources, including storage.

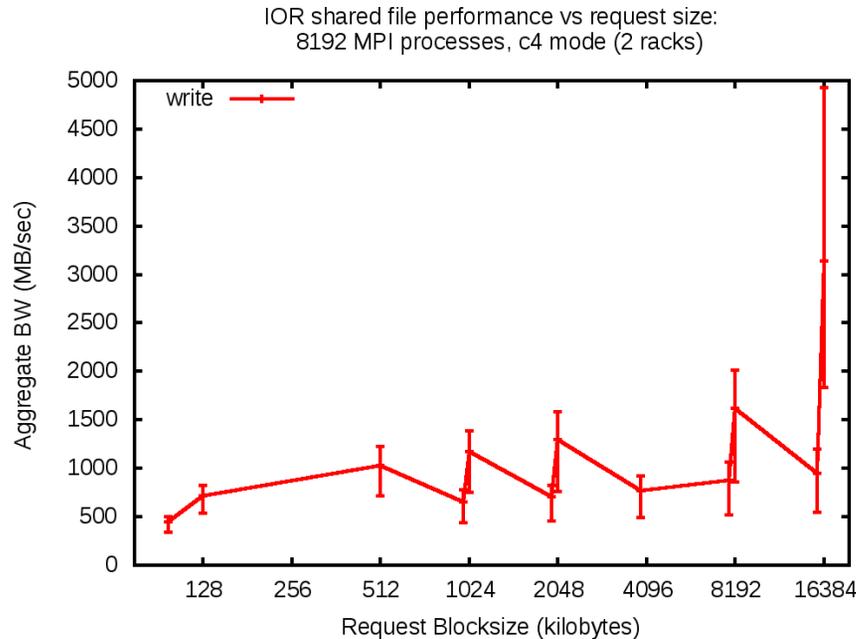
**Storage nodes** run the parallel file system.

# Request Size and I/O Rate

Interconnect latency has a significant impact on effective rate of I/O. Typically I/Os should be in the O(Mbytes) range.



2K processes of IBM Blue Gene/P at ANL.



8k processes of IBM Blue Gene /Q at ANL



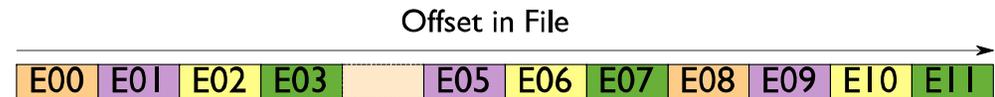
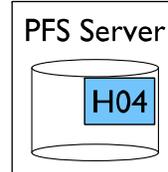
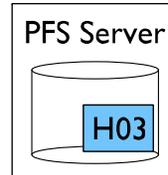
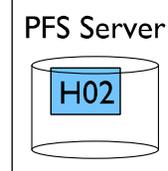
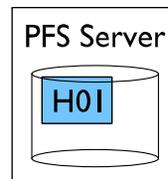
# Data Distribution in Parallel File Systems

**Distribution across multiple servers allows concurrent access.**

Logically a file is an extendable sequence of bytes that can be referenced by offset into the sequence.

Metadata associated with the file specifies a mapping of this sequence of bytes into a set of objects on PFS servers.

Extents in the byte sequence are mapped into objects on PFS servers. This mapping is usually determined at file creation time and is often a round-robin distribution of a fixed extent size over the allocated objects.



Space is allocated on demand, so unwritten "holes" in the logical file do not consume disk space.

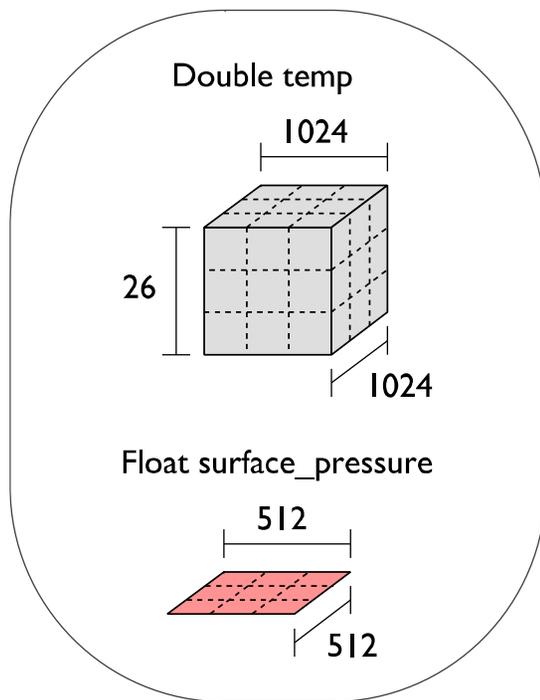
A static mapping from logical file to objects allows clients to easily calculate server(s) to contact for specific regions, eliminating need to interact with a metadata server on each I/O operation.



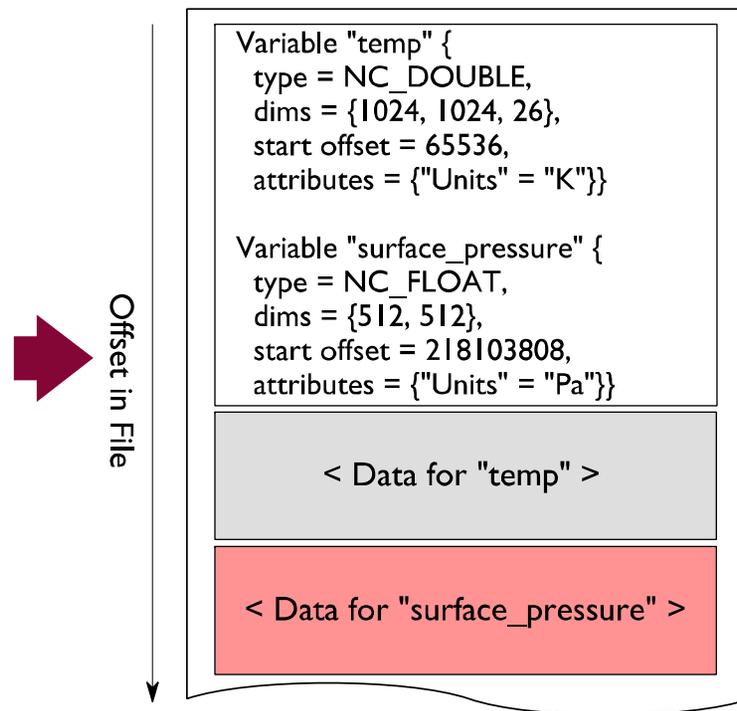
# Storing and Organizing Data: Application Model(s)

Application data models are supported via libraries that map down to files (and sometimes directories).

Application Data Structures



netCDF File "checkpoint07.nc"



netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

Data for variables is stored in contiguous blocks, encoded in a portable binary format according to the variable's type.

# HPC I/O Software Stack

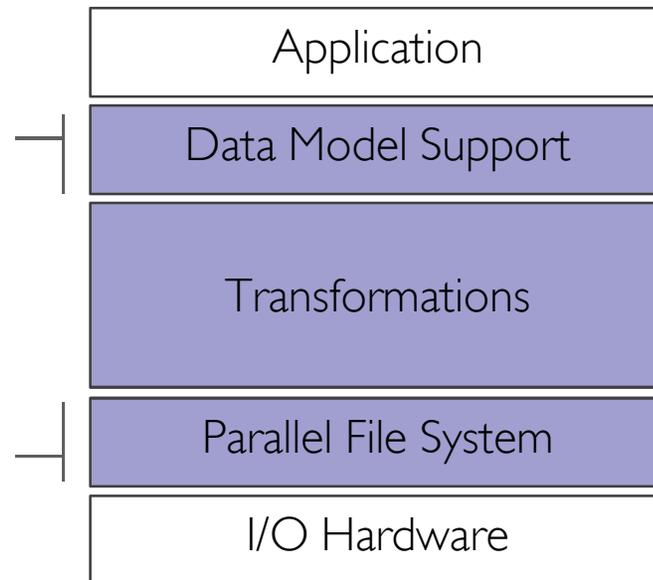
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack*.

**Data Model Libraries** map application abstractions onto storage abstractions and provide data portability.

*HDF5, Parallel netCDF, ADIOS*

**Parallel file system** maintains logical file model and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*



**I/O Middleware** organizes accesses from many processes, especially those using collective I/O.

*MPI-IO, GLEAN, PLFS*

**I/O Forwarding** transforms I/O from many clients into fewer, larger request; reduces lock contention; and bridges between the HPC system and external storage.

*IBM ciod, IOFSL, Cray DVS*

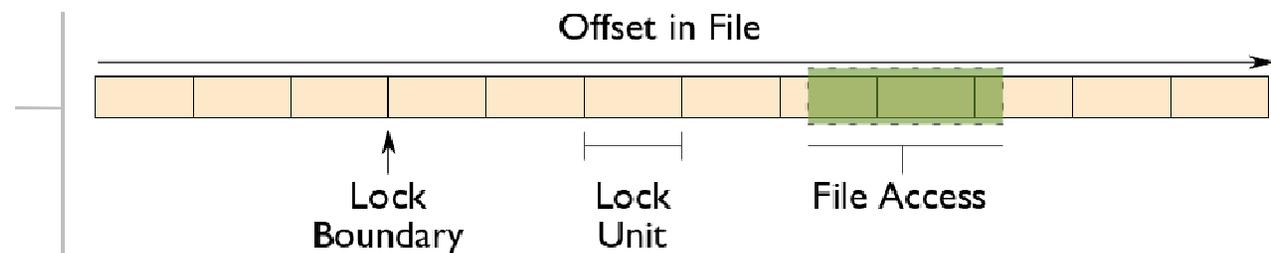
# How It Works: HPC I/O Performance

# Managing Concurrent Access

**Files are treated like global shared memory regions. Locks are used to manage concurrent access:**

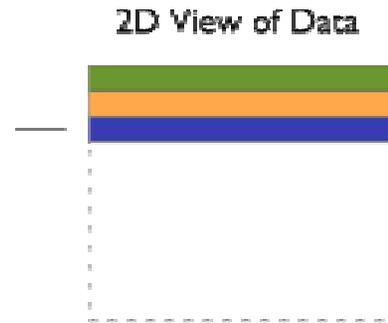
- Files are broken up into lock units
  - Unit boundaries are dictated by the storage system regardless of access pattern
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.



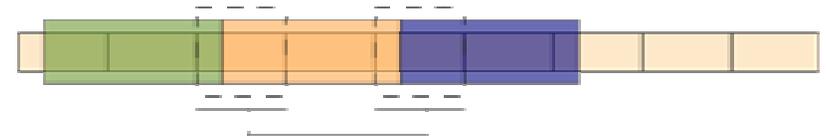
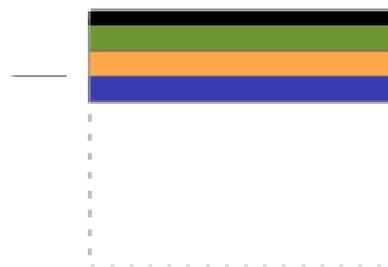
# Implications of Locking in Concurrent Access

The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.



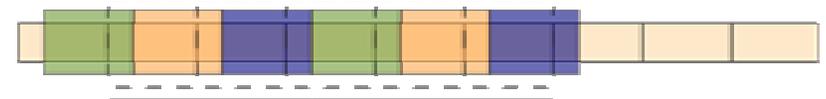
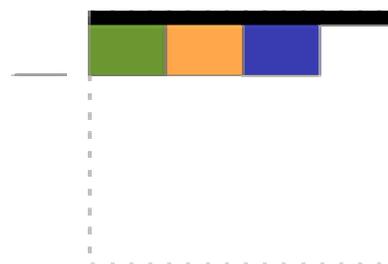
When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.



These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

In this example, processes exhibit a block-block access pattern (e.g. accessing a subarray). This results in many interleaved accesses in the file.



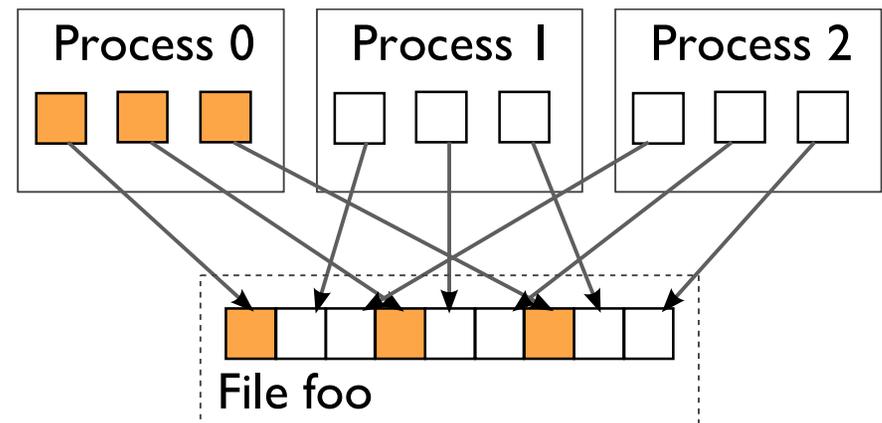
When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.



# I/O Transformations

Software between the application and the PFS performs transformations, primarily to improve performance.

- Goals of transformations:
  - Reduce number of operations to PFS (avoiding latency)
  - Avoid lock contention (increasing level of concurrency)
  - Hide number of clients (more on this later)
- With “transparent” transformations, data ends up in the same locations in the file
  - i.e., the file system is still aware of the actual data organization

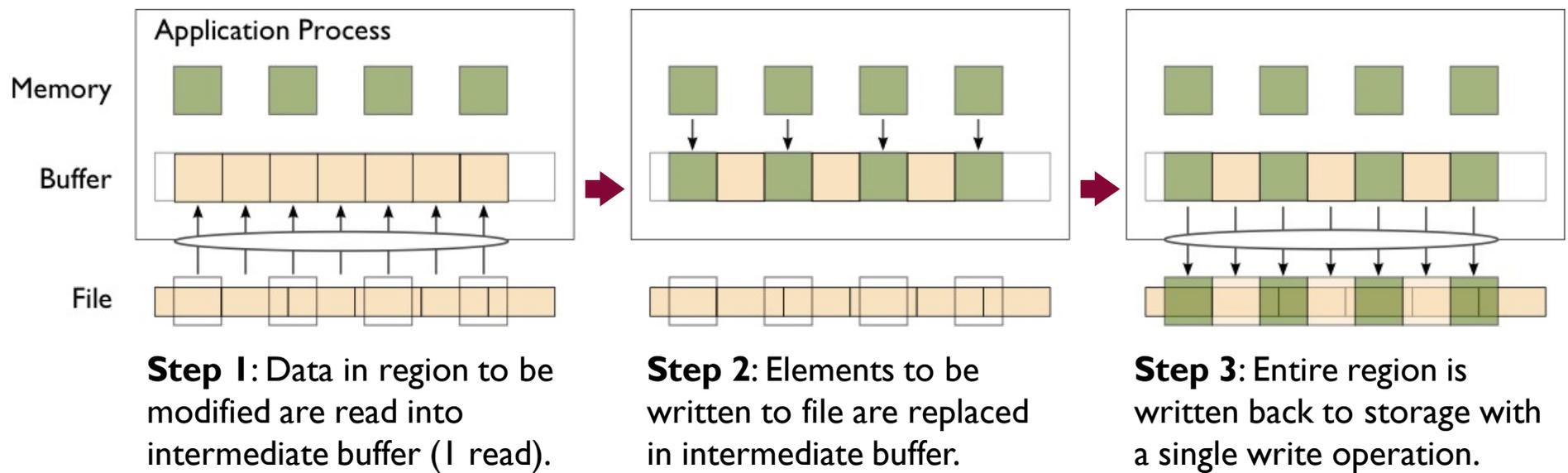


When we think about I/O transformations, we consider the mapping of data between application processes and locations in file.

# Reducing Number of Operations

Since most operations go over the network, I/O to a PFS incurs more latency than with a local FS. *Data sieving* is a technique to address I/O latency by combining operations:

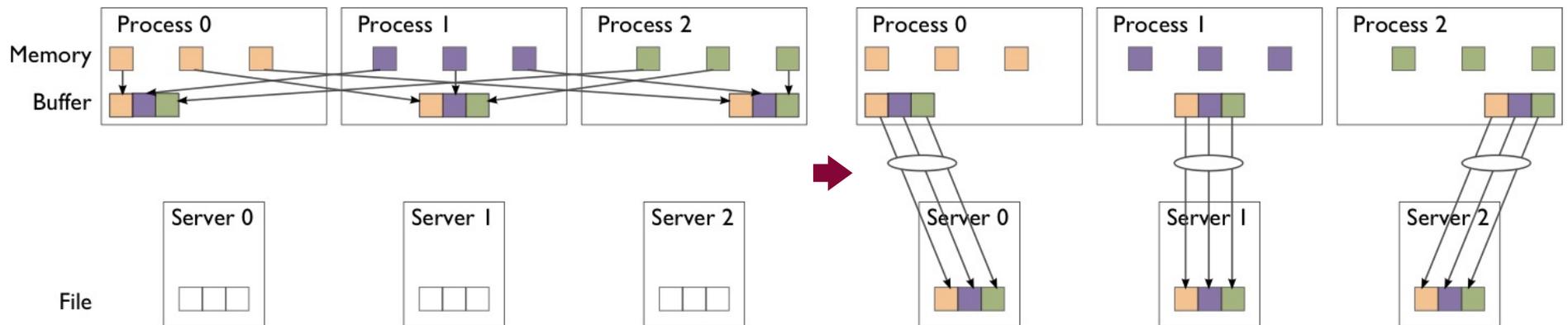
- When reading, application process reads a large region holding all needed data and pulls out what is needed
- When writing, three steps required (below)
- Somewhat counter-intuitive: do extra I/O to avoid contention



# Avoiding Lock Contention

To avoid lock contention when writing to a shared file, we can reorganize data between processes. *Two-phase I/O* splits I/O into a data reorganization phase and an interaction with the storage system (two-phase write depicted):

- Data exchanged between processes to match file layout
- 0<sup>th</sup> phase determines exchange schedule (not shown)



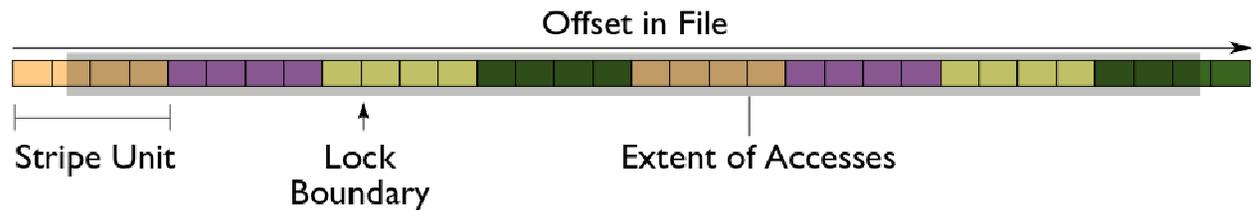
**Phase 1:** Data are exchanged between processes based on organization of data in file.

**Phase 2:** Data are written to file (storage servers) with large writes, no contention.

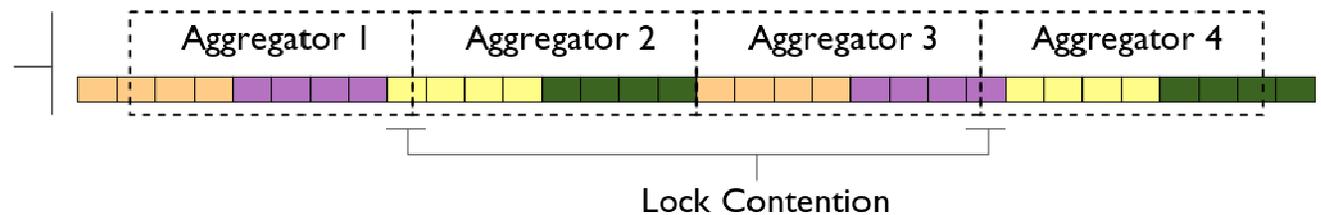
# Two-Phase I/O Algorithms

(or, You don't want to do this yourself...)

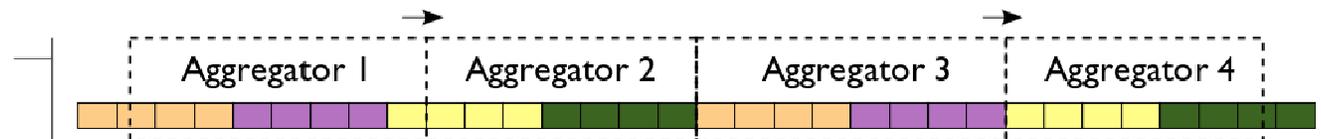
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



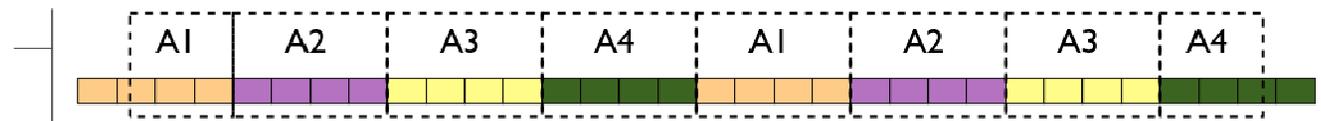
One approach is to evenly divide the region accessed across aggregators.



Aligning regions with lock boundaries eliminates lock contention.



Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).

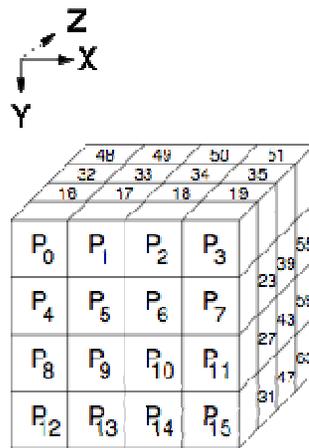
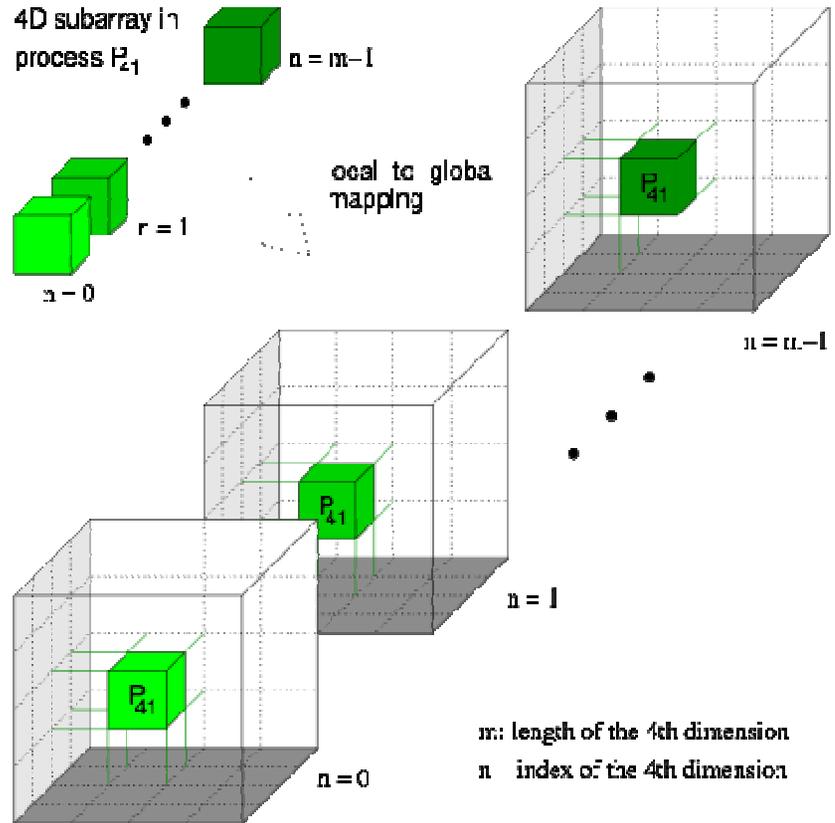
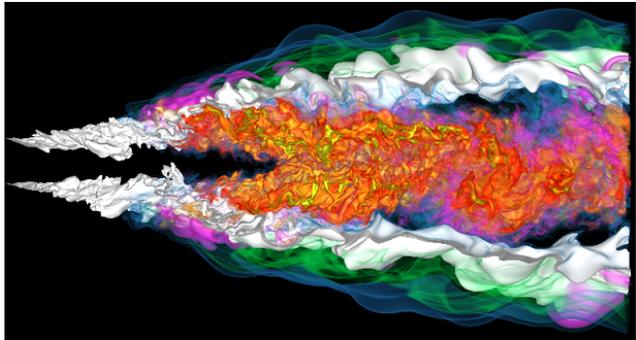


For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.



# S3D Turbulent Combustion Code

- S3D is a turbulent combustion application using a direct numerical simulation solver from Sandia National Laboratory
- Checkpoints consist of four global arrays
  - 2 3-dimensional
  - 2 4-dimensional
  - 50x50x50 fixed subarrays



Thanks to Jackie Chen (SNL), Ray Grout (SNL), and Wei-Keng Liao (NWU) for providing the S3D I/O benchmark, Wei-Keng Liao for providing this diagram, C. Wang, H. Yu, and K.-L. Ma of UC Davis for image.



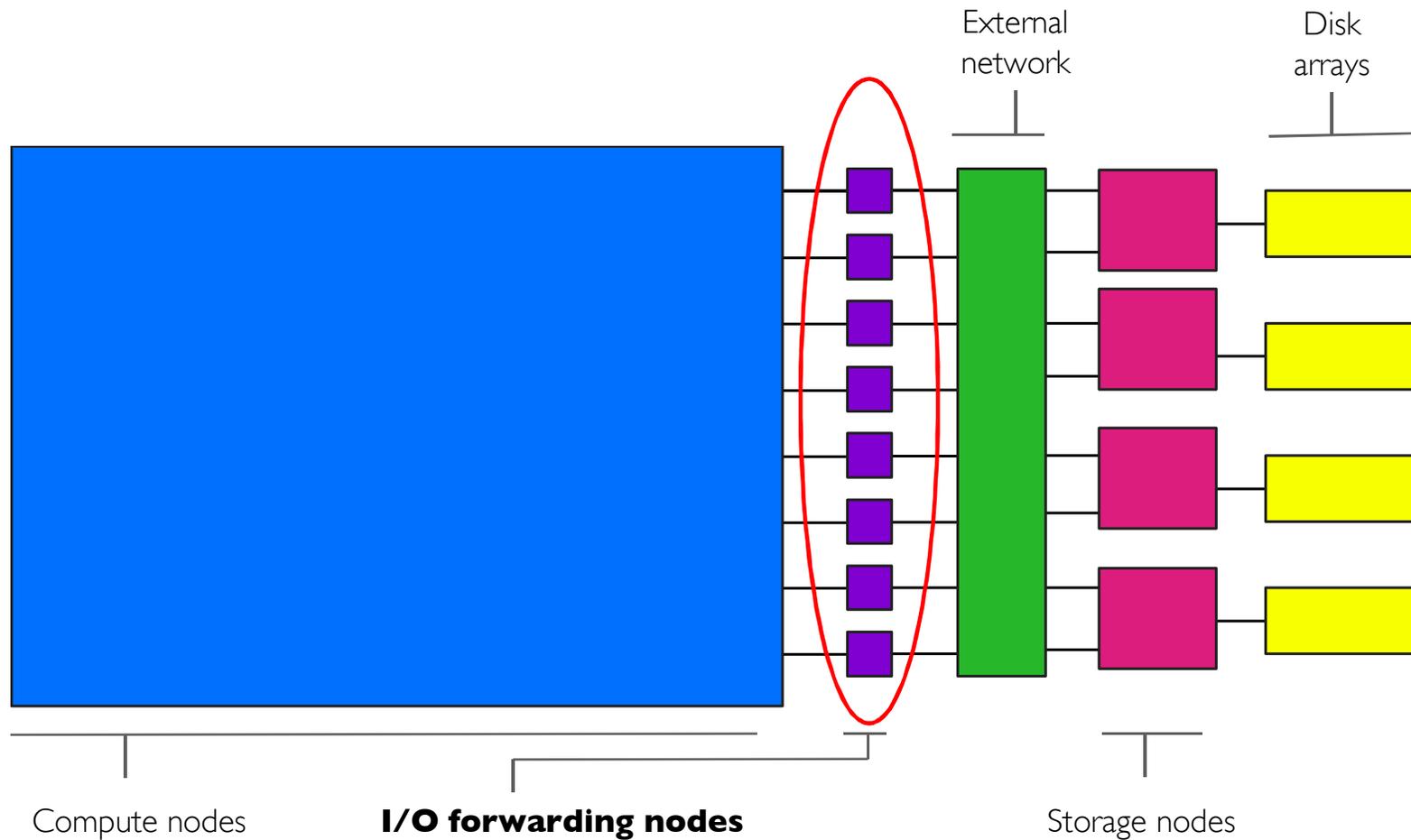
# Impact of Transformations on S3D I/O

- Testing with PnetCDF output to single file, three configurations, 16 processes
  - All MPI-IO optimizations (collective buffering and data sieving) disabled
  - Independent I/O optimization (data sieving) enabled
  - Collective I/O optimization (collective buffering, a.k.a. two-phase I/O) enabled

	Coll. Buffering and Data Sieving Disabled	Data Sieving Enabled	Coll. Buffering Enabled (incl. Aggregation)
POSIX writes	102,401	81	<b>5</b>
POSIX reads	0	80	0
MPI-IO writes	64	64	64
Unaligned in file	102,399	80	4
Total written (MB)	6.25	<b>87.11</b>	6.25
Runtime (sec)	1443	11	6.0
Avg. MPI-IO time per proc (sec)	<b>1426.47</b>	4.82	0.60



# Transformations in the I/O Forwarding Step



**I/O forwarding nodes**  
(or I/O gateways) shuffle data between compute nodes and external resources, including storage.



## Transformations in the I/O Forwarding Step

Another way of transforming data access by clients is by introducing new hardware: *I/O forwarding nodes*.

- I/O forwarding nodes serve a number of functions:
  - Bridge between internal and external networks
  - Run PFS client software, allowing lighter-weight solutions internally
  - Perform I/O operations on behalf of multiple clients
  - Transparently transform data on its way to and from the file system
- Transformations can take many forms:
  - Performing one file open on behalf of many processes
  - Combining small accesses into larger ones
  - Caching of data (sometimes between I/O forwarding nodes)

Note: Current vendor implementations don't aggressively aggregate.

- Compute nodes can be allocated to provide a similar service



# “Not So Transparent” Transformations

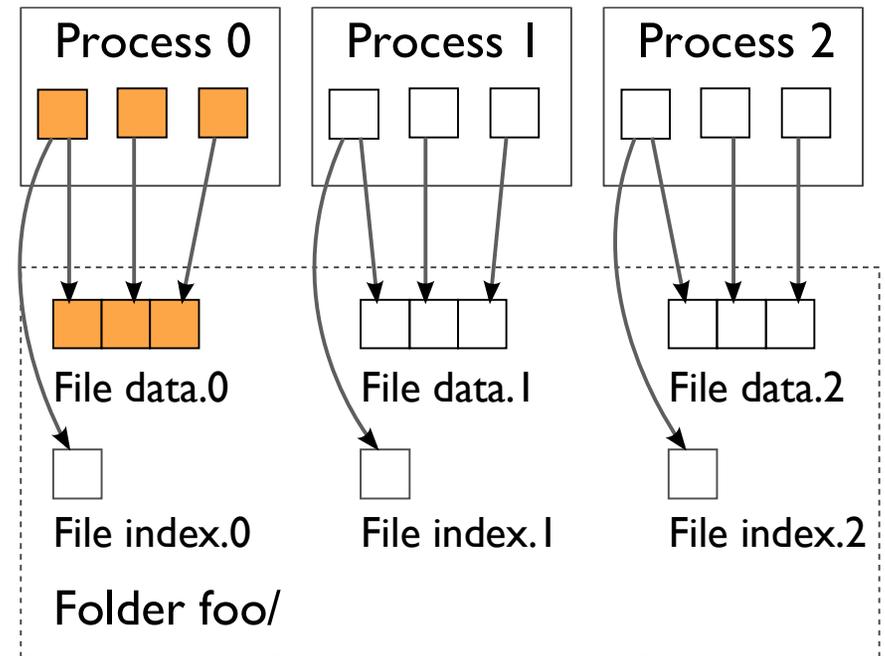
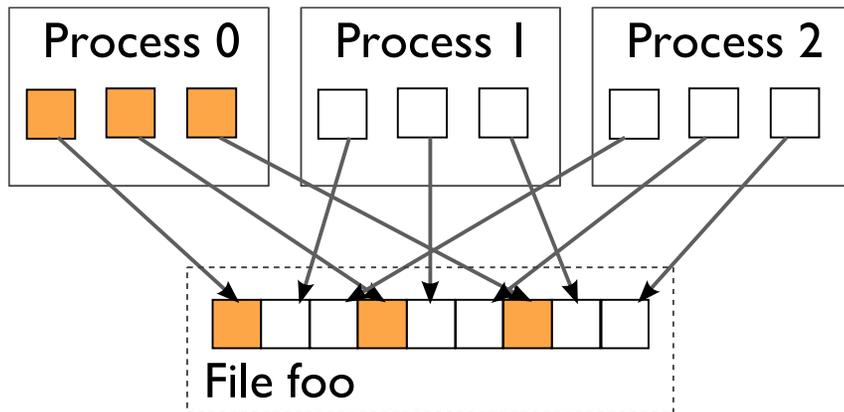
**Some transformations result in file(s) with different data organizations than the user requested.**

- If processes are writing to different files, then they will not have lock conflicts
- What if we convert writes to the same file into writes to different files?
  - Need a way to group these files together
  - Need a way to track what we put where
  - Need a way to reconstruct on reads
- Parallel Log-Structured File System software does this
  - It is transparent from the application/user perspective (it presents a virtual view of the data) but not from the storage system perspective

See J. Bent et al. PLFS: a checkpoint filesystem for parallel applications. SC2009. Nov. 2009.



# Parallel Log Structured File System



Application intends to interleave data regions into single file.

Transparent transformations such as data sieving and two-phase I/O preserve data order on the file system.

**PLFS remaps I/O** into separate log files per process, with indices capturing locations of data in these files.

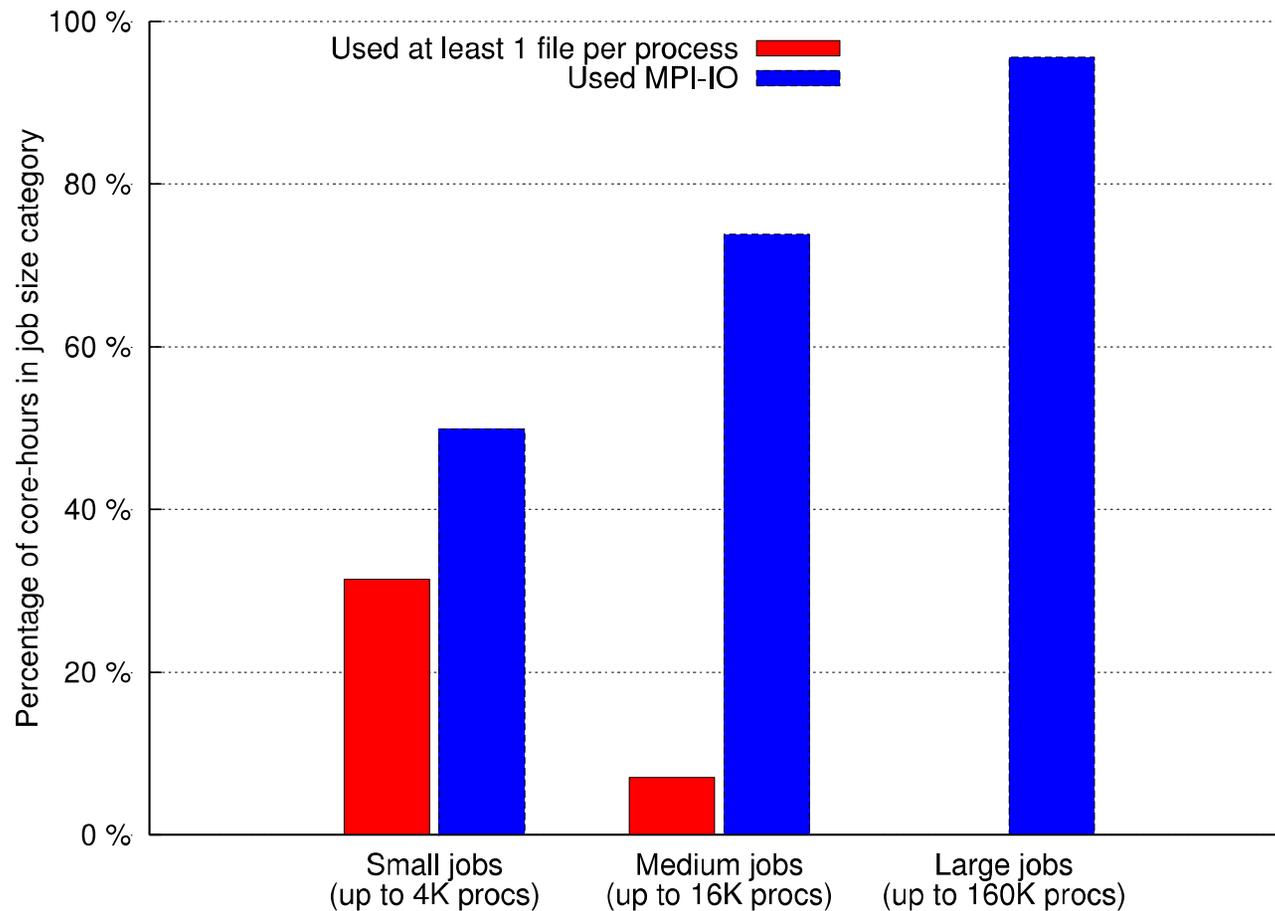
**PLFS software needed when reading** to reconstruct the file view.

See J. Bent et al. PLFS: a checkpoint filesystem for parallel applications. SC2009. Nov. 2009.



# Why not just write a file per process?

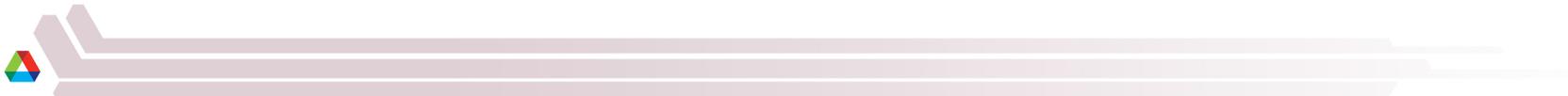
**File per process vs. shared file access as function of job size on Intrepid Blue Gene/P system**



# I/O Transformations and the Storage Data Model

Historically, the storage data model has been the POSIX file model, and the PFS has been responsible for managing it.

- Transparent transformations work within these limitations
- When data model libraries are used:
  - Transforms can take advantage of more knowledge
    - e.g., dimensions of multidimensional datasets
  - Doesn't matter so much whether there is a single file underneath
  - Or in what order the data is stored
  - As long as portability is maintained
- Single stream of bytes in a file is inconvenient for parallel access
  - Future designs might provide a different underlying model



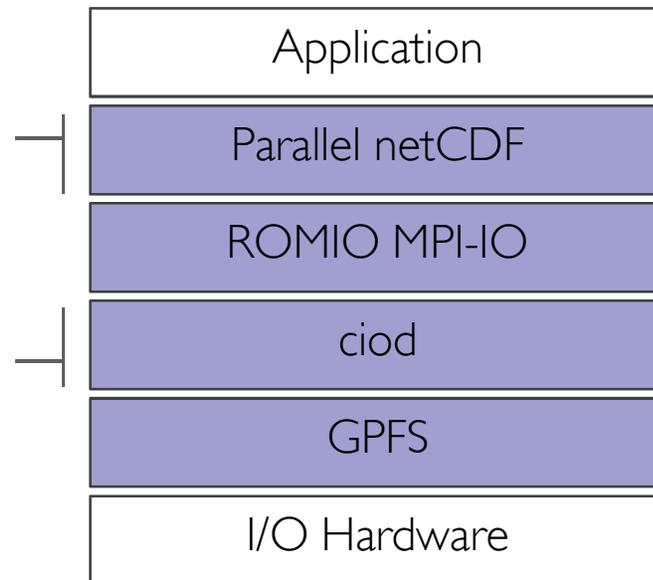
# How It Works: Today's I/O Systems

# An Example HPC I/O Software Stack

This example I/O stack captures the software stack used in some applications on the IBM Blue Gene/Q system at Argonne.

**Parallel netCDF** is used in numerous climate and weather applications running on DOE systems. Built in collaboration with NWU.

**ciod** is the I/O forwarding implementation on the IBM Blue Gene/P and Blue Gene/Q systems.

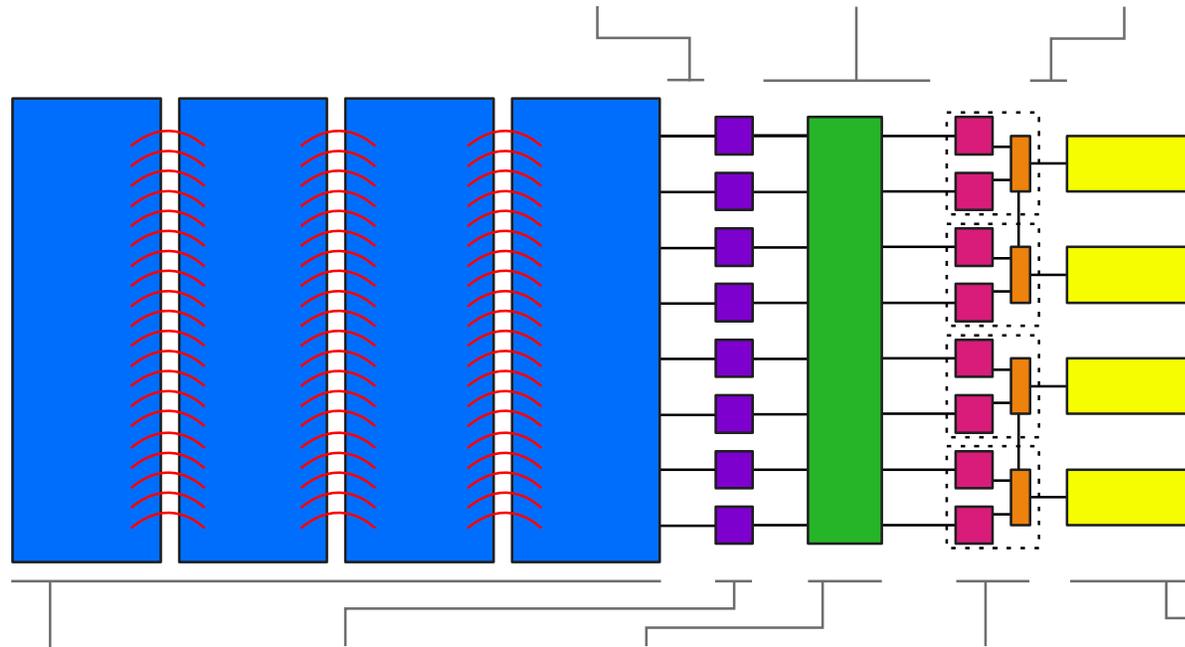


**ROMIO** is the basis for virtually all MPI-IO implementations on all platforms today and the starting point for nearly all MPI-IO research. Incorporates research from NWU and patches from vendors.

**GPFS** is a production parallel file system provided by IBM.

# Mira Blue Gene/Q and its Storage System

BG/Q Optical 2x16 Gbit/sec      QDR InfiniBand 32 Gbit/sec      Serial ATA 6.0 Gbit/sec



**Compute nodes**  
run applications and  
some I/O middleware.

*768K cores with 1 Gbyte  
of RAM each*

**Gateway nodes**  
run parallel file system  
client software and  
forward I/O operations  
from HPC clients.

*384 16-core PowerPC  
A2 nodes with 16 Gbytes  
of RAM each*

**Commodity  
network** primarily  
carries storage traffic.

*QDR Infiniband  
Federated Switch*

**Storage nodes**  
run parallel file system  
software and manage  
incoming FS traffic  
from gateway nodes.

*SFA12KE hosts VM  
running GPFS servers*

**Enterprise storage**  
controllers and large racks  
of disks are connected via  
InfiniBand.

*32 DataDirect SFA12KE;  
560 3 Tbyte drives + 32  
200 GB SSD; 16  
InfiniBand ports per pair*



# Takeaways

- Parallel file systems provide the underpinnings of HPC I/O solutions
- Data model libraries provide alternative data models for applications
  - PnetCDF and HDF5 will both be discussed in detail later in the day
- Characteristics of PFSEs lead to the need for transformations in order to achieve high performance
  - Implemented in a number of different software layers
  - Some preserving file organization, others breaking it
- Number of layers complicates performance debugging
  - Some ways of approaching this discussed later in the day



# Building an I/O API

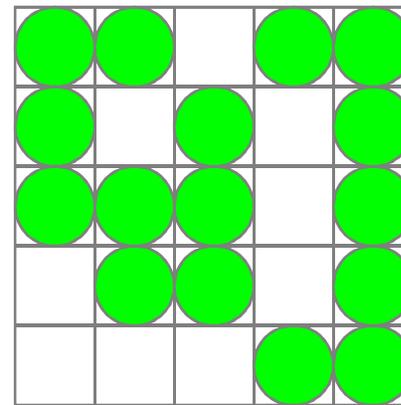
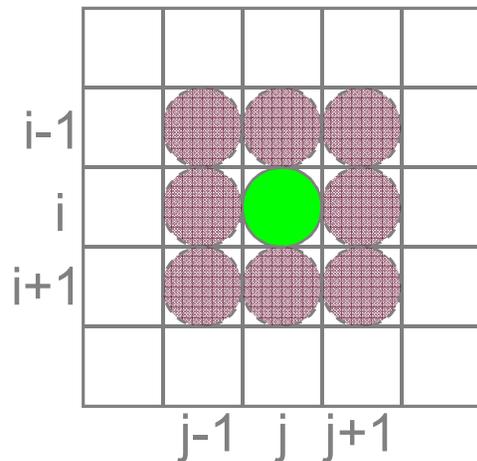
# Conway's Game of Life

- We use Conway's Game of Life as a simple example to illustrate the program issues common to many codes that use regular meshes, such as PDE solvers
  - Allows us to concentrate on the I/O issues
- Game of Life is a cellular automaton
  - Described in 1970 Scientific American
  - Many interesting behaviors; see:
    - <http://www.ibiblio.org/lifepatterns/october1970.html>



# Rules for Life

- Matrix values  $A(i,j)$  initialized to 1 (live) or 0 (dead)
- In each iteration,  $A(i,j)$  is set to
  - 1 (live) if either
    - the sum of the values of its 8 neighbors is 3, or
    - the value was already 1 and the sum of its 8 neighbors is 2 or 3
  - 0 (dead) otherwise



# Implementing Life

- For the non-parallel version, we:
  - Allocate a 2D matrix to hold state
    - Actually two matrices, and we will swap them between steps
  - Initialize the matrix
    - Force boundaries to be “dead”
    - Randomly generate states inside
  - At each time step:
    - Calculate each new cell state based on previous cell states (including neighbors)
    - Store new states in second matrix
    - Swap new and old matrices

All code examples in this tutorial can be found in hands-on repo:

[xgitlab.cels.anl.gov/ATPESC-IO/hands-on-2015](https://xgitlab.cels.anl.gov/ATPESC-IO/hands-on-2015)



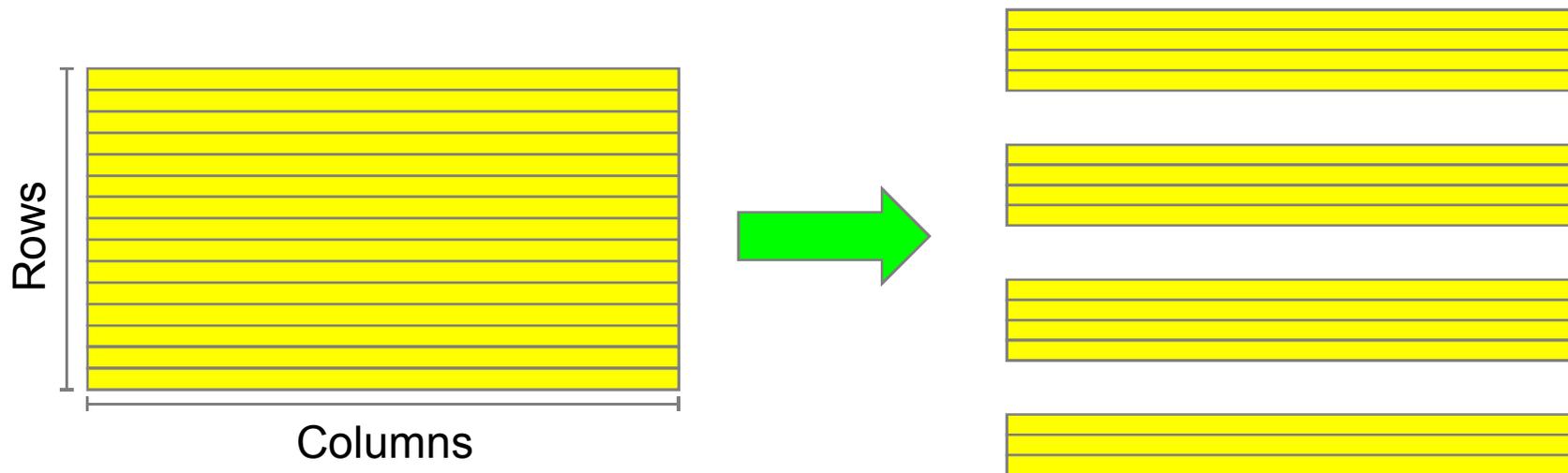
## Steps in Designing a Parallel Game of Life

- Start with the “global” array as the main object
  - Natural for output – result we’re computing
- Describe decomposition in terms of global array
- Describe communication of data, still in terms of the global array
- Define the “local” arrays and the communication between them by referring to the global array



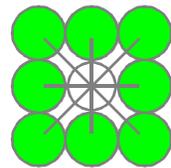
# Step 1: Description of Decomposition

- By rows (1D or row-block)
  - Each process gets a group of adjacent rows

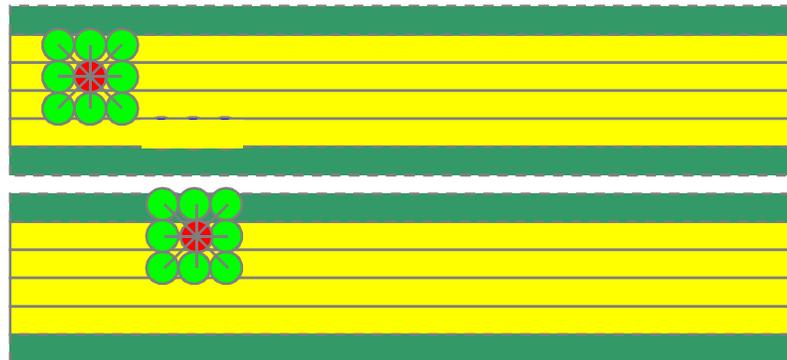


## Step 2: Communication

- “Stencil” requires read access to data from neighbor cells



- We allocate extra space on each process to store neighbor cells
- Use send/recv or RMA to update prior to computation



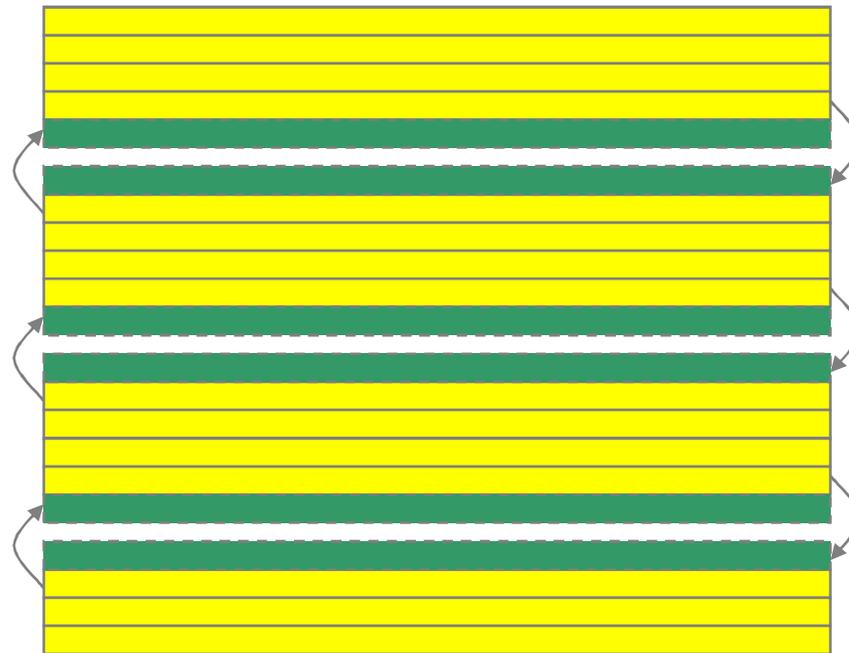
## Step 3: Define the Local Arrays

- Correspondence between the local and global array
- “Global” array is an abstraction
  - There is no one global array allocated anywhere
- Instead, we compute parts of it (the local arrays) on each process
- Provide ways to output the global array by combining the values on each process (parallel I/O!)



## Boundary Regions

- In order to calculate next state of cells in edge rows, need data from adjacent rows
- Need to communicate these regions at each step



# Building an I/O API for Game of Life

## Supporting Checkpoint/Restart

- For long-running applications, the cautious user checkpoints
- Application-level checkpoint involves the application saving its own state
  - Portable!
- A canonical representation is preferred
  - Independent of number of processes
- Restarting is then possible
  - Canonical representation aids restarting with a different number of processes
- Also eases data analysis (when using same output)



# Defining a Checkpoint

- Need enough to restart
  - Header information
    - Size of problem (e.g. matrix dimensions)
    - Description of environment (e.g. input parameters)
  - Program state
    - Should represent the global (canonical) view of the data
- Ideally stored in a convenient container
  - Single file!
- If all processes checkpoint at once, naturally a parallel, collective operation



# Life Checkpoint/Restart API

- Define an interface for checkpoint/restart for the row-block distributed Life code
- Five functions:
  - MLIFEIO\_Init
  - MLIFEIO\_Finalize
  - MLIFEIO\_Checkpoint
  - MLIFEIO\_Can\_restart
  - MLIFEIO\_Restart
- All functions are collective
  - i.e., all processes must make the call
- We can implement API for different back-end formats



# Life Checkpoint

```
▪ MLIFEIO_Checkpoint(char    *prefix,  
                      int     **matrix,  
                      int     rows,  
                      int     cols,  
                      int     iter,  
                      MPI_Info info);
```

- Prefix is used to set filename
- Matrix is a reference to the data to store
- Rows, cols, and iter describe the data (header)
- Info is used for tuning purposes



# Life stdout “checkpoint”

- The first implementation is one that simply prints out the “checkpoint” in an easy-to-read format
- MPI standard does not specify that all stdout will be collected in any particular way
  - Pass data back to rank 0 for printing
  - Portable!
  - Not scalable, but ok for the purpose of stdio

```
# Iteration 9
1:  **      **      **      ** *
2:  * **    * *    * *    ***** * *   *** **
3:  **      **      **      * * * * *   **
4:          **      * * * * * * * *
5:          * * **   ** * * * * * * *
6:          *  * **   *  * * * *
7:          ***      *  ** *   ***
8:          *** * ** ***   *  * ***** *** **
9:          *** *      * ** *   *** **
10:  * * * *      *      *      * * * *
11:  *  **      **      **      * *
12:          **      *****   * ** * * * *
13:          **      *** * **   *  *** *
14:          *  ** *   *  * * * *
15:          ** **   *****   *  * *
16:          ****   *****   *  * *
17:  ***   *** *   ***   ****
18:  ***   ** **
19:  *  **      **      *      **      *
20: *  *      *  * **   **      ***
21: * * * **   * * * * *   ***      * * **
22: * * **   *  ***** *   **      *  *   *** **
23: *  **      ** ***** ***   *      * * ** *
24:          ***   *  *   **      *  *****
25:          ***      **      ****
```



# stdio Life Checkpoint Code Walkthrough

- Points to observe:
  - All processes call checkpoint routine
    - Collective I/O from the viewpoint of the program
  - Interface describes the global array
  - Output is independent of the number of processes

See `mlife-io-stdout.c` pp. 1-3 for code example.



```
1: /* SLIDE: stdio Life Checkpoint Code Walkthrough */
2: /* -*- Mode: C; c-basic-offset:4 ; -*- */
3: /*
4:  * (C) 2004 by University of Chicago.
5:  * See COPYRIGHT in top-level directory.
6:  */
7:
8: #include <stdio.h>
9: #include <stdlib.h>
10: #include <unistd.h>
11:
12: #include <mpi.h>
13:
14: #include "mlife.h"
15: #include "mlife-io.h"
16:
17: /* stdout implementation of checkpoint (no restart) for MPI Life
18:  *
19:  * Data output in matrix order: spaces represent dead cells,
20:  * '*'s represent live ones.
21:  */
22: static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
23:                                       int cols,
24:                                       MPI_Datatype *newtype);
25: static void MLIFEIO_Row_print(int *data, int cols, int rownr);
26: static void MLIFEIO_msleep(int msec);
27:
28: static MPI_Comm mlifeio_comm = MPI_COMM_NULL;
```



```
29: /* SLIDE: stdio Life Checkpoint Code Walkthrough */
30: int MLIFEIO_Init(MPI_Comm comm)
31: {
32:     int err;
33:
34:     err = MPI_Comm_dup(comm, &mlifeio_comm);
35:
36:     return err;
37: }
38:
39: int MLIFEIO_Finalize(void)
40: {
41:     int err;
42:
43:     err = MPI_Comm_free(&mlifeio_comm);
44:
45:     return err;
46: }
```



```
47: /* SLIDE: Life stdout "checkpoint" */
48: /* MLIFEIO_Checkpoint
49:  *
50:  * Parameters:
51:  * prefix - prefix of file to hold checkpoint (ignored)
52:  * matrix - data values
53:  * rows   - number of rows in matrix
54:  * cols   - number of columns in matrix
55:  * iter   - iteration number of checkpoint
56:  * info   - hints for I/O (ignored)
57:  *
58:  * Returns MPI_SUCCESS on success, MPI error code on error.
59:  */
60: int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
61:                       int cols, int iter, MPI_Info info)
62: {
63:     int err = MPI_SUCCESS, rank, nprocs, myrows, myoffset;
64:     MPI_Datatype type;
65:
66:     MPI_Comm_size(mlifeio_comm, &nprocs);
67:     MPI_Comm_rank(mlifeio_comm, &rank);
68:
69:     myrows    = MLIFE_myrows(rows, rank, nprocs);
70:     myoffset  = MLIFE_myrowoffset(rows, rank, nprocs);
71:
```



```
72: /* SLIDE: Describing Data */
73:     if (rank != 0) {
74:         /* send all data to rank 0 */
75:
76:         MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
77:         MPI_Type_commit(&type);
78:         err = MPI_Send(MPI_BOTTOM, 1, type, 0, 1, mlifeio_comm);
79:         MPI_Type_free(&type);
80:     }
81:     else {
82:         int i, procrows, totrows;
83:
84:         printf("\033[H\033[2J# Iteration %d\n", iter);
85:
86:         /* print rank 0 data first */
87:         for (i=1; i < myrows+1; i++) {
88:             MLIFEIO_Row_print(&matrix[i][1], cols, i);
89:         }
90:         totrows = myrows;
91:
```



```
92:  /* SLIDE: Describing Data */
93:      /* receive and print others' data */
94:      for (i=1; i < nprocs; i++) {
95:          int j, *data;
96:
97:          procrows = MLIFE_myrows(rows, i, nprocs);
98:          data = (int *) malloc(procrows * cols * sizeof(int));
99:
100:         err = MPI_Recv(data, procrows * cols, MPI_INT, i, 1,
101:                        mlifeio_comm, MPI_STATUS_IGNORE);
102:
103:         for (j=0; j < procrows; j++) {
104:             MLIFEIO_Row_print(&data[j * cols], cols,
105:                               totrows + j + 1);
106:         }
107:         totrows += procrows;
108:
109:         free(data);
110:     }
111: }
112:
113: MLIFEIO_msleep(250); /* give time to see the results */
114:
115: return err;
116: }
```



# Describing Data



- Lots of rows, all the same size
  - Rows are all allocated as one big block
  - Perfect for MPI\_Type\_vector

```
MPI_Type_vector(count = myrows,  
                blklen = cols, stride = cols+2, MPI_INT, &vectype);
```
  - Second type gets memory offset right (allowing use of MPI\_BOTTOM in MPI\_File\_write\_all)

```
MPI_Type_hindexed(count = 1, len = 1,  
                  disp = &matrix[1][1], vectype, &type);
```

See `mlife-io-stdout.c` pp. 4-6 for code example.

```
117: /* SLIDE: Describing Data */
118: /* MLIFEIO_Type_create_rowblk
119:  *
120:  * Creates a MPI_Datatype describing the block of rows of data
121:  * for the local process, not including the surrounding boundary
122:  * cells.
123:  *
124:  * Note: This implementation assumes that the data for matrix is
125:  *       allocated as one large contiguous block!
126:  */
127: static int MLIFEIO_Type_create_rowblk (int **matrix, int myrows,
128:                                       int cols,
129:                                       MPI_Datatype *newtype)
130: {
131:     int err, len;
132:     MPI_Datatype vectype;
133:     MPI_Aint disp;
134:
135:     /* since our data is in one block, access is very regular! */
136:     err = MPI_Type_vector(myrows, cols, cols+2, MPI_INT,
137:                          &vectype);
138:     if (err != MPI_SUCCESS) return err;
139:
140:     /* wrap the vector in a type starting at the right offset */
141:     len = 1;
142:     MPI_Address(&matrix[1][1], &disp);
143:     err = MPI_Type_hindexed(1, &len, &disp, vectype, newtype);
144:
145:     MPI_Type_free(&vectype); /* decrement reference count */
```



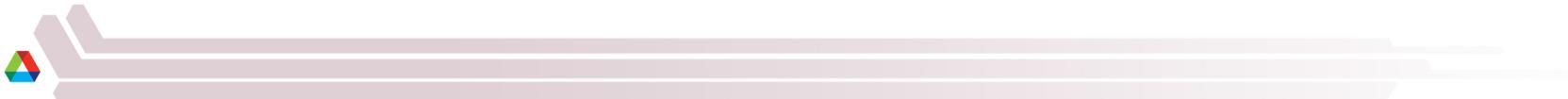
```
146:
147:     return err;
148: }
149:
150: static void MLIFEIO_Row_print(int *data, int cols, int rownr)
151: {
152:     int i;
153:
154:     printf("%3d: ", rownr);
155:     for (i=0; i < cols; i++) {
156:         printf("%c", (data[i] == BORN) ? '*' : ' ');
157:     }
158:     printf("\n");
159: }
160:
161: int MLIFEIO_Can_restart(void)
162: {
163:     return 0;
164: }
165:
166: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
167:                    int cols, int iter, MPI_Info info)
168: {
169:     return MPI_ERR_IO;
170: }
```



# Parallelizing our I/O API

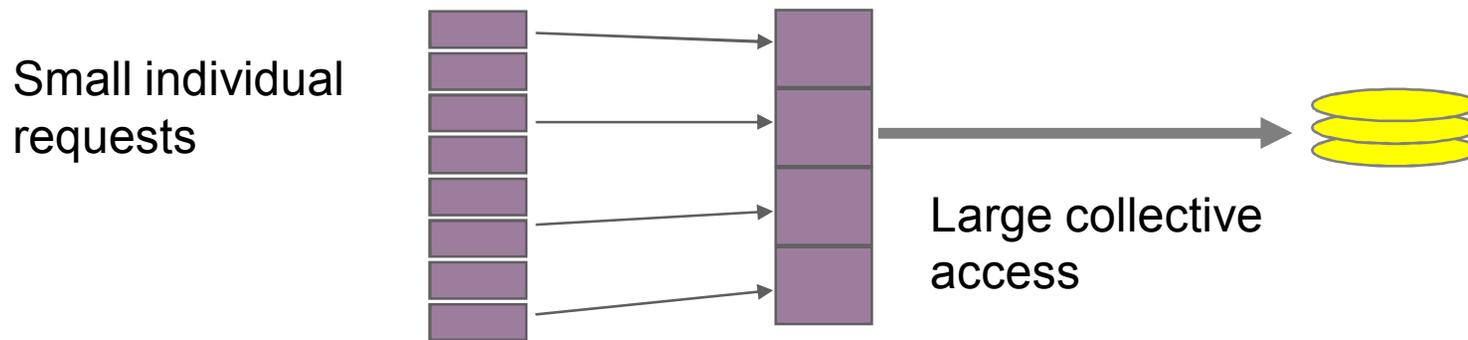
## Parallel I/O and MPI

- The stdio checkpoint routine works but is not parallel
  - One process is responsible for all I/O
  - Wouldn't want to use this approach for real
- How can we get the full benefit of a parallel file system?
  - We first look at how parallel I/O works in MPI
  - We then implement a fully parallel checkpoint routine
- MPI is a good setting for parallel I/O
  - Writing is like sending and reading is like receiving
  - Any parallel I/O system will need:
    - collective operations
    - user-defined datatypes to describe both memory and file layout
    - communicators to separate application-level message passing from I/O-related message passing
    - non-blocking operations
  - i.e., lots of MPI-like machinery



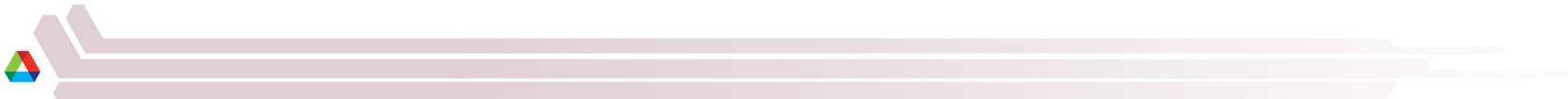
# Collective I/O

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of “big picture” to file system
  - Framework for I/O optimizations at the MPI-IO layer
  - e.g., two-phase I/O



## Collective MPI I/O Functions

- Not going to go through the MPI-IO API in excruciating detail
  - Can talk during hands-on
- **MPI\_File\_write\_at\_all**, etc.
  - **\_all** indicates that all processes in the group specified by the communicator passed to MPI\_File\_open will call this function
  - **\_at** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate “seek” call
- Each process specifies only its own access information
  - the argument list is the same as for the non-collective functions



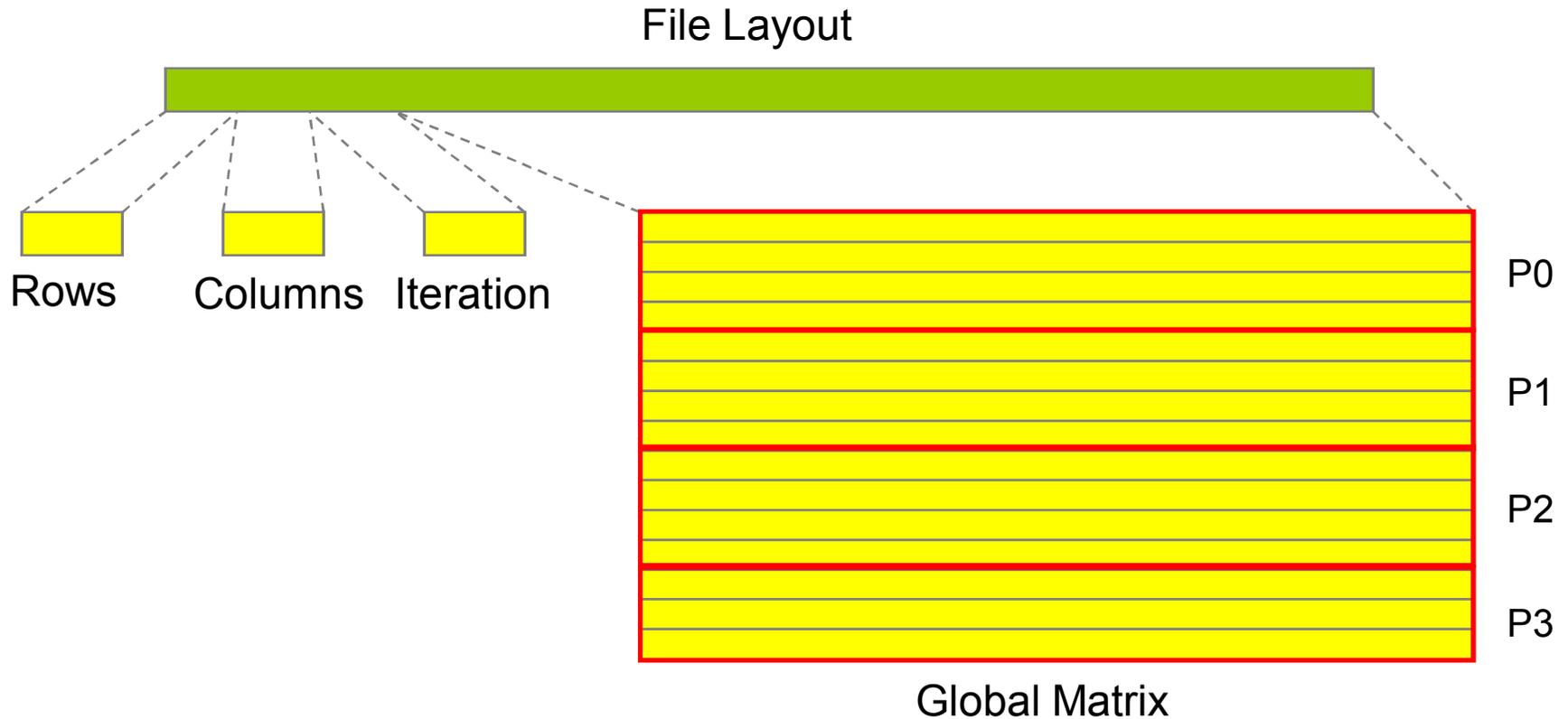
# MPI-IO Life Checkpoint Code Walkthrough

- Points to observe:
  - Use of a user-defined MPI datatype to handle the local array
  - Use of MPI\_Offset for the offset into the file
    - “Automatically” supports files larger than 2GB if the underlying file system supports large files
  - Collective I/O calls
    - Extra data on process 0

See `mlife-io-mpio.c` pp. 1-2 for code example.



# Data Layout in MPI-IO Checkpoint File



Note: We store the matrix in global, canonical order with no ghost cells.

See `mlife-io-mpio.c` pp. 1-9 for code example.



## Life MPI-IO Checkpoint/Restart

- We can map our collective checkpoint directly to a single collective MPI-IO file write: `MPI_File_write_at_all`
  - Process 0 writes a little extra (the header)
- On restart, two steps are performed:
  - Everyone reads the number of rows and columns from the header in the file with `MPI_File_read_at_all`
    - Sometimes faster to read individually and bcast (see later example)
  - If they match those in current run, a second collective call used to read the actual data
    - Number of processors can be different

See `mlife-io-mpio.c` pp. 3-6 for code example.



```
56: /* SLIDE: Life MPI-IO Checkpoint/Restart */
57: int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
58:                       int cols, int iter, MPI_Info info)
59: {
60:     int err;
61:     int amode = MPI_MODE_WRONLY | MPI_MODE_CREATE |
62:               MPI_MODE_UNIQUE_OPEN;
63:     int rank, nprocs;
64:     int myrows, myoffset;
65:
66:     MPI_File fh;
67:     MPI_Datatype type;
68:     MPI_Offset myfileoffset;
69:     char filename[64];
70:
71:     MPI_Comm_size(mlifeio_comm, &nprocs);
72:     MPI_Comm_rank(mlifeio_comm, &rank);
73:
74:     myrows = MLIFE_myrows(rows, rank, nprocs);
75:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
76:
77:     snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
78:     err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
79:     if (err != MPI_SUCCESS) {
80:         fprintf(stderr, "Error opening %s.\n", filename);
81:         return err;
82:     }
83:
84:
```



```
85:  /* SLIDE: Life MPI-IO Checkpoint/Restart */
86:  if (rank == 0) {
87:      MLIFEIO_Type_create_hdr_rowblk(matrix, myrows, &rows,
88:                                     &cols, &iter, &type);
89:      myfileoffset = 0;
90:  }
91:  else {
92:      MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
93:      myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
94:  }
95:
96:  MPI_Type_commit(&type);
97:  err = MPI_File_write_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
98:                              type, MPI_STATUS_IGNORE);
99:  MPI_Type_free(&type);
100:
101:  err = MPI_File_close(&fh);
102:  return err;
103: }
104:
```



```
105: /* SLIDE: Life MPI-IO Checkpoint/Restart */
106: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
107:                   int cols, int iter, MPI_Info info)
108: {
109:     int err, gErr;
110:     int amode = MPI_MODE_RDONLY | MPI_MODE_UNIQUE_OPEN;
111:     int rank, nprocs;
112:     int myrows, myoffset;
113:     int buf[3]; /* rows, cols, iteration */
114:
115:     MPI_File fh;
116:     MPI_Datatype type;
117:     MPI_Offset myfileoffset;
118:     char filename[64];
119:
120:     MPI_Comm_size(mlifeio_comm, &nprocs);
121:     MPI_Comm_rank(mlifeio_comm, &rank);
122:
123:     myrows = MLIFE_myrows(rows, rank, nprocs);
124:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
125:
126:     snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
127:     err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
128:     if (err != MPI_SUCCESS) return err;
129:
130:     /* check that rows and cols match */
131:     err = MPI_File_read_at_all(fh, 0, buf, 3, MPI_INT,
132:                               MPI_STATUS_IGNORE);
133:
```



```
134: /* SLIDE: Life MPI-IO Checkpoint/Restart */
135: /* Have all process check that nothing went wrong */
136: MPI_Allreduce(&err, &gErr, 1, MPI_INT, MPI_MAX, mlifeio_comm);
137: if (gErr || buf[0] != rows || buf[1] != cols) {
138:     if (rank == 0) fprintf(stderr, "restart failed.\n");
139:     return MPI_ERR_OTHER;
140: }
141:
142: MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
143: myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
144:
145: MPI_Type_commit(&type);
146: err = MPI_File_read_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
147:                             type, MPI_STATUS_IGNORE);
148: MPI_Type_free(&type);
149:
150: err = MPI_File_close(&fh);
151: return err;
152: }
153:
```



## Describing Header and Data

- Data is described just as before
- Create a struct wrapped around this to describe the header as well:
  - no. of rows
  - no. of columns
  - Iteration no.
  - data (using previous type)

See `mlife-io-mpio.c` pp. 7 for code example.



```
154: /* SLIDE: Describing Header and Data */
155: /* MLIFEIO_Type_create_hdr_rowblk
156:  *
157:  * Used by process zero to create a type that describes both
158:  * the header data for a checkpoint and its contribution to
159:  * the stored matrix.
160:  *
161:  * Parameters:
162:  * matrix - pointer to the matrix, including boundaries
163:  * myrows - number of rows held locally
164:  * rows_p - pointer to # of rows in matrix (so we can get its
165:  *         address for use in the type description)
166:  * cols_p - pointer to # of cols in matrix
167:  * iter_p - pointer to iteration #
168:  * newtype - pointer to location to store new type ref.
169:  */
170: static int MLIFEIO_Type_create_hdr_rowblk (int **matrix,
171:                                           int myrows,
172:                                           int *rows_p,
173:                                           int *cols_p,
174:                                           int *iter_p,
175:                                           MPI_Datatype *newtype)
176: {
177:     int err;
178:     int lens[4] = { 1, 1, 1, 1 };
179:     MPI_Aint disps[4];
180:     MPI_Datatype types[4];
181:     MPI_Datatype rowblk;
182:
```

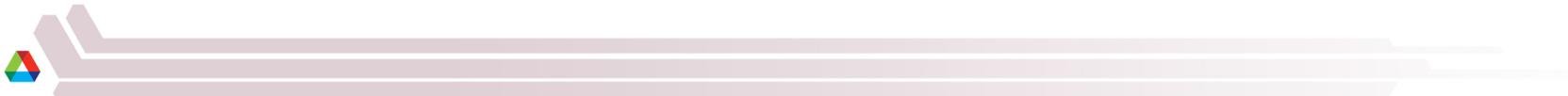


```
183: /* SLIDE: Describing Header and Data */
184:     MLIFEIO_Type_create_rowblk(matrix, myrows, *cols_p, &rowblk);
185:
186:     MPI_Address(rows_p, &disps[0]);
187:     MPI_Address(cols_p, &disps[1]);
188:     MPI_Address(iter_p, &disps[2]);
189:     disps[3] = (MPI_Aint) MPI_BOTTOM;
190:     types[0] = MPI_INT;
191:     types[1] = MPI_INT;
192:     types[2] = MPI_INT;
193:     types[3] = rowblk;
194:
195: #if defined(MPI_VERSION) && MPI_VERSION >= 2
196:     err = MPI_Type_create_struct(3, lens, disps, types, newtype);
197: #else
198:     err = MPI_Type_struct(3, lens, disps, types, newtype);
199: #endif
200:
201:     MPI_Type_free(&rowblk);
202:
203:     return err;
204: }
205:
```



## MPI-IO Takeaway

- Sometimes it makes sense to build a custom library that uses MPI-IO (or maybe even MPI + POSIX) to write a custom format
  - e.g., a data format for your domain already exists, need parallel API
- We've only touched on the API here
  - There is support for data that is noncontiguous in file and memory
  - There are independent calls that allow processes to operate without coordination
- In general we suggest using data model libraries
  - They do more for you
  - Performance can be competitive



# Using Data Model Libraries: A Parallel netCDF Example

# HPC I/O Software Stack

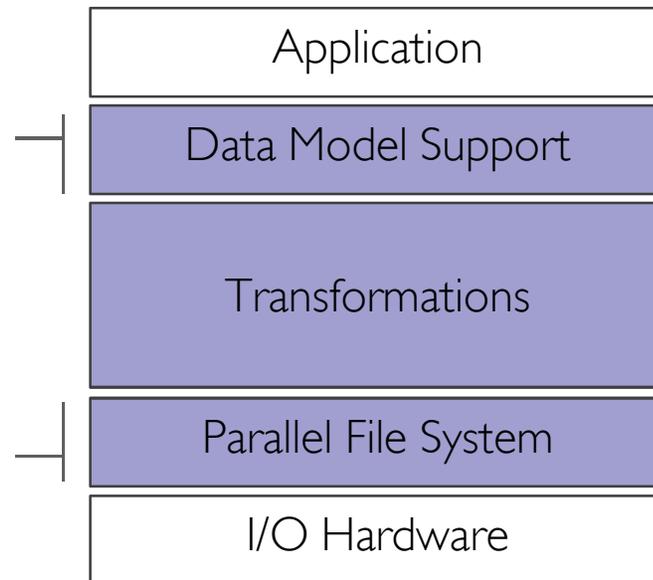
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack*.

**Data Model Libraries** map application abstractions onto storage abstractions and provide data portability.

*HDF5, Parallel netCDF, ADIOS*

**Parallel file system** maintains logical file model and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*



**I/O Middleware** organizes accesses from many processes, especially those using collective I/O.

*MPI-IO, GLEAN, PLFS*

**I/O Forwarding** transforms I/O from many clients into fewer, larger request; reduces lock contention; and bridges between the HPC system and external storage.

*IBM ciod, IOFSL, Cray DVS*

## Data Model Libraries

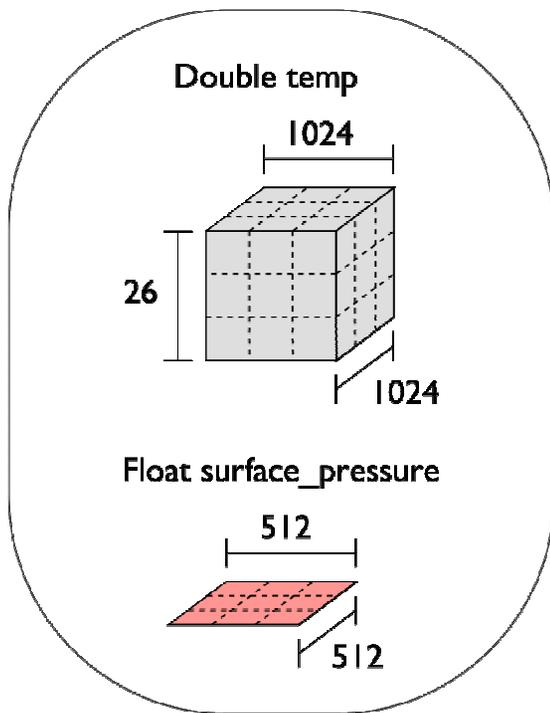
- Scientific applications work with structured data and desire more self-describing file formats
- PnetCDF and HDF5 are two popular “higher level” I/O libraries
  - Abstract away details of file layout
  - Provide standard, portable file formats
  - Include metadata describing contents
- For parallel machines, these use MPI and probably MPI-IO
  - MPI-IO implementations are sometimes poor on specific platforms, in which case libraries might directly call POSIX calls instead



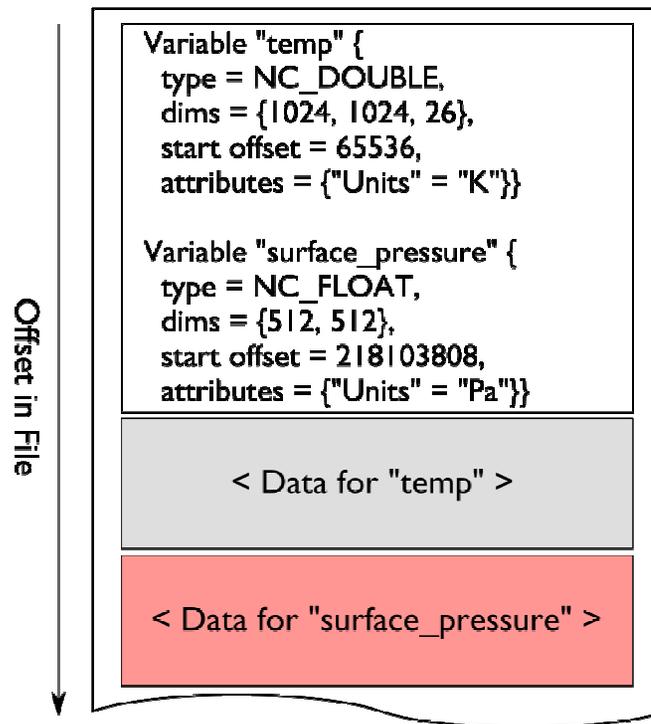
# netCDF Data Model

The netCDF model provides a means for storing multiple, multi-dimensional arrays in a single file.

Application Data Structures



netCDF File "checkpoint07.nc"



netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

Data for variables is stored in contiguous blocks, encoded in a portable binary format according to the variable's type.

# Parallel netCDF (PnetCDF)

- (Serial) netCDF
  - API for accessing multi-dimensional data sets
  - Portable file format
  - Popular in both fusion and climate communities
- Parallel netCDF
  - Very similar API to netCDF
  - Tuned for better performance in today's computing environments
  - Retains the file format so netCDF and PnetCDF applications can share files
  - PnetCDF builds on top of any MPI-IO implementation

Cluster



IBM Blue Gene



# PnetCDF Life Checkpoint/Restart Code Walkthrough

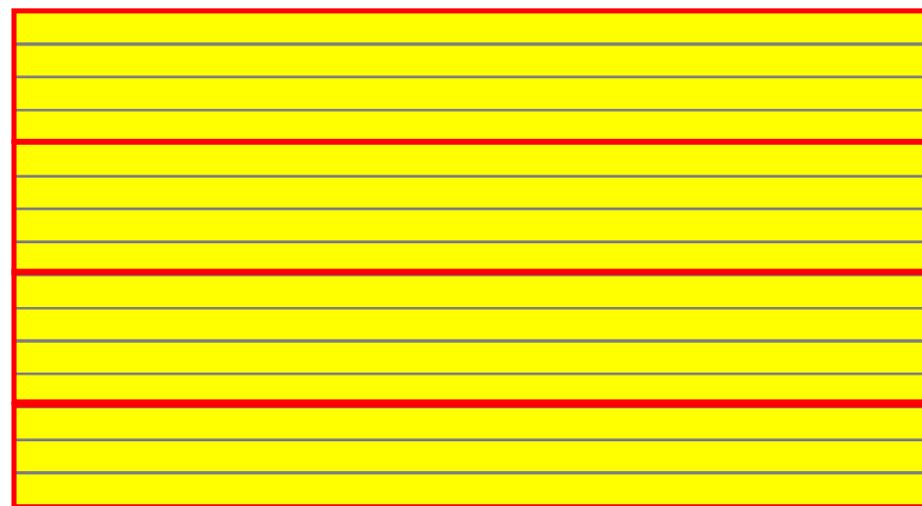
- Stores matrix as a two-dimensional array of integers
  - Same canonical ordering as in MPI-IO version
- Iteration number stored as an attribute

integer iter

integer "matrix" [rows][cols]



Iteration



P0

P1

P2

P3

Global Matrix

See `mlife-io-pnetcdf.c` pp. 1-5 for code example.



```
45: /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
46: int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
47:                       int cols, int iter, MPI_Info info)
48: {
49:     int err;
50:     int cmode = 0;
51:     int rank, nprocs;
52:     int myrows, myoffset;
53:
54:     int ncid, varid, coldim, rowdim, dims[2];
55:     MPI_Offset start[2];
56:     MPI_Offset count[2];
57:     int i, j, *buf;
58:     char filename[64];
59:
60:     MPI_Comm_size(mlifeio_comm, &nprocs);
61:     MPI_Comm_rank(mlifeio_comm, &rank);
62:
63:     myrows = MLIFE_myrows(rows, rank, nprocs);
64:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
65:
66:     snprintf(filename, 63, "%s-%d.nc", prefix, iter);
67:
68:     err = ncmpi_create(mlifeio_comm, filename, cmode, info, &ncid);
69:     if (err != 0) {
70:         fprintf(stderr, "Error opening %s.\n", filename);
71:         return MPI_ERR_IO;
72:     }
73:
```



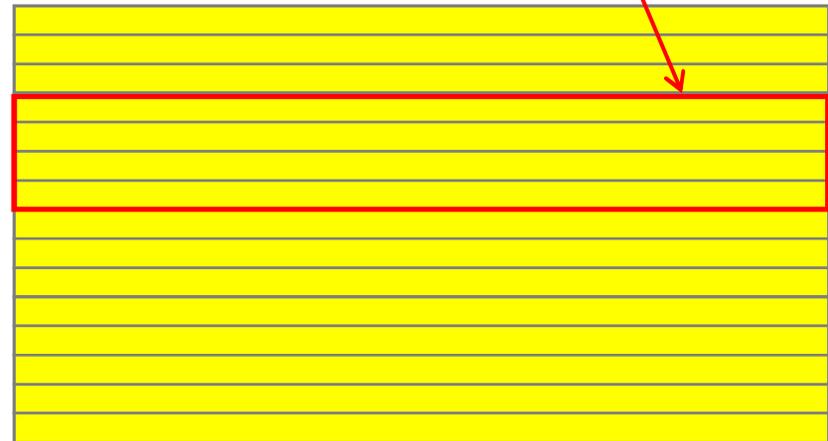
# Describing Subarray Access in PnetCDF

- PnetCDF provides calls for reading/writing subarrays in a single (collective) call:

```
ncmpi_put_vara_all(ncid,  
varid,  
start[], count[],  
buf, count,  
datatype)
```



Local Sub-matrix  
in memory



Global Matrix in PnetCDF File

P  
1

```
74:  /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
75:  ncmpi_def_dim(ncid, "col", cols, &coldim);
76:  ncmpi_def_dim(ncid, "row", rows, &rowdim);
77:  dims[0] = coldim;
78:  dims[1] = rowdim;
79:  ncmpi_def_var(ncid, "matrix", NC_INT, 2, dims, &varid);
80:
81:  /* store iteration as global attribute */
82:  ncmpi_put_att_int(ncid, NC_GLOBAL, "iter", NC_INT, 1, &iter);
83:
84:  ncmpi_enddef(ncid);
85:
86:  start[0] = 0; /* col start */
87:  start[1] = myoffset; /* row start */
88:  count[0] = cols;
89:  count[1] = myrows;
90:
91:  MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
92:  MPI_Type_commit(&type);
93:
94:  ncmpi_put_vara_all(ncid, varid, start, count, MPI_BOTTOM, 1,
95:                   type);
96:
97:  MPI_Type_free(&type);
98:
99:  ncmpi_close(ncid);
100:  return MPI_SUCCESS;
101: }
102:
```



```
103: /* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
104: int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
105:                   int cols, int iter, MPI_Info info)
106: {
107:     int err = MPI_SUCCESS;
108:     int rank, nprocs;
109:     int myrows, myoffset;
110:     int flag;
111:
112:     int cmode = 0;
113:     int ncid, varid, dims[2];
114:     MPI_Offset start[2];
115:     MPI_Offset count[2];
116:     MPI_Offset coldimsz, rowdimsz;
117:     int i, j, *buf;
118:     char filename[64];
119:
120:     MPI_Comm_size(mlifeio_comm, &nprocs);
121:     MPI_Comm_rank(mlifeio_comm, &rank);
122:
123:     myrows = MLIFE_myrows(rows, rank, nprocs);
124:     myoffset = MLIFE_myrowoffset(rows, rank, nprocs);
125:
126:     snprintf(filename, 63, "%s-%d.nc", prefix, iter);
127:     err = ncmpi_open(mlifeio_comm, filename, cmode, info, &ncid);
128:     if (err != 0) {
129:         fprintf(stderr, "Error opening %s.\n", filename);
130:         return MPI_ERR_IO;
131:     }
```



## Discovering Variable Dimensions

- Because netCDF is self-describing, applications can inquire about data in netCDF files:

```
err = ncmpi_inq_dimlen(ncid,  
                      dims[0],  
                      &coldimsz);
```

- Allows us to discover the dimensions of our matrix at restart time

See `mfile-io-pnetcdf.c` pp. 6-7 for code example.



```
132: /* SLIDE: Discovering Variable Dimensions */
133:     err = ncmpi_inq_varid(ncid, "matrix", &varid);
134:     if (err != 0) {
135:         return MPI_ERR_IO;
136:     }
137:
138:     /* verify that dimensions in file are same as input row/col */
139:     err = ncmpi_inq_vardimid(ncid, varid, dims);
140:     if (err != 0) {
141:         return MPI_ERR_IO;
142:     }
143:
144:     err = ncmpi_inq_dimlen(ncid, dims[0], &coldimsz);
145:     if (coldimsz != cols) {
146:         fprintf(stderr, "cols does not match\n");
147:         return MPI_ERR_IO;
148:     }
149:
150:     err = ncmpi_inq_dimlen(ncid, dims[1], &rowdimsz);
151:     if (rowdimsz != rows) {
152:         fprintf(stderr, "rows does not match\n");
153:         return MPI_ERR_IO;
154:     }
155:
```



```
156:  /* SLIDE: Discovering Variable Dimensions */
157:  buf = (int *) malloc(myrows * cols * sizeof(int));
158:  flag = (buf == NULL);
159:  /* See if any process failed to allocate memory */
160:  MPI_Allreduce(MPI_IN_PLACE, &flag, 1, MPI_INT, MPI_LOR,
161:              mlifeio_comm);
162:  if (flag) {
163:      return MPI_ERR_IO;
164:  }
165:
166:  start[0] = 0; /* col start */
167:  start[1] = myoffset; /* row start */
168:  count[0] = cols;
169:  count[1] = myrows;
170:  ncmpi_get_vara_int_all(ncid, varid, start, count, buf);
171:
172:  for (i=0; i < myrows; i++) {
173:      for (j=0; j < cols; j++) {
174:          matrix[i+1][j] = buf[(i*cols) + j];
175:      }
176:  }
177:
178:  free(buf);
179:
180:  return MPI_SUCCESS;
181: }
```



## Takeaway from PnetCDF Game of Life Example

- PnetCDF abstracts away the file system model, giving us something closer to (many) domain models
  - Arrays
  - Types
  - Attributes
- Captures metadata for us (e.g., rows, columns, types) and allows us to programmatically explore datasets
- Uses MPI-IO underneath, takes advantage of data sieving and two-phase I/O when possible
- Next we will spend a bit of time on PnetCDF itself



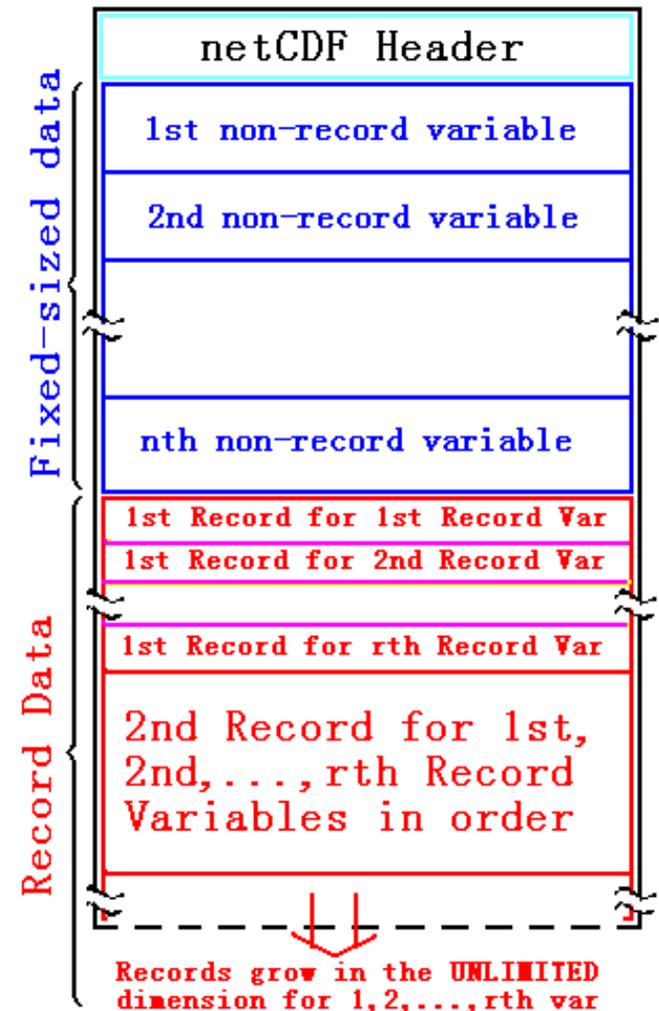
# How It Works: The Parallel netCDF Interface and File Format

Thanks to Wei-Keng Liao, Alok Choudhary, and Kui Gao (NWU) for their help in the development of PnetCDF.

[www.mcs.anl.gov/parallel-netcdf](http://www.mcs.anl.gov/parallel-netcdf)

# Record Variables in netCDF

- Record variables are defined to have a single “unlimited” dimension
  - Convenient when a dimension size is unknown at time of variable creation
- Record variables are stored after all the other variables in an interleaved format
  - Using more than one in a file is likely to result in poor performance due to number of noncontiguous accesses



## Inside PnetCDF Define Mode

- In define mode (collective)
  - Use `MPI_File_open` to create file at create time
  - Set hints as appropriate (more later)
  - Locally cache header information in memory
    - All changes are made to local copies at each process
- At `ncmpi_enddef`
  - Process 0 writes header with `MPI_File_write_at`
  - `MPI_Bcast` result to others
  - Everyone has header data in memory, understands placement of all variables
    - No need for any additional header I/O during data mode!



## Inside PnetCDF Data Mode

### ■ Inside `ncmpi_put_vara_all` (once per variable)

- Each process performs data conversion into internal buffer
- Uses `MPI_File_set_view` to define file region
  - Contiguous region for each process in FLASH case
- `MPI_File_write_all` collectively writes data

### ■ At `ncmpi_close`

- `MPI_File_close` ensures data is written to storage

### ■ MPI-IO performs optimizations

- Two-phase possibly applied when writing variables

### ■ MPI-IO makes PFS calls

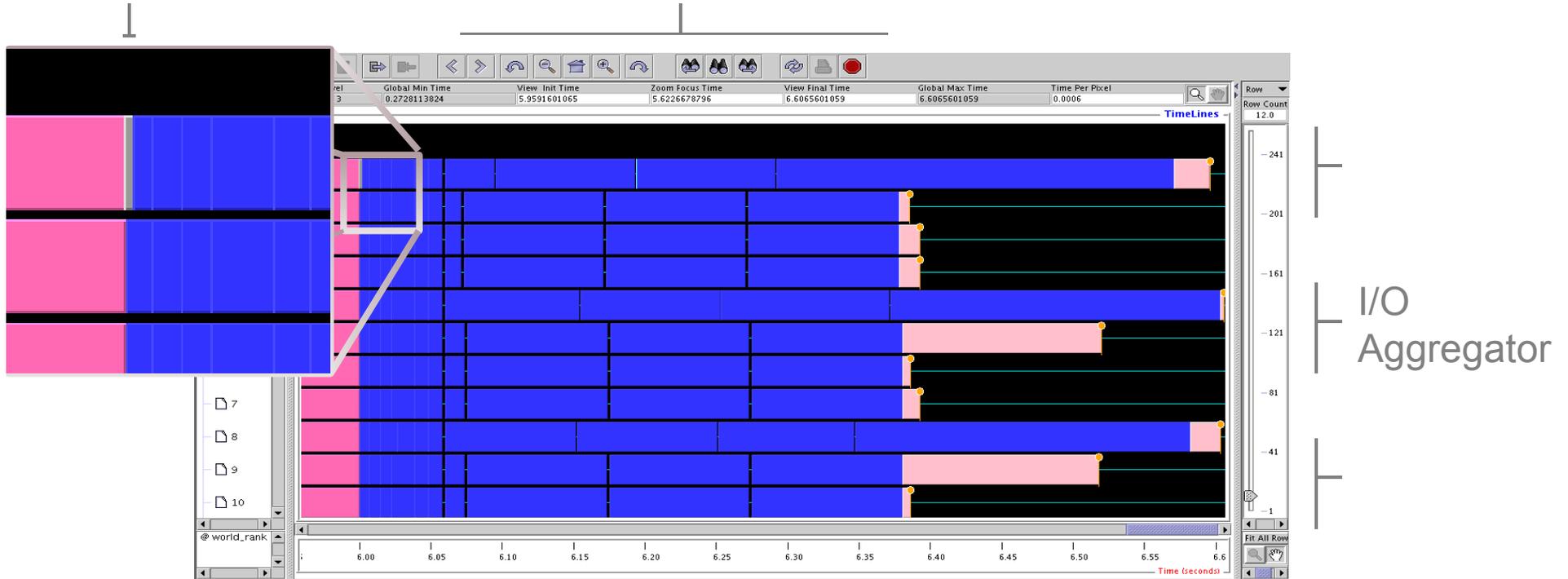
- PFS client code communicates with servers and stores data



# Inside Parallel netCDF: Jumpshot view

1: Rank 0 write header  
(independent I/O)

3: Collectively  
write 4 variables



2: Collectively write  
app grid, AMR data

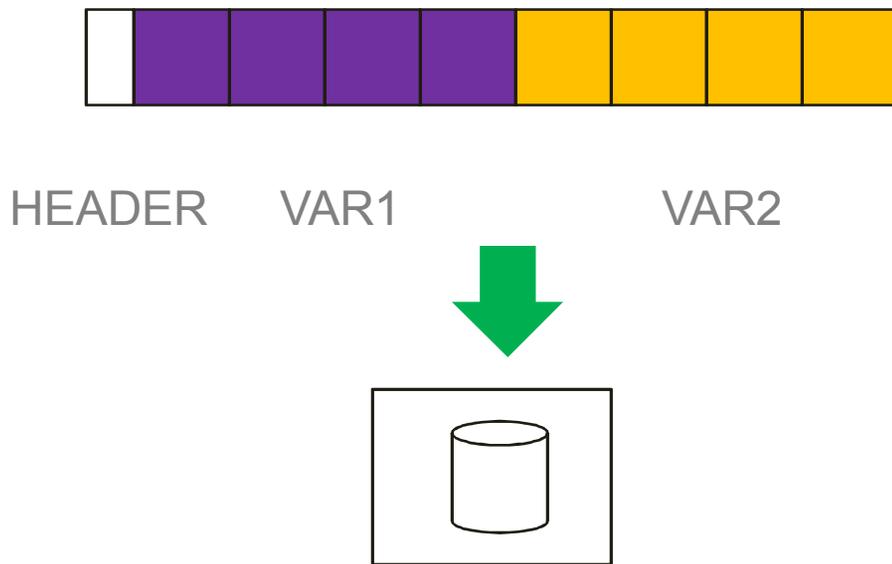
4: Close file

File open
  Indep. write
  Collective write
  File close



# Parallel-NetCDF write-combining optimization

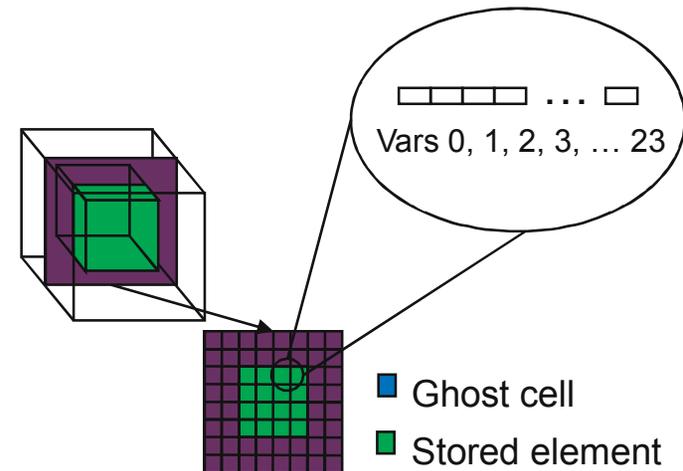
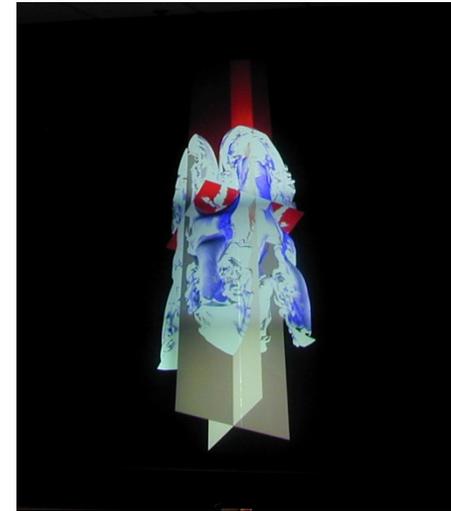
```
ncmpi_iput_vara(ncfile, varid1,  
               &start, &count, &data,  
               count, MPI_INT, &requests[0]);  
ncmpi_iput_vara(ncfile, varid2,  
               &start, &count, &data,  
               count, MPI_INT, &requests[1]);  
ncmpi_wait_all(ncfile, 2, requests, statuses);
```



- netCDF variables laid out contiguously
- Applications typically store data in separate variables
  - temperature(lat, long, elevation)
  - Velocity\_x(x, y, z, timestep)
- Operations posted independently, completed collectively
  - Defer, coalesce synchronization
  - Increase average request size

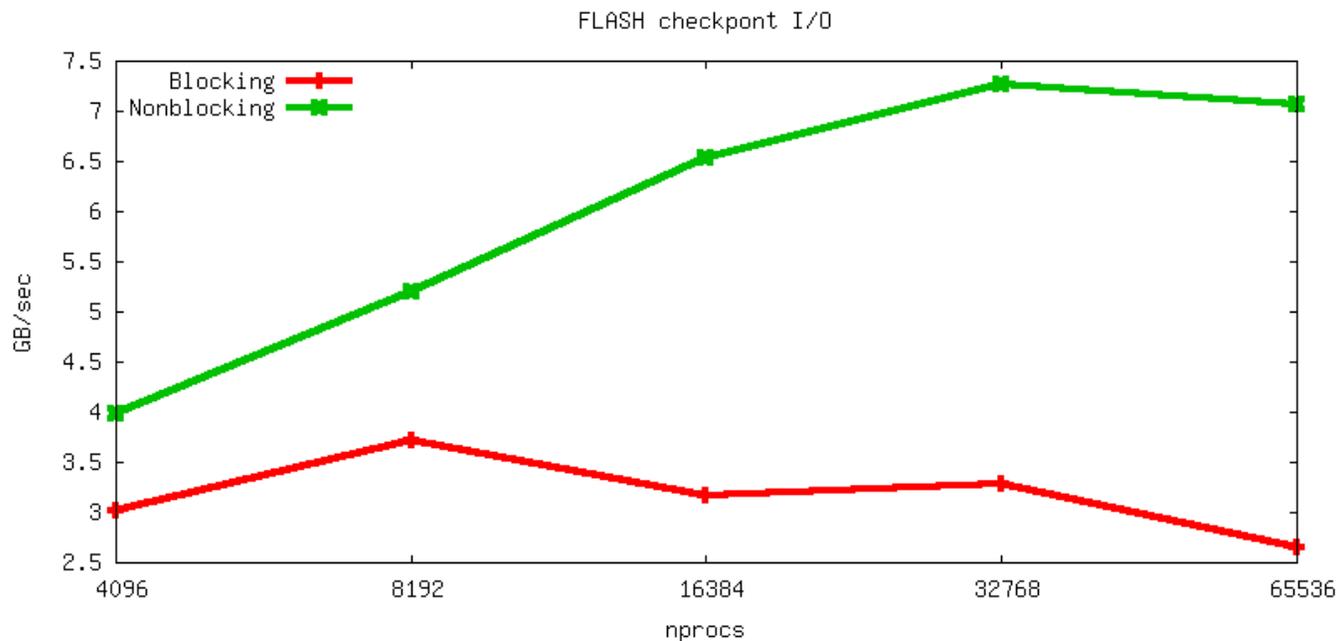
# Example: FLASH Astrophysics

- FLASH is an astrophysics code for studying events such as supernovae
  - Adaptive-mesh hydrodynamics
  - Scales to 1000s of processors
  - MPI for communication
- Frequently checkpoints:
  - Large blocks of typed variables from all processes
  - Portable format
  - Canonical ordering (different than in memory)
  - Skipping ghost cells



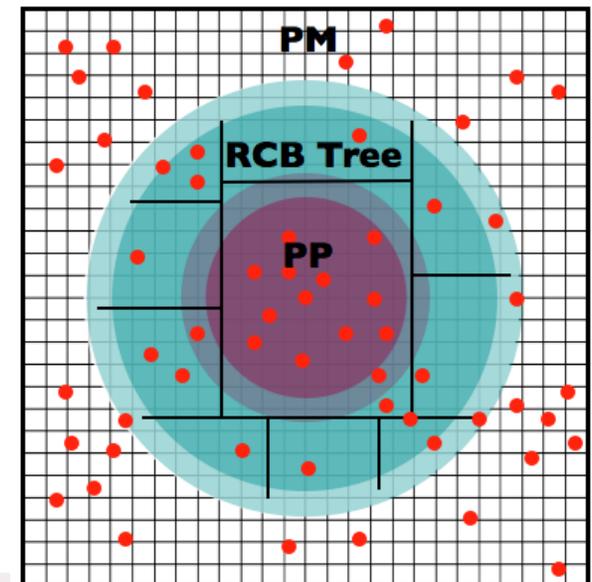
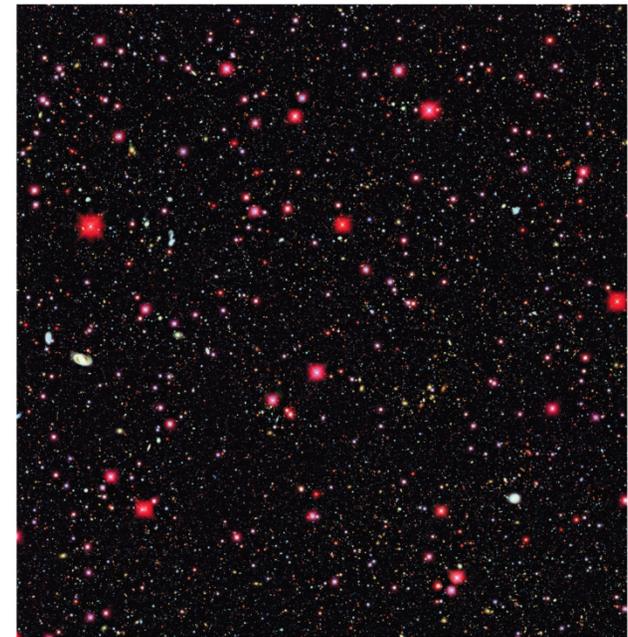
# FLASH Astrophysics and the write-combining optimization

- FLASH writes one variable at a time
- Could combine all 4D variables (temperature, pressure, etc) into one 5D variable
  - Altered file format (conventions) requires updating entire analysis toolchain
- Write-combining provides improved performance with same file conventions
  - Larger requests, less synchronization.



# HACC: understanding cosmos via simulation

- “Cosmology = Physics + Simulation “  
(Salman Habib)
- Sky surveys collecting massive amounts of data
  - (~100 PB)
- Understanding of these massive datasets rests on modeling distribution of cosmic entities
- Seed simulations with initial conditions
- Run for 13 billion (simulated) years
- Comparison with observed data validates physics model.
- I/O challenges:
  - Checkpointing
  - analysis





## HACC particles with pnetcdf: metadata (1/2)

```
/* class constructor creates dataset */
IO::IO(int mode, char *filename, MPI_Comm comm) {
    ncmpi_create(comm, filename, NC_64BIT_DATA,
                MPI_INFO_NULL, &ncfile);
}

/* describe simulation metadata, not pnetcdf metadata */
void IO::WriteMetadata(char *notes, float *block_size,
    float *global_min, int *num_blocks,
    int first_time_step, int last_time_step,
    int this_time_step, int num_secondary_keys,
    char **secondary_keys) {
    ncmpi_put_att_text(ncfile, NC_GLOBAL, "notes",
        strlen(notes), notes);
    ncmpi_put_att_float(ncfile, NC_GLOBAL, "global_min_z",
        NC_FLOAT, 1, &global_min[2]);
}
```



## HACC particles with pnetcdf: metadata (2/2)

```
void IO::DefineDims() {
    ncmpi_def_dim(ncfile, "KeyIndex", key_index,
        &dim_keyindex);
    char str_attribute[100 =
        "num_blocks_x * num_blocks_y * num_blocks_z *
        num_kys";

    /* variable with no dimensions: "scalar" */
    ncmpi_def_var(ncfile, "KeyIndex", NC_INT, 0,
        NULL, &var_keyindex);
    ncmpi_put_att_text(ncfile, var_keyindex, "Key_Index",
        strlen(str_attribute), str_attribute);
    /* pnetcdf knows shape and type, but application must
        annotate with units */
    strcpy(unit, "km/s");
    ncmpi_def_var(ncfile, "Velocity", NC_FLOAT,
        ndims, dimpids, &var_velid);
    ncmpi_put_att_text(ncfile, var_velid,
        "unit_of_velocity", strlen(unit), unit);
}
```



# HACC particles with pnetcdf: data

```
void IO::WriteData(int num_particles, float *xx, float *yy, float
*zz,
                  float *vx, float *vy, float *vz,
                  float *phi, int64_t *pid, float *mins,
                  float *maxs) {
// calculate total number of particles and individual array offsets
nParticles = num_particles; // typecast to MPI_Offset
myOffset    = 0; // particle offset of this process
MPI_Exscan(&nParticles, &myOffset, 1, MPI_OFFSET, MPI_SUM, comm);
MPI_Allreduce(MPI_IN_PLACE, &nParticles, 1, MPI_OFFSET,
MPI_SUM, comm);

start[0] = myOffset; start[1] = 0;
count[0] = num_particles; count[1] = 3; /* ZYX dimensions */

// write "Velocity" in parallel, partitioned
// along dimension nParticles
// "Velocity" is of size nParticles x nDimensions
// data_vel array set up based on method parameters
ncmpi_put_vara_float_all(ncfile, var_valid, start, count,
                        &data_vel[0][0]);
}
```



## Parallel-NetCDF Inquiry routines

- Talked a lot about writing, but what about reading?
- Parallel-NetCDF QuickTutorial contains examples of several approaches to reading and writing
- General approach
  1. Obtain simple counts of entities (similar to MPI datatype “envelope”)
  2. Inquire about length of dimensions
  3. Inquire about type, associated dimensions of variable
- Real application might assume convention, skip some steps
- A full parallel reader would, after determining shape of variables, assign regions of variable to each rank (“decompose”).
  - Next slide focuses only on inquiry routines. (See website for I/O code)



# Parallel NetCDF Inquiry Routines

```
int main(int argc, char **argv) {
    /* extracted from
     * http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/QuickTutorial
     * "Reading Data via standard API" */
    MPI_Init(&argc, &argv);
    ncmpi_open(MPI_COMM_WORLD, argv[1], NC_NOWRITE,
               MPI_INFO_NULL, &ncfile);

    /* reader knows nothing about dataset, but we can interrogate with
     * query routines: ncmpi_inq tells us how many of each kind of
     * "thing" (dimension, variable, attribute) we will find in file */

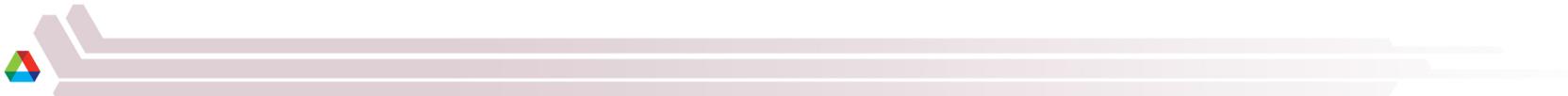
    1 ncmpi_inq(ncfile, &ndims, &nvars, &ngatts, &has_unlimited);
    /* no communication needed after ncmpi_open: all processors have a
     * cached view of the metadata once ncmpi_open returns */

    dim_sizes = calloc(ndims, sizeof(MPI_Offset));
    /* netcdf dimension identifiers are allocated sequentially starting
     * at zero; same for variable identifiers */
    2 for (i=0; i<ndims; i++) {
        ncmpi_inq_dimlen(ncfile, i, &(dim_sizes[i]) );
    }
    3 for (i=0; i<nvars; i++) {
        ncmpi_inq_var(ncfile, i, varname, &type, &var_ndims, dimids,
                     &var_natts);
        printf("variable %d has name %s with %d dimensions"
              " and %d attributes\n",
              i, varname, var_ndims, var_natts);
    }
    ncmpi_close(ncfile);
    MPI_Finalize();
}
```



# PnetCDF Wrap-Up

- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead
- Some limits on (old, common CDF-2) file format:
  - Fixed-size variable: < 4 GiB
  - Per-record size of record variable: < 4 GiB
  - $2^{32} - 1$  records
  - New extended file format to relax these limits (CDF-5, released in pnetcdf-1.1.0, November 2009)



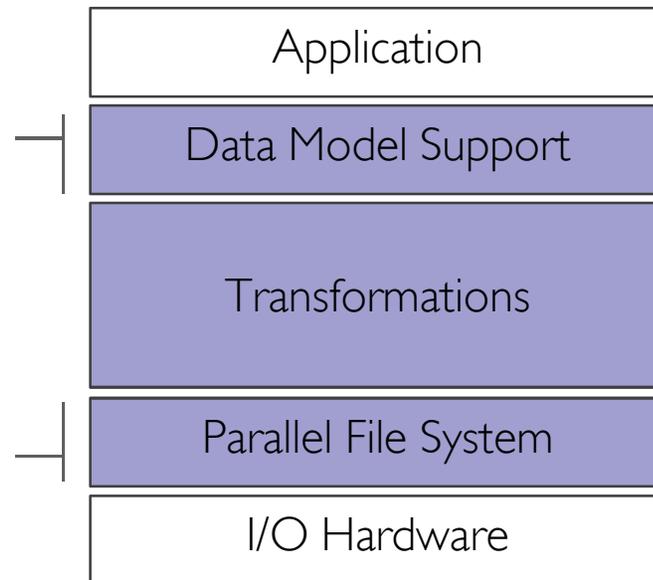
# Additional I/O Interfaces

**Data Model Libraries** map application abstractions onto storage abstractions and provide data portability.

*HDF5, Parallel netCDF, ADIOS*

**Parallel file system** maintains logical file model and provides efficient access to data.

*PVFS, PanFS, GPFS, Lustre*



**I/O Middleware** organizes accesses from many processes, especially those using collective I/O.

*MPI-IO, GLEAN, PLFS*

**I/O Forwarding** transforms I/O from many clients into fewer, larger request; reduces lock contention; and bridges between the HPC system and external storage.

*IBM ciod, IOFSL, Cray DVS*



## Data Model I/O libraries

- Parallel-NetCDF: <http://www.mcs.anl.gov/pnetcdf>
- HDF5: <http://www.hdfgroup.org/HDF5/>
- NetCDF-4: <http://www.unidata.ucar.edu/software/netcdf/netcdf-4/>
  - netCDF API with HDF5 back-end
- ADIOS: <http://adiosapi.org>
  - Configurable (xml) I/O approaches
- SILO: <https://wci.llnl.gov/codes/silo/>
  - A mesh and field library on top of HDF5 (and others)
- H5part: <http://vis.lbl.gov/Research/AcceleratorSAPP/>
  - simplified HDF5 API for particle simulations
- GIO: <https://svn.pnl.gov/gcrm>
  - Targeting geodesic grids as part of GCRM
- PIO:
  - climate-oriented I/O library; supports raw binary, parallel-netcdf, or serial-netcdf (from master)
- ... Many more: consider existing libs before deciding to make your own.



# Understanding I/O Behavior and Performance

Thanks to the following for much of this material:

**Kevin Harms, Charles Bacon, Sam Lang, Bill Allcock**

Math and Computer Science Division and  
Argonne Leadership Computing Facility  
Argonne National Laboratory

**Yushu Yao and Katie Antypas**

National Energy Research Scientific  
Computing Center  
Lawrence Berkeley National Laboratory

For more information, see:

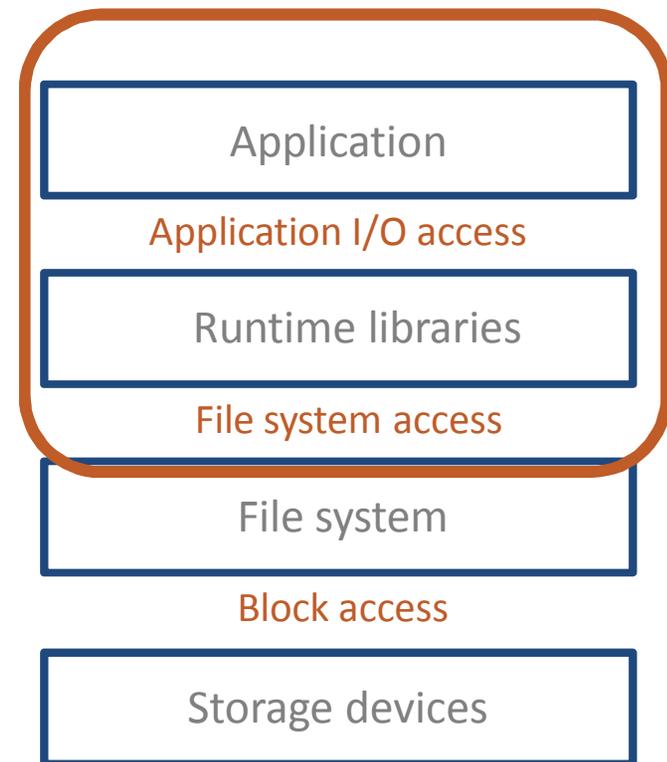
- P. Carns et al. Understanding and improving computational science storage access through continuous characterization. ACM TOS. 2011.
- P. Carns et al. Production I/O characterization on the Cray XE6. CUG 2013. May, 2013.

# Characterizing Application I/O

How are applications using the I/O system, and how successful are they at attaining high performance?

- The best way to answer these questions is by observing behavior at the application and library level
- What did the application intend to do, and how much time did it take to do it?
- In this portion of the training course we will focus on ***Darshan***, a scalable tool for characterizing application I/O activity.

Simplified HPC I/O stack



# What does Darshan do

**Darshan** (Sanskrit for “sight”) is a tool we developed for I/O characterization at extreme scale:

- No code changes, easy to enable
  - Enabled by default at ALCF and NERSC, optionally available at OLCF
- **Negligible performance impact:** just “leave it on”
- Produces a summary of I/O activity for each job
- Captures:
  - Counters for file I/O and MPI-IO calls, some PnetCDF and HDF5 calls
  - Counters for unaligned, sequential, consecutive, and strided access
  - Timing of opens, closes, first and last reads and writes
  - Cumulative data read and written
  - Histograms of access, stride, datatype, and extent sizes



sequential



consecutive

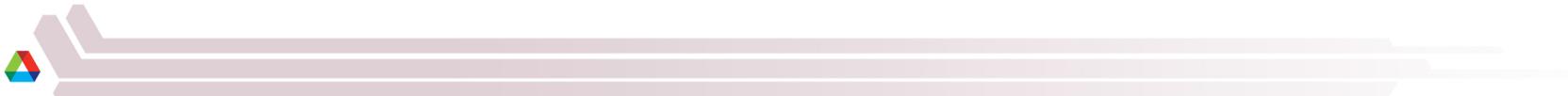


strided



# The technology behind Darshan

- Intercepts I/O functions using link-time wrappers
  - No code modification
  - Can be transparently enabled in MPI compiler scripts
  - Compatible with all major C, C++, and Fortran compilers
- Record statistics independently at each process, for each file
  - Bounded memory consumption
  - Compact summary rather than verbatim record
- Collect, compress, and store results at shutdown time
  - Aggregate shared file data using custom MPI reduction operator
  - Compress remaining data in parallel with zlib
  - Write results with collective MPI-IO
  - Result is a single gzip-compatible file containing characterization information
- Works for Linux clusters, Blue Gene, and Cray systems



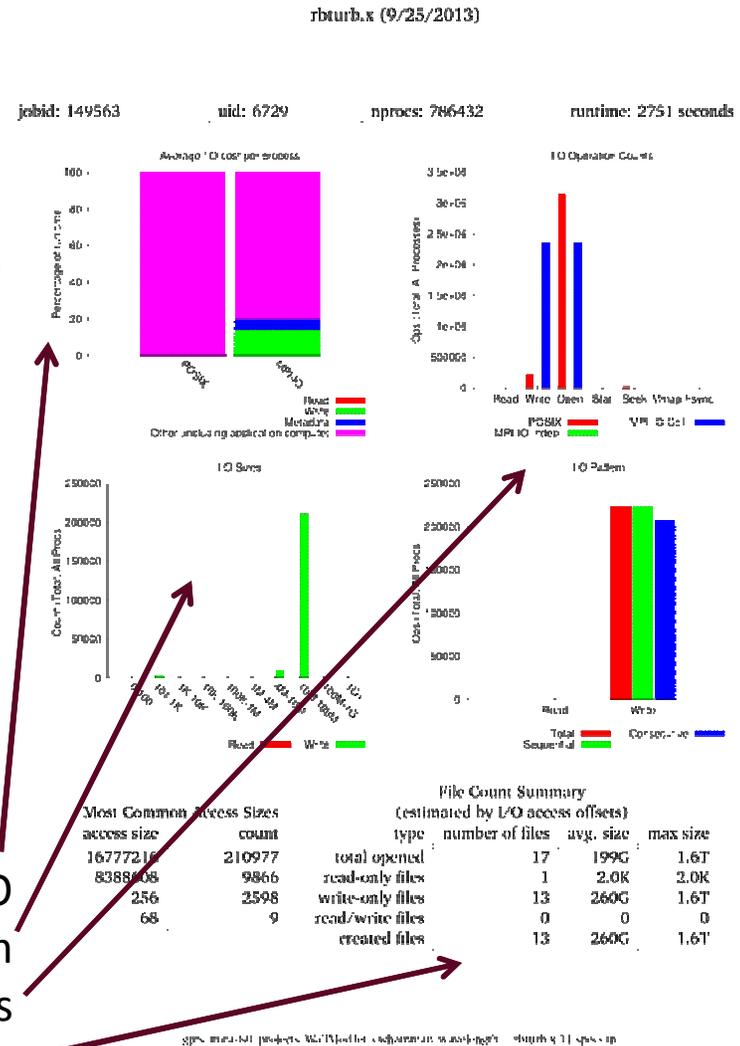
## How to use Darshan

- Compile a C, C++, or FORTRAN program that uses MPI
- Run the application
- Look for the Darshan log file
- This will be in a particular directory (depending on your system's configuration)
  - `<dir>/<year>/<month>/<day>/<username>_<appname>*.darshan.gz`
  - Mira: see `/projects/logs/darshan/`
  - Edison: see `/scratch1/scratchdirs/darshanlogs/`
- Application must run to completion and call `MPI_Finalize()` to generate a log file
- Use Darshan command line tools to analyze the log file
- Warning/disclaimer: Darshan does not currently work for *F90* programs on Mira

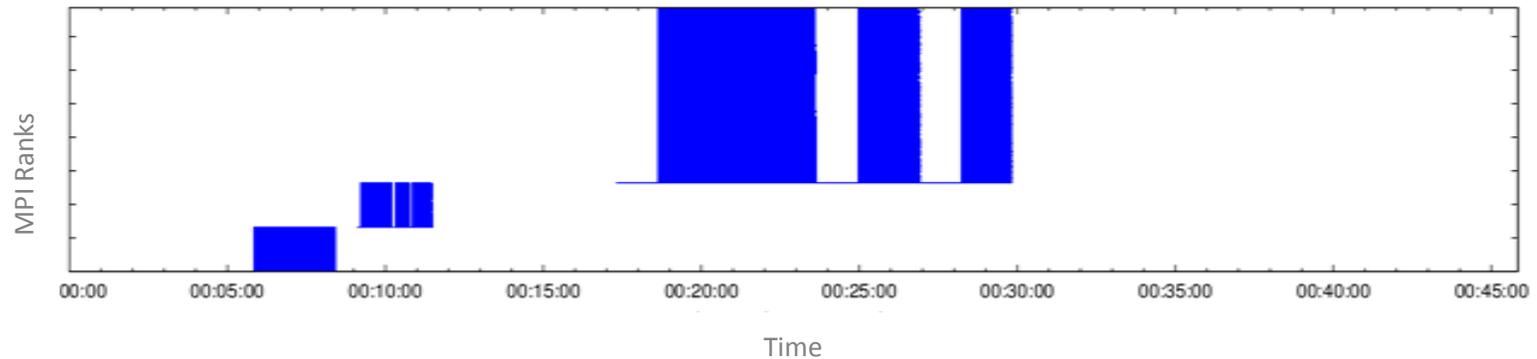


# Darshan analysis example

- Each job instrumented with Darshan produces a single characterization log file
- Darshan command line utilities are used to analyze these log files
- Example: Darshan-job-summary.pl produces a 3-page PDF file summarizing various aspects of I/O performance
- This figure shows the I/O behavior of a 786,432 process turbulence simulation (production run) on the Mira system at ANL
- Application is write intensive and benefits greatly from collective buffering



## Darshan analysis example (page 2)



This graph (and others like it) are on the second page of the [darshan-job-summary.pl](#) output. This example shows intervals of I/O activity from each MPI process.



## Available Darshan analysis tools

- <http://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html>
- Key tools:
  - **Darshan-job-summary.pl**: creates pdf with graphs for initial analysis
  - **Darshan-summary-per-file.sh**: similar to above, but produces a separate pdf summary for every file opened by application
  - **Darshan-parser**: dumps all information into text format

Darshan-parser example (see all counters related to write operations):

```
“darshan-parser user_app_numbers.darshan.gz |grep WRITE”
```

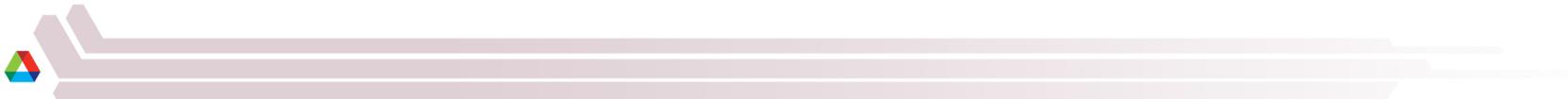
See documentation above for definition of output fields





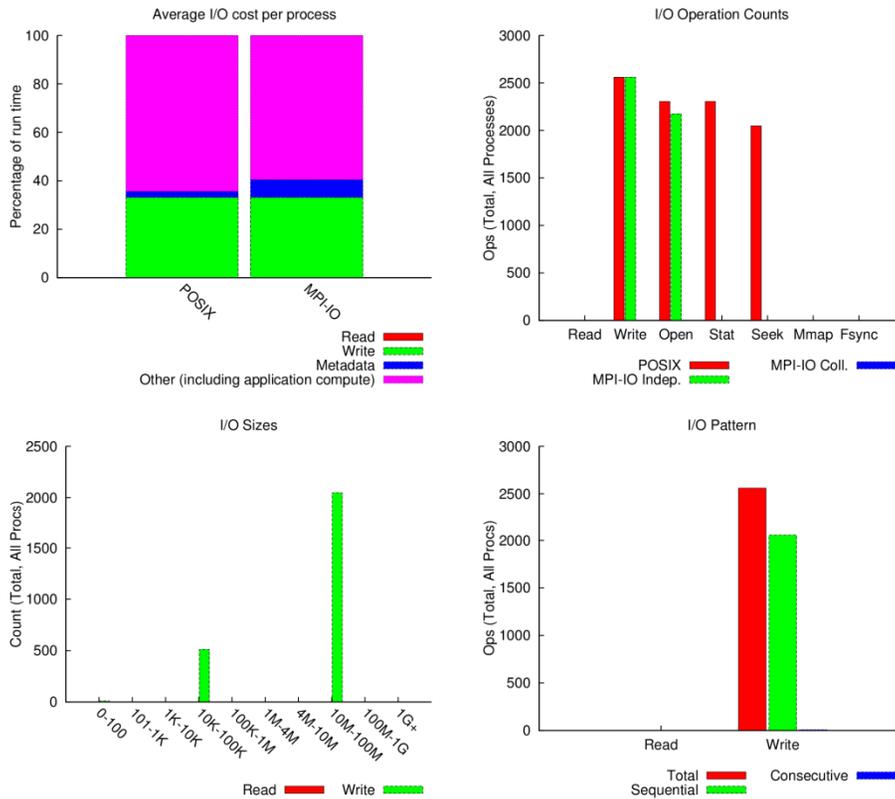
## Looking for I/O Performance Problems

- Many I/O problems can be seen from these logs
- The next few slides will show some examples



# Example: checking user expectations

jobid: | uid: | nprocs: 4096 | runtime: 175 seconds



- User opened 129 files (one “control” file, and 128 data files)
- Should be one header, about 40 KiB, per data file
- This example shows 512 headers being written
- Code bug: header was written 4x per file

access size	count
67108864	2048
41120	512
8	4
4	3

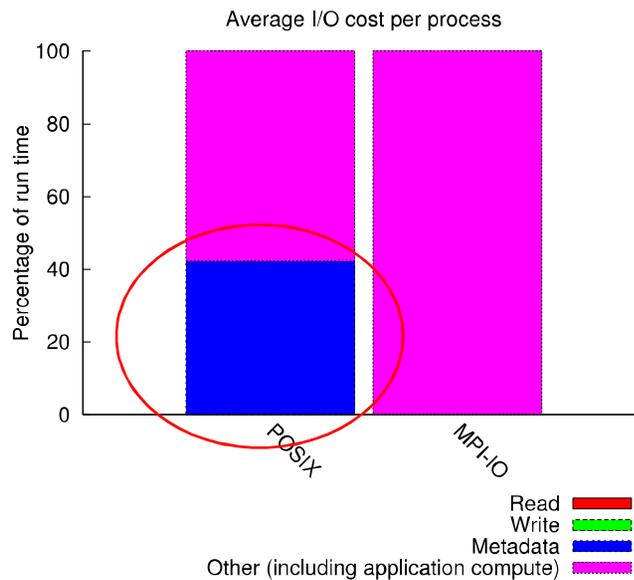
type	number of files	avg. size	max size
total opened	129	1017M	1.1G
read-only files	0	0	0
write-only files	129	1017M	1.1G
read/write files	0	0	0
created files	129	1017M	1.1G



# Performance Debugging: Simulation Output

- HSCD combustion physics application
  - HSCD was writing 2-3 files per process with up to 32,768 cores
  - Darshan attributed 99% of the I/O time to metadata (on Intrepid BG/P)

jobid: 0	uid: 1817	nprocs: 8192	runtime: 863 seconds
----------	-----------	--------------	----------------------



File Count Summary

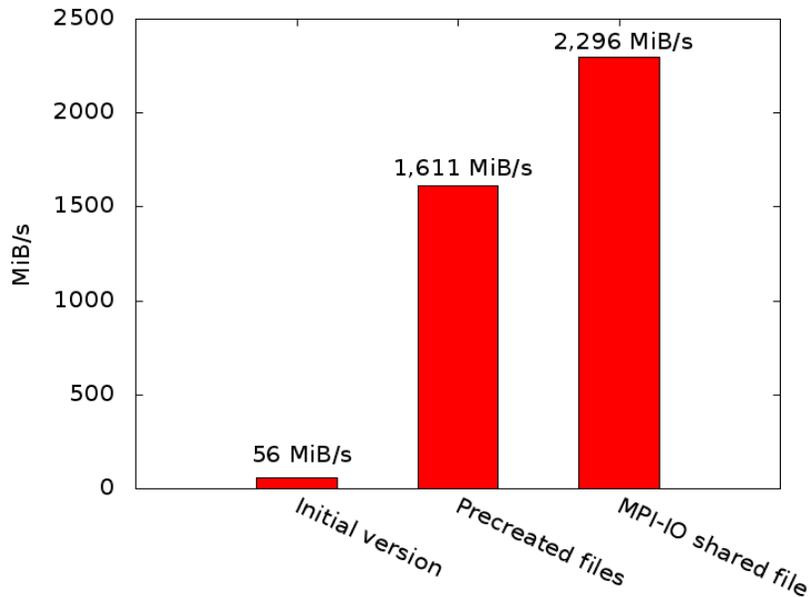
type	number of files	avg. size	max size
total opened	16388	2.5M	8.1M
read-only files	0	0	0
write-only files	16388	2.5M	8.1M
read/write files	0	0	0
created files	16388	2.5M	8.1M



# Simulation Output (continued)

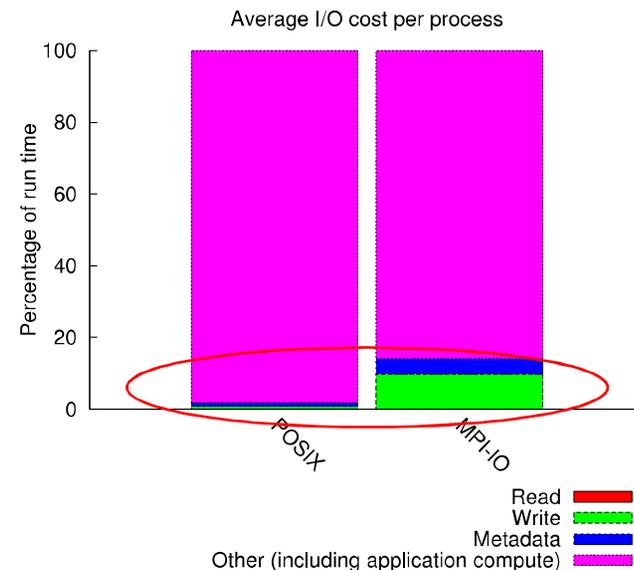
- With help from ALCF catalysts and Darshan instrumentation, we developed an I/O strategy that used MPI-IO collectives and a new file layout to reduce metadata overhead
- Impact: 41X improvement in I/O throughput for production application**

HSCD I/O performance with 32,768 cores

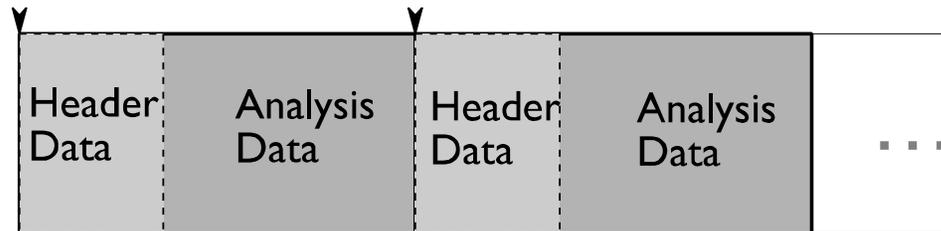


File Count Summary

type	number of files	avg. size	max size
total opened	8	515M	2.0G
read-only files	2	2.2K	3.7K
write-only files	6	686M	2.0G
read/write files	0	0	0
created files	6	686M	2.0G



# Performance Debugging: An Analysis I/O Example



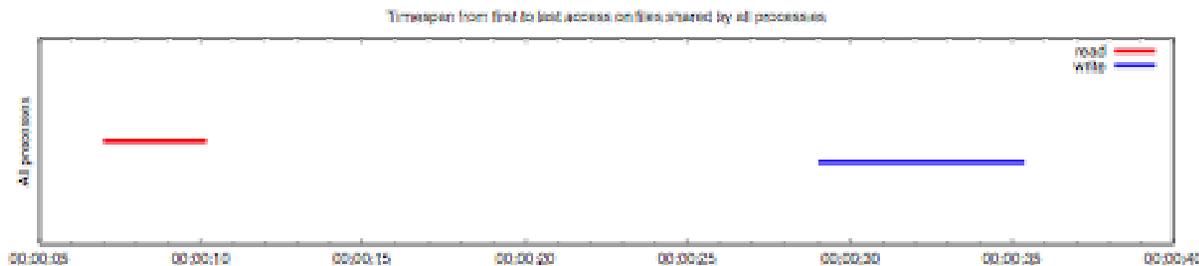
- Variable-size analysis data requires headers to contain size information
- Original idea: all processes collectively write headers, followed by all processes collectively write analysis data
- Use MPI-IO, collective I/O, all optimizations
- 4 GB output file (not very large)
- Why does the I/O take so long in this case?

Processes	I/O Time (s)	Total Time (s)
8,192	8	60
16,384	16	47
32,768	32	57

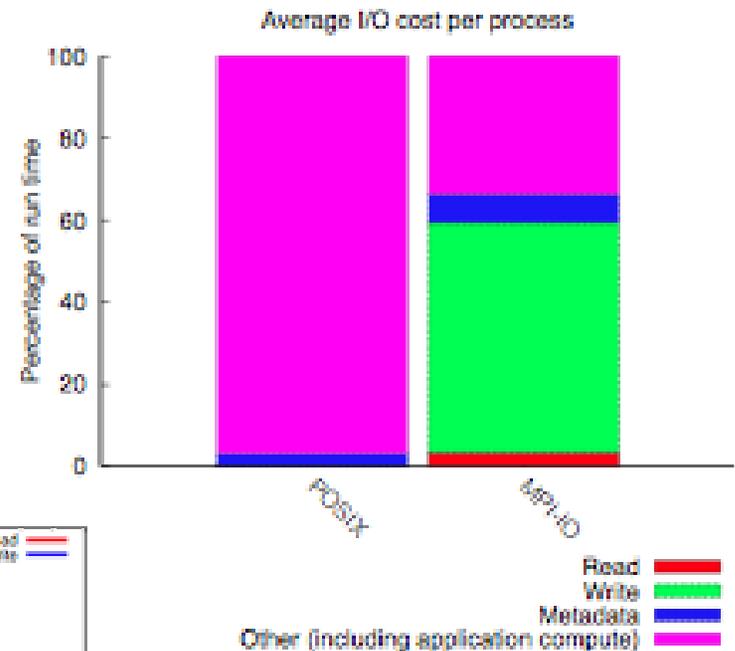


## An Analysis I/O Example (continued)

- **Problem:** More than 50% of time spent writing output at 32K processes. Cause: Unexpected RMW pattern, difficult to see at the application code level, was identified from Darshan summaries.
- What we expected to see, read data followed by write analysis:

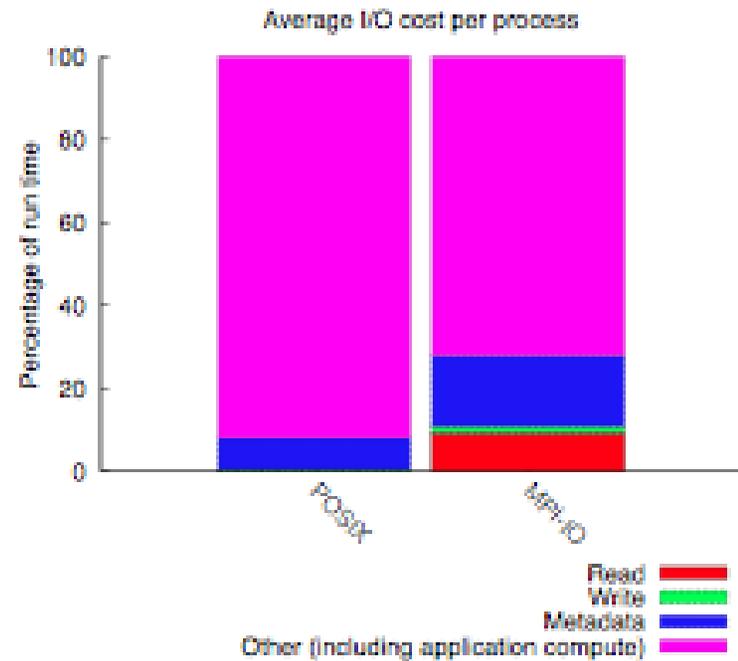


- What we saw instead: RMW during the writing shown by overlapping red (read) and blue (write), and a very long write as well.



# An Analysis I/O Example (continued)

- Solution:** Reorder operations to combine writing block headers with block payloads, so that "holes" are not written into the file during the writing of block headers, to be filled when writing block payloads
- Result:** Less than 25% of time spent writing output, output time 4X shorter, overall run time 1.7X shorter
- Impact:** Enabled parallel Morse-Smale computation to scale to 32K processes on Rayleigh-Taylor instability data



Processes	I/O Time (s)	Total Time (s)
8,192	7	60
16,384	6	40
32,768	7	33



# Example: redundant read traffic

- Scenario: Applications that read more bytes of data from the file system than were present in the file
  - Even with caching effects, this type of job can cause disruptive I/O network traffic
  - Candidates for aggregation or collective I/O

- Example:

- Scale: 6,138 processes
- Run time: 6.5 hours
- Avg. I/O time per process: 27 minutes

File Count Summary  
(estimated by I/O access offsets)

type	number of files	avg. size	max size
total opened	1299	1.1G	8.0G
read-only files	1187	1.1G	8.0G
write-only files	112	418M	2.6G
read/write files	0	0	0
created files	112	418M	2.6G

- 1.3 TiB of file data
- 500+ TiB read!

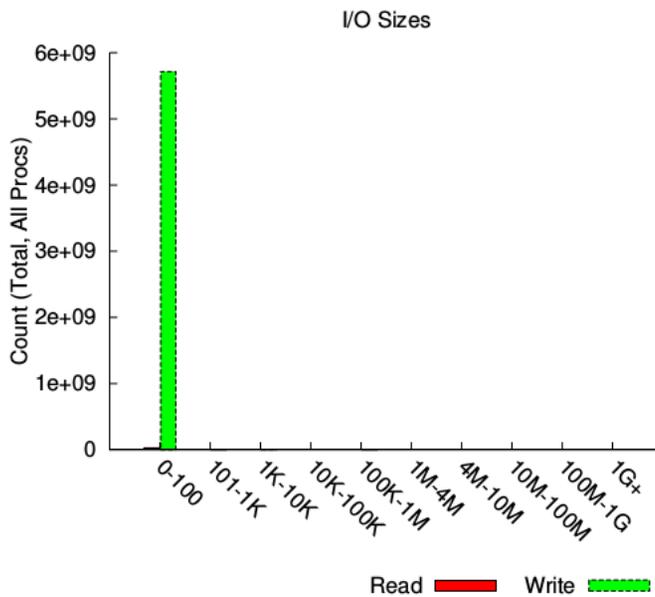
Data Transfer Per Filesystem

File System	Write		Read	
	MiB	Ratio	MiB	Ratio
/	47161.47354	1.00000	575224145.24837	1.00000



# Example: small writes to shared files

- Scenario: Small writes can contribute to poor performance
  - Particularly when writing to shared files
  - Candidates for collective I/O or batching/buffering of write operations
- Example:
  - Issued 5.7 billion writes to shared files, each less than 100 bytes in size
  - Averaged just over 1 MiB/s per process during shared write phase

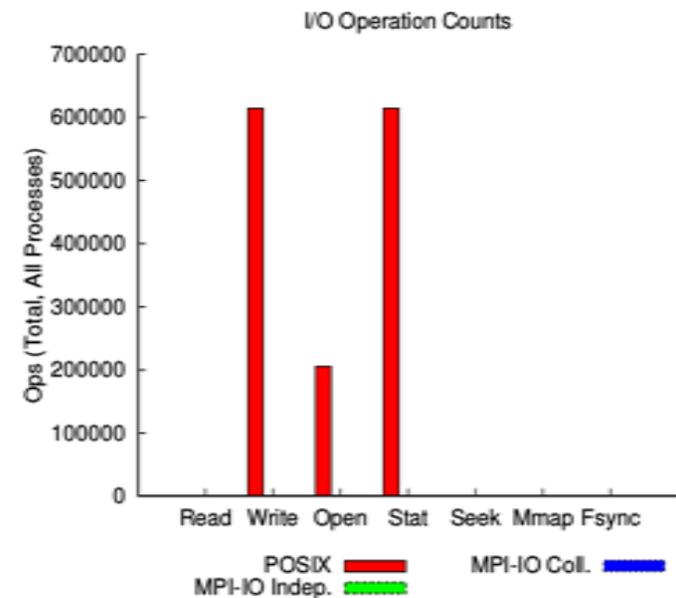
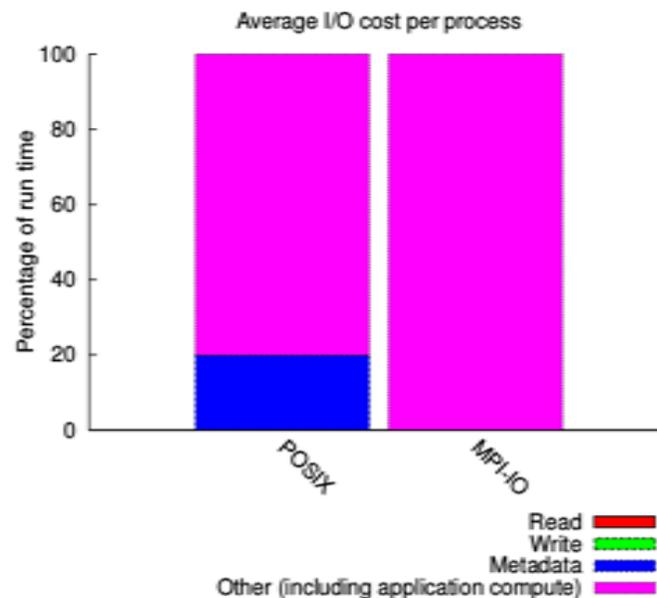


access size	count
1	3418409696
15	2275400442
24	42289948
12	14725053



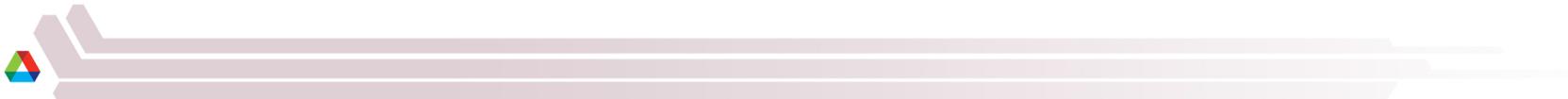
## Example: excessive metadata overhead

- Scenario: Very high percentage of I/O time spent performing metadata operations such as open(), close(), stat(), and seek()
  - Close() cost can be misleading due to write-behind cache flushing
  - Candidates for coalescing files and eliminating extra metadata calls
- Example:
  - Scale: 40,960 processes for 229 seconds, 103 seconds of I/O
  - 99% of I/O time in metadata operations
  - Generated 200,000+ files with 600,000+ write() and 600,000+ stat() calls



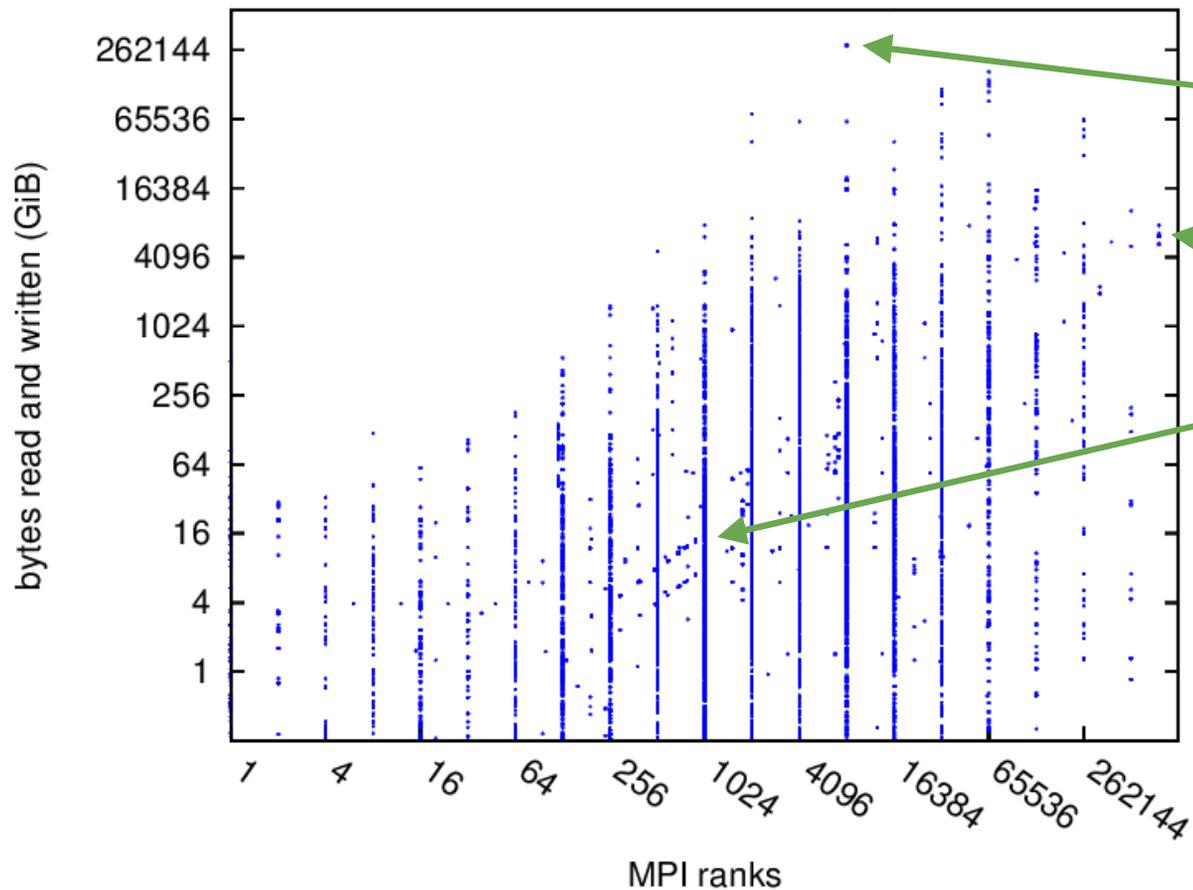
## Metadata side topic: what's so bad about stat()?

- stat() is actually quite cheap on most file systems
- But not a large-scale HPC I/O system!
- The usual problem is that stat() requires a consistent size calculation for the file
- To do this, a PFS has two options:
  - Store a precalculated size on the metadata server, which becomes a source of contention
  - Calculate size on demand, which might cause a storm of requests to \*all\* servers
- No present-day PFS deployments respond very well when thousands of processes stat() the same file at once



# Example: system-wide analysis

- Job size vs. data volume for Mira BG/Q system in 2014 (~128,000 logs as of October, ~8 PiB of traffic)

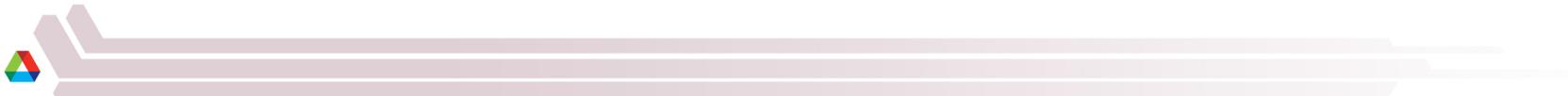


- Biggest by volume: ~300 TiB
- Biggest by scale: 768K processes
- Probably some scaling experiments?
- Most jobs use power of 2 numbers of processes on Mira



# I/O Understanding Takeaway

- Scalable tools like Darshan can yield useful insight
  - Identify characteristics that make applications successful ...and those that cause problems.
  - Identify problems to address through I/O research
- Petascale performance tools require special considerations
  - Target the problem domain carefully to minimize amount of data
  - Avoid shared resources
  - Use collectives where possible
- For more information, see:  
<http://www.mcs.anl.gov/research/projects/darshan>



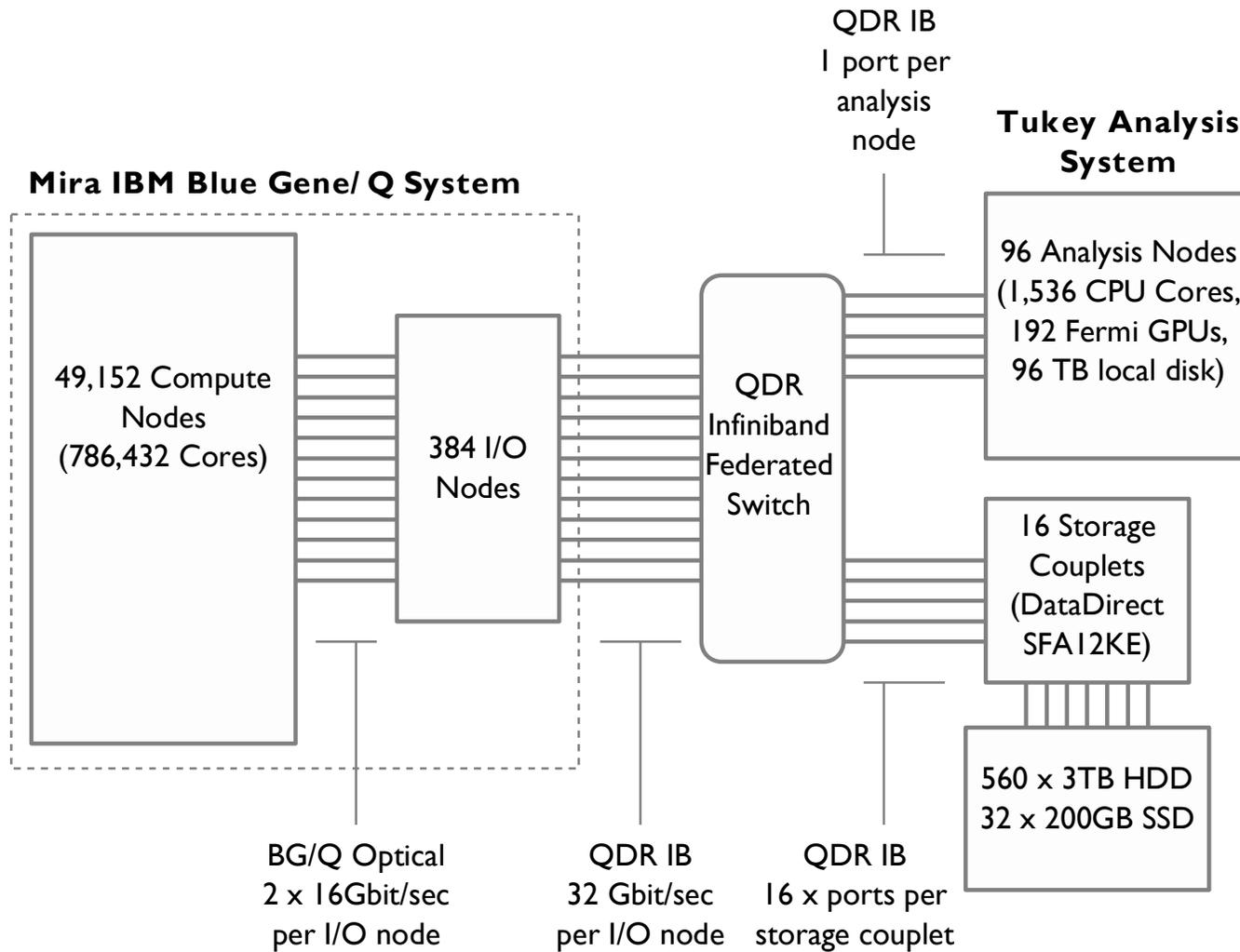
## I/O Performance Tuning “Rules of thumb”

- Use collectives when possible
- Use high-level libraries (e.g. HDF5 or PnetCDF) when possible
- A few large I/O operations are better than many small I/O operations
- Avoid unnecessary metadata operations, especially *stat()*
- Avoid writing to shared files with POSIX
- Avoid leaving gaps/holes in files to be written later
- Use tools like Darshan to check assumptions about behavior



# Changes in Data Analysis Workflows

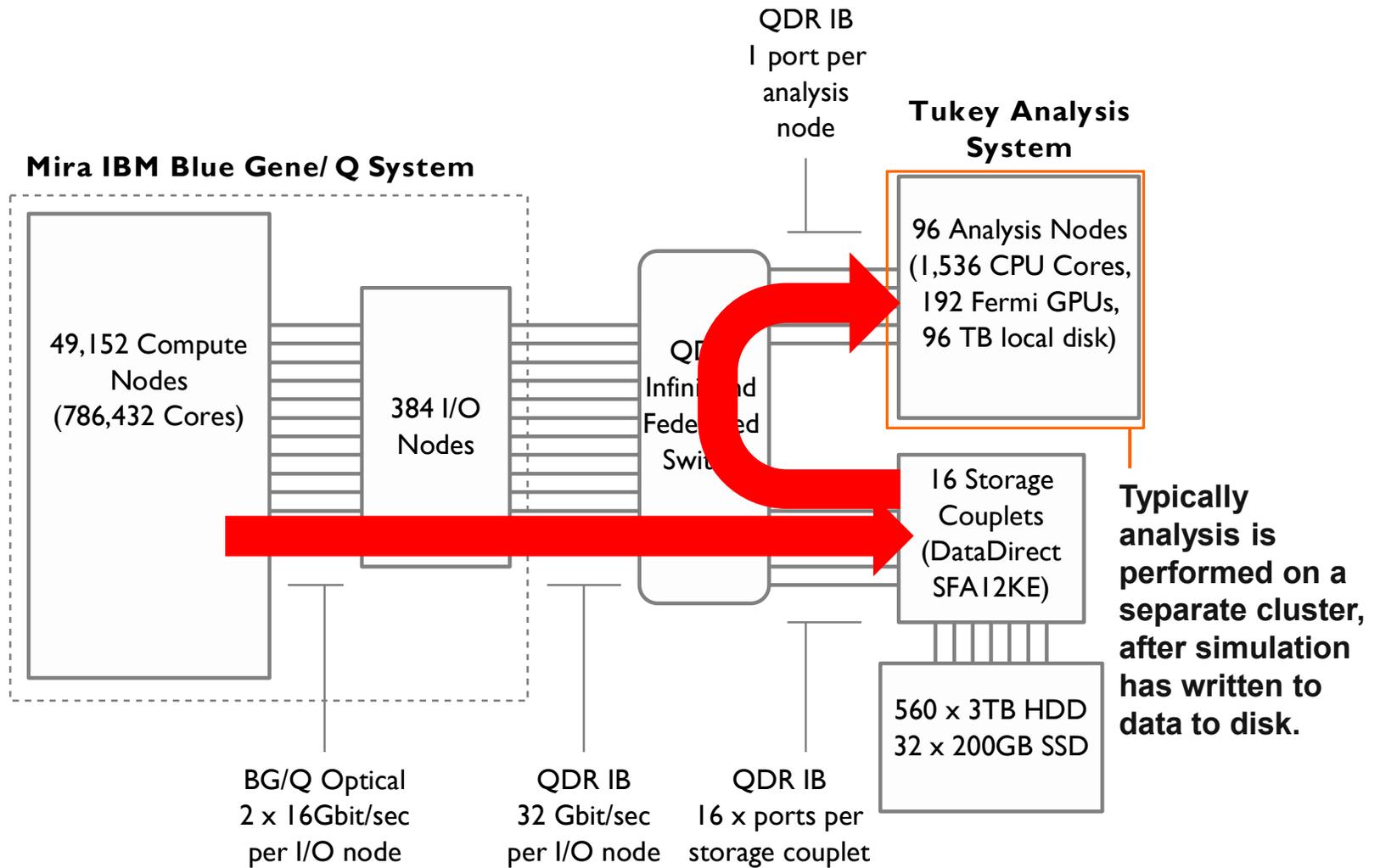
# Our Example Leadership System Architecture



High-level diagram of 10 Pflop IBM Blue Gene/Q system at Argonne Leadership Computing Facility



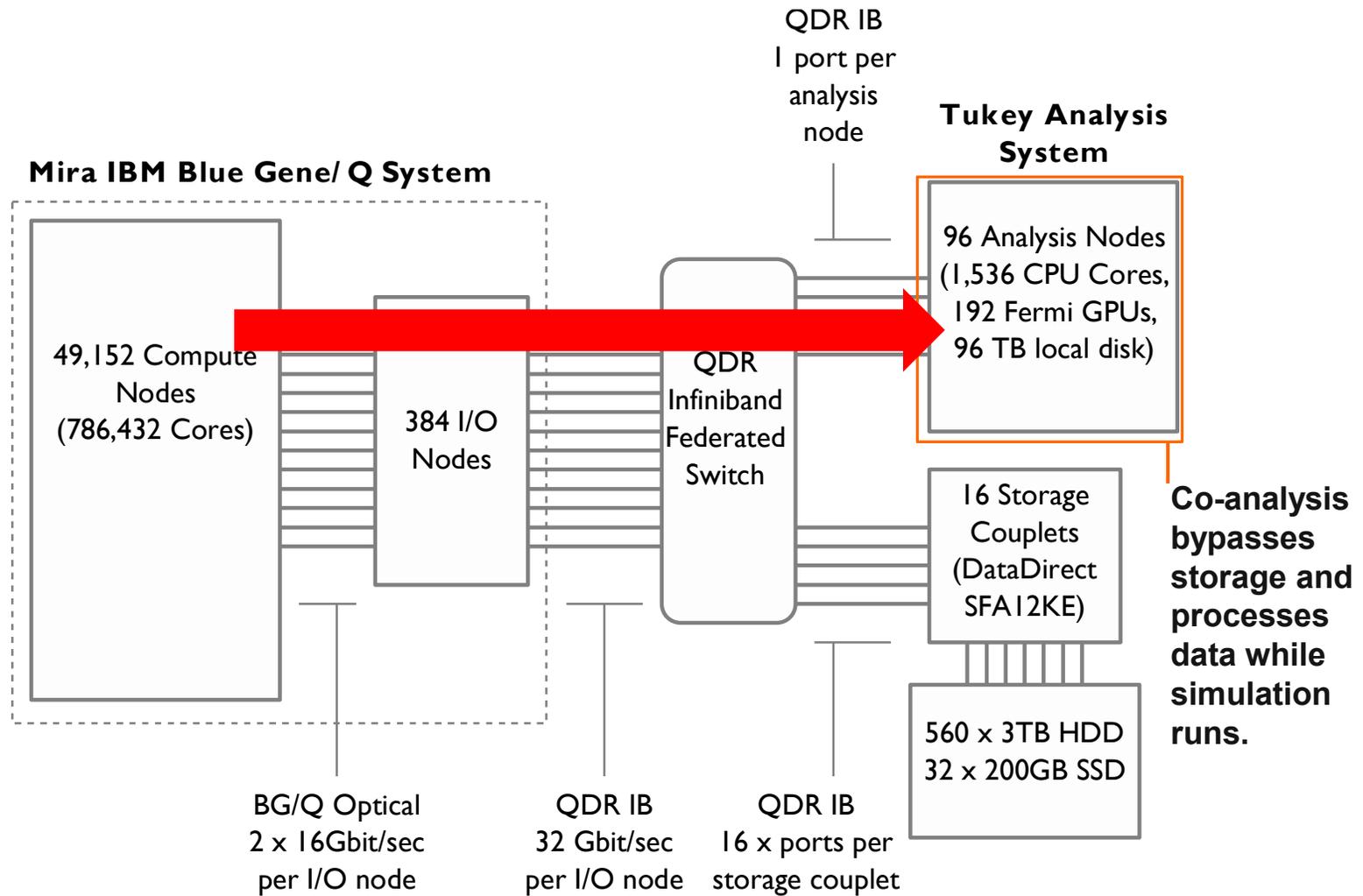
# Analyzing Data: Traditional Post-Processing



High-level diagram of 10 Pflop IBM Blue Gene/Q system at Argonne Leadership Computing Facility



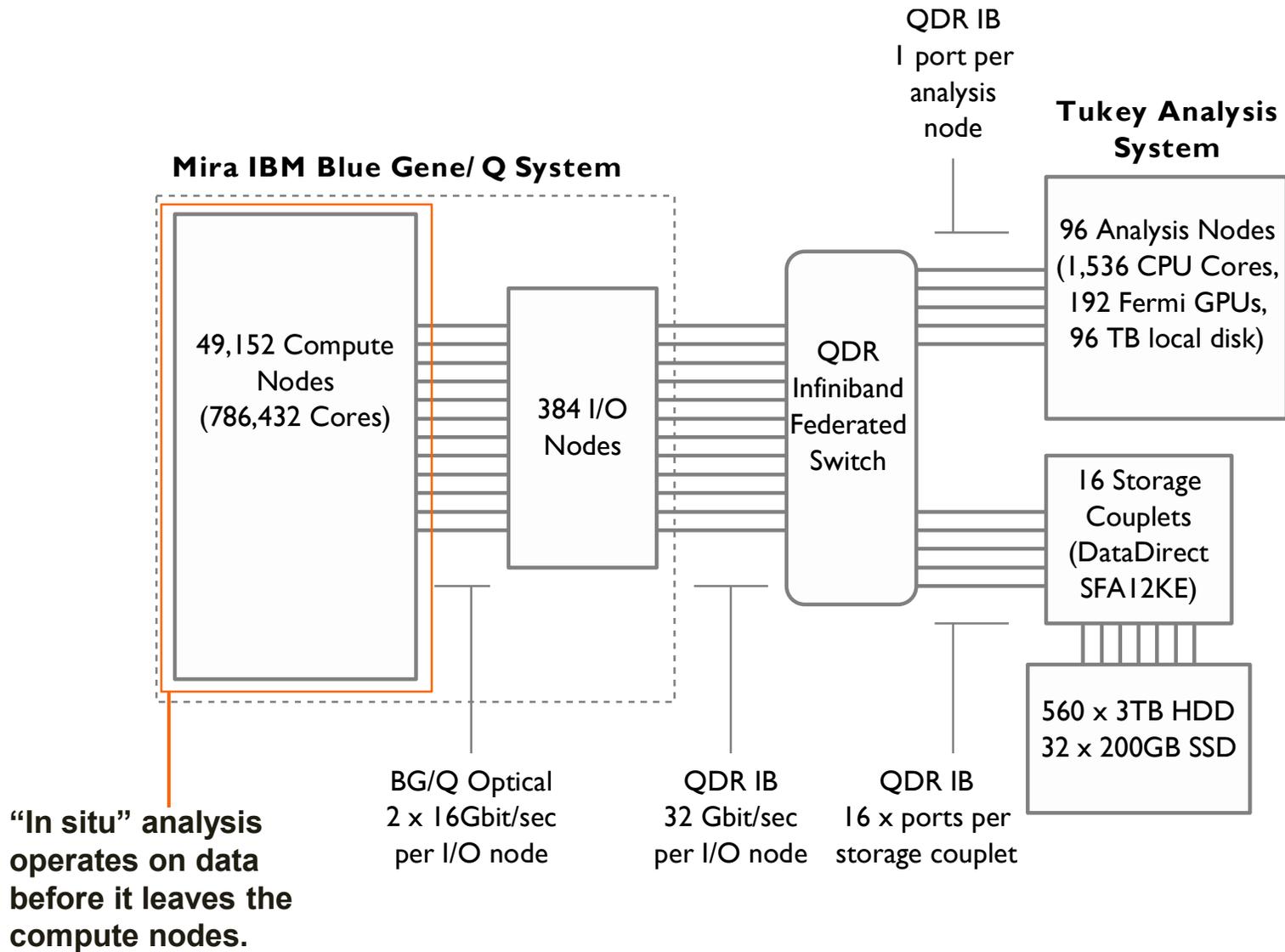
# Analyzing Data: Co-Analysis



High-level diagram of 10 Pflop IBM Blue Gene/Q system at Argonne Leadership Computing Facility



# Analyzing Data: In Situ Analysis

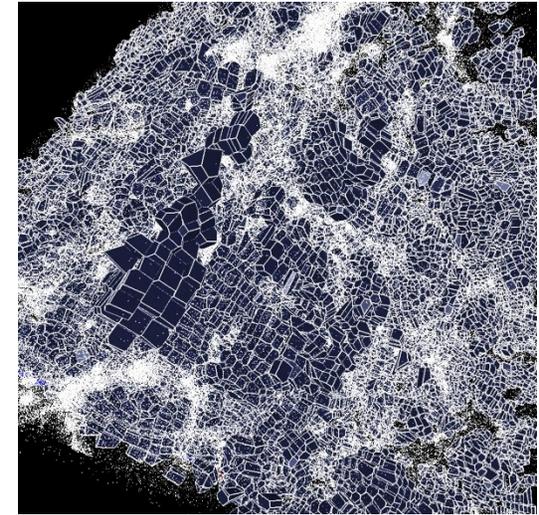


High-level diagram of 10 Pflop IBM Blue Gene/Q system at Argonne Leadership Computing Facility

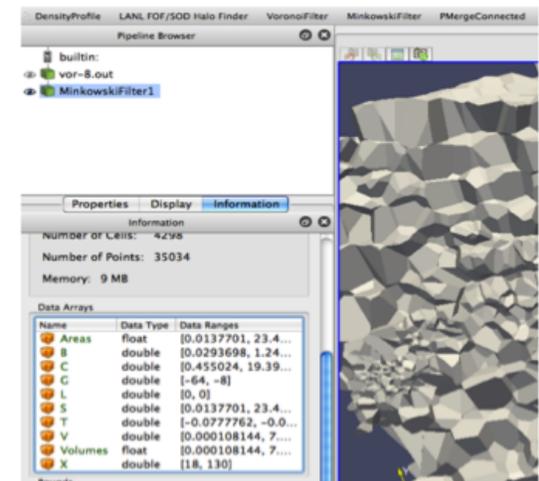


# In Situ Analysis and Data Reduction: HACC

- On the HPC side, analysis is increasingly performed during runtime to avoid subsequent I/O
- HACC cosmology code employing Voronoi tessellation
  - Converts from particles into unstructured grid based on particle density
  - Adaptive, retains full dynamic range of input
  - DIY toolkit (open source) used to implement analysis routines
- ParaView environment used for visual exploration, custom tools for analysis



Voronoi tessellation reveals regions of irregular low-density voids amid high-density halos.



ParaView plugin provides interactive feature exploration.

Collaboration with Kitware and U. of Tennessee



# In-System Storage

Many thanks to:

**Ning Liu**

Illinois Institute of Technology

**Chris Carothers**

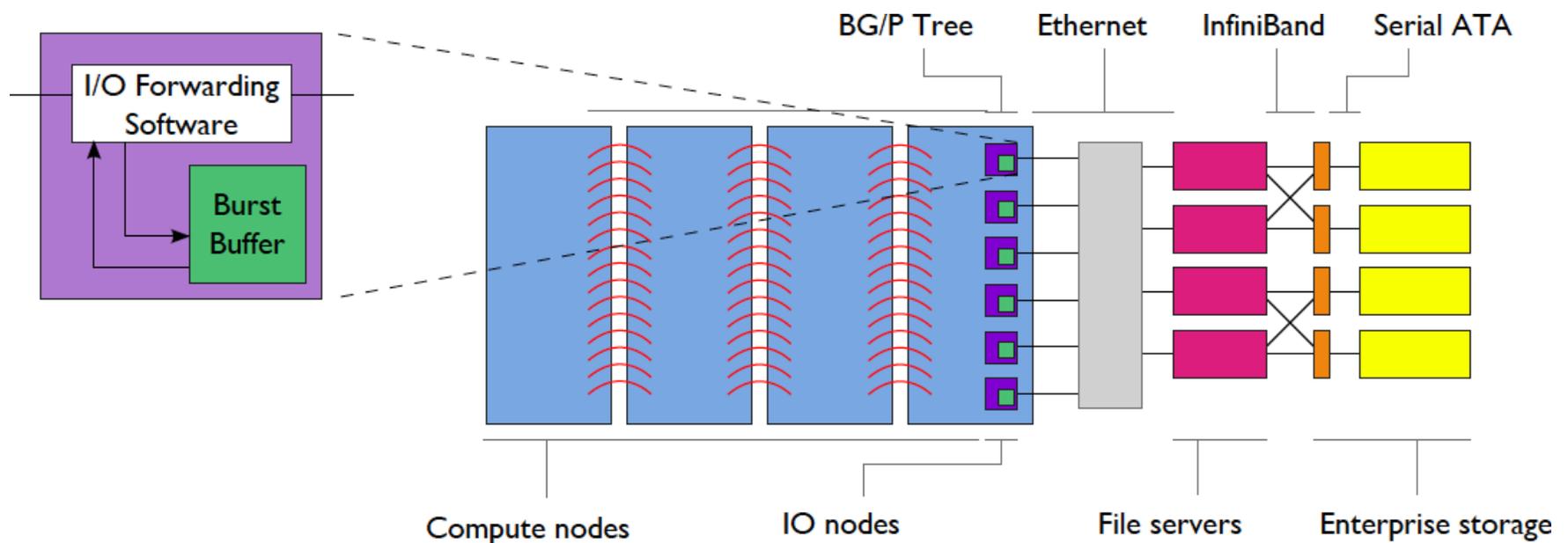
Rensselaer Polytechnic Institute

**Jason Cope**

DataDirect Networks

# Adding In System Storage to the Storage Model

The inclusion of NVRAM storage in future systems is a compelling way to deal with the burstiness of I/O in HPC systems, reducing the peak I/O requirements for external storage. In this case the NVRAM is called a “burst buffer”.



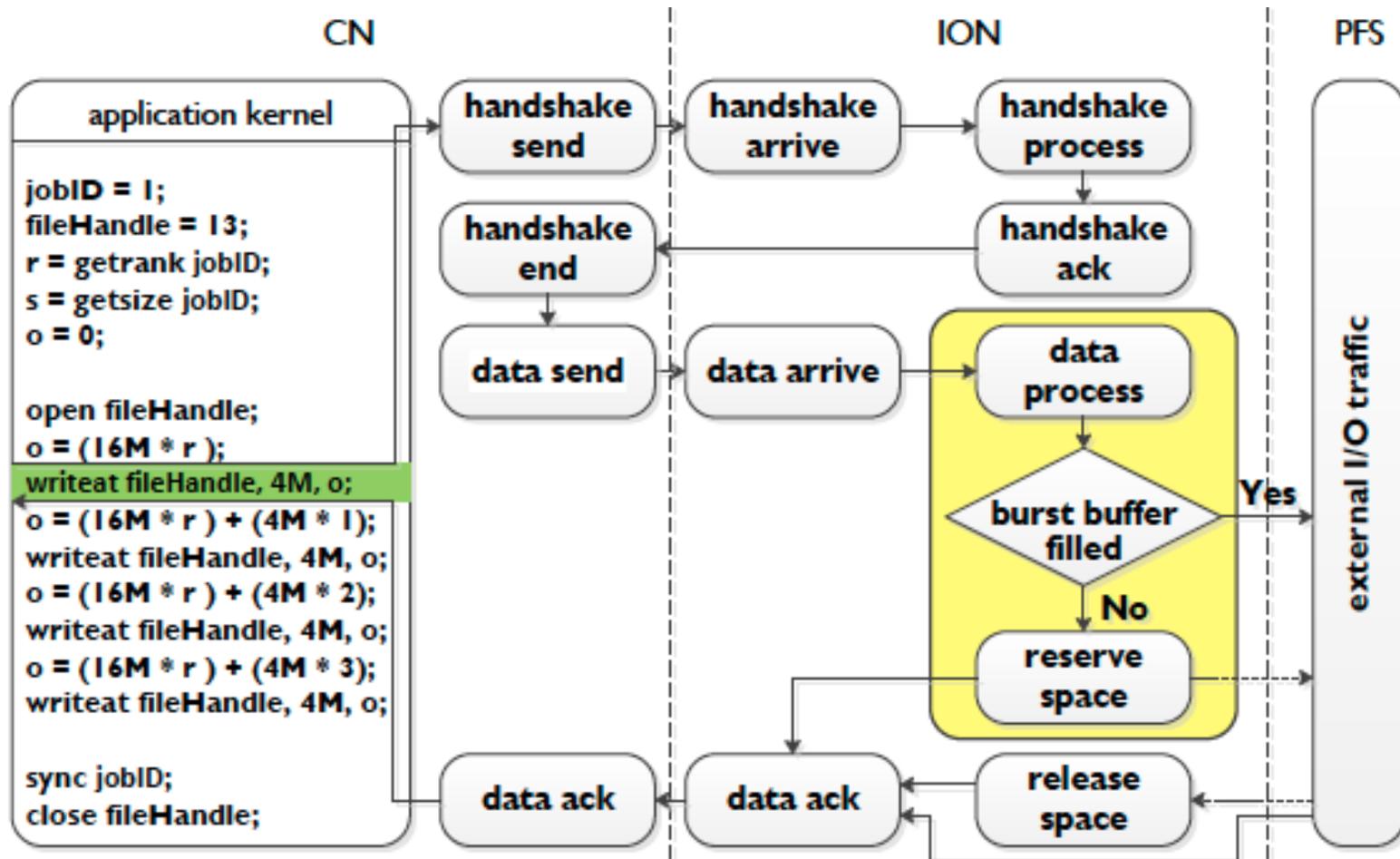
# What's a Burst?

- We quantified the I/O behavior by analyzing one month of production I/O activity on Blue Gene/P from December 2011
  - Application-level access pattern information with per process and per file granularity
  - Adequate to provide estimate of I/O bursts

Project	Procs	Nodes	Total Written	Run Time (hours)	Avg. Size and Subsequent Idle Time for Write Bursts > 1 GiB				
					Count	Size	Size/Node	Size/ION	Idle Time (sec)
PlasmaPhysics	131,072	32,768	67.0 TiB	10.4	1	33.5 TiB	1.0 GiB	67.0 GiB	7554
					1	33.5 TiB	1.0 GiB	67.0 GiB	end of job
Turbulence1	131,072	32,768	8.9 TiB	11.5	5	128.2 GiB	4.0 MiB	256.4 MiB	70
					1	128.2 GiB	4.0 MiB	256.4 MiB	end of job
					421	19.6 GiB	627.2 KiB	39.2 MiB	70
AstroPhysics	32,768	8,096	8.8 TiB	17.7	1	550.9 GiB	68.9 MiB	4.3 GiB	end of job
					8	423.4 GiB	52.9 MiB	3.3 GiB	240
					37	131.5 GiB	16.4 MiB	1.0 GiB	322
					140	1.6 GiB	204.8 KiB	12.8 MiB	318
Turbulence2	4,096	4,096	5.1 TiB	11.6	21	235.8 GiB	59.0 MiB	3.7 GiB	1.2
					1	235.8 GiB	59.0 MiB	3.7 GiB	end of job

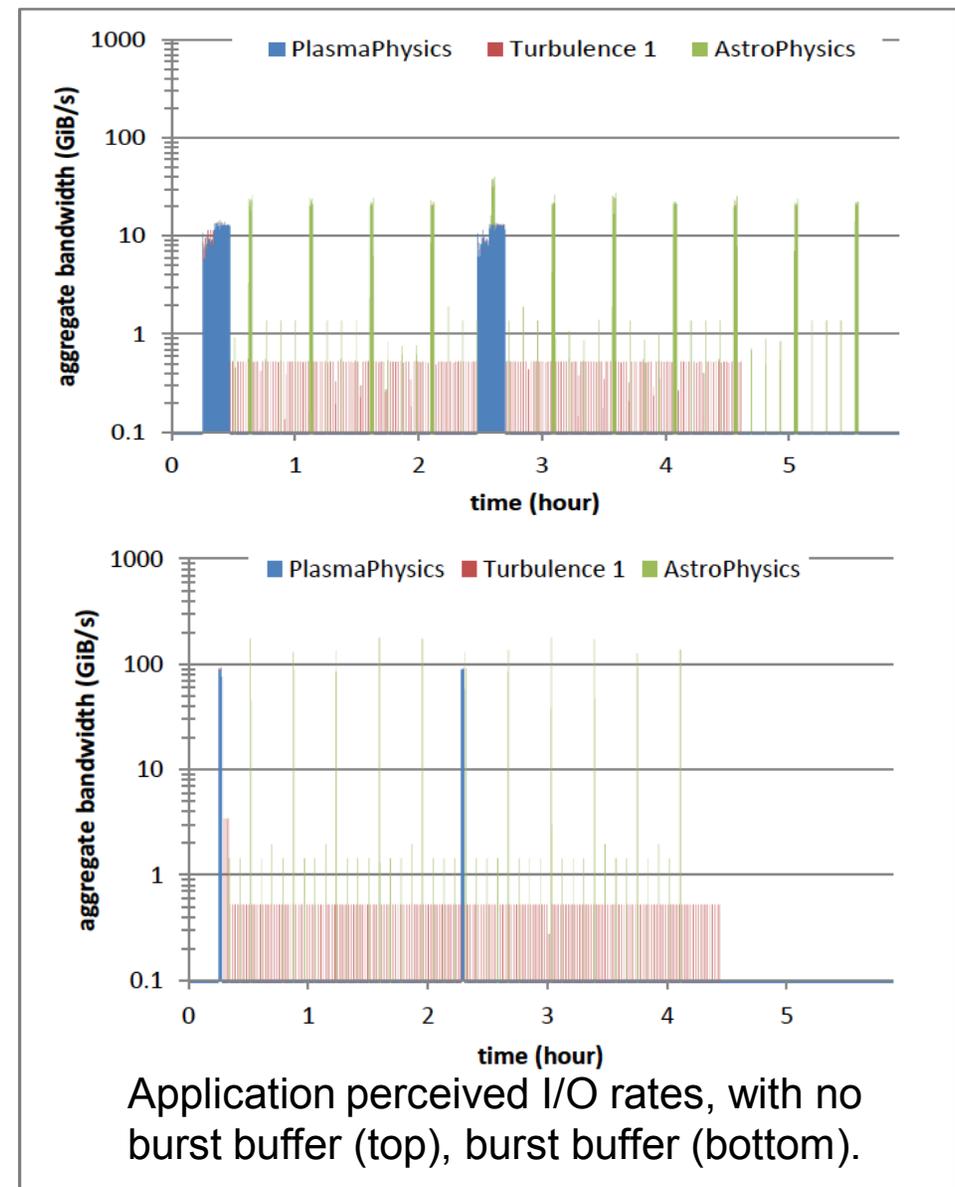


# Studying Burst Buffers with Parallel Discrete Event Simulation



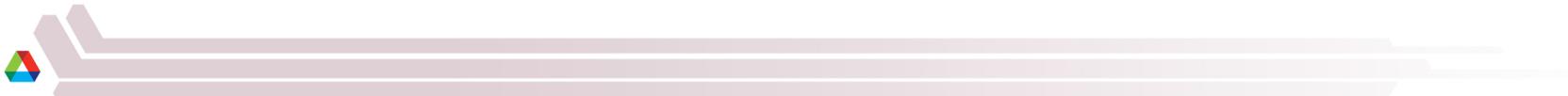
# Burst Buffers Work for Multi-application Workloads

- Early simulation results, 2011
- Burst buffers improve application perceived throughput under mixed I/O workloads.
- Applications' time to solution decrease with burst buffers enabled (from 5.5 to 4.4 hours)
- Peak bandwidth of the external I/O system may be reduced by 50% without a perceived change on the application side
- Tool for co-design



## Burst buffers: from research to production in 2015

- NERSC's Cori system will be the first DOE platform with burst buffers in production
  - Phase 1: Fall 2015, Phase 2 late 2016
  - Architecture: dedicated “burst buffer nodes” with PCI-attached SSD storage
  - A faster, intermediate tier of storage between compute nodes and parallel file system
  - Software: Cray Datawarp
  - Up to 1.6 TiB/s data rate



## Burst buffers on other next-gen platforms

Multiple approaches to in-system storage and how to use it in upcoming Trinity and CORAL procurements

- LANL/Sandia: Trinity (2015/2016)
  - Similar architecture to NERSC/Cori, dedicated burst buffer nodes
- ORNL: Summit (2018)
  - 800 GiB NVRAM per compute node
- LLNL: Sierra (2017)
  - 800 GiB NVRAM per compute node
- ANL: Theta (2016)
  - 128 GiB SSD per compute node
- ANL: Aurora (2018)
  - NVRAM per compute node and SSD burst buffers



## How will applications use burst buffers?

- Usage model possibilities:
  - Provisioned per job or user
  - Migration (staging) of data to and from parallel file system
  - Transparent caching layer
  - In-transit analysis of data
- Initial adoption will require minimal (if any) change to applications
- Specify parameters/mode at job submission time



# Beyond Burst Buffers

- Obviously lots of other potential uses
  - Checkpointing location
  - Out-of-core computation
  - Holding area for analysis data (e.g., temporal analysis, in situ)
  - Code coupling
  - Input data staging
  - ...
- Improves memory capacity of systems
  - More data intensive applications?
- Placement of NVRAM will matter
  - On I/O forwarding nodes (as in simulation example)
  - On some/all compute nodes?
  - Both?



# Future data models: beyond files

# Are there alternatives to the file/directory model?

(Yes!)

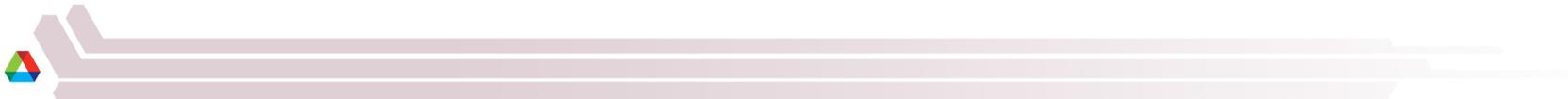
- Internet services and big data deployments have successfully utilized a variety of alternatives
  - Key/value: Cassandra, Dynamo, Memcached
  - Objects: S3, Openstack/Swift
  - Files with non-POSIX semantics: Hadoop and GoogleFS
  - Graph: Neo4j, OrientDB
  - Documents: MongoDB
- Research underway to make these foundational data models and other available to HPC applications on future systems
- No clear consensus on which models will reach production yet
- Storage services likely to be customizable in the future instead of a “one size fits all” approach



# Wrapping Up

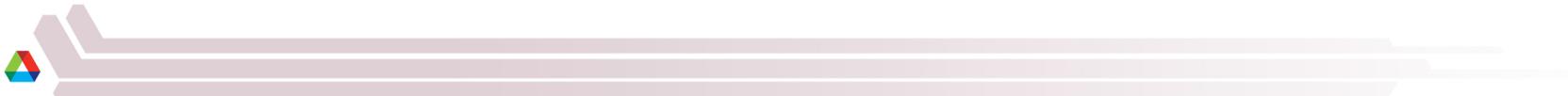
## Wrapping Up

- HPC storage is a complex hardware/software system
- Some effort is necessary to make best use of these resources
- Many tools are available to:
  - Increase productivity
  - Improve portability of data and capture additional provenance
  - Assist in understanding performance problems
- We hope we have shed some light on these systems



## Wrapping Up (but wait, there's more!)

- Hands on exercises tonight:
  - Feel free to work on your own code and ask us questions
  - There are some planned exercises if you would like to experiment with the tools and libraries presented in this talk
  - Example code from presentation is available as well
- Check for email about hands-on material and ATPESC mailing list
- Next up are presentations from **Scot Breitenfeld** and **Rachana Ananthakrishnan**
- See you after dinner for more discussion, maybe some hacking?





## Feedback

- What technologies are most interesting for your work?
- What would you like to have heard more about?
- Let us know any time, or catch us one-on-one tonight

[robl@mcs.anl.gov](mailto:robl@mcs.anl.gov) (Rob)

[carns@mcs.anl.gov](mailto:carns@mcs.anl.gov) (Phil)

Thanks for spending the day with us!

