HPC I/O for Computational Scientists

Rob Latham and Rob Ross
Math and Computer Science Division
Argonne National Laboratory
robl@mcs.anl.gov, rross@mcs.anl.gov

Quincey Koziol
The HDF Group
koziol@hdfgroup.org

Many thanks for their contributions go to:

Brent Welch
Panasas, Inc.
welch@panasas.com

William Gropp
University of Illinois
wgropp@illinois.edu

Katie Antypas
NERSC
kantypas@lbl.gov

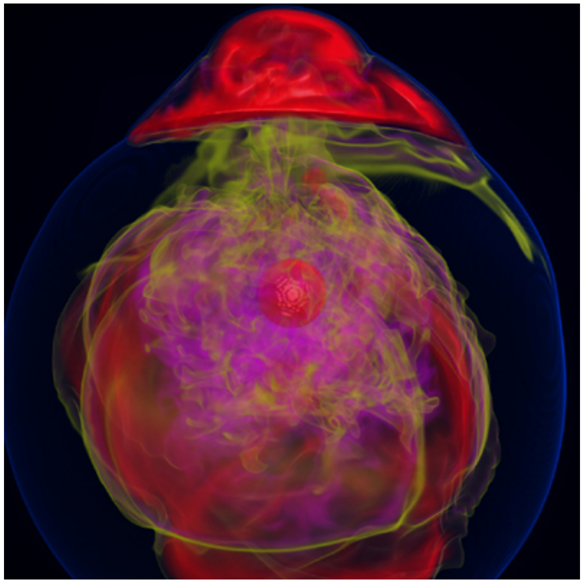
Rusty Lusk
lusk@mcs.anl.gov

Rajeev Thakur
Argonne National Laboratory
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:

    http://www.mcs.anl.gov/mpi/tutorial/advmpi/mpi2tutorial.tar.gz

About Us

- Rob Latham
  - Principle Software Development Specialist, MCS Division, Argonne National Laboratory
  - ROMIO MPI-IO implementation
  - Parallel netCDF high-level I/O library
  - Application outreach
- Rob Ross
  - Computer Scientist, MCS Division, Argonne National Laboratory
  - Parallel Virtual File System
  - Deputy Director, Scientific Data Management, Analysis, and Visualization Institute (SDAV)
- Quincey Koziol
  - HDF5
  - Department of Energy Fast Forward: Exascale Storage

Thinking about HPC I/O Systems

HPC I/O Systems

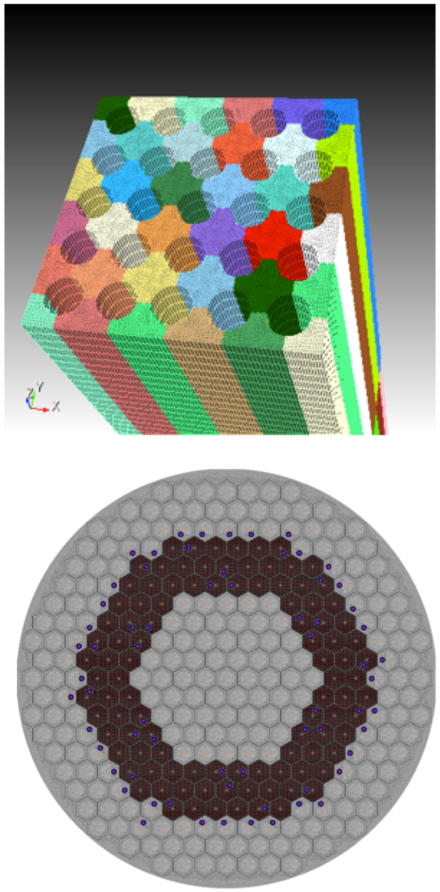
HPC I/O system is the hardware and software that assists in accessing data during simulations and analysis and retaining data between these activities.

- Hardware: disks, disk enclosures, servers, networks, etc.
- Software: parallel file system, libraries, parts of the OS

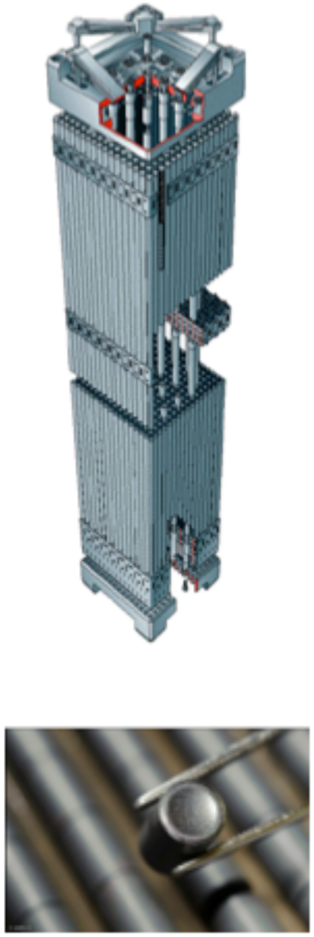
- Two "flavors" of I/O from applications:
  - **Defensive**: storing data to protect results from data loss due to system faults
  - **Productive**: storing/retrieving data as part of the scientific workflow
  - Note: Sometimes these are combined (i.e., data stored both protects from loss and is used 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.



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.

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

Data Volumes in Computational Science

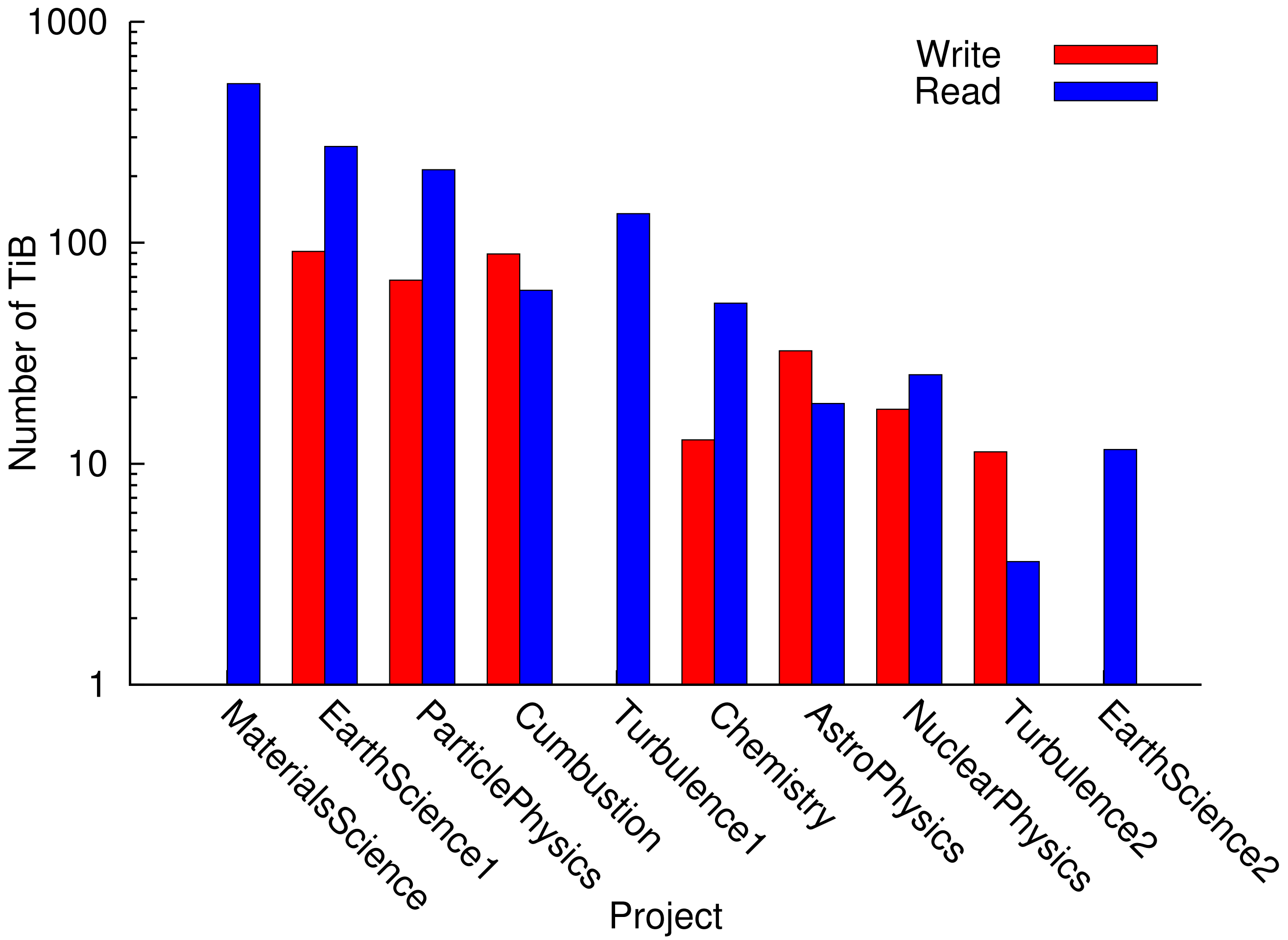
Science teams are routinely working with tens and hundreds of terabytes (TBs) of data.

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

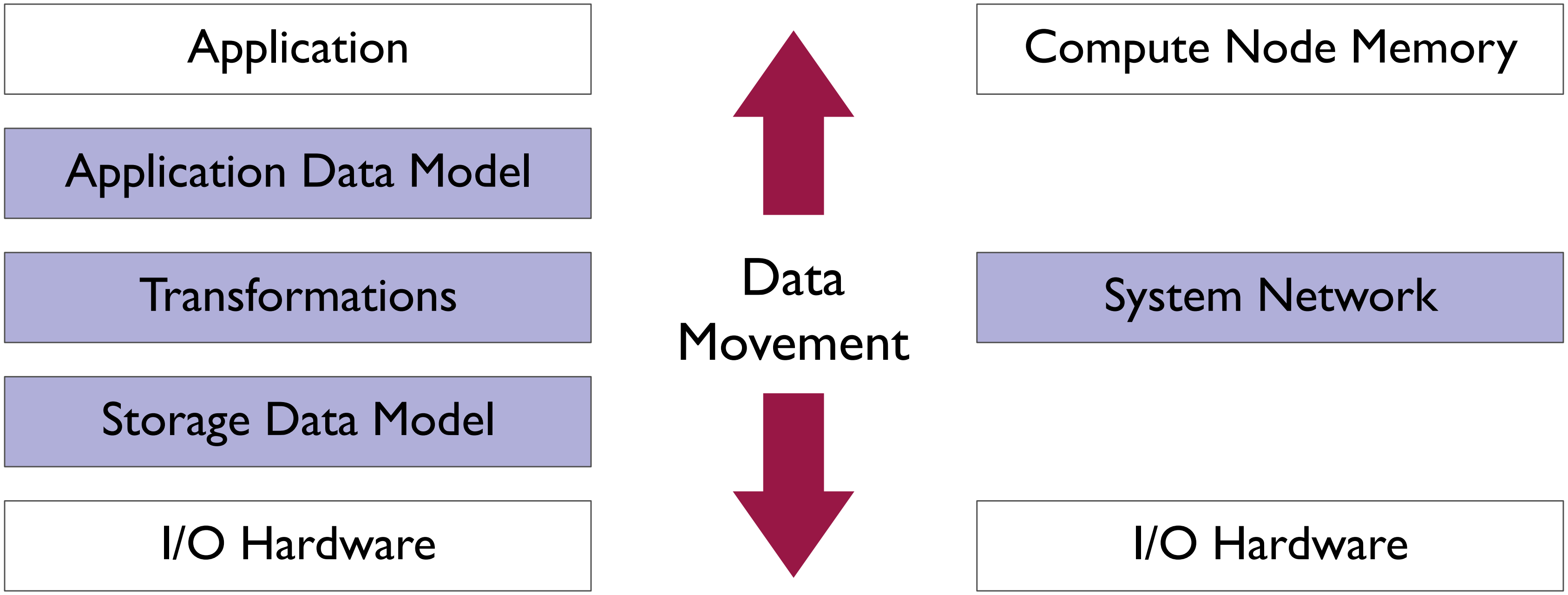
It’s not just checkpoints – scientists are reading large volumes of data *into* HPC systems as part of their science.



Top 10 data producer/consumers instrumented with Darshan over the month of July, 2011.

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.

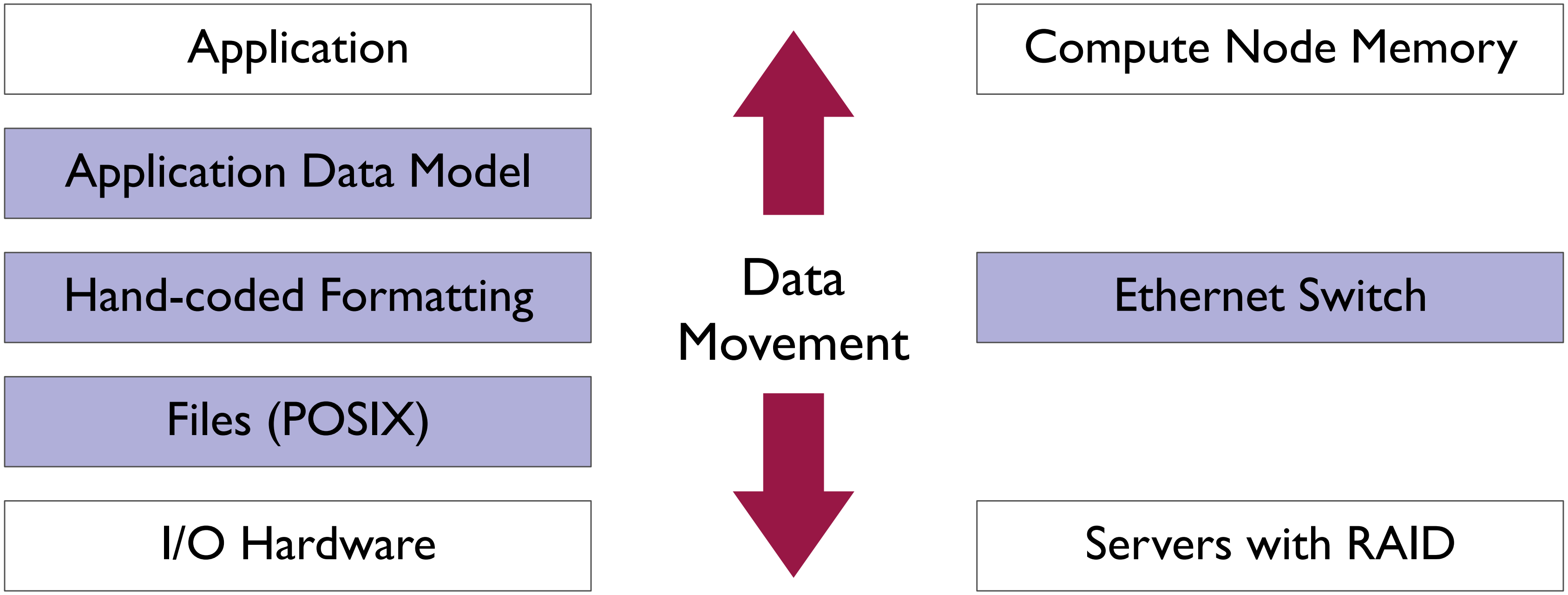


Logical (data model) view of data access.

Physical (hardware) view of data access.

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.



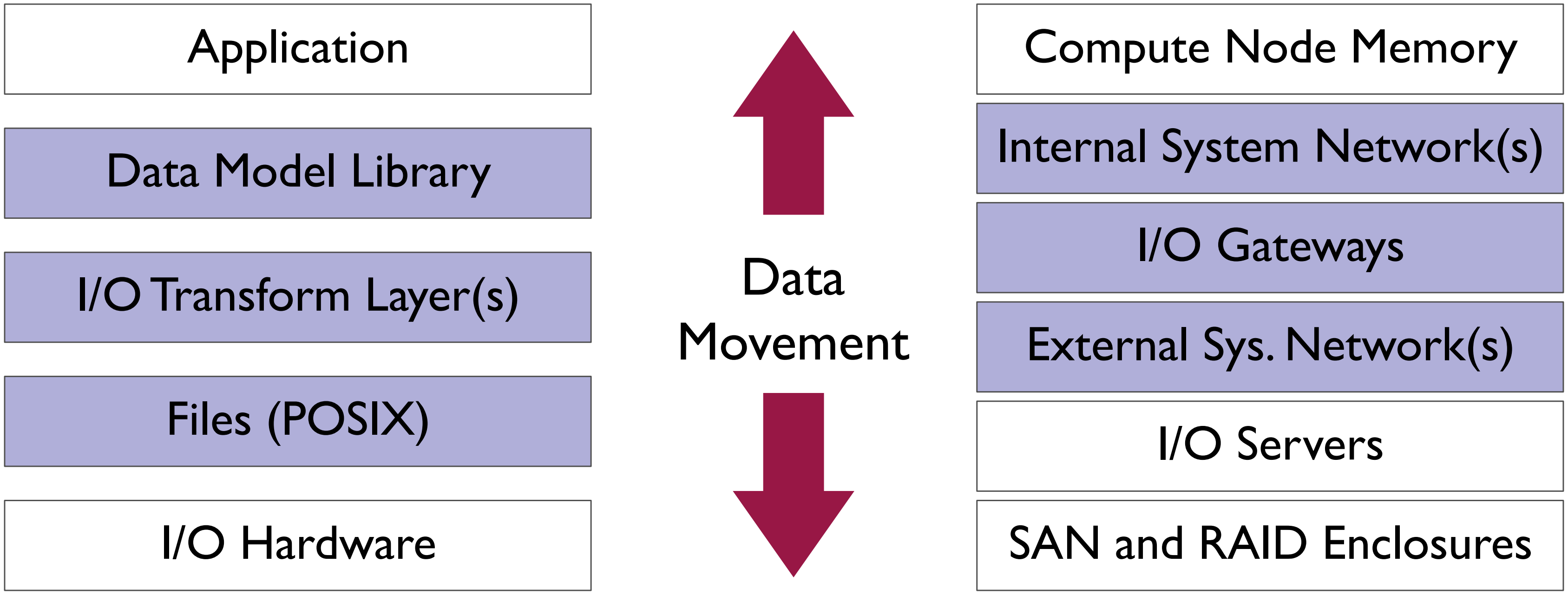
Logical (data model) view of data access.

Physical (hardware) view of data access.

* 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.



Application

Data Model Library

I/O Transform Layer(s)

Files (POSIX)

I/O Hardware

Data Movement

Compute Node Memory

Internal System Network(s)

I/O Gateways

External Sys. Network(s)

I/O Servers

SAN and RAID Enclosures

Logical (data model) view of data access.

Physical (hardware) view of data access.

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

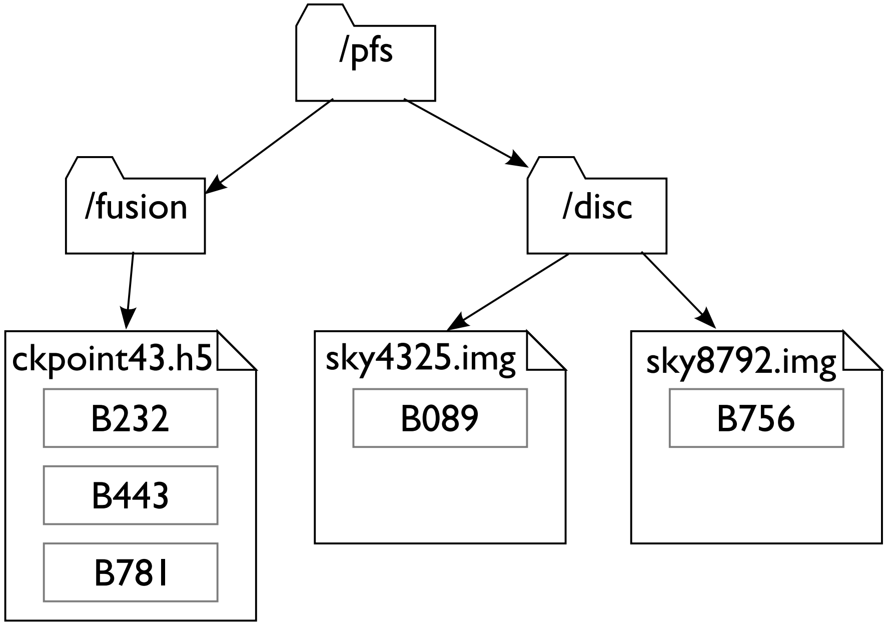
- Next we will dive down into some of the details of HPC I/O

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!

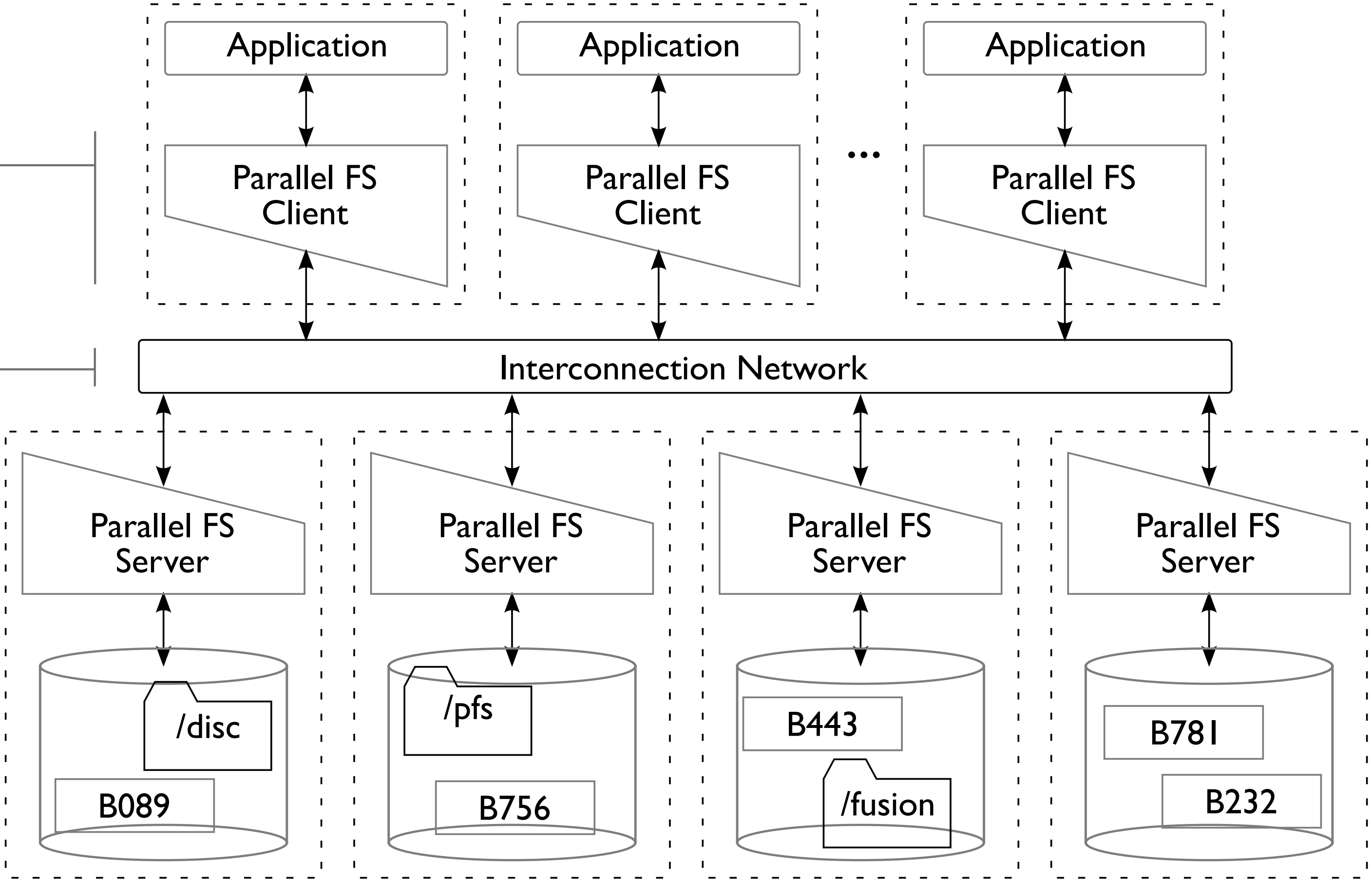
Reading and Writing Data (etc.)

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

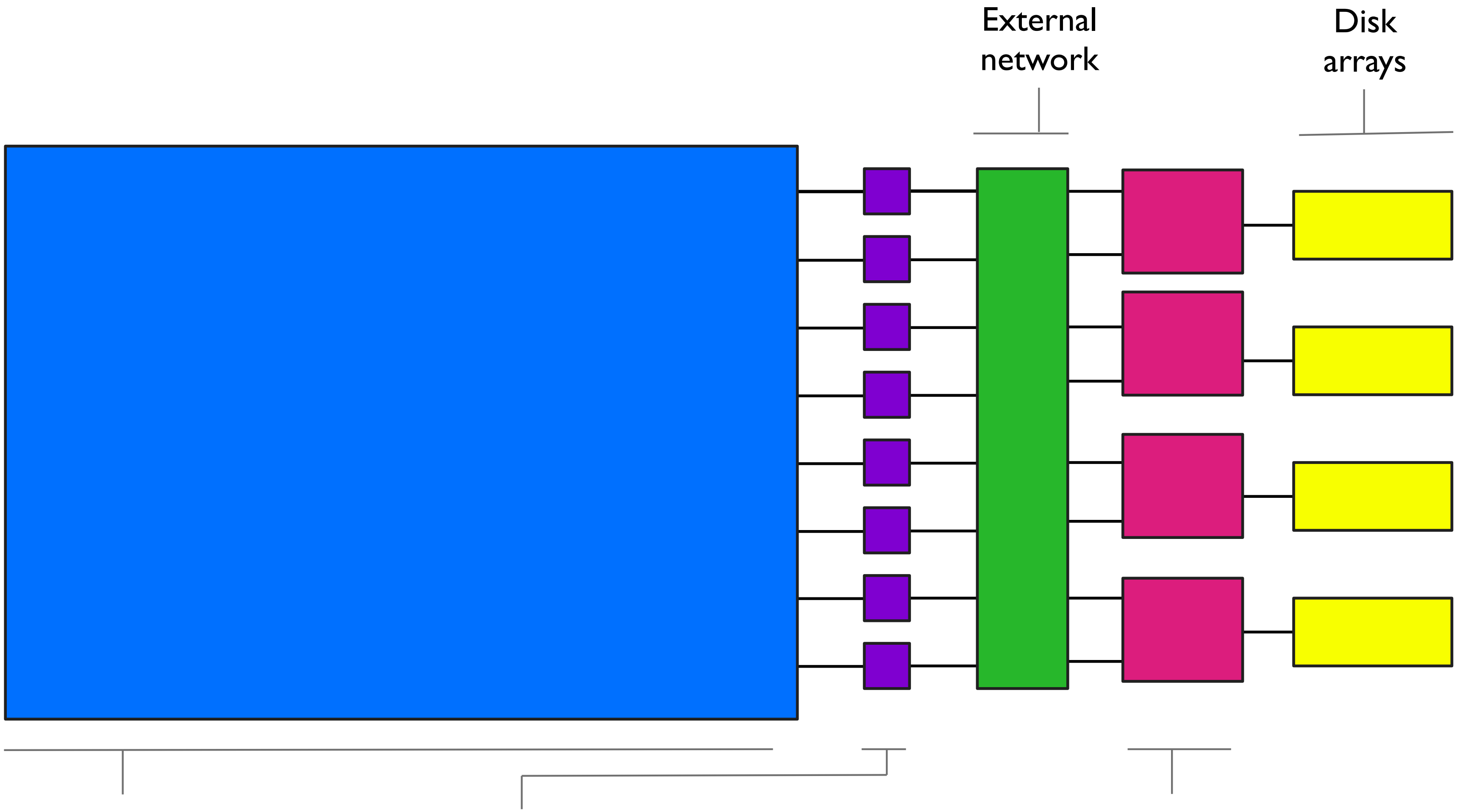
Requests pass over the interconnect, thus **each request incurs some latency**.

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



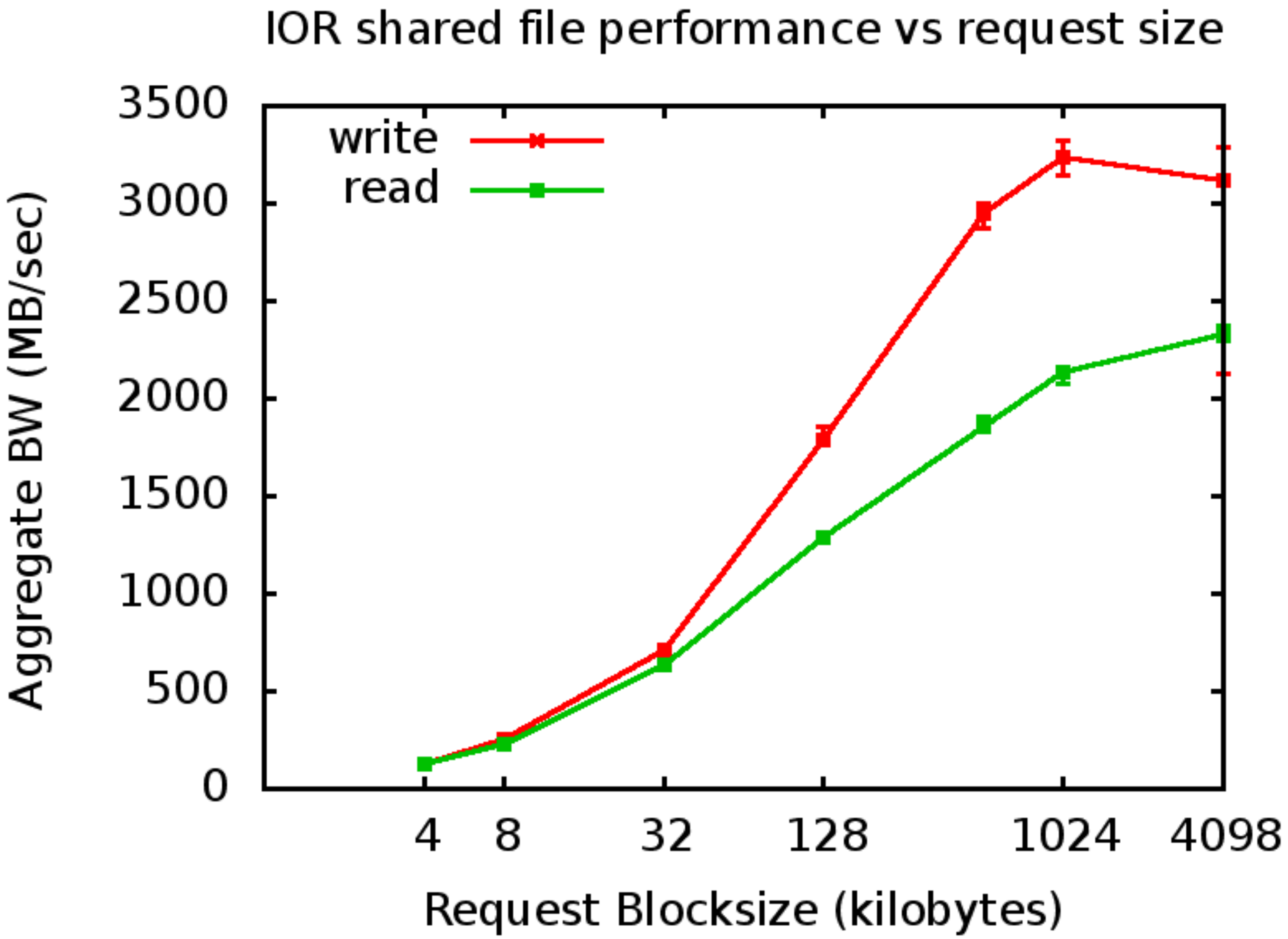
Compute nodes run application processes. Data model software also runs here, and some I/O transformations are performed here.

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.



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

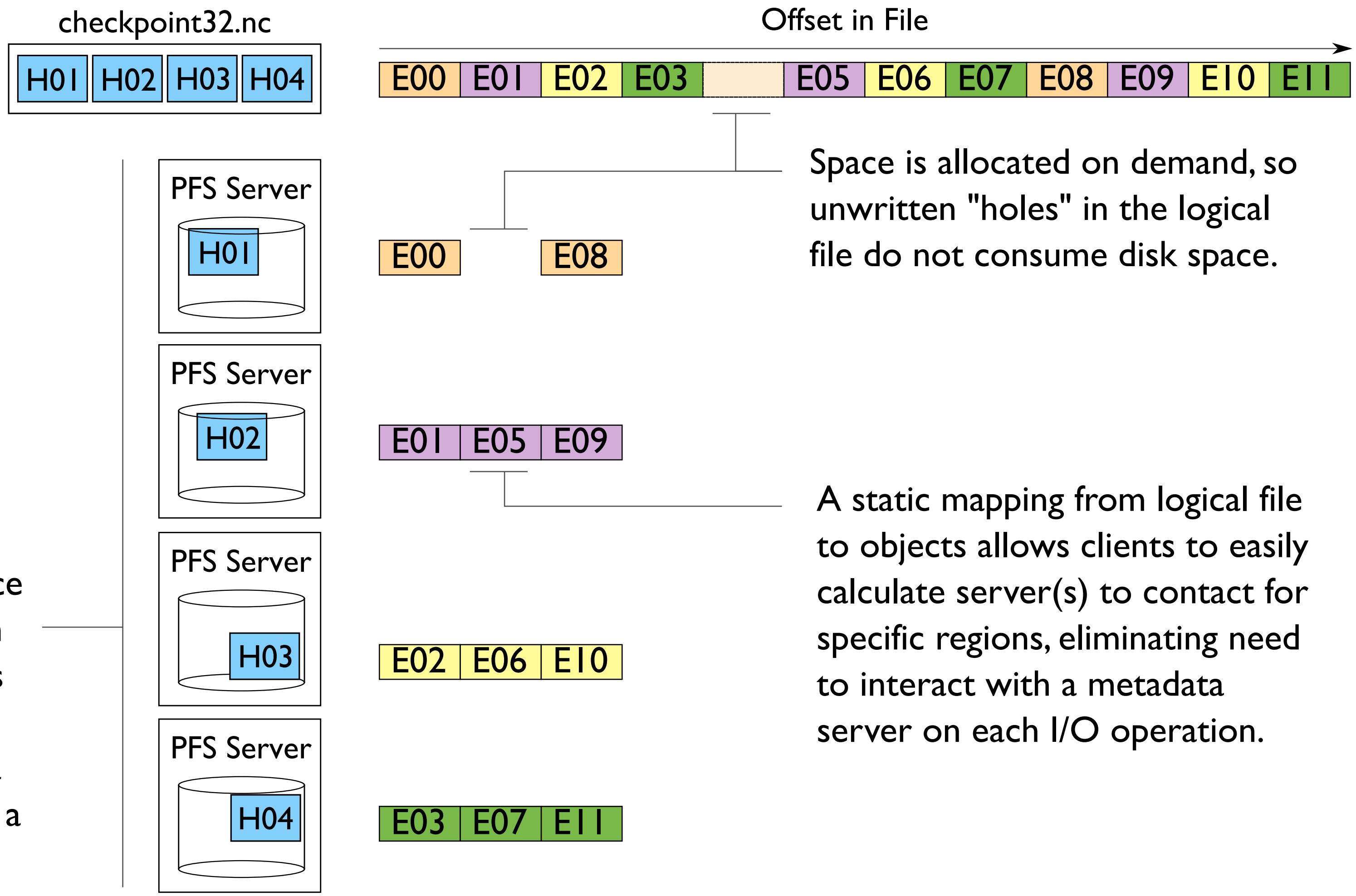
Data Distribution in Parallel File Systems

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.

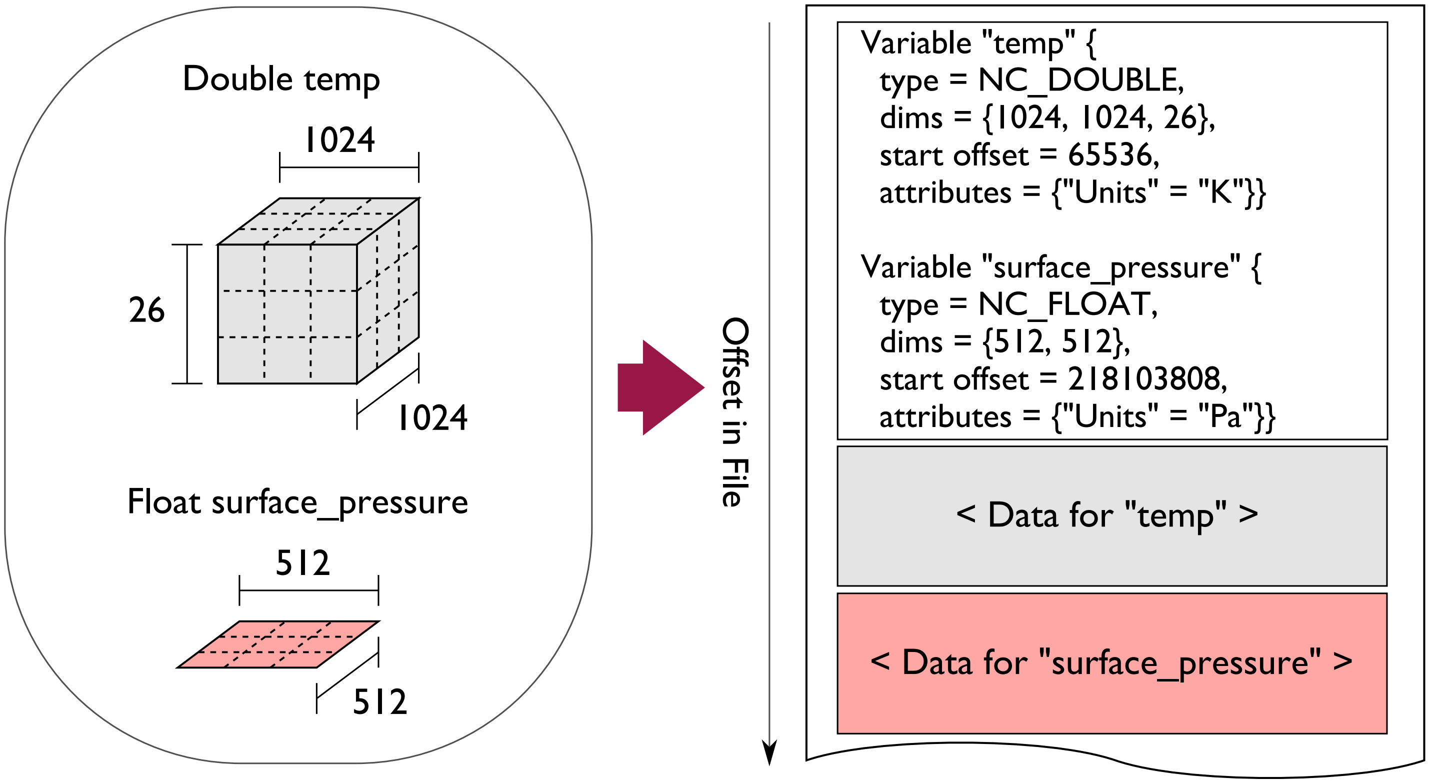
A static mapping from logical file to objects allows clients to easily calculate server(s) to contact for specific regions, eliminating need to interact with a metadata server on each I/O operation.

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"



```
Variable "temp" {
  type = NC_DOUBLE,
  dims = {1024, 1024, 26},
  start offset = 65536,
  attributes = {"Units" = "K"}}

Variable "surface_pressure" {
  type = NC_FLOAT,
  dims = {512, 512},
  start offset = 218103808,
  attributes = {"Units" = "Pa"}}
```

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

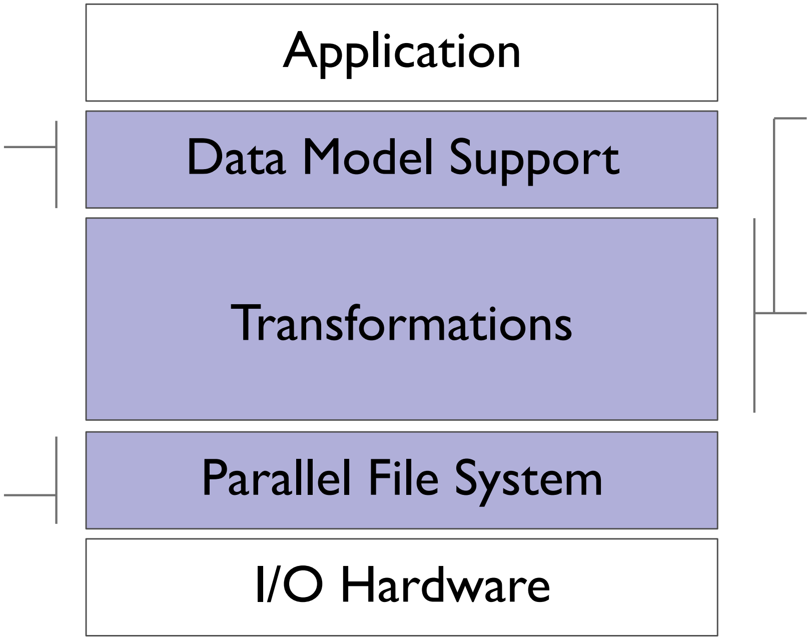
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack.*

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



Application

Data Model Support

Transformations

Parallel File System

I/O Hardware

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

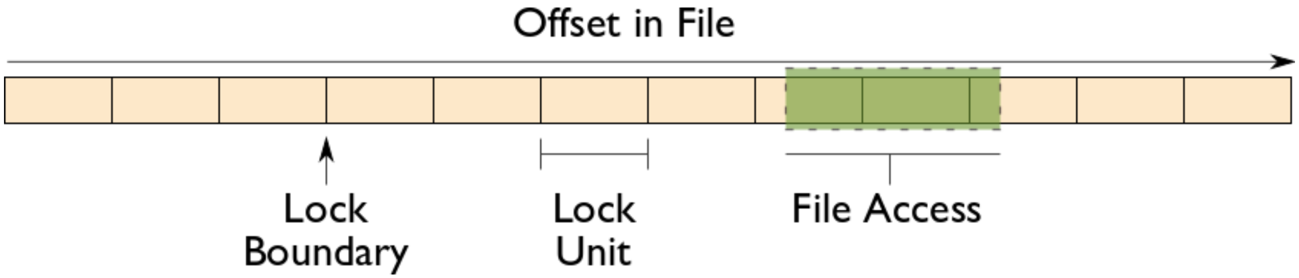
How It Works: HPC I/O Performance

Managing Concurrent Access

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

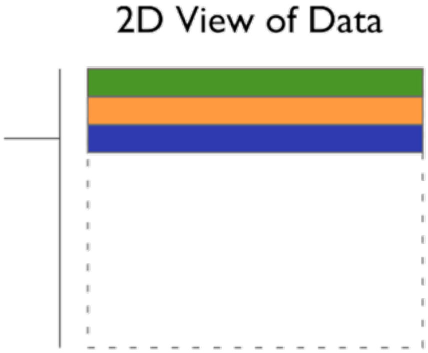
- Files are broken up into lock units
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

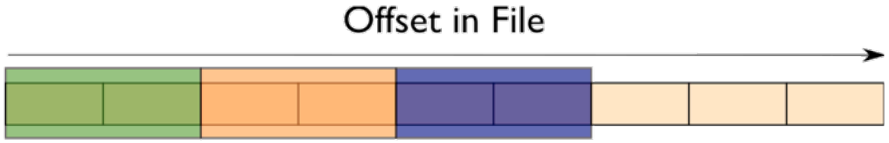
If an access touches any data in a lock unit, the lock for that region must be obtained before access occurs.



Implications of Locking in Concurrent Access

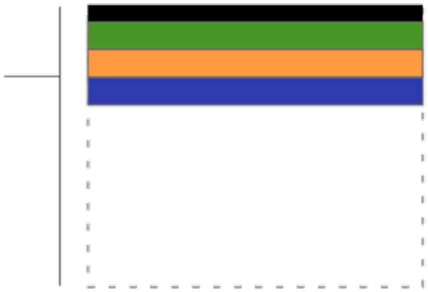
The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.

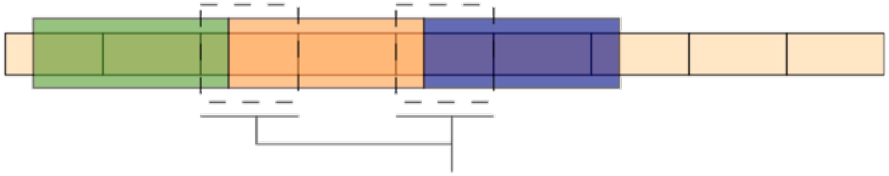




When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

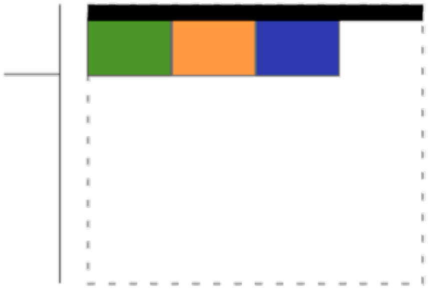
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

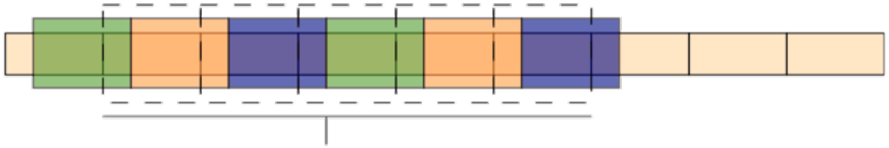




These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

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



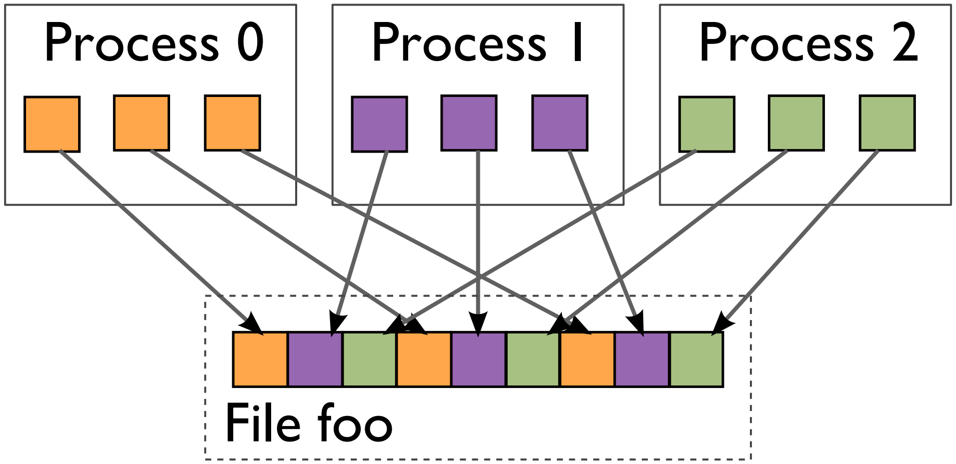


When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

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

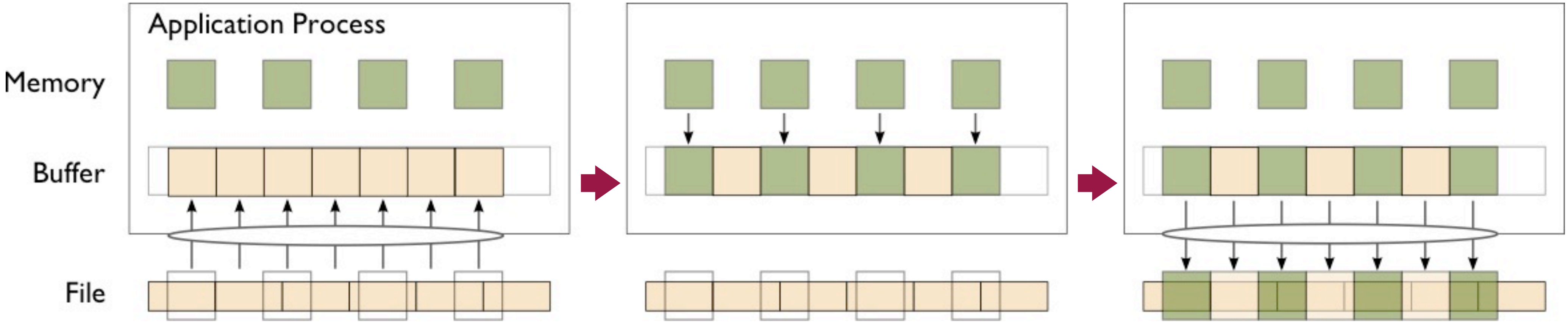


When we think about I/O transformations, we consider the mapping of data between application processes and locations in file.

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)



Step 1: Data in region to be modified are read into intermediate buffer (1 read).

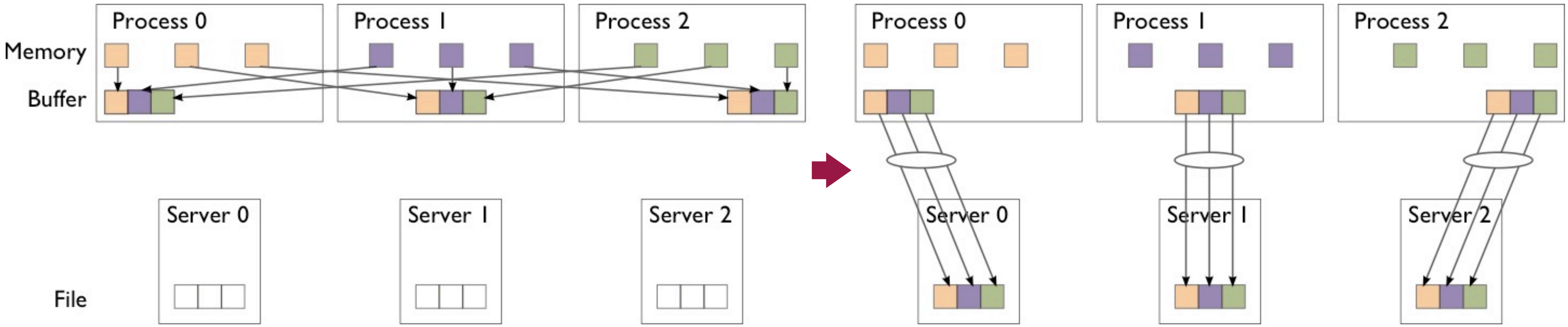
Step 2: Elements to be written to file are replaced in intermediate buffer.

Step 3: Entire region is written back to storage with a single write operation.

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
- 0th phase determines exchange schedule (not shown)

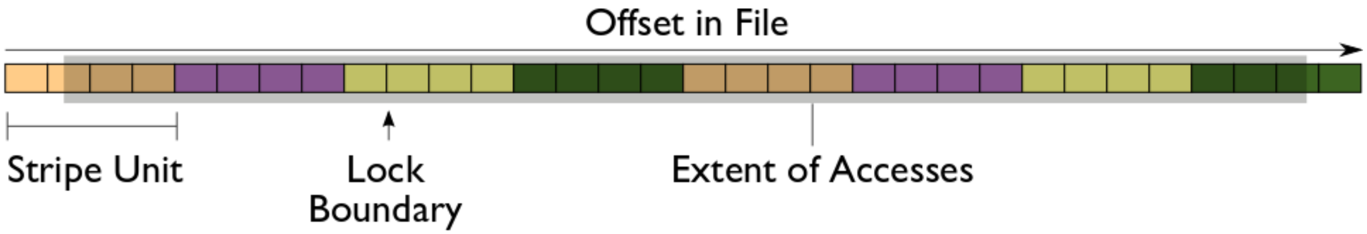


Phase 1: Data are exchanged between processes based on organization of data in file.

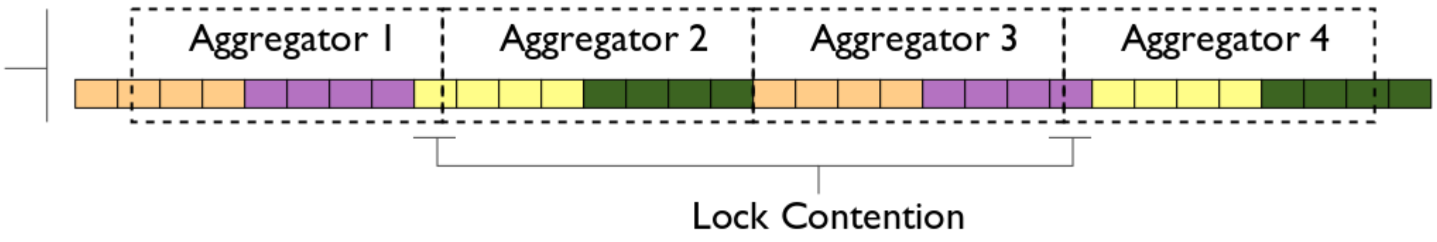
Phase 2: Data are written to file (storage servers) with large writes, no contention.

Two-Phase I/O Algorithms
(or, You don't want to do this yourself…)

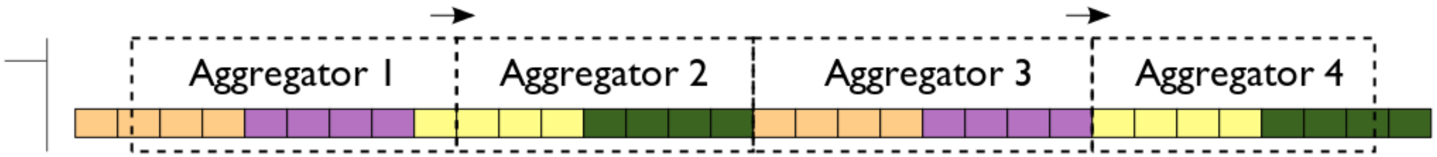
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



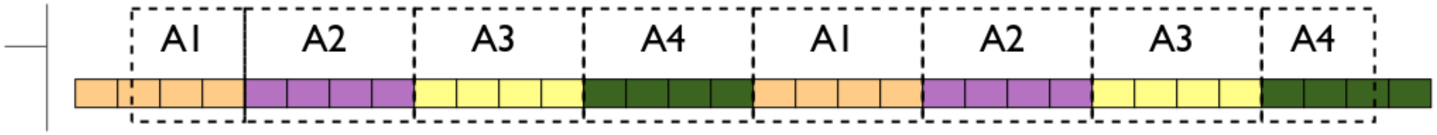
One approach is to evenly divide the region accessed across aggregators.



Aligning regions with lock boundaries eliminates lock contention.



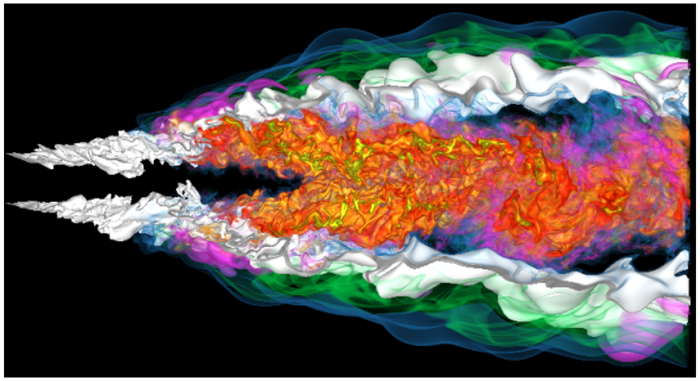
Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).

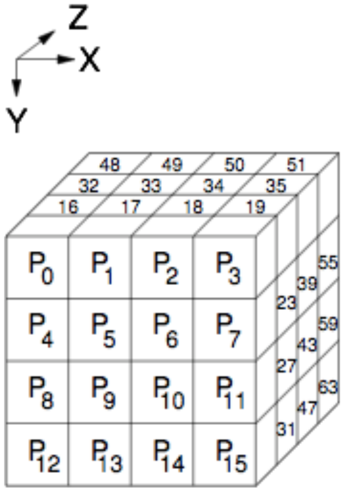


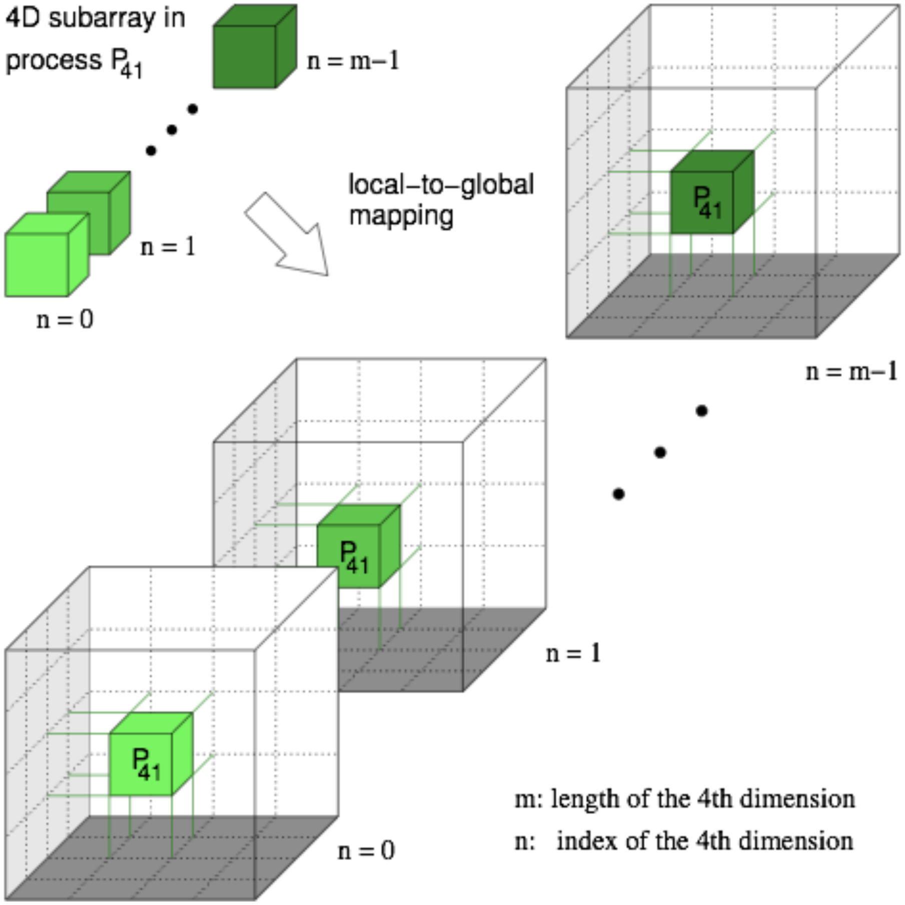
For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

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






4D subarray in process P41
n = m−1
n = 1
n = 0
local–to–global mapping
P41
P41
P41
n = m−1
n = 1
n = 0
m: length of the 4th dimension
n: index of the 4th dimension

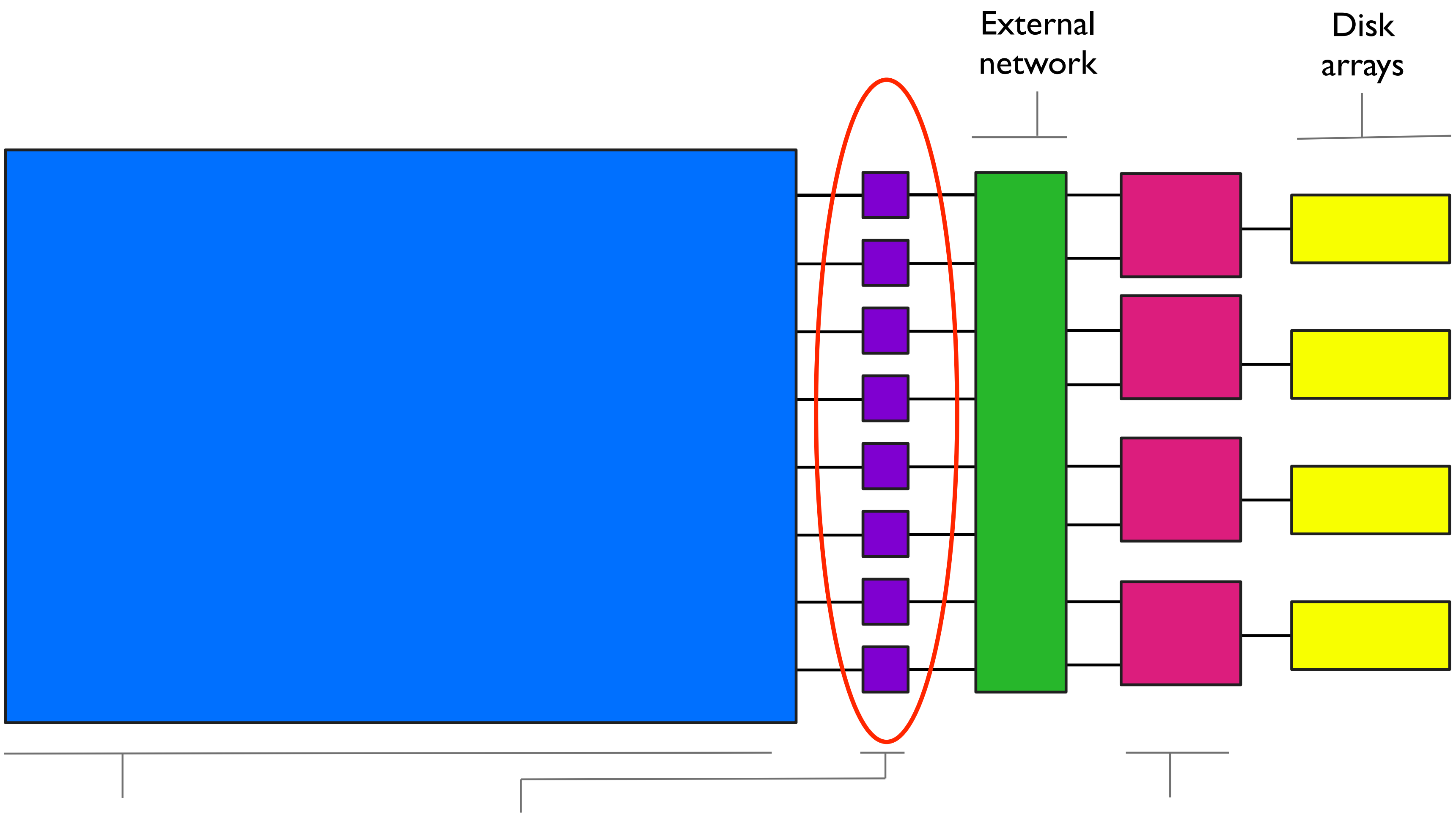
Thanks to Jackie Chen (SNL), Ray Grout (SNL), and Wei-Keng Liao (NWU) for providing the S3D I/O benchmark, Wei-Keng Liao for providing this diagram, C. Wang, H.Yu, and K.-L. Ma of UC Davis for image.

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	**5**
POSIX reads	0	80	0
MPI-IO writes	64	64	64
Unaligned in file	102,399	80	4
Total written (MB)	6.25	**87.11**	6.25
Runtime (sec)	1443	11	6.0
Avg. MPI-IO time per proc (sec)	**1426.47**	4.82	0.60

Transformations in the I/O Forwarding Step



Compute nodes

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

Storage nodes

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

- 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

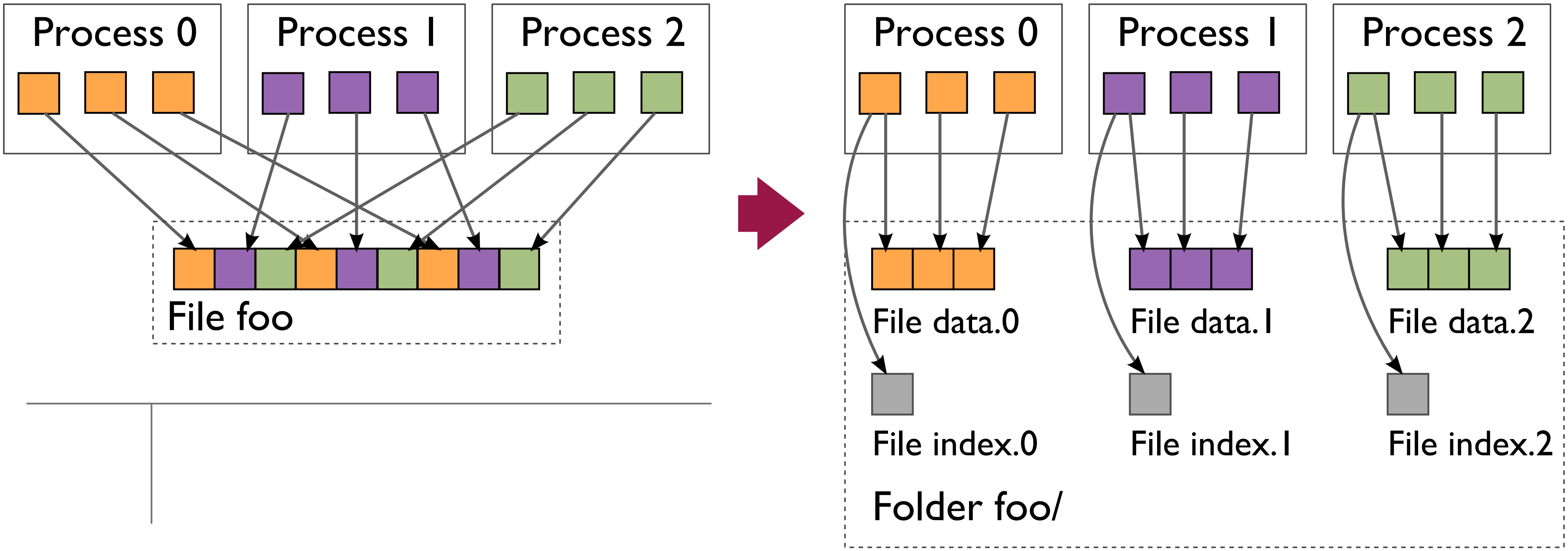
“Not So Transparent” Transformations

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

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.

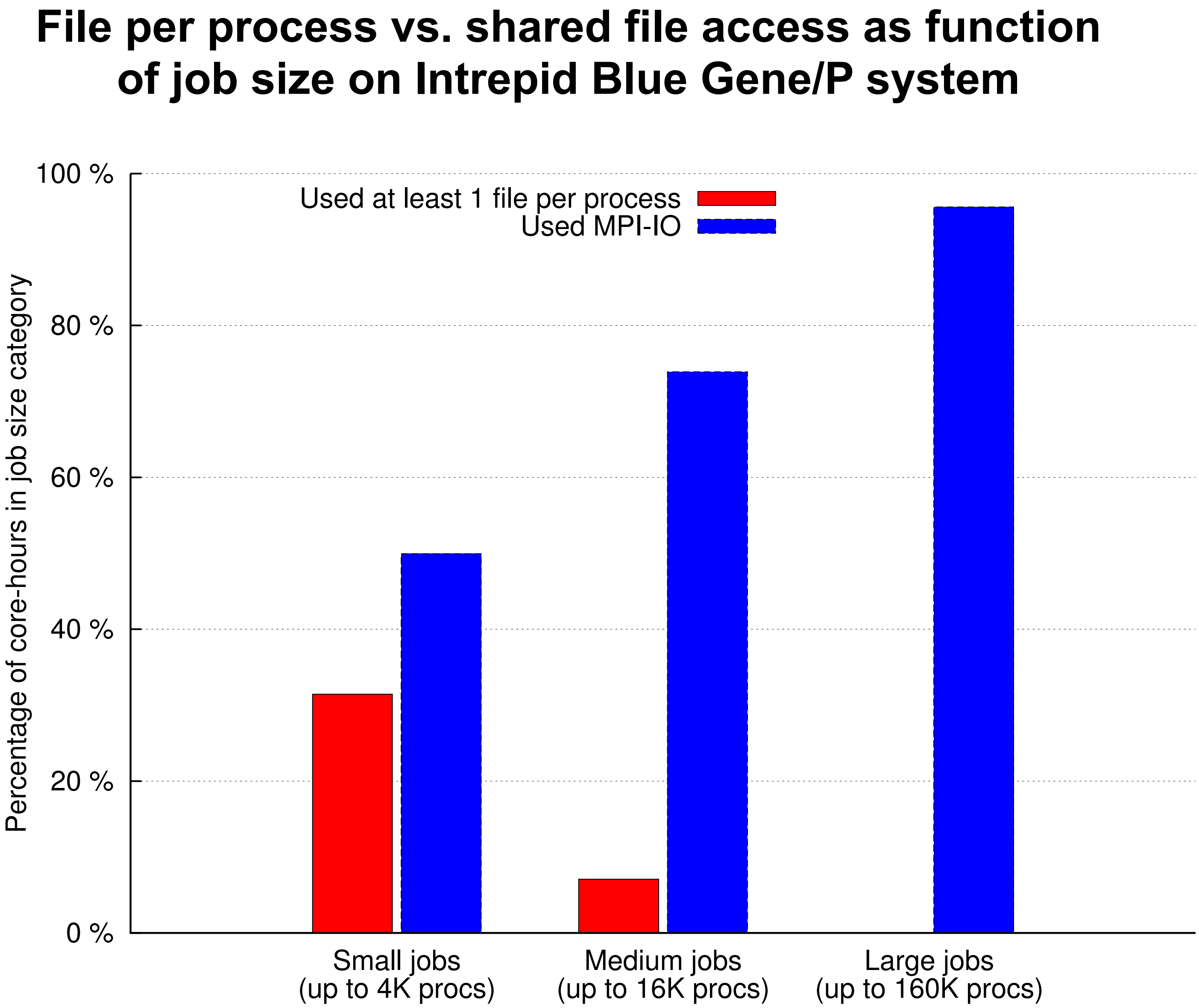
Transparent transformations such as data sieving and two-phase I/O preserve data order on the file system.

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?



I/O Transformations and the Storage Data Model

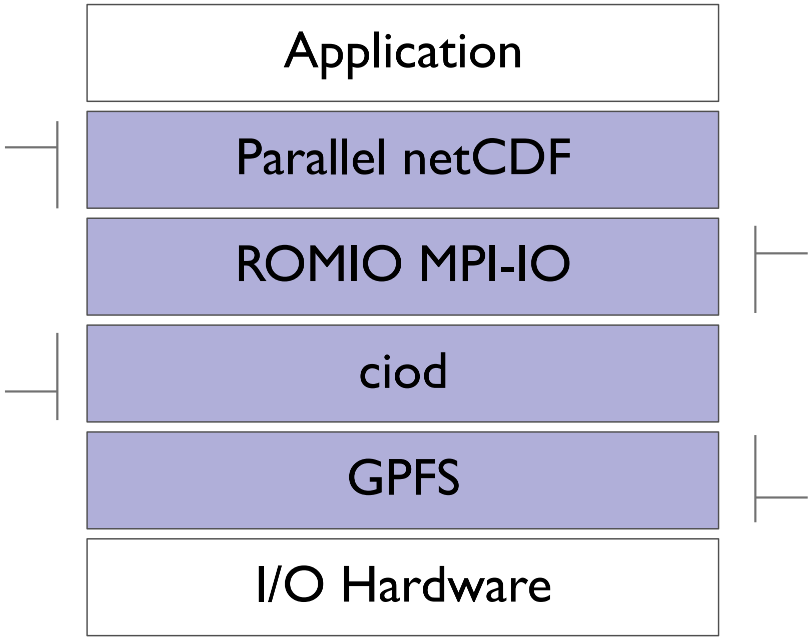
Historically, the storage data model has been the POSIX file model, and the PFS has been responsible for managing it.

- 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
  - Later will discuss efforts to 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.



Application

Parallel netCDF

ROMIO MPI-IO

ciod

GPFS

I/O Hardware

Parallel netCDF is used in numerous climate and weather applications running on DOE systems.

Built in collaboration with NWU.

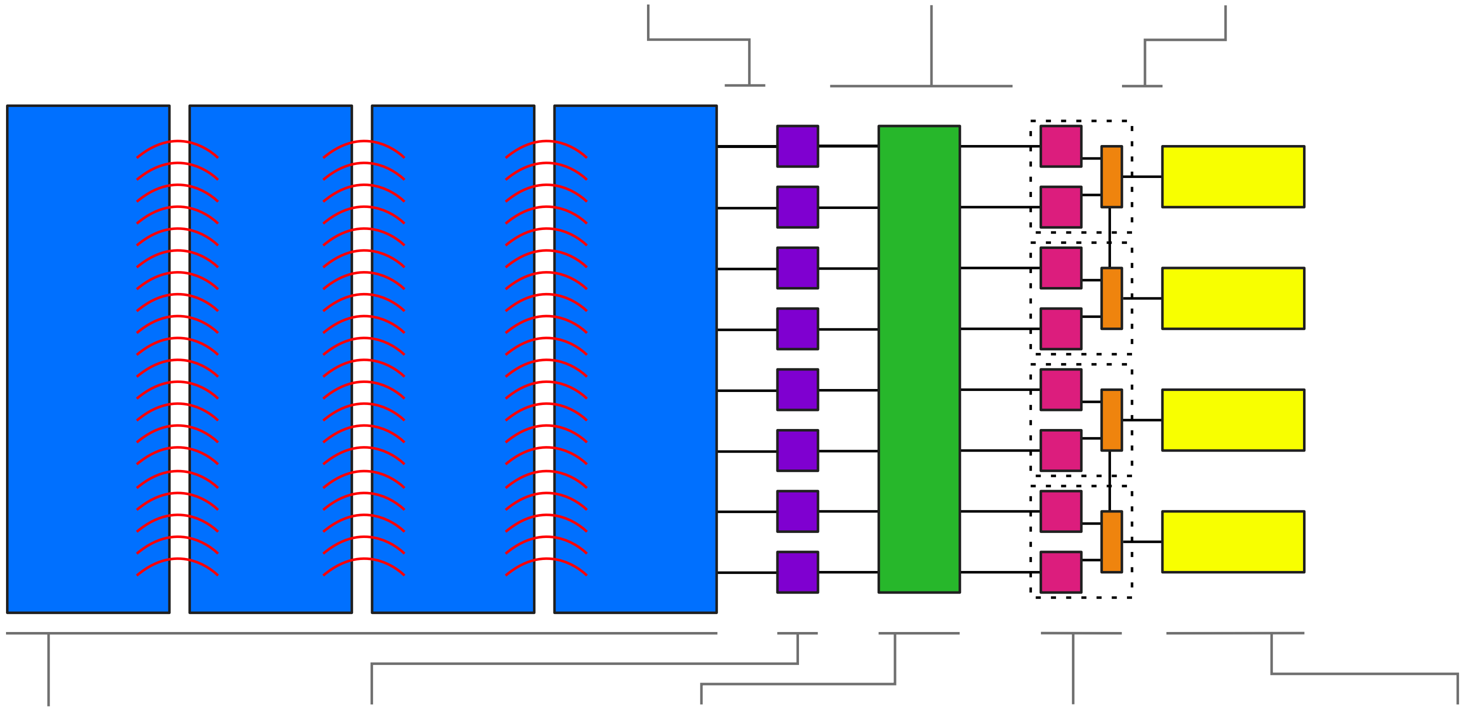
ROMIO is the basis for virtually all MPI-IO implementations on all platforms today and the starting point for nearly all MPI-IO research.

Incorporates research from NWU and patches from vendors.

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

GPFS is a production parallel file system provided by IBM.

Mira Blue Gene/Q and its Storage System



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

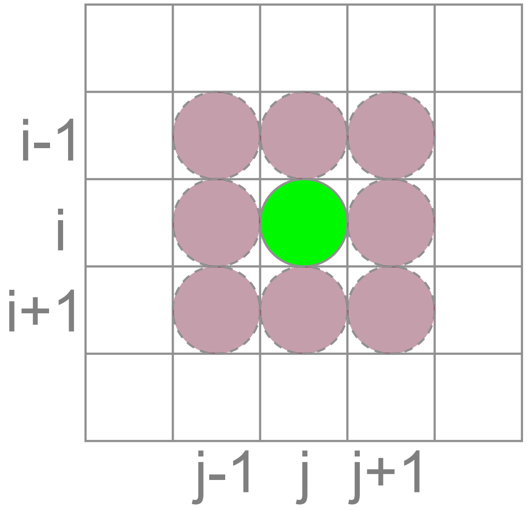
Conway's Game of Life

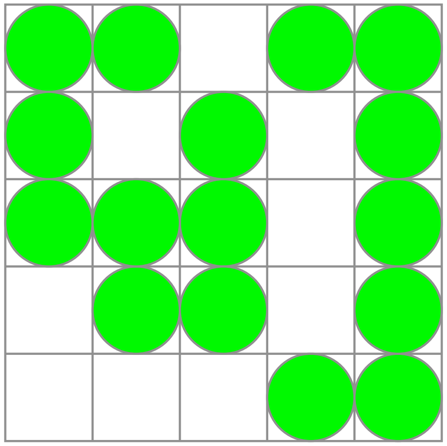
- We use Conway's Game of Life as a simple example to illustrate the program issues common to many codes that use regular meshes, such as PDE solvers
  - Allows us to concentrate on the I/O issues

- 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

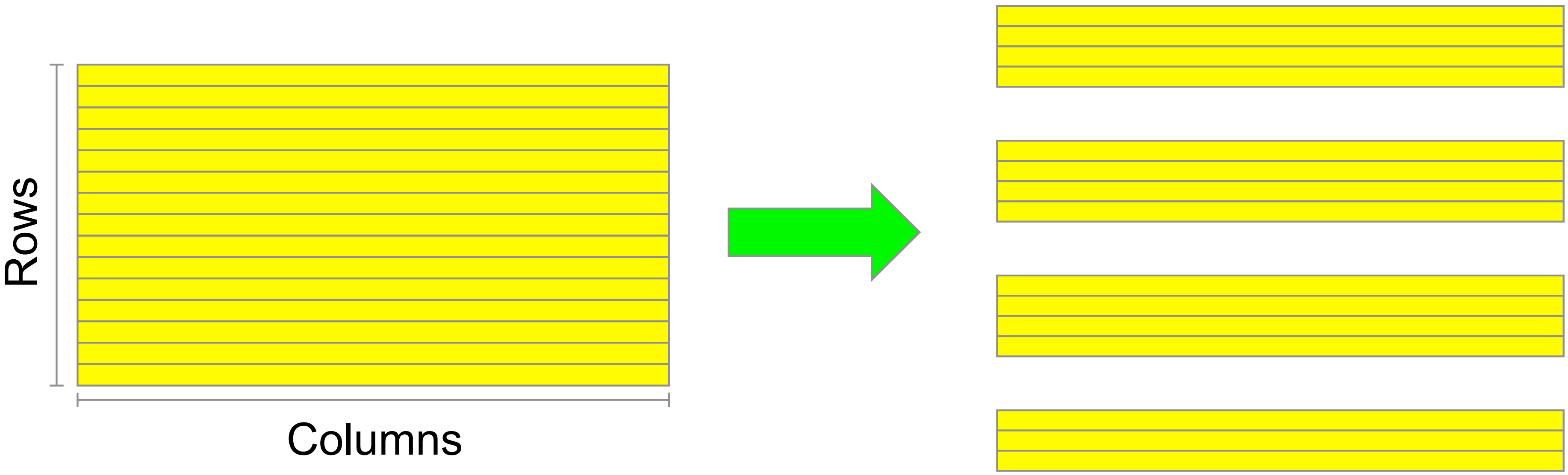
All code examples in this tutorial can be downloaded from www.mcs.anl.gov/mpi/tutorial/advmpi/mpi2tutorial.tar.gz

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

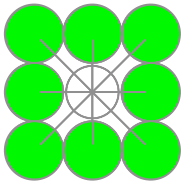
Step 1: Description of Decomposition

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

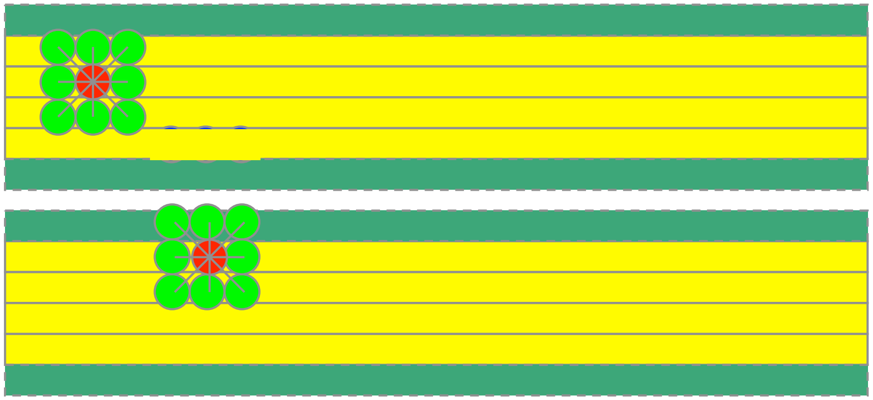


Step 2: Communication

- "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

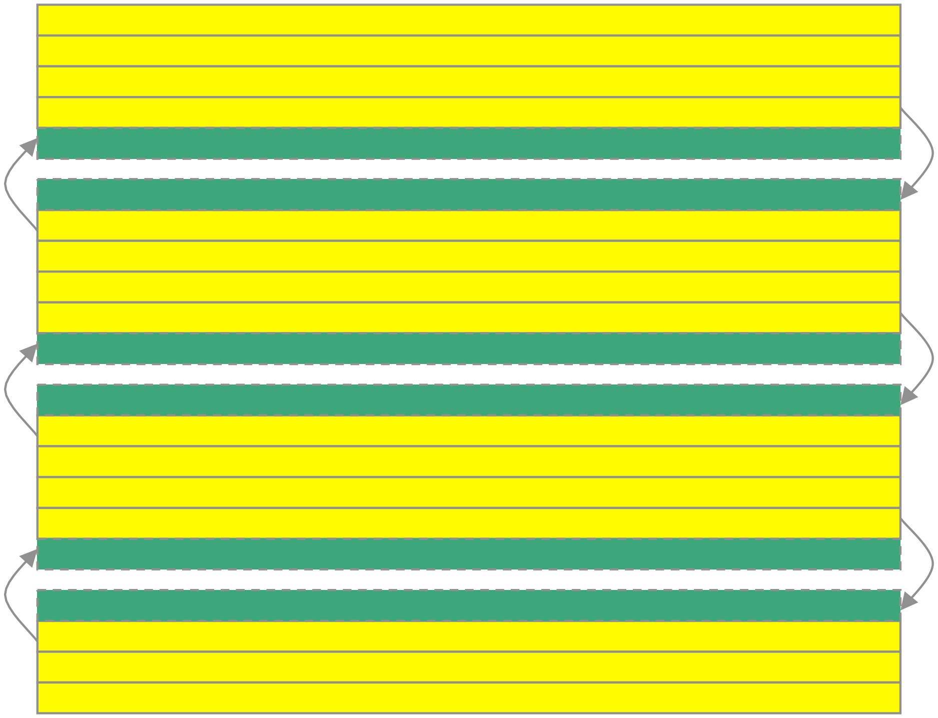


Step 3: Define the Local Arrays

- 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:                       ***           **                        ****
```

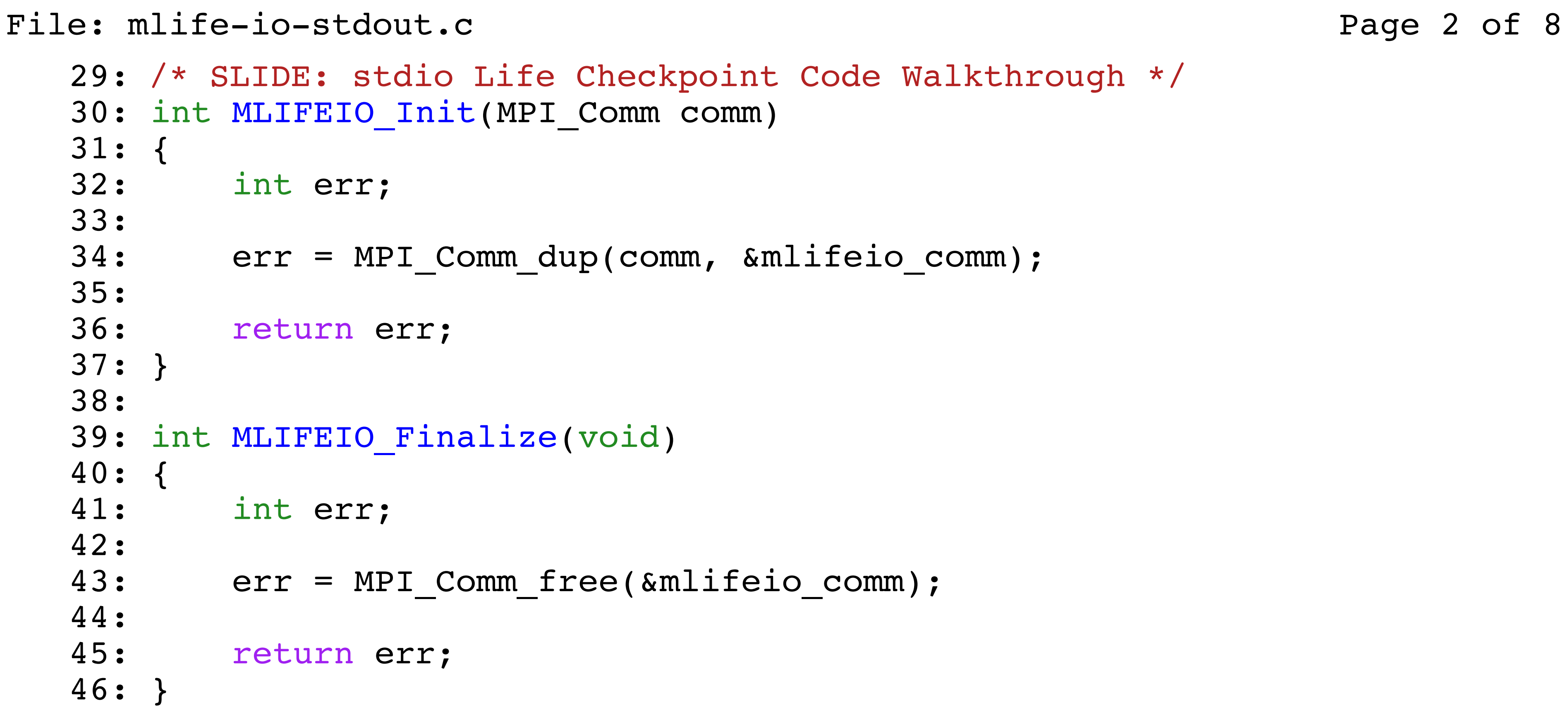
stdio Life Checkpoint Code Walkthrough

- 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.

File: mlife-io-stdout.c   Page 1 of 8

```c
/* SLIDE: stdio Life Checkpoint Code Walkthrough */
/* -*- Mode: C; c-basic-offset:4 ; -*- */
/*
 *  (C) 2004 by University of Chicago.
 *      See COPYRIGHT in top-level directory.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <mpi.h>

#include "mlife.h"
#include "mlife-io.h"

/* stdout implementation of checkpoint (no restart) for MPI Life
 *
 * Data output in matrix order: spaces represent dead cells,
 * '*'s represent live ones.
 */
static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
                                      int cols,
                                      MPI_Datatype *newtype);
static void MLIFEIO_Row_print(int *data, int cols, int rownr);
static void MLIFEIO_msleep(int msec);

static MPI_Comm mlifeio_comm = MPI_COMM_NULL;
```

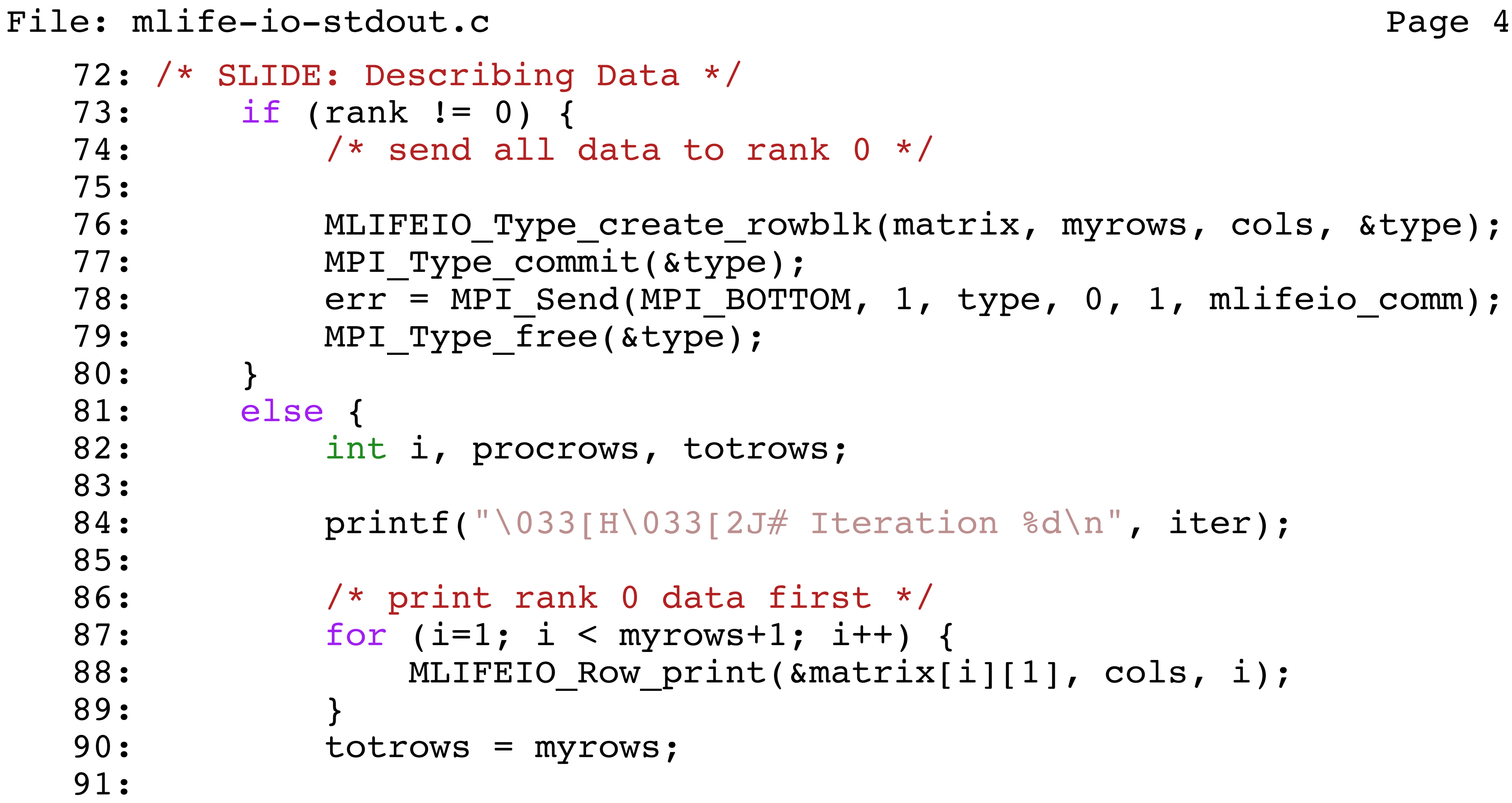


File: mlife-io-stdout.c Page 2 of 8

```c
/* SLIDE: stdio Life Checkpoint Code Walkthrough */
int MLIFEIO_Init(MPI_Comm comm)
{
    int err;

    err = MPI_Comm_dup(comm, &mlifeio_comm);

    return err;
}

int MLIFEIO_Finalize(void)
{
    int err;

    err = MPI_Comm_free(&mlifeio_comm);

    return err;
}
```

File: mlife-io-stdout.c Page 3 of 8

```c
/* SLIDE: Life stdout "checkpoint" */
/* MLIFEIO_Checkpoint
 *
 * Parameters:
 * prefix - prefix of file to hold checkpoint (ignored)
 * matrix - data values
 * rows   - number of rows in matrix
 * cols   - number of columns in matrix
 * iter   - iteration number of checkpoint
 * info   - hints for I/O (ignored)
 *
 * Returns MPI_SUCCESS on success, MPI error code on error.
 */
int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
                       int cols, int iter, MPI_Info info)
{
    int err = MPI_SUCCESS, rank, nprocs, myrows, myoffset;
    MPI_Datatype type;

    MPI_Comm_size(mlifeio_comm, &nprocs);
    MPI_Comm_rank(mlifeio_comm, &rank);

    myrows   = MLIFE_myrows(rows, rank, nprocs);
    myoffset = MLIFE_myrowoffset(rows, rank, nprocs);

```

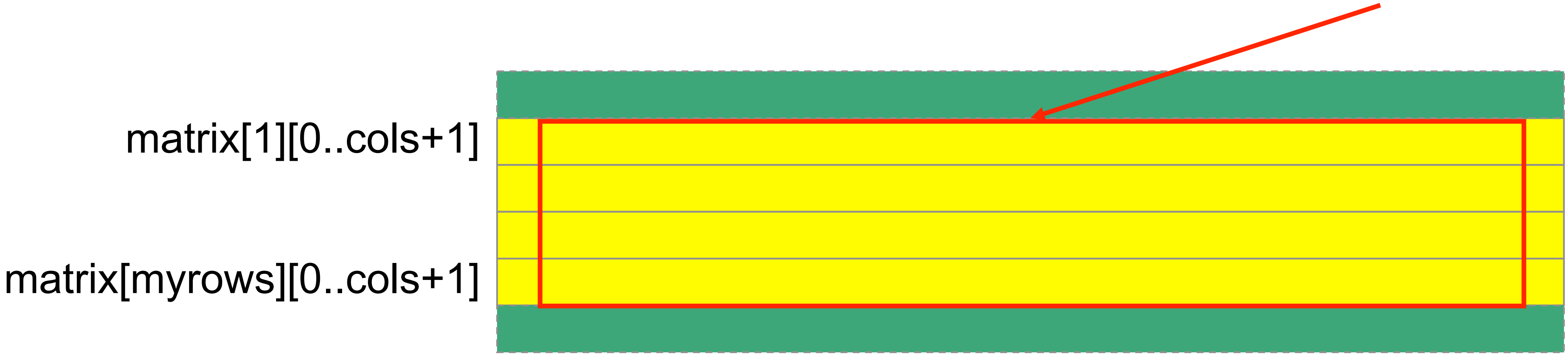


File: mlife-io-stdout.c Page 4 of 8

```
/* SLIDE: Describing Data */
    if (rank != 0) {
        /* send all data to rank 0 */

        MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
        MPI_Type_commit(&type);
        err = MPI_Send(MPI_BOTTOM, 1, type, 0, 1, mlifeio_comm);
        MPI_Type_free(&type);
    }
    else {
        int i, procrows, totrows;

        printf("\033[H\033[2J# Iteration %d\n", iter);

        /* print rank 0 data first */
        for (i=1; i < myrows+1; i++) {
            MLIFEIO_Row_print(&matrix[i][1], cols, i);
        }
        totrows = myrows;

/* SLIDE: Describing Data */
        /* receive and print others' data */
        for (i=1; i < nprocs; i++) {
                    int j, *data;

            procrows = MLIFE_myrows(rows, i, nprocs);
            data = (int *) malloc(procrows * cols * sizeof(int));

            err = MPI_Recv(data, procrows * cols, MPI_INT, i, 1,
                           mlifeio_comm, MPI_STATUS_IGNORE);

            for (j=0; j < procrows; j++) {
                MLIFEIO_Row_print(&data[j * cols], cols,
                                  totrows + j + 1);
            }
            totrows += procrows;

            free(data);
        }
    }

    MLIFEIO_msleep(250); /* give time to see the results */

    return err;
}
```

Describing Data

Need to save this region in the array



- 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);
```

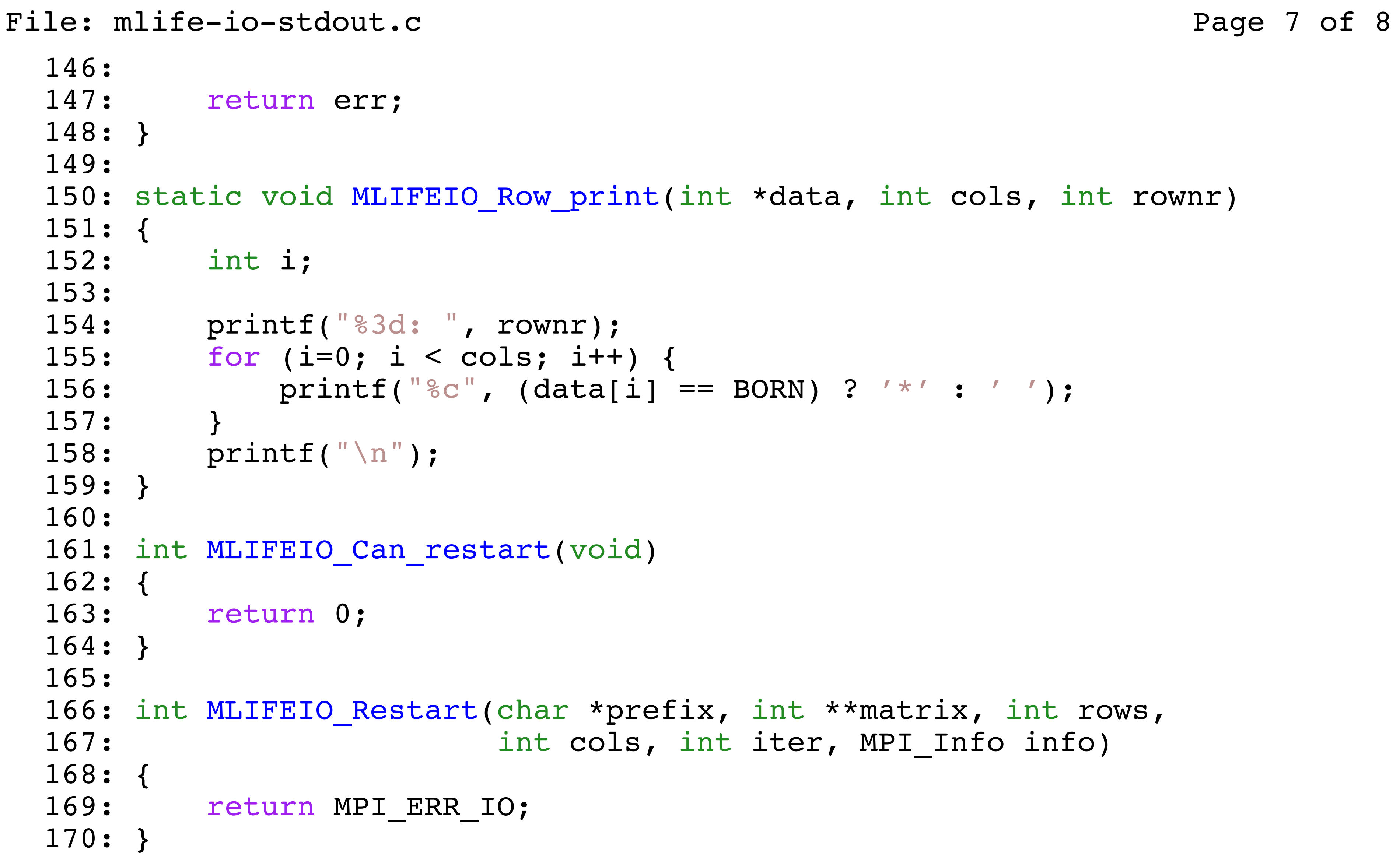
  - Second type gets memory offset right (allowing use of MPI_BOTTOM in MPI_File_write_all)

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

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

File: mlife-io-stdout.c   Page 6 of 8

```c
/* SLIDE: Describing Data */
/* MLIFEIO_Type_create_rowblk
 *
 * Creates a MPI_Datatype describing the block of rows of data
 * for the local process, not including the surrounding boundary
 * cells.
 *
 * Note: This implementation assumes that the data for matrix is
 *       allocated as one large contiguous block!
 */
static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
                                      int cols,
                                      MPI_Datatype *newtype)
{
    int err, len;
    MPI_Datatype vectype;
    MPI_Aint disp;

    /* since our data is in one block, access is very regular! */
    err = MPI_Type_vector(myrows, cols, cols+2, MPI_INT,
                          &vectype);
    if (err != MPI_SUCCESS) return err;

    /* wrap the vector in a type starting at the right offset */
    len = 1;
    MPI_Address(&matrix[1][1], &disp);
    err = MPI_Type_hindexed(1, &len, &disp, vectype, newtype);

    MPI_Type_free(&vectype); /* decrement reference count */
```



File: mlife-io-stdout.c Page 7 of 8

```c

    return err;
}

static void MLIFEIO_Row_print(int *data, int cols, int rownr)
{
    int i;

    printf("%3d: ", rownr);
    for (i=0; i < cols; i++) {
        printf("%c", (data[i] == BORN) ? '*' : ' ');
    }
    printf("\n");
}

int MLIFEIO_Can_restart(void)
{
    return 0;
}

int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
                    int cols, int iter, MPI_Info info)
{
    return MPI_ERR_IO;
}
```

File: mlife-io-stdout.c Page 8 of 8

```c

#ifdef HAVE_NANOSLEEP
#include <time.h>
static void MLIFEIO_msleep(int msec)
{
    struct timespec t;


    t.tv_sec = msec / 1000;
    t.tv_nsec = 1000000 * (msec - t.tv_sec);

    nanosleep(&t, NULL);
}
#else
static void MLIFEIO_msleep(int msec)
{
    if (msec < 1000) {
        sleep(1);
    }
    else {
        sleep(msec / 1000);
    }
}
#endif
```

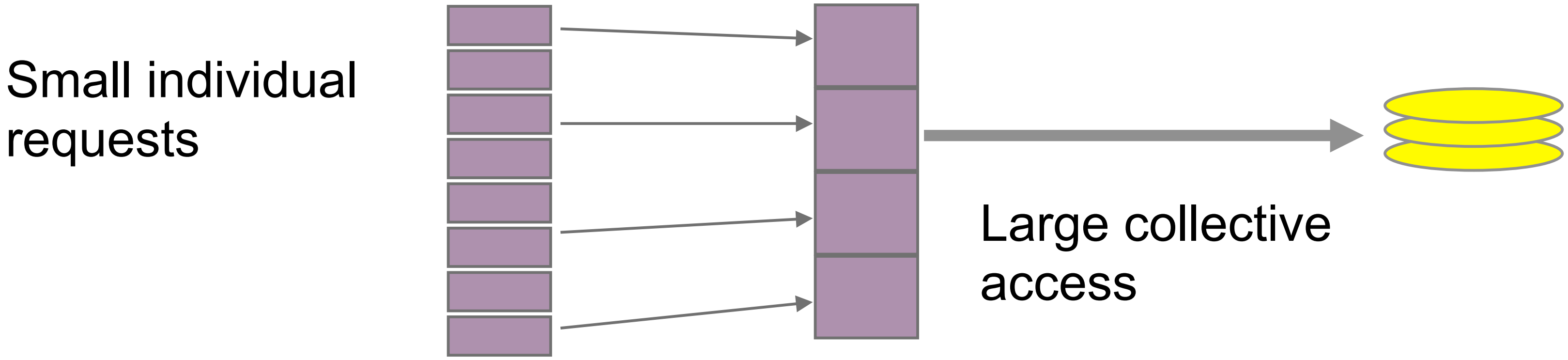
Parallelizing our I/O API

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

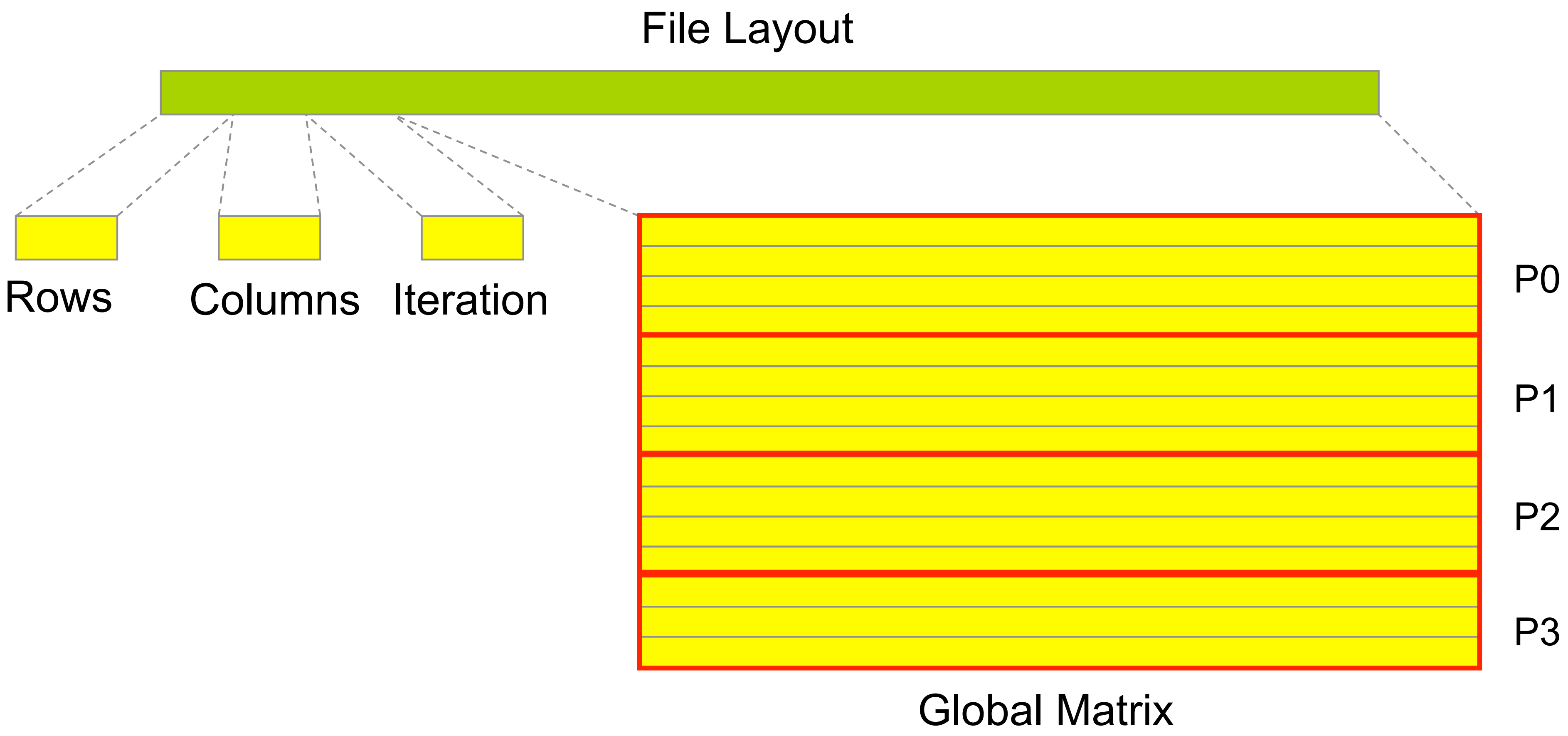
- **MPI_File_write_at_all**, etc.
  - **_all** indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
  - **_at** indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than 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-mpiio.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-mpiio.c pp. 1-9 for code example.

File: mlife-io-mpiio.c   Page 1 of 9

```c
/* SLIDE: MPI-IO Life Checkpoint Code Walkthrough */
/* -*- Mode: C; c-basic-offset:4 ; -*- */
/*
 *
 *  (C) 2004 by University of Chicago.
 *      See COPYRIGHT in top-level directory.
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "mlife-io.h"

/* MPI-IO implementation of checkpoint and restart for MPI Life
 *
 * Data stored in matrix order, with a header consisting of three
 * integers: matrix size in rows and columns, and iteration no.
 *
 * Each checkpoint is stored in its own file.
 */
static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
                                      int cols,
                                      MPI_Datatype *newtype);
static int MLIFEIO_Type_create_hdr_rowblk(int **matrix,
                                          int myrows,
                                          int *rows_p,
                                          int *cols_p,
                                          int *iter_p,
                                          MPI_Datatype *newtype);
```

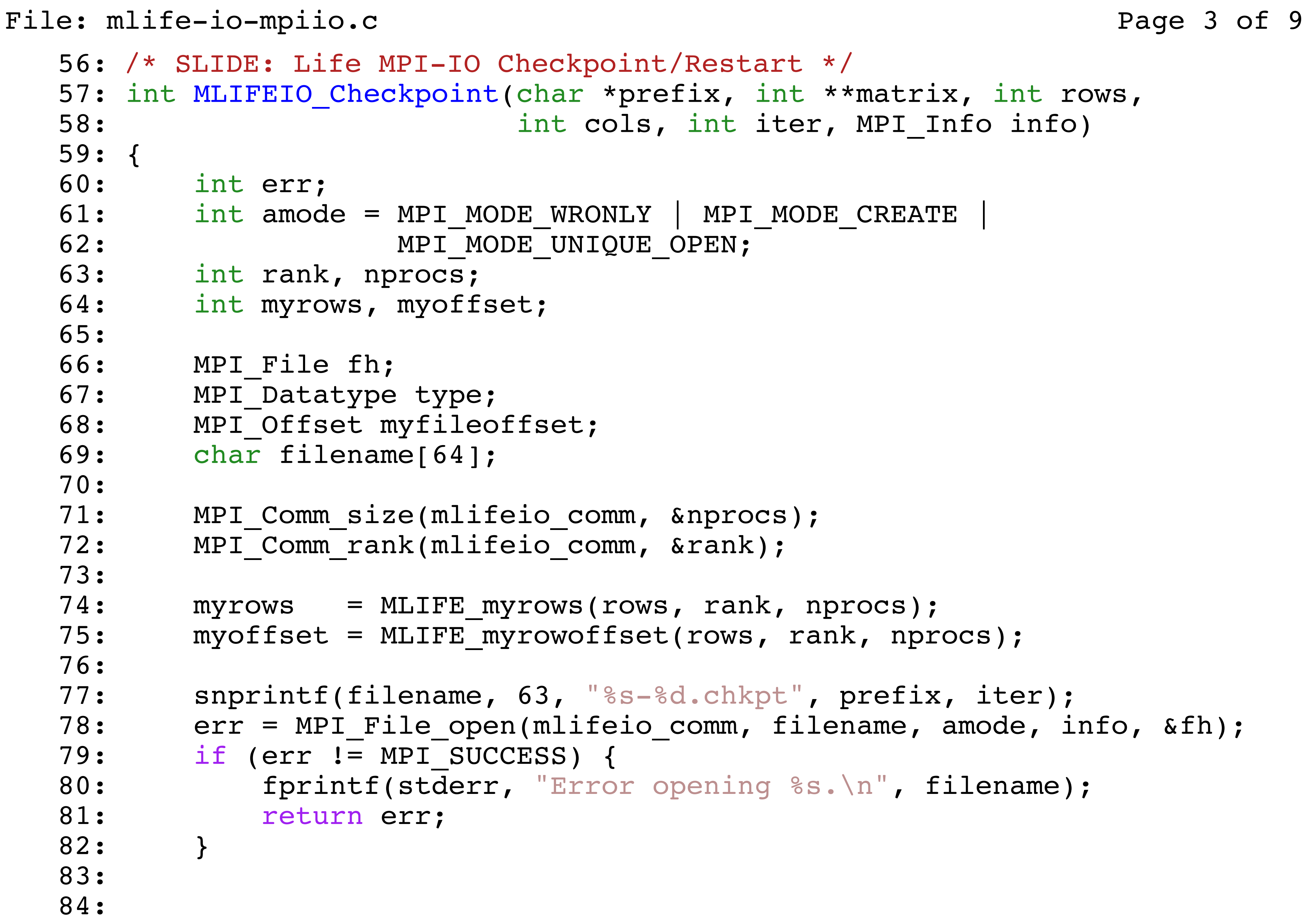
File: mlife-io-mpiio.c   Page 2 of 9

```
/* SLIDE: MPI-IO Life Checkpoint Code Walkthrough */
static MPI_Comm mlifeio_comm = MPI_COMM_NULL;

int MLIFEIO_Init(MPI_Comm comm)
{
    int err;

    err = MPI_Comm_dup(comm, &mlifeio_comm);

    return err;
}

int MLIFEIO_Finalize(void)
{
    int err;

    err = MPI_Comm_free(&mlifeio_comm);

    return err;
}

int MLIFEIO_Can_restart(void)
{
    return 1;
}

```

Life MPI-IO Checkpoint/Restart

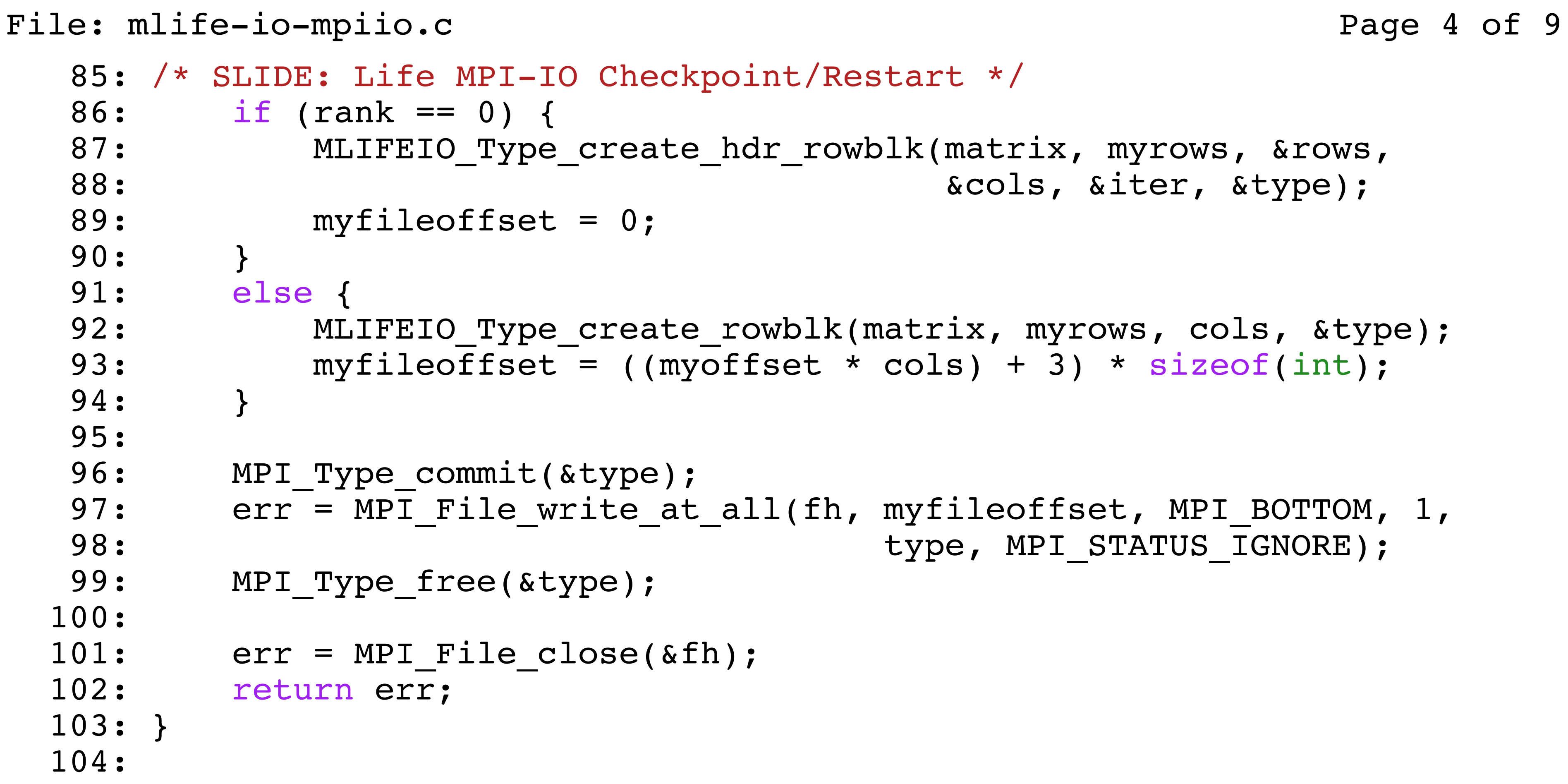
- We can map our collective checkpoint directly to a single collective MPI-IO file write: MPI_File_write_at_all
  - Process 0 writes a little extra (the header)
- On restart, two steps are performed:
  - Everyone reads the number of rows and columns from the header in the file with MPI_File_read_at_all
    - 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-mpiio.c pp. 3-6 for code example.



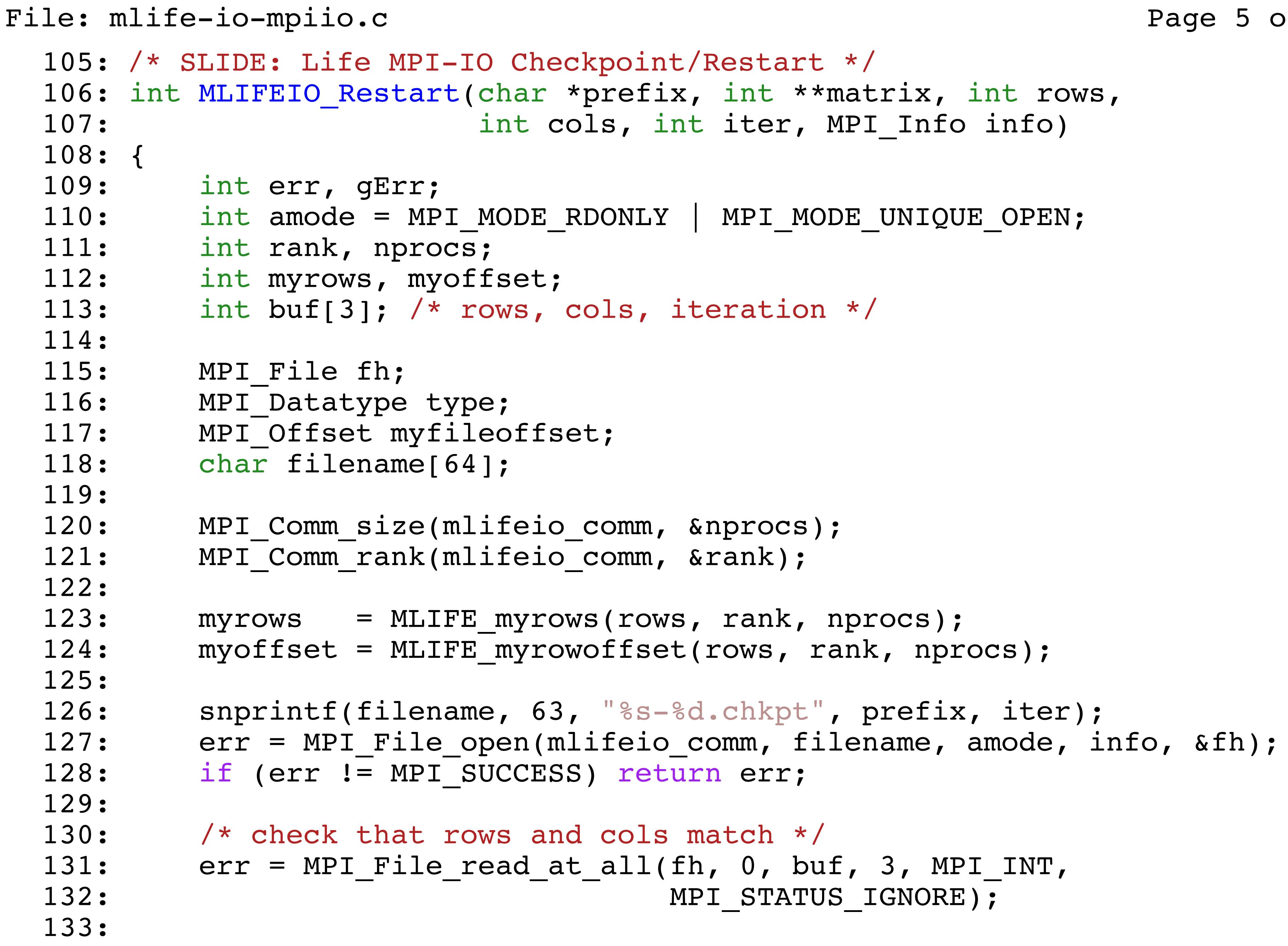
File: mlife-io-mpiio.c   Page 3 of 9

```
/* SLIDE: Life MPI-IO Checkpoint/Restart */
int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
                       int cols, int iter, MPI_Info info)
{
    int err;
    int amode = MPI_MODE_WRONLY | MPI_MODE_CREATE |
                MPI_MODE_UNIQUE_OPEN;
    int rank, nprocs;
    int myrows, myoffset;

    MPI_File fh;
    MPI_Datatype type;
    MPI_Offset myfileoffset;
    char filename[64];

    MPI_Comm_size(mlifeio_comm, &nprocs);
    MPI_Comm_rank(mlifeio_comm, &rank);

    myrows   = MLIFE_myrows(rows, rank, nprocs);
    myoffset = MLIFE_myrowoffset(rows, rank, nprocs);

    snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
    err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
    if (err != MPI_SUCCESS) {
        fprintf(stderr, "Error opening %s.\n", filename);
        return err;
    }


```



File: mlife-io-mpiio.c   Page 4 of 9

```c
/* SLIDE: Life MPI-IO Checkpoint/Restart */
    if (rank == 0) {
        MLIFEIO_Type_create_hdr_rowblk(matrix, myrows, &rows,
                                       &cols, &iter, &type);
        myfileoffset = 0;
    }
    else {
        MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
        myfileoffset = ((myoffset * cols) + 3) * sizeof(int);
    }

    MPI_Type_commit(&type);
    err = MPI_File_write_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
                                    type, MPI_STATUS_IGNORE);
    MPI_Type_free(&type);

    err = MPI_File_close(&fh);
    return err;
}

```



File: mlife-io-mpiio.c Page 5 of 9

```c
/* SLIDE: Life MPI-IO Checkpoint/Restart */
int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
                    int cols, int iter, MPI_Info info)
{
    int err, gErr;
    int amode = MPI_MODE_RDONLY | MPI_MODE_UNIQUE_OPEN;
    int rank, nprocs;
    int myrows, myoffset;
    int buf[3]; /* rows, cols, iteration */

    MPI_File fh;
    MPI_Datatype type;
    MPI_Offset myfileoffset;
    char filename[64];

    MPI_Comm_size(mlifeio_comm, &nprocs);
    MPI_Comm_rank(mlifeio_comm, &rank);

    myrows   = MLIFE_myrows(rows, rank, nprocs);
    myoffset = MLIFE_myrowoffset(rows, rank, nprocs);

    snprintf(filename, 63, "%s-%d.chkpt", prefix, iter);
    err = MPI_File_open(mlifeio_comm, filename, amode, info, &fh);
    if (err != MPI_SUCCESS) return err;

    /* check that rows and cols match */
    err = MPI_File_read_at_all(fh, 0, buf, 3, MPI_INT,
                               MPI_STATUS_IGNORE);

```

File: mlife-io-mpiio.c   Page 6 of 9

```
/* SLIDE: Life MPI-IO Checkpoint/Restart */
    /* Have all process check that nothing went wrong */
    MPI_Allreduce(&err, &gErr, 1, MPI_INT, MPI_MAX, mlifeio_comm);
    if (gErr || buf[0] != rows || buf[1] != cols) {
        if (rank == 0) fprintf(stderr, "restart failed.\n");
        return MPI_ERR_OTHER;
    }

    MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
    myfileoffset = ((myoffset * cols) + 3) * sizeof(int);

    MPI_Type_commit(&type);
    err = MPI_File_read_at_all(fh, myfileoffset, MPI_BOTTOM, 1,
                               type, MPI_STATUS_IGNORE);
    MPI_Type_free(&type);

    err = MPI_File_close(&fh);
    return err;
}

```

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-mpiio.c pp. 7 for code example.

File: mlife-io-mpiio.c Page 7 of 9

```c
/* SLIDE: Describing Header and Data */
/* MLIFEIO_Type_create_hdr_rowblk
 *
 * Used by process zero to create a type that describes both
 * the header data for a checkpoint and its contribution to
 * the stored matrix.
 *
 * Parameters:
 * matrix  - pointer to the matrix, including boundaries
 * myrows  - number of rows held locally
 * rows_p  - pointer to # of rows in matrix (so we can get its
 *           address for use in the type description)
 * cols_p  - pointer to # of cols in matrix
 * iter_p  - pointer to iteration #
 * newtype - pointer to location to store new type ref.
 */
static int MLIFEIO_Type_create_hdr_rowblk(int **matrix,
                                          int myrows,
                                          int *rows_p,
                                          int *cols_p,
                                          int *iter_p,
                                          MPI_Datatype *newtype)
{
    int err;
    int lens[4] = { 1, 1, 1, 1 };
    MPI_Aint disps[4];
    MPI_Datatype types[4];
    MPI_Datatype rowblk;

```

File: mlife-io-mpiio.c Page 8 of 9

```c
/* SLIDE: Describing Header and Data */
    MLIFEIO_Type_create_rowblk(matrix, myrows, *cols_p, &rowblk);

    MPI_Address(rows_p, &disps[0]);
    MPI_Address(cols_p, &disps[1]);
    MPI_Address(iter_p, &disps[2]);
    disps[3] = (MPI_Aint) MPI_BOTTOM;
    types[0] = MPI_INT;
    types[1] = MPI_INT;
    types[2] = MPI_INT;
    types[3] = rowblk;

#if defined(MPI_VERSION) && MPI_VERSION >= 2
    err = MPI_Type_create_struct(3, lens, disps, types, newtype);
#else
    err = MPI_Type_struct(3, lens, disps, types, newtype);
#endif

    MPI_Type_free(&rowblk);

    return err;
}

```

File: mlife-io-mpiio.c Page 9 of 9

```c
/* SLIDE: Placing Data in Checkpoint */
/* MLIFEIO_Type_create_rowblk
 *
 * See stdio version for details (this is a copy).
 */
static int MLIFEIO_Type_create_rowblk(int **matrix, int myrows,
                                      int cols,
                                      MPI_Datatype *newtype)
{
    int err, len;
    MPI_Datatype vectype;

    MPI_Aint disp;

    /* since our data is in one block, access is very regular */
    err = MPI_Type_vector(myrows, cols, cols+2, MPI_INTEGER,
                          &vectype);
    if (err != MPI_SUCCESS) return err;

    /* wrap the vector in a type starting at the right offset */
    len = 1;
    MPI_Address(&matrix[1][1], &disp);
    err = MPI_Type_hindexed(1, &len, &disp, vectype, newtype);

    MPI_Type_free(&vectype); /* decrement reference count */

    return err;
}
```

Building an I/O API for Sparse Matrices

Sparse Matrix Operations

- A typical operation is a matrix-vector multiply
- Consider an example where the sparse matrix is produced by one application and you wish to use a parallel program to solve the linear system

Sparse Matrix Format

n – number of rows/cols (matrix dimensions)
nz – number of nonzero elements
ia[0..n] – index into data for first element in row i
ja[0..nz-1] – column location for element j
a[0..nz-1] – actual data

(0, 0, 0, 0, 4
1, 0, 3, 0, 0
5, 2, 0, 0, 8
0, 6, 7, 0, 0
0, 0, 0, 9, 0)

→

n = 5
nz = 9
ia[] = (0, 1, 3, 6, 8, 9)
ja[] = (4, 0, 2, 0, 1, 4, 1, 2, 3)
a[] = (4, 1, 3, 5, 2, 8, 6, 7, 9)

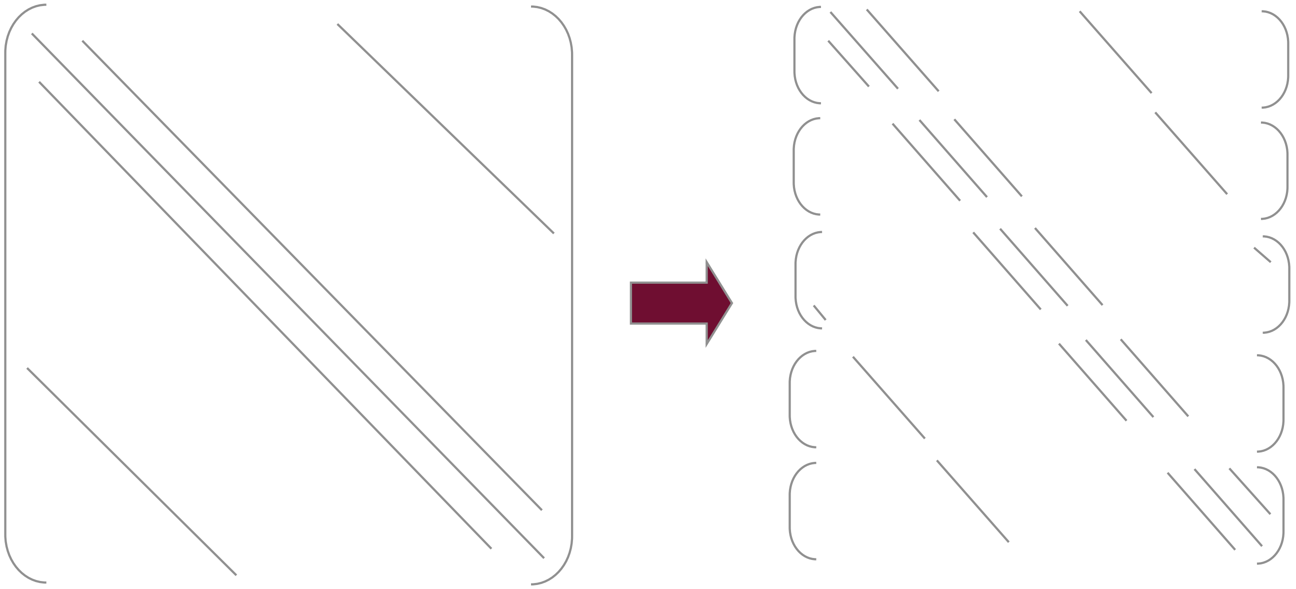
(known as CSR or AIJ format)
Note: Format isn’t a win for a matrix of this size and density.

Steps in Designing the Parallel Version

- Same as our other examples:
  - Decomposition
  - Communication (for the matrix-vector product)
  - Define the local representation

Step 1: Description of Decomposition

- By rows (matches equations)
- In practice, the diagonal block and off-diagonal block are stored separately
  - For simplicity, we will ignore this



Step 2: Communication

- For matrix-vector product, we would need
  - Elements of vector (also distributed in the same way as the matrix) from other processes corresponding to columns in which there are non-zero entries
- Can be implemented with send and receive or with RMA
  - For simplicity, we will not show this part of the code

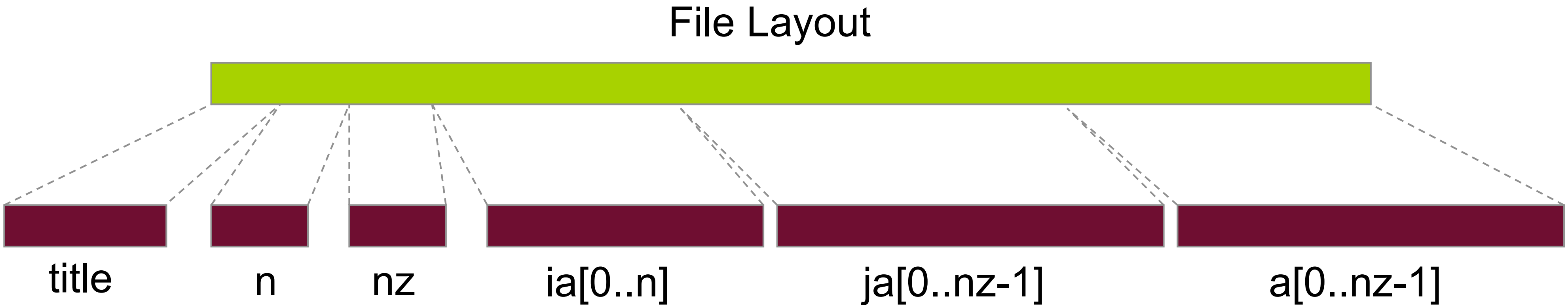
Step 3: Define the Local Arrays

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

I/O in Sparse Matrix Codes

- Define the file format
- We want the file to be independent of the number of processes
- File requires:
  - Header information
    - Size of matrix, number of non-zeros
    - Name of matrix
  - ia, ja, and A vectors

Placing Data in Checkpoint



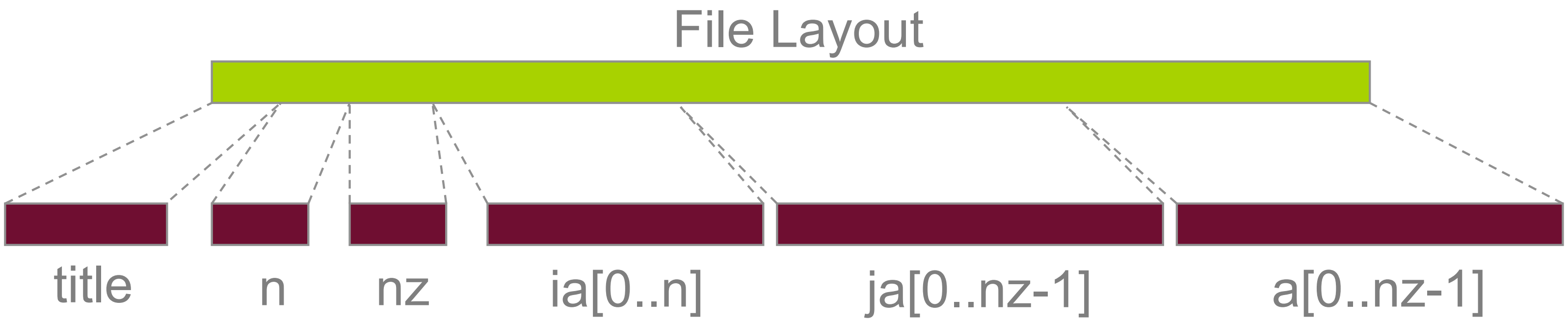
- Unlike data layout in the Life case, positioning of data for a given process depends on the values held by other processes (number of nonzero values)!
- Each process has pieces that are spread out in the file (noncontiguous!)

stdio CSRIO Code Walkthrough

- Points to observe
  - MPI_Exscan and MPI_Allreduce to discover starting locations and complete sizes of vectors
  - Passing data to rank 0 for printing
  - Converting ia from local to global references

See csrio-stdout.c pp. 1-2 for code example.

Writing Sparse Matrices (stdout)



- Steps:
  - MPI_Exscan to get count of nonzeros from all previous processes
    - gives starting offset in ja[] and a[] arrays and value to add to ia[] elements
  - MPI_Allreduce to get total count of nonzeros (nz) – gives size of ja[] and a[] arrays
  - Process zero writes header (title, n, nz)
  - Copy ia[] and adjust to refer to global matrix locations
  - Pass data back to rank zero for printing

See csrio-stdout.c pp. 3-8 for code example.

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

Data Model Libraries

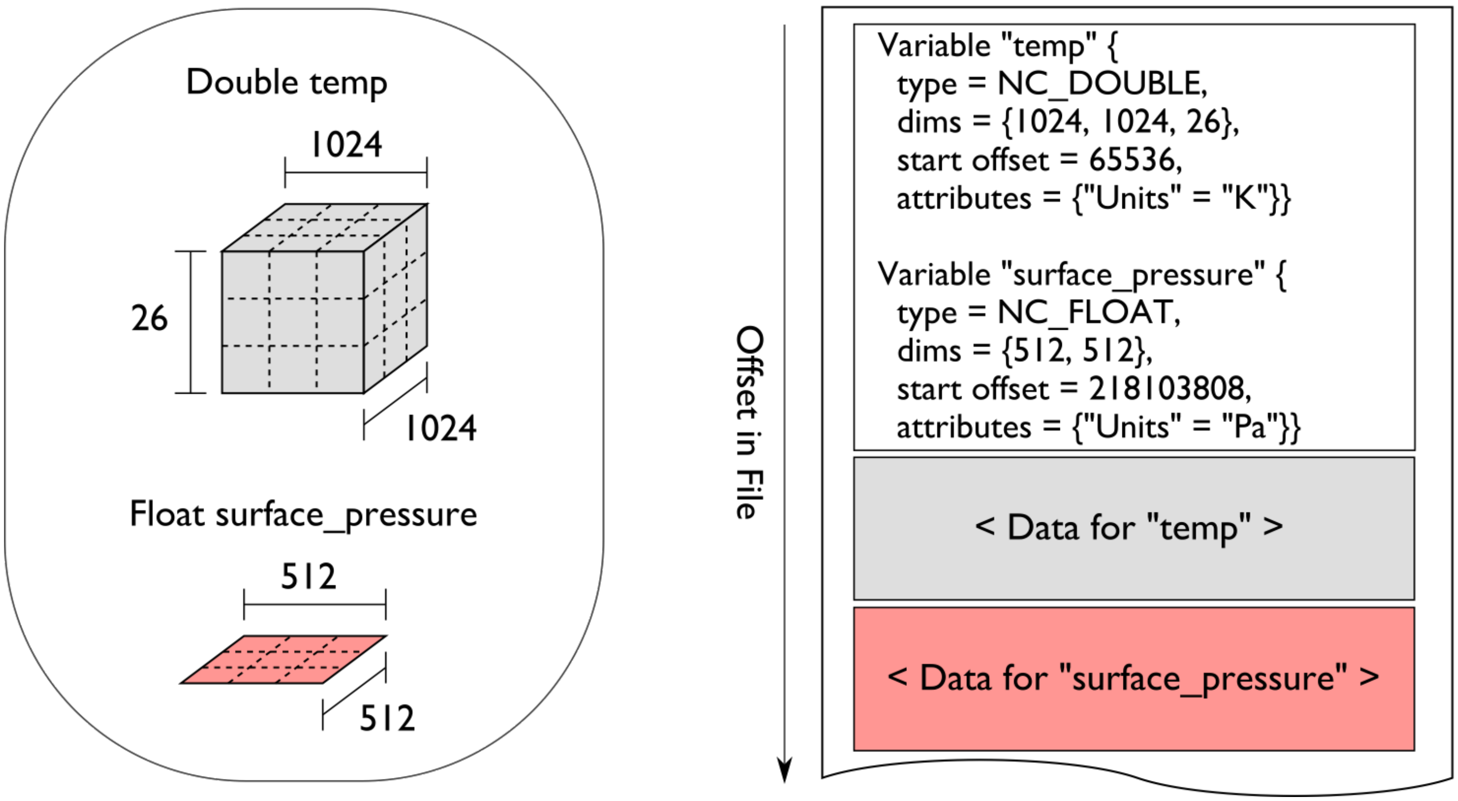
- Scientific applications work with structured data and desire more self-describing file formats
- PnetCDF and HDF5 are two popular “higher level” I/O libraries
  - Abstract away details of file layout
  - Provide standard, portable file formats
  - Include metadata describing contents
- For parallel 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"



```
Variable "temp" {
  type = NC_DOUBLE,
  dims = {1024, 1024, 26},
  start offset = 65536,
  attributes = {"Units" = "K"}}

Variable "surface_pressure" {
  type = NC_FLOAT,
  dims = {512, 512},
  start offset = 218103808,
  attributes = {"Units" = "Pa"}}
```

< Data for "temp" >

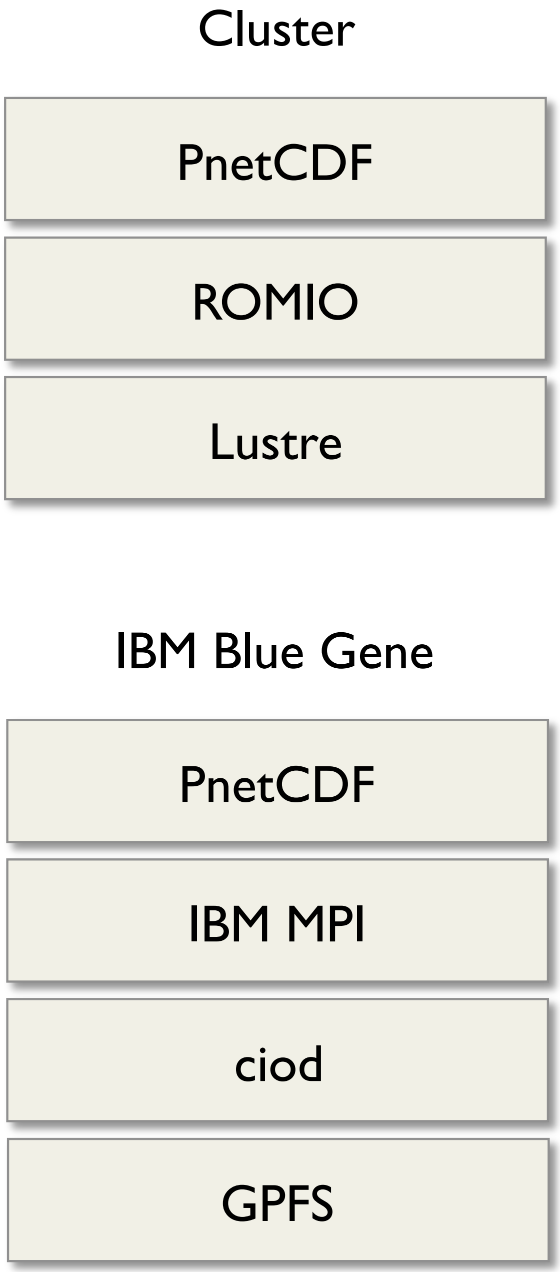
< Data for "surface_pressure" >

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

PnetCDF

ROMIO

Lustre

IBM Blue Gene

PnetCDF

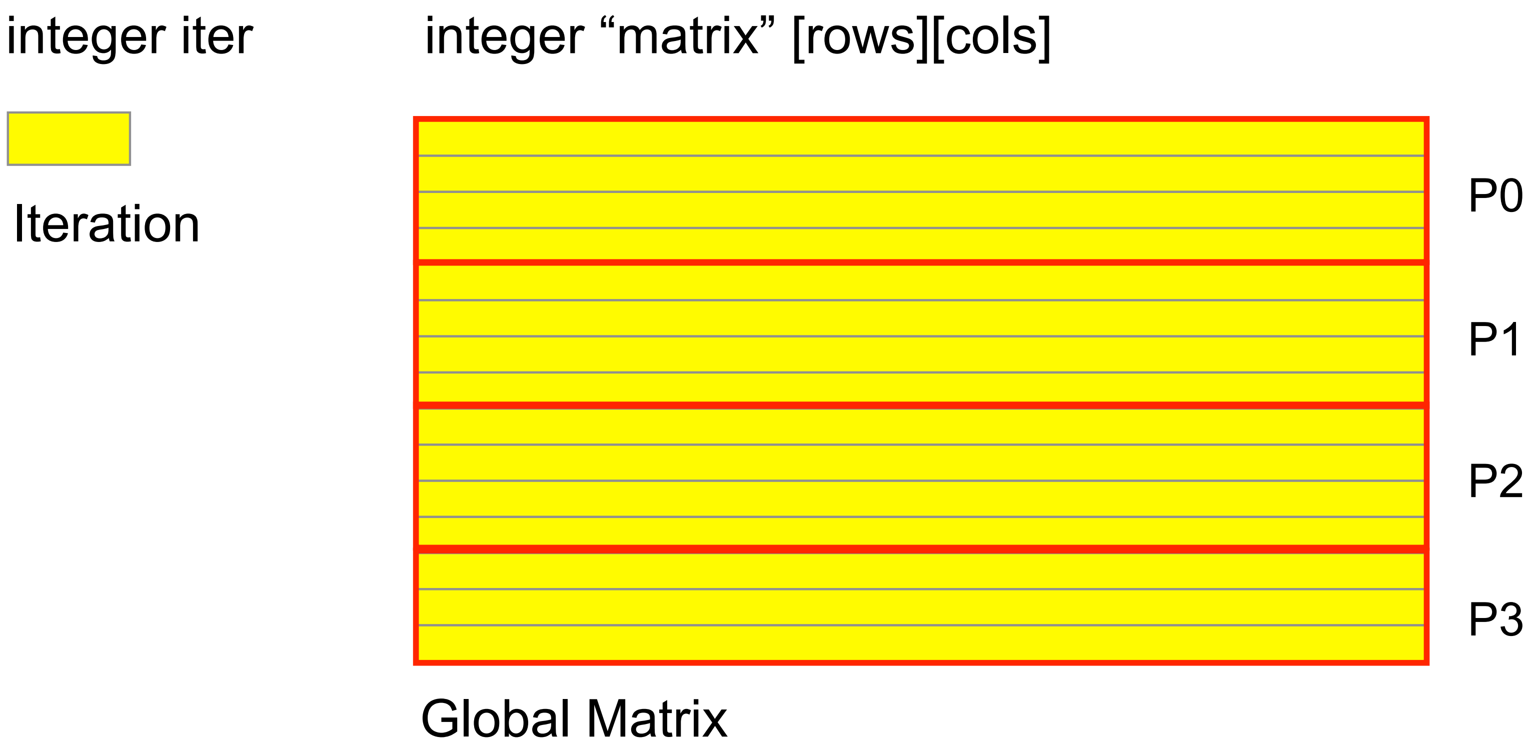
IBM MPI

ciod

GPFS

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

Iteration

integer “matrix” [rows][cols]

P0

P1

P2

P3

Global Matrix

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

File: mlife-io-pnetcdf.c   Page 1 of 7

```c
/* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
/* -*- Mode: C; c-basic-offset:4 ; -*- */
/*
 *  (C) 2004 by University of Chicago.
 *      See COPYRIGHT in top-level directory.
 */
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <pnetcdf.h>
#include "mlife-io.h"

/* Parallel netCDF implementation of checkpoint and restart for
 * MPI Life
 *
 * Data stored in a 2D variable called "matrix" in matrix order,
 * with dimensions "row" and "col".
 *
 * Each checkpoint is stored in its own file.
 */
static MPI_Comm mlifeio_comm = MPI_COMM_NULL;

int MLIFEIO_Init(MPI_Comm comm)
{
    int err;
    err = MPI_Comm_dup(comm, &mlifeio_comm);
    return err;
}

```

File: mlife-io-pnetcdf.c   Page 2 of 7

```
/* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
int MLIFEIO_Finalize(void)
{
    int err;

    err = MPI_Comm_free(&mlifeio_comm);

    return err;
}

int MLIFEIO_Can_restart(void)
{
    return 1;
}

```

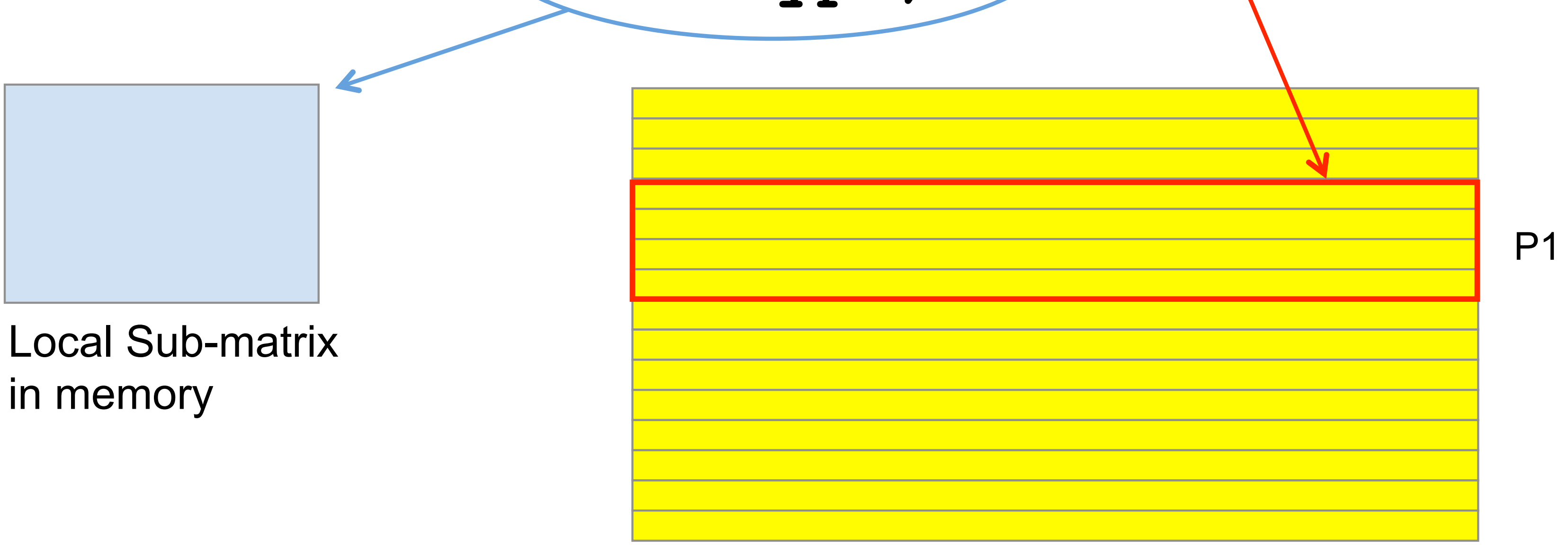
File: mlife-io-pnetcdf.c   Page 3 of 7

```c
/* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
int MLIFEIO_Checkpoint(char *prefix, int **matrix, int rows,
                       int cols, int iter, MPI_Info info)
{
    int err;
    int cmode = 0;
    int rank, nprocs;
    int myrows, myoffset;

    int ncid, varid, coldim, rowdim, dims[2];
    MPI_Offset start[2];
    MPI_Offset count[2];
    int i, j, *buf;
    char filename[64];

    MPI_Comm_size(mlifeio_comm, &nprocs);
    MPI_Comm_rank(mlifeio_comm, &rank);

    myrows   = MLIFE_myrows(rows, rank, nprocs);
    myoffset = MLIFE_myrowoffset(rows, rank, nprocs);

    snprintf(filename, 63, "%s-%d.nc", prefix, iter);

    err = ncmpi_create(mlifeio_comm, filename, cmode, info, &ncid);
    if (err != 0) {
        fprintf(stderr, "Error opening %s.\n", filename);
        return MPI_ERR_IO;
    }

```

Describing Subarray Access in PnetCDF

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

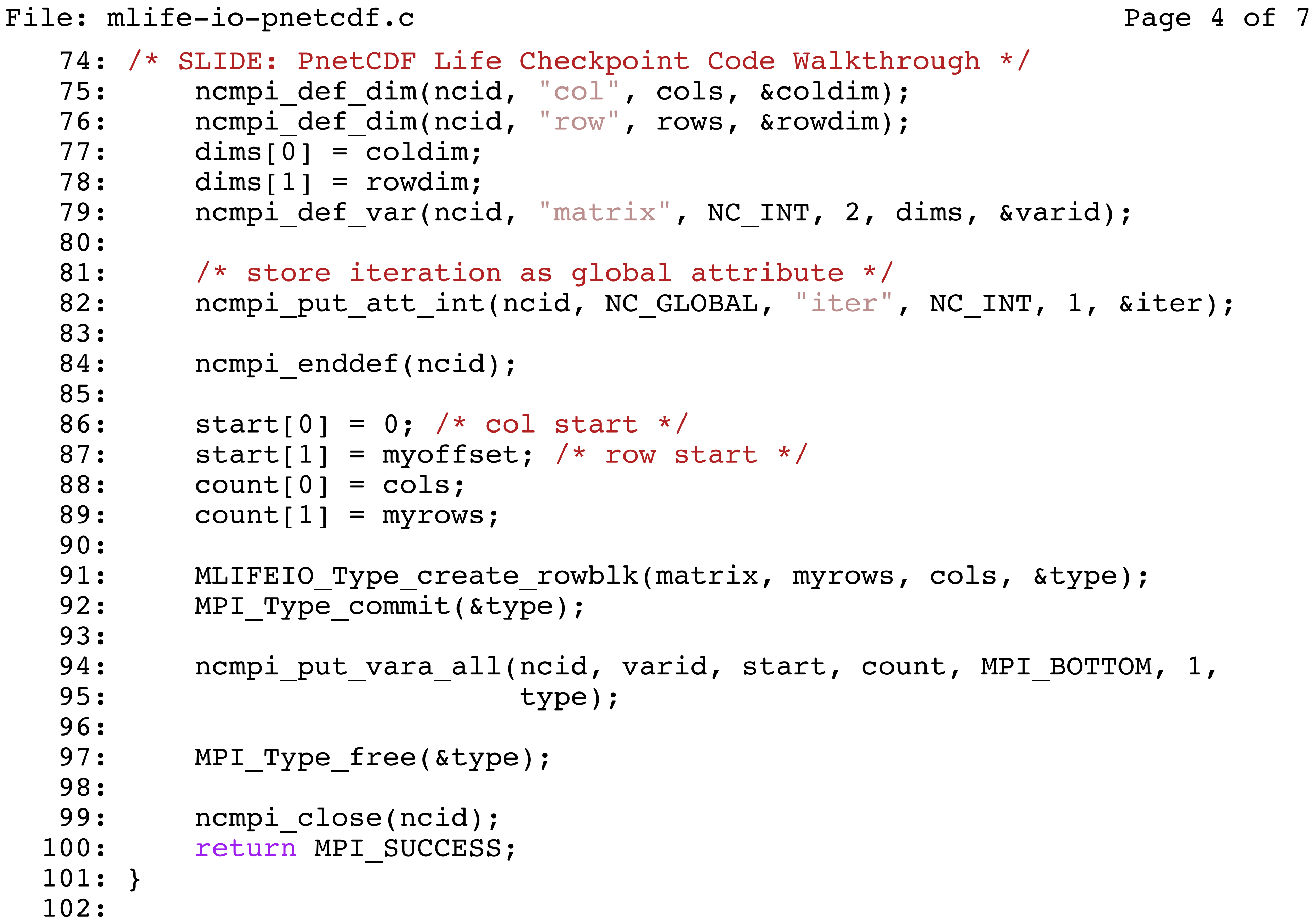
```
 ncmpi_put_vara_all(ncid,
                    varid,
                    start[], count[],
                    buf, count,
                    datatype)
```



Local Sub-matrix in memory

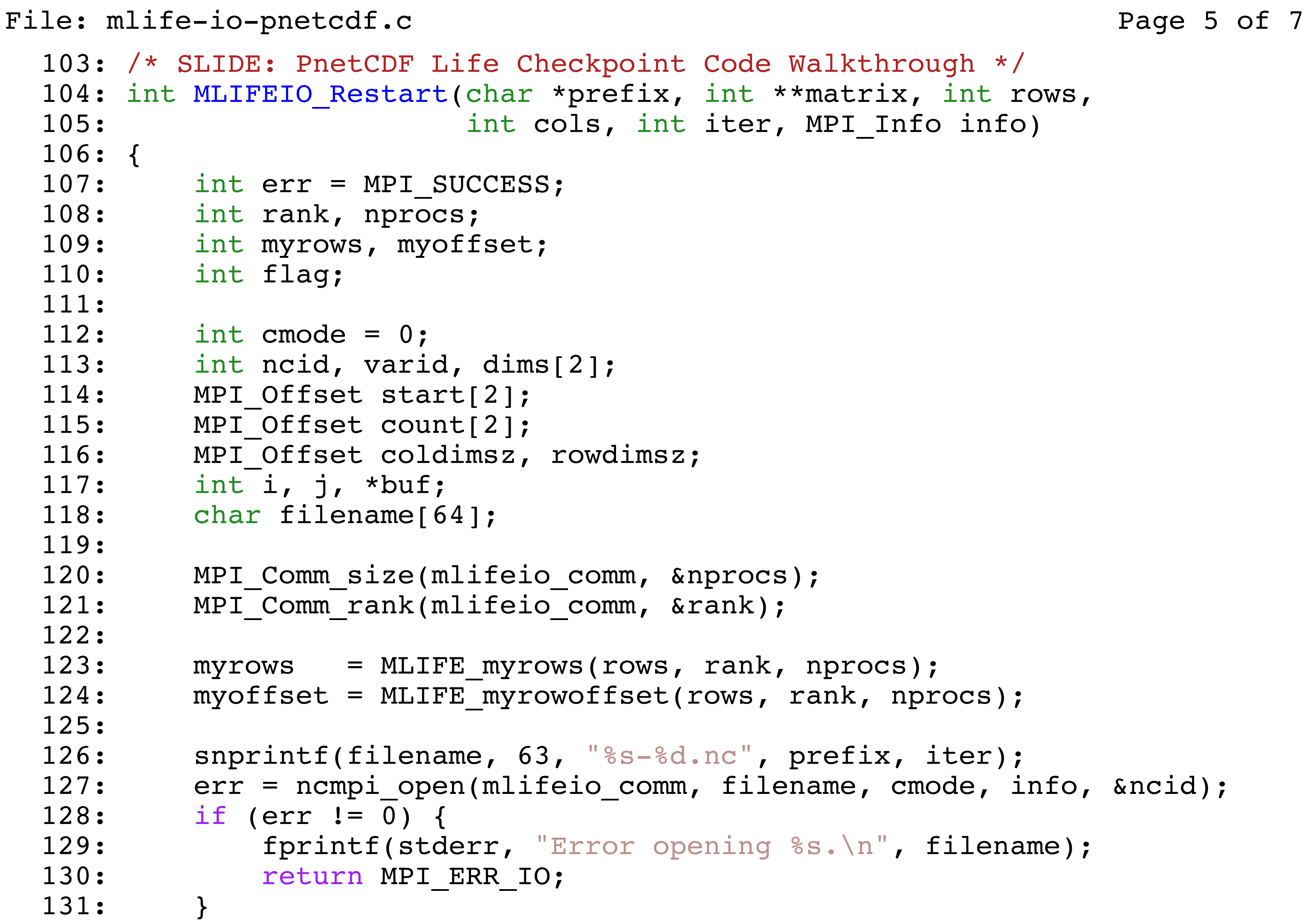
P1

Global Matrix in PnetCDF File



File: mlife-io-pnetcdf.c Page 4 of 7

```
/* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
    ncmpi_def_dim(ncid, "col", cols, &coldim);
    ncmpi_def_dim(ncid, "row", rows, &rowdim);
    dims[0] = coldim;
    dims[1] = rowdim;
    ncmpi_def_var(ncid, "matrix", NC_INT, 2, dims, &varid);

    /* store iteration as global attribute */
    ncmpi_put_att_int(ncid, NC_GLOBAL, "iter", NC_INT, 1, &iter);

    ncmpi_enddef(ncid);

    start[0] = 0; /* col start */
    start[1] = myoffset; /* row start */
    count[0] = cols;
    count[1] = myrows;

    MLIFEIO_Type_create_rowblk(matrix, myrows, cols, &type);
    MPI_Type_commit(&type);

    ncmpi_put_vara_all(ncid, varid, start, count, MPI_BOTTOM, 1,
                       type);

    MPI_Type_free(&type);

    ncmpi_close(ncid);
    return MPI_SUCCESS;
}
```



File: mlife-io-pnetcdf.c Page 5 of 7

```
/* SLIDE: PnetCDF Life Checkpoint Code Walkthrough */
int MLIFEIO_Restart(char *prefix, int **matrix, int rows,
                    int cols, int iter, MPI_Info info)
{
    int err = MPI_SUCCESS;
    int rank, nprocs;
    int myrows, myoffset;
    int flag;

    int cmode = 0;
    int ncid, varid, dims[2];
    MPI_Offset start[2];
    MPI_Offset count[2];
    MPI_Offset coldimsz, rowdimsz;
    int i, j, *buf;
    char filename[64];

    MPI_Comm_size(mlifeio_comm, &nprocs);
    MPI_Comm_rank(mlifeio_comm, &rank);

    myrows   = MLIFE_myrows(rows, rank, nprocs);
    myoffset = MLIFE_myrowoffset(rows, rank, nprocs);

    snprintf(filename, 63, "%s-%d.nc", prefix, iter);
    err = ncmpi_open(mlifeio_comm, filename, cmode, info, &ncid);
    if (err != 0) {
        fprintf(stderr, "Error opening %s.\n", filename);
        return MPI_ERR_IO;
    }
```

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 mlife-io-pnetcdf.c pp. 6-7 for code example.

```c
/* SLIDE: Discovering Variable Dimensions */
    err = ncmpi_inq_varid(ncid, "matrix", &varid);
    if (err != 0) {
        return MPI_ERR_IO;
    }

    /* verify that dimensions in file are same as input row/col */
    err = ncmpi_inq_vardimid(ncid, varid, dims);
    if (err != 0) {
        return MPI_ERR_IO;
    }

    err = ncmpi_inq_dimlen(ncid, dims[0], &coldimsz);
    if (coldimsz != cols) {
        fprintf(stderr, "cols does not match\n");
        return MPI_ERR_IO;
    }

    err = ncmpi_inq_dimlen(ncid, dims[1], &rowdimsz);
    if (rowdimsz != rows) {
        fprintf(stderr, "rows does not match\n");
        return MPI_ERR_IO;
    }

```

File: mlife-io-pnetcdf.c   Page 7 of 7

```c
/* SLIDE: Discovering Variable Dimensions */
    buf = (int *) malloc(myrows * cols * sizeof(int));
    flag = (buf == NULL);
    /* See if any process failed to allocate memory */
    MPI_Allreduce(MPI_IN_PLACE, &flag, 1, MPI_INT, MPI_LOR,
                  mlifeio_comm);
    if (flag) {
        return MPI_ERR_IO;
    }

    start[0] = 0; /* col start */
    start[1] = myoffset; /* row start */
    count[0] = cols;
    count[1] = myrows;
    ncmpi_get_vara_int_all(ncid, varid, start, count, buf);

    for (i=0; i < myrows; i++) {
        for (j=0; j < cols; j++) {
            matrix[i+1][j] = buf[(i*cols) + j];
        }
    }

    free(buf);

    return MPI_SUCCESS;
}
```

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 look at how some other domain models can be mapped into PnetCDF

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.

HPC I/O Software Stack

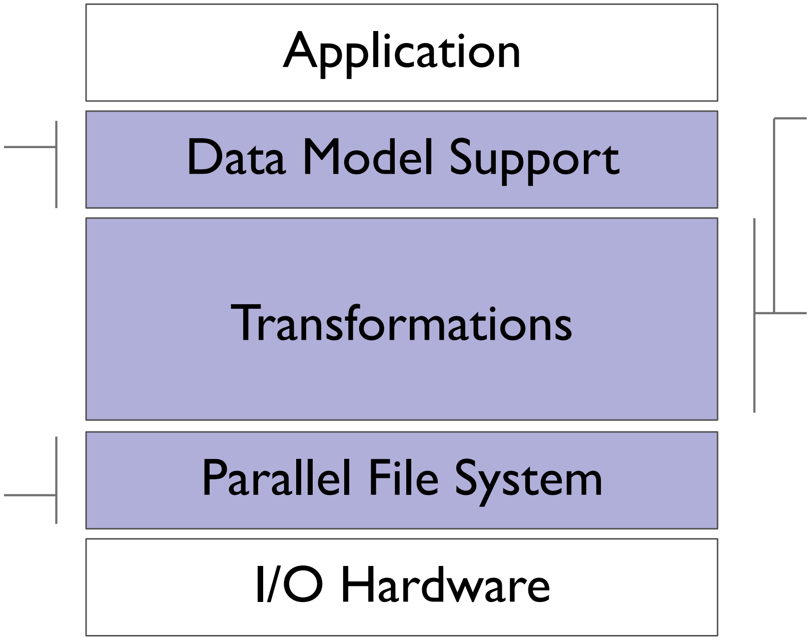
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack.*

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



Application

Data Model Support

Transformations

Parallel File System

I/O Hardware

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 Interfaces

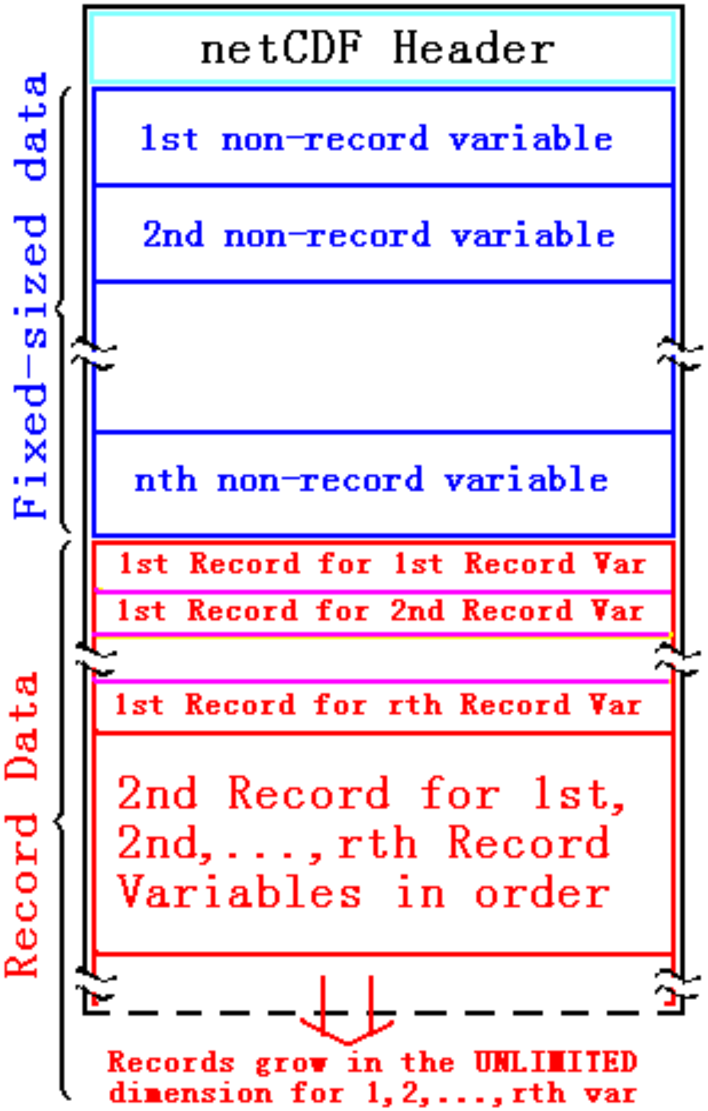
- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents
- Present APIs more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays and I/O on subsets of these arrays
- Both of our example interfaces are implemented on top of MPI-IO

Parallel NetCDF (PnetCDF)

- Based on original “Network Common Data Format” (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C, Fortran, and F90 interfaces
  - Portable data format (identical to netCDF)
  - Noncontiguous I/O in memory using MPI datatypes
  - Noncontiguous I/O in file using sub-arrays
  - Collective I/O
  - Non-blocking I/O
- Unrelated to netCDF-4 work
- Parallel-NetCDF tutorial:
  - http://trac.mcs.anl.gov/projects/parallel-netcdf/wiki/QuickTutorial

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

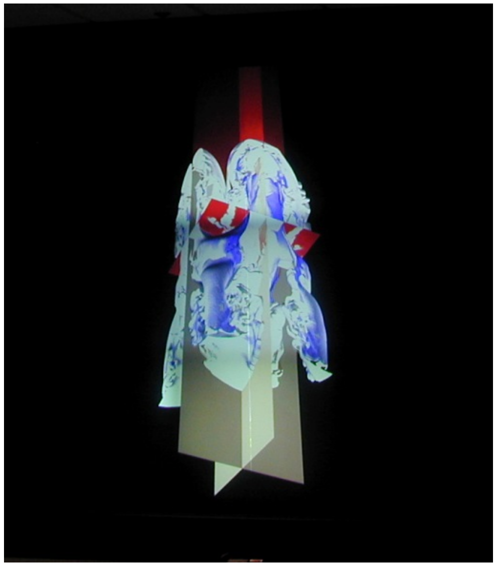

netCDF Header
Fixed-sized data
1st non-record variable
2nd non-record variable
nth non-record variable
Record Data
1st Record for 1st Record Var
1st Record for 2nd Record Var
1st Record for rth Record Var
2nd Record for 1st, 2nd,...,rth Record Variables in order
Records grow in the UNLIMITED dimension for 1,2,...,rth var

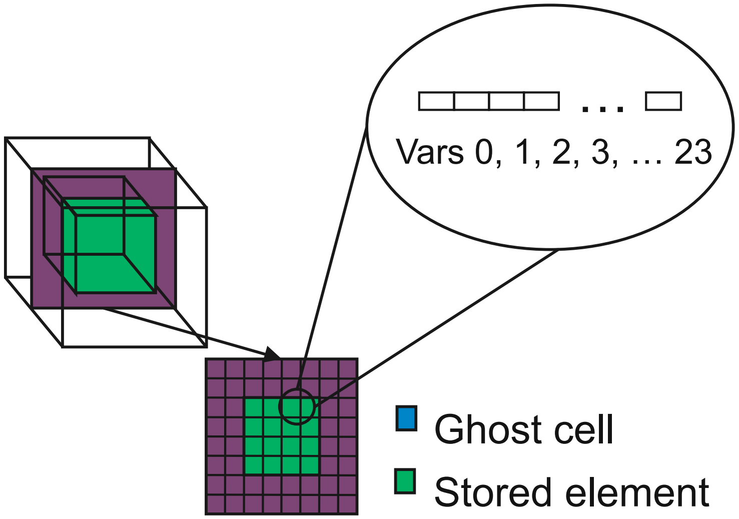
Storing Data in PnetCDF

- Create a dataset (file)
  - Puts dataset in define mode
  - Allows us to describe the contents
    - Define dimensions for variables
    - Define variables using dimensions
    - Store attributes if desired (for variable or dataset)
- Switch from define mode to data mode to write variables
- Store variable data
- Close the dataset

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





Example: FLASH with PnetCDF

- FLASH AMR structures do not map directly to netCDF multidimensional arrays
- Must create mapping of the in-memory FLASH data structures into a representation in netCDF multidimensional arrays
- Chose to
  - Place all checkpoint data in a single file
  - Impose a linear ordering on the AMR blocks
    - Use 4D variables
  - Store each FLASH variable in its own netCDF variable
    - Skip ghost cells
  - Record attributes describing run time, total blocks, etc.

Defining Dimensions

```
int status, ncid, dim_tot_blks, dim_nxb,
  dim_nyb, dim_nzb;
MPI_Info hints;
/* create dataset (file) */
status = ncmpi_create(MPI_COMM_WORLD, filename,
  NC_CLOBBER, hints, &file_id);
/* define dimensions */
status = ncmpi_def_dim(ncid, "dim_tot_blks",
  tot_blks, &dim_tot_blks);
status = ncmpi_def_dim(ncid, "dim_nxb",
  nzones_block[0], &dim_nxb);
status = ncmpi_def_dim(ncid, "dim_nyb",
  nzones_block[1], &dim_nyb);
status = ncmpi_def_dim(ncid, "dim_nzb",
  nzones_block[2], &dim_nzb);
```

Each dimension gets a unique reference

Creating Variables

```
int dims = 4, dimids[4];
int varids[NVARS];
/* define variables (X changes most quickly) */
dimids[0] = dim_tot_blks;
dimids[1] = dim_nzb;
dimids[2] = dim_nyb;
dimids[3] = dim_nxb;
for (i=0; i < NVARS; i++) {
   status = ncmpi_def_var(ncid, unk_label[i],
     NC_DOUBLE, dims, dimids, &varids[i]);
}
```

Same dimensions used for all variables

Storing Attributes

```
/* store attributes of checkpoint */
status = ncmpi_put_att_text(ncid, NC_GLOBAL,
  "file_creation_time", string_size,
  file_creation_time);
status = ncmpi_put_att_int(ncid, NC_GLOBAL,
  "total_blocks", NC_INT, 1, tot_blks);
status = ncmpi_enddef(file_id);

/* now in data mode … */
```

Writing Variables

```
double *unknowns; /* unknowns[blk][nzb][nyb][nxb]
   */
size_t start_4d[4], count_4d[4];
start_4d[0] = global_offset; /* different for each
   process */
start_4d[1] = start_4d[2] = start_4d[3] = 0;
count_4d[0] = local_blocks;
count_4d[1] = nzb;  count_4d[2] = nyb;
   count_4d[3] = nxb;
for (i=0; i < NVARS; i++) {
    /* ... build datatype “mpi_type” describing
      values of a single variable ... */
    /* collectively write out all values of a
      single variable */
    ncmpi_put_vara_all(ncid, varids[i], start_4d,
      count_4d, unknowns, 1, mpi_type);
}
status = ncmpi_close(file_id);
```

Typical MPI buffer-count-type tuple

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

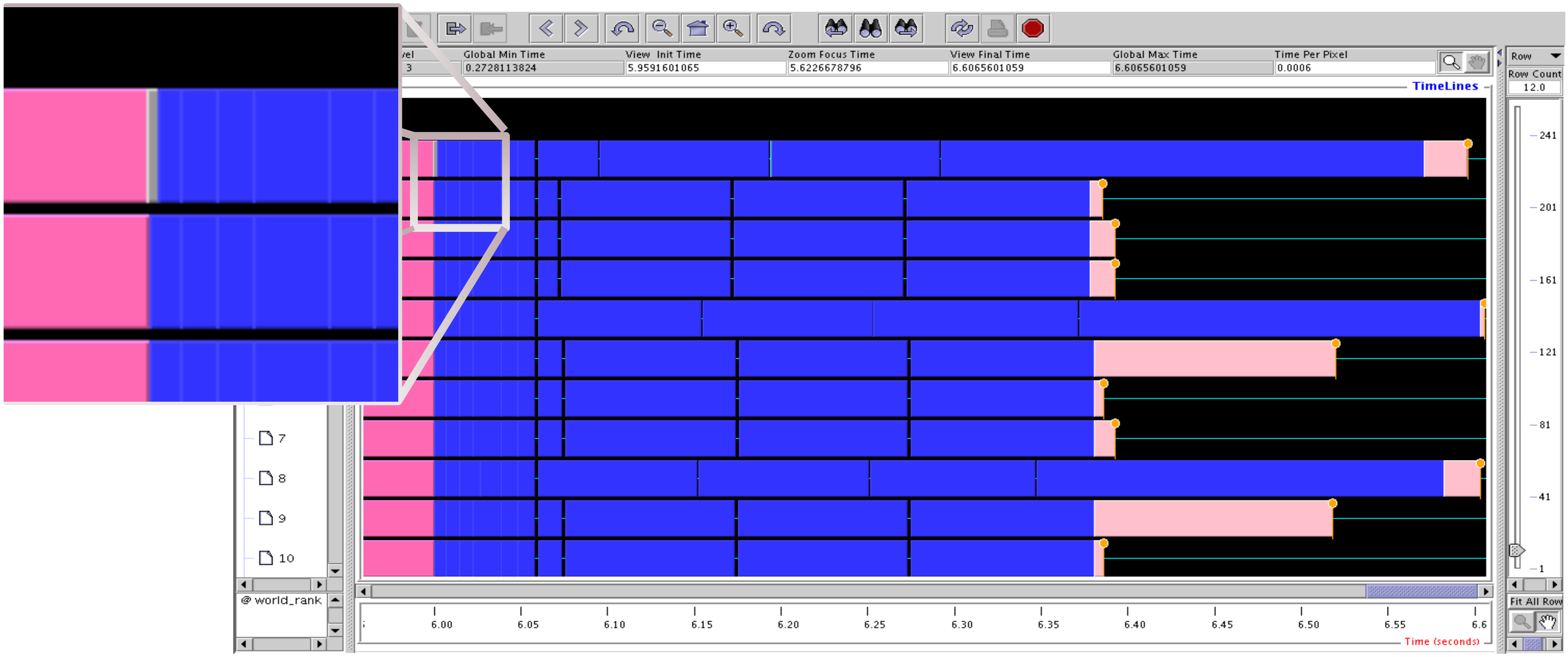
- Inside `ncmpi_put_vara_all` (once per variable)
  - Each process performs data conversion into internal buffer
  - Uses `MPI_File_set_view` to define file region
    - Contiguous region for each process in FLASH case
  - `MPI_File_write_all` collectively writes data
- At ncmpi_close
  - `MPI_File_close` 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



vel	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
3	0.2728113824	5.9591601065	5.6226678796	6.6065601059	6.6065601059	0.0006

Row Count 12.0

I/O Aggregator

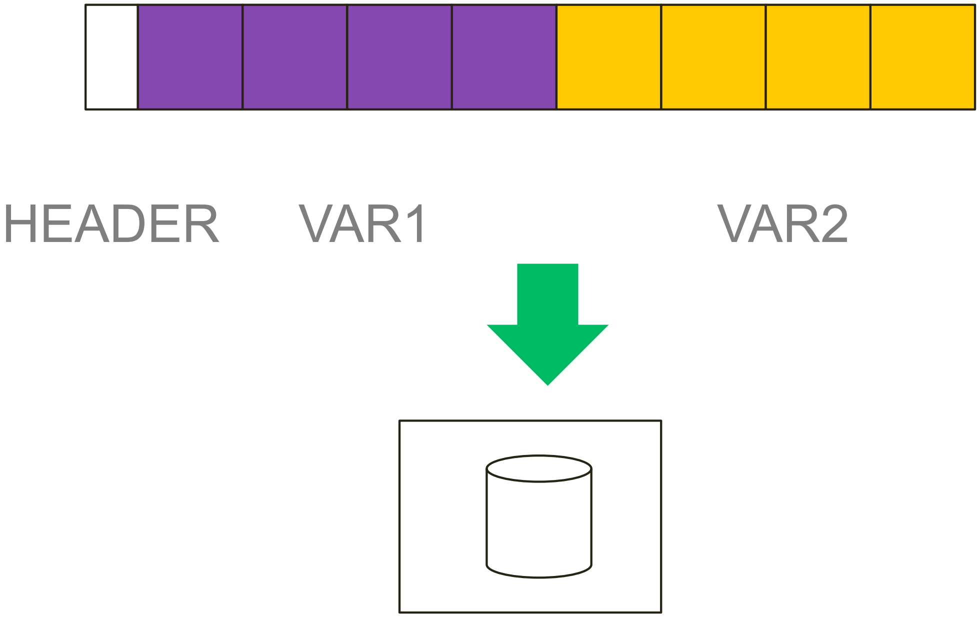
2: Collectively write app grid, AMR data

4: Close file

🟧 File open ⬜ Indep. write 🟦 Collective write ⬜ File close

Parallel-NetCDF write-combining optimization

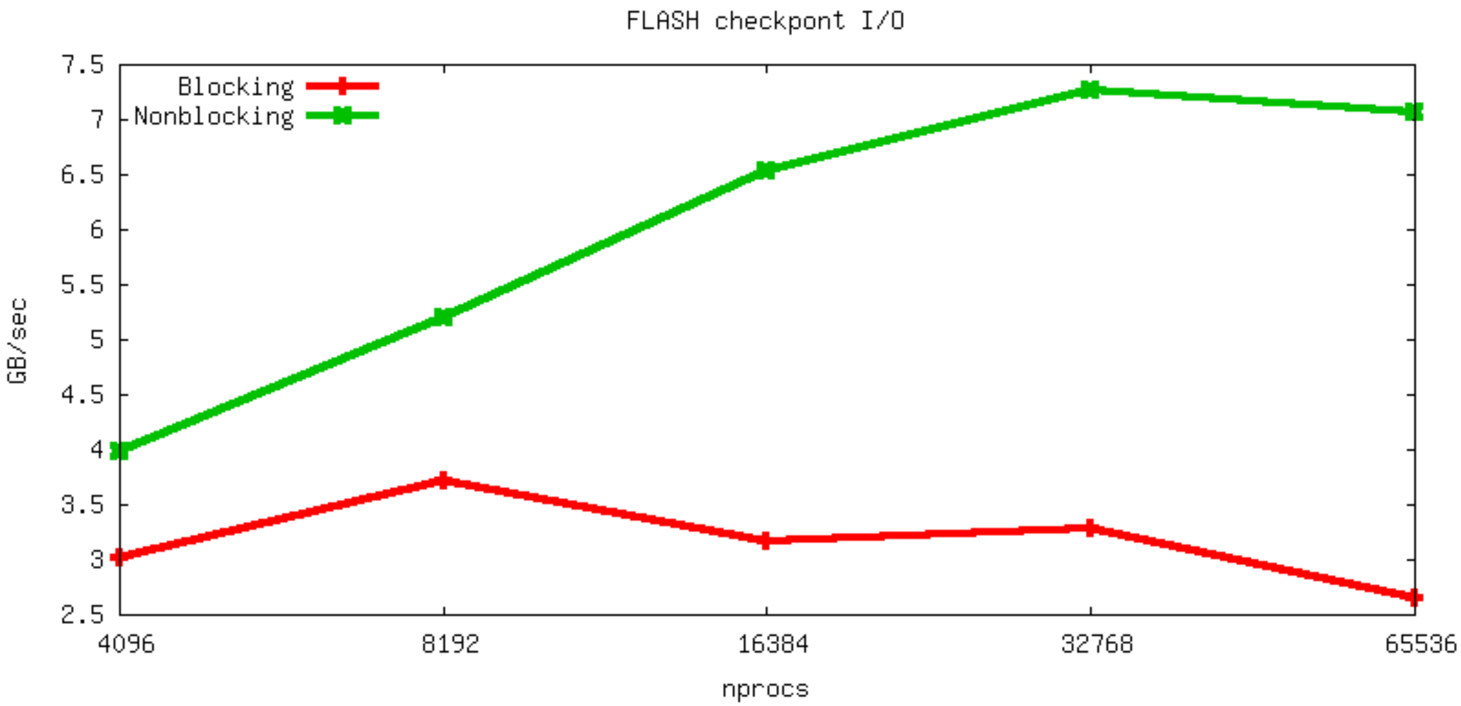
```
ncmpi_iput_vara(ncfile, varid1,
    &start, &count, &data,
    count, MPI_INT, &requests[0]);
ncmpi_iput_vara(ncfile, varid2,
    &start,&count, &data,
    count, MPI_INT, &requests[1]);
ncmpi_wait_all(ncfile, 2, requests, statuses);
```



- 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

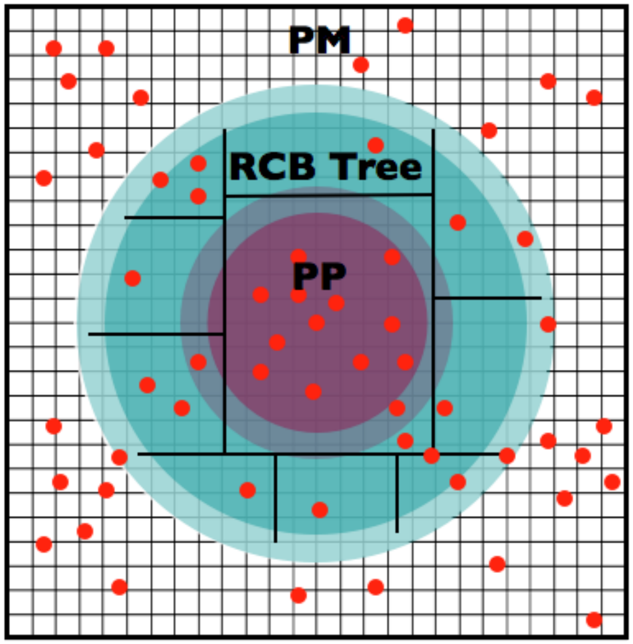
FLASH and write-combining optimization

- 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.



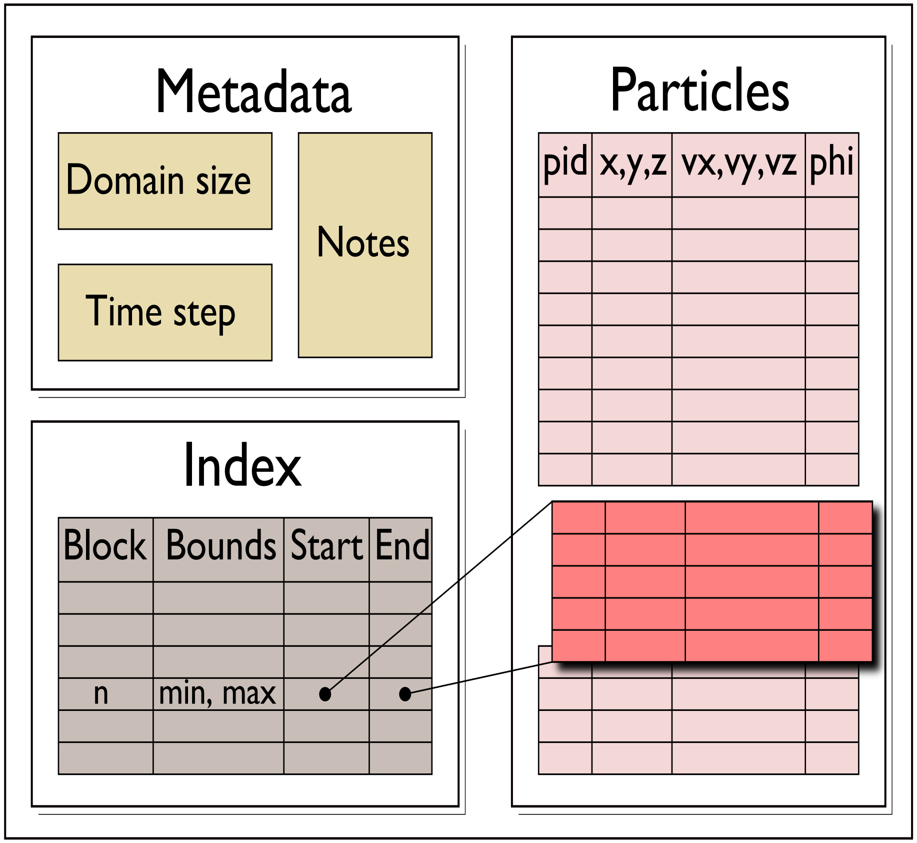
HACC: understanding cosmos via simulation

- "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



Parallel NetCDF Particle Output Collaboration with Northwestern and Argonne

- Metadata, index, and particle data
- Self-describing portable format
- Can be read with different number of processes than written
- Can be queried for particles within spatial bounds



Metadata

Domain size

Time step

Notes

Index

Block	Bounds	Start	End
n	min, max	●	●

Particles

pid	x,y,z	vx,vy,vz	phi

File schema for analysis output enables spatial queries of particle data in a high-level self-describing format.

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_velid, 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 */
    for(i=0; i<ndims; i++)  {
2       ncmpi_inq_dimlen(ncfile, i, &(dim_sizes[i]) );
    }
    for(i=0; i<nvars; i++) {
3       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)

Additional I/O Interfaces for Applications

HPC I/O Software Stack

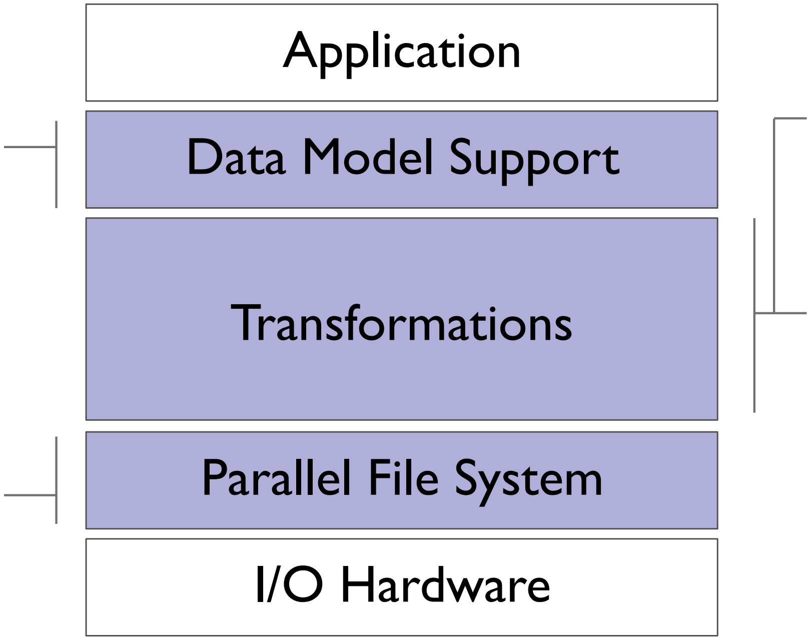
The software used to provide data model support and to transform I/O to better perform on today's I/O systems is often referred to as the *I/O stack.*

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 I/O Interfaces

- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents
- Present APIs more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays and I/O on subsets of these arrays
- Both of our example interfaces are implemented on top of MPI-IO

Other 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: my point: it's ok to make your own.

netCDF-4

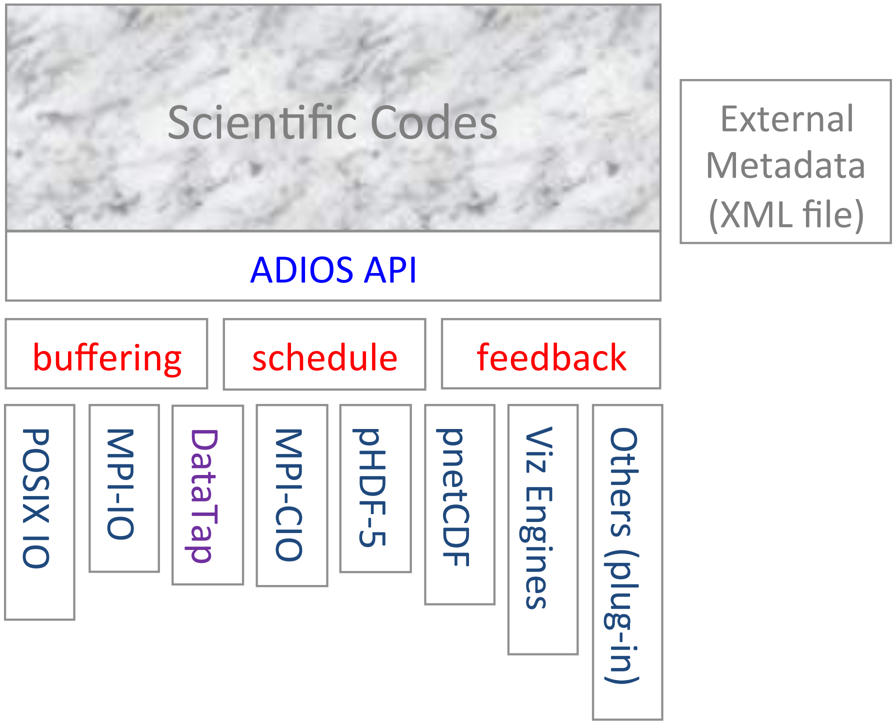
- Joint effort between Unidata (netCDF) and NCSA (HDF5)
  - Initial effort NASA funded.
  - Ongoing development Unidata/UCAR funded.
- Combine netCDF and HDF5 aspects
  - HDF5 file format (still portable, self-describing)
  - netCDF API
- Features
  - Parallel I/O
  - C, Fortran, and Fortran 90 language bindings (C++ in development)
  - per-variable compression
  - multiple unlimited dimensions
  - higher limits for file and variable sizes
  - backwards compatible with “classic” datasets
  - Groups
  - Compound types
  - Variable length arrays
  - Data chunking and compression (parallel reads only – serial writes)

NetCDF 4 API Summary

- `nc_create_par("demo", NC_MPIIO|NC_NETCDF4,`
  `MPI_COMM_WORLD, MPI_INFO_NULL, &ncfile);`
  - New flag ‘NC_NETCDF4’; MPI Communicator, Info
- `nc_open_par("demo", NC_MPIIO,`
  `MPI_COMM_WORLD, MPI_INFO_NULL, &ncfile);`
  - Can select POSIX (`NC_MPIPOSIX`) or MPI-IO (`NC_MPIIO`)
- `nc_var_par_access(ncfile, varid, NC_COLLECTIVE);`
  - Enable/disable collective I/O access (enabled by default).
- No longer need “send-to-master” model

ADIOS

- Allows plug-ins for different I/O implementations
- Abstracts the API from the method used for I/O
- Simple API, almost as easy as F90 write statement
- Best practice/optimized IO routines for all supported transports “for free”
- Thin API
- XML description of structure
  - data groupings with annotation
  - IO method selection
  - buffer sizes
- Supports common optimizations
  - Buffering
  - Scheduling
- Pluggable IO routines



ADIOS API C Example

XML configuration file:

```
<adios-config host-language=“C”>
<adios-group name=“restart”>
<var name=“n” path=“/” type=“integer” />
<var name=“mi” path=“/param” type=“integer”/>
    … <!-- declare more data elements -->
<var name=“zion” type=“real” dimensions=“n,4,2,mi”/>
<attribute name=“units” path=“/param” value=“m/s”/>
</adios-group>
… <!-- declare additional adios-groups -->

<method method=“MPI” group=“restart”/>
<method priority=“2” method=“DATATAP” iterations=“1”
    type=“diagnosis”>srv=ewok001.ccs.ornl.gov</method>
<!-- add more methods -->
<buffer size-MB=“100” allocate-time=“now”/>
</adios-config>
```

C code:

```
// parse the XML file and determine buffer sizes
adios_init (“config.xml”);
// open and write the retrieved type
adios_open (&h1, “restart”, “restart.n1”, “w”);
adios_group_size (h1, size, &total_size, comm);
adios_write (h1, “n”, n);  // int n;
adios_write (h1, “mi”, mi); // int mi;
adios_write (h1, “zion”, zion); // float zion [10][20][30][40];
// write more variables
...
// commit the writes for synchronous transmission, or
// generally initiate the write for asynchronous transmission
adios_close (h1);
// do more work
...
// shutdown the system at the end of my run
adios_finalize (mype);
```

Understanding I/O Behavior and Performance

Thanks to the following for much of this material:

Phil Carns, 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:

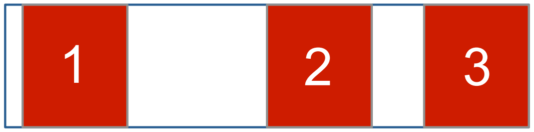
- P. Carns et al. Understanding and improving computational science storage access through continuous characterization. ACM TOS. 2011.
- P. Carns et al. Production I/O characterization on the Cray XE6. CUG 2013. May, 2013.

Characterizing Application I/O

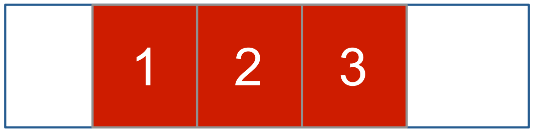
How are applications using the I/O system, and how successful are they at attaining high performance?

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

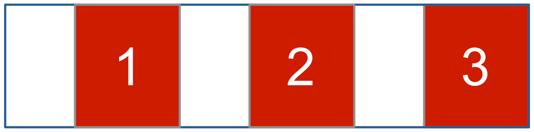
- No code changes, small and tunable memory footprint (~2MB default)
- Characterization data aggregated and compressed prior to writing
- Captures:
  - Counters for file I/O and MPI-IO calls, some PnetCDF and HDF5 calls
  - Counters for unaligned, sequential, consecutive, and strided access
  - Timing of opens, closes, first and last reads and writes
  - Cumulative data read and written
  - Histograms of access, stride, datatype, and extent sizes



sequential



consecutive

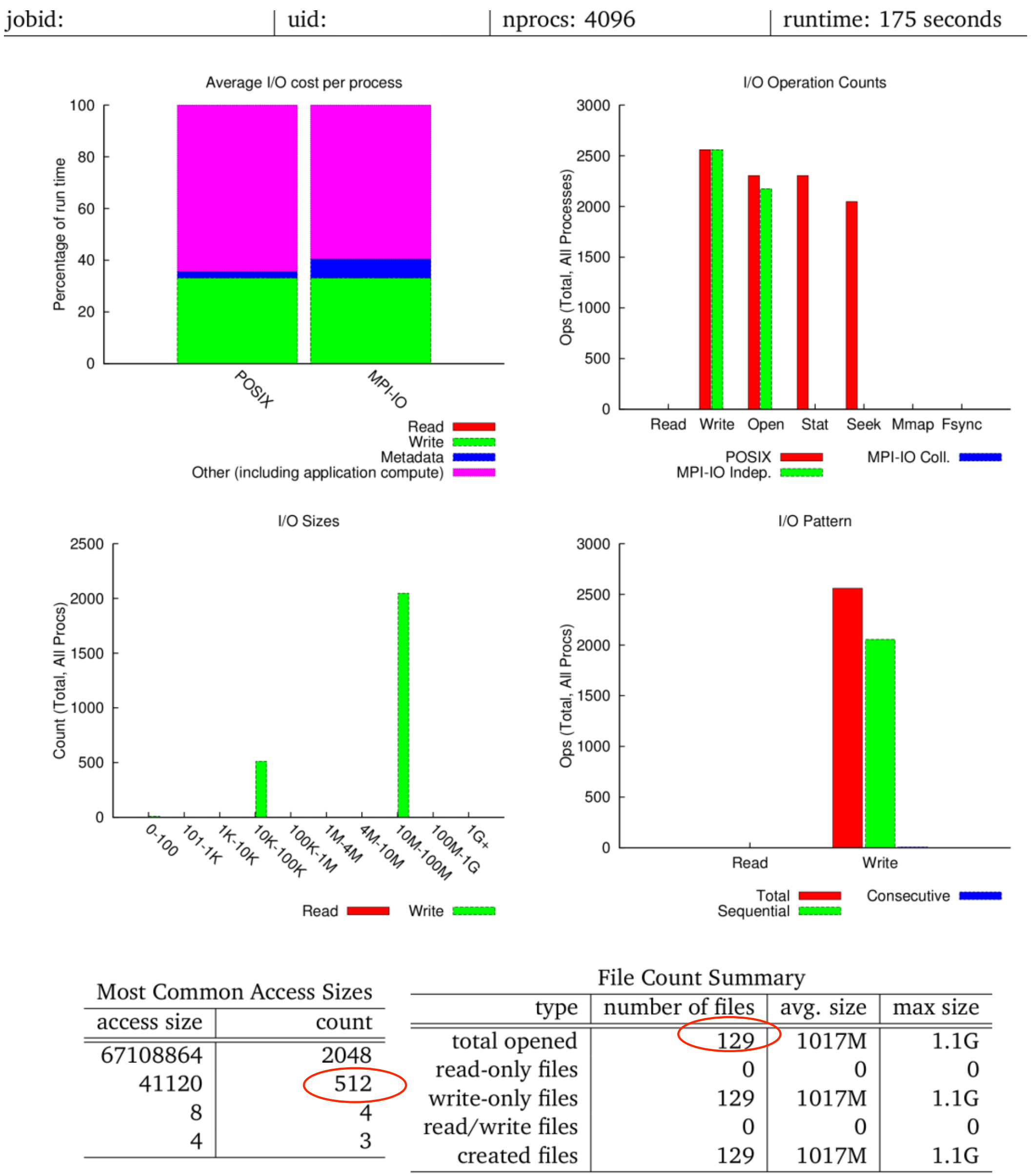


strided

How Darshan Works

- Use PMPI and ld wrappers to intercept I/O functions
  - Requires re-linking, but no code modification
  - Can be transparently included in mpicc
  - Compatible with a variety of compilers
- Record statistics independently at each process
  - Compact summary rather than verbatim record
  - Independent data for each file
- 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, IBM Blue Gene/P and /Q, Cray XE6

Darshan Job Summary



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

Most Common Access Sizes

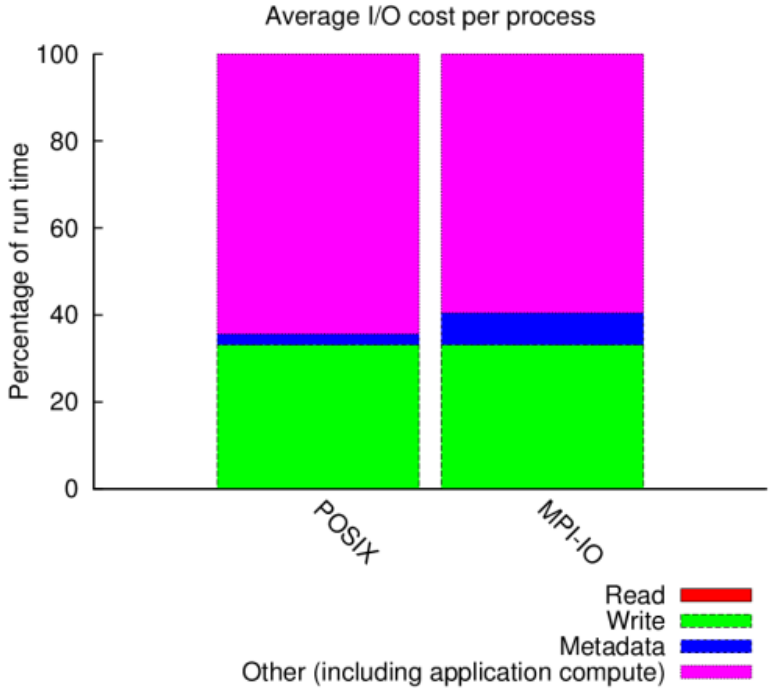
access size	count
67108864	2048
41120	512
8	4
4	3

File Count Summary

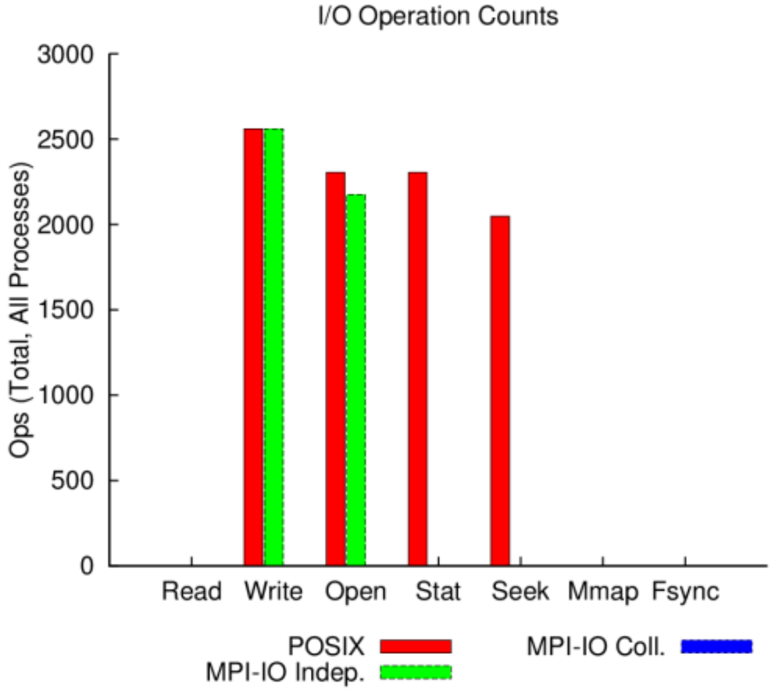
type	number of files	avg. size	max size
total opened	129	1017M	1.1G
read-only files	0	0	0
write-only files	129	1017M	1.1G
read/write files	0	0	0
created files	129	1017M	1.1G

Job summary tool shows characteristics "at a glance":

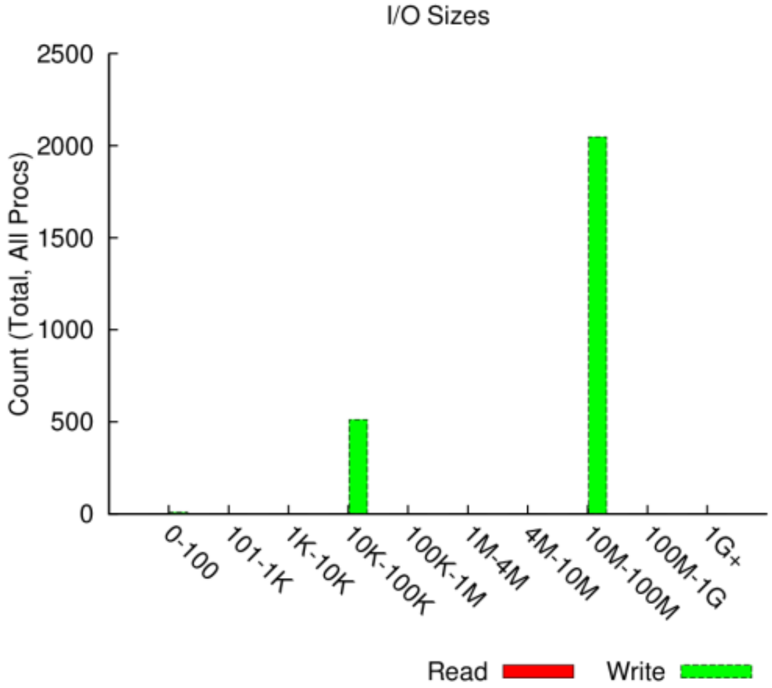
- Early indication of I/O behavior and where to explore in further
- Example: Mismatch between number of files (R) vs. number of header writes (L)
  - The same header is being overwritten 4 times in each data file



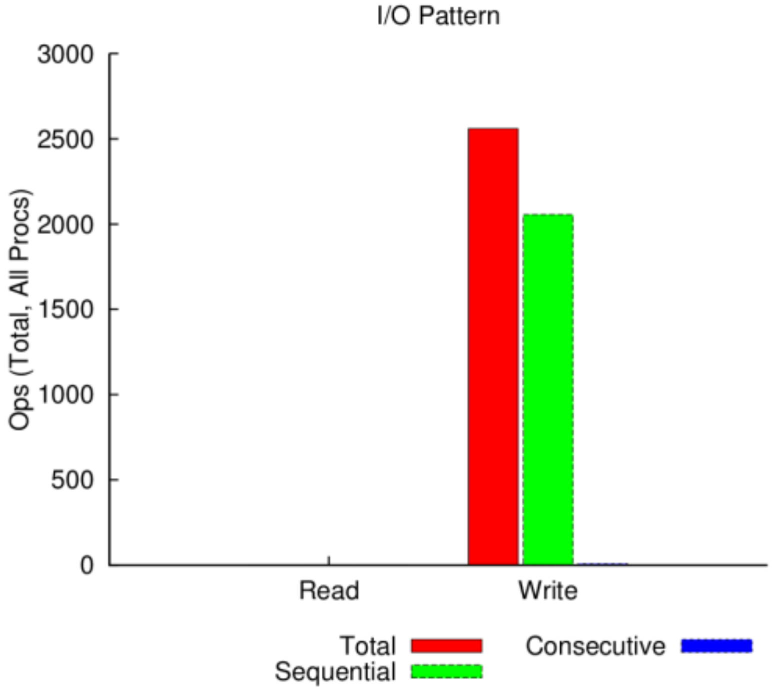
Average I/O cost per process



I/O Operation Counts



I/O Sizes



I/O Pattern

Looking for I/O Performance Problems

- Many I/O problems can be seen from these logs
- More formally, we can look at specific metrics that we believe indicate performance problems
  - Administrators can filter logs using metrics to identify applications that may benefit from tuning assistance
- We explored three example metrics that can be quickly computed from Darshan log data:
  - Applications that read more bytes of data from the file system than were present in the file
  - Percentage of I/O time spent performing metadata operations such as open(), close(), stat(), and seek() exceeds a threshold (e.g., 25%)
  - Jobs that wrote less than 1 MiB per operation to shared files without using any collective operations
  - Data from Hopper Cray XE6 at NERSC, 261,890 jobs analyzed

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 through redundant file system transfers
  - Candidates for aggregation or collective I/O

- **Summary of analysis:**
  - Threshold: > 1 TiB
  - Matching jobs: 671
- **Top Example:**
  - Scale: 6,138 processes
  - Run time: 6.5 hours
  - Avg. I/O time per process: 27 minutes
  - Read 548.6 TiB of data from a 1.2 TiB collection of read-only files
  - Used 8 KiB read operations and generated 457x redundant read traffic

Time in Metadata Operations

- **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
  - Most relevant for jobs that performed a significant amount of I/O
  - Candidates for coalescing files and eliminating extra metadata calls

- **Summary of analysis:**
  - Thresholds:

    meta_time_per_proc > 30 sec && nprocs >= 192 && meta_ratio >= 25%
  - Matching jobs: 252
- **Top Example:**
  - Scale: 40,960 processes
  - Run time: 229 seconds
  - Max. I/O time per process: 103 seconds
  - 99% of I/O time in metadata operations
  - Generated 200,000+ files with 600,000+ write() and 600,000+ stat() calls

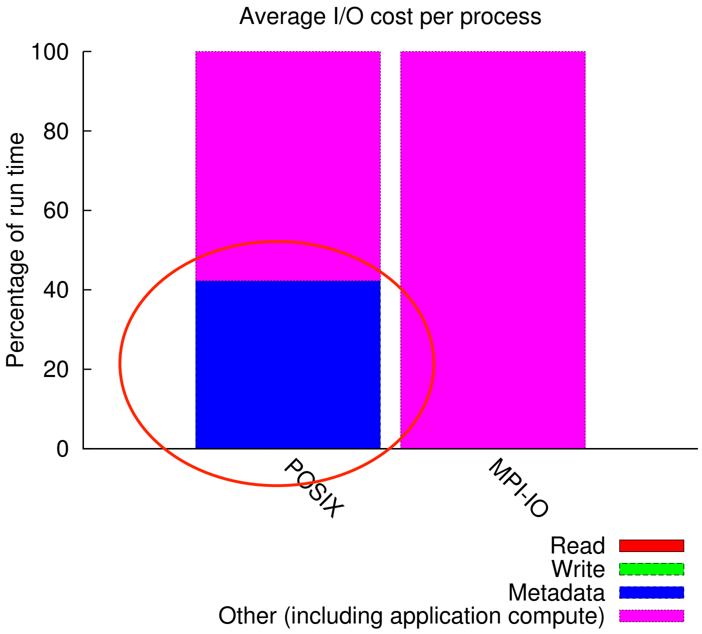
Small Writes to Shared Files

- **Scenario:** Small writes can contribute to poor performance
  - We searched for jobs that wrote less than 1 MiB per operation to shared files without using any collective operations
  - Candidates for collective I/O or batching/buffering of write operations
- **Summary of analysis:**
  - Thresholds: > 100 million small writes && 0 collective writes
  - Matching jobs: 230
- **Top Example:**
  - Scale: 128 processes
  - Run time: 30 minutes
  - Max. I/O time per process: 12 minutes
  - 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

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

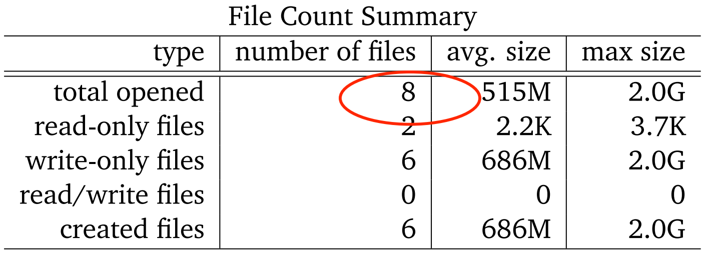


File Count Summary

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)

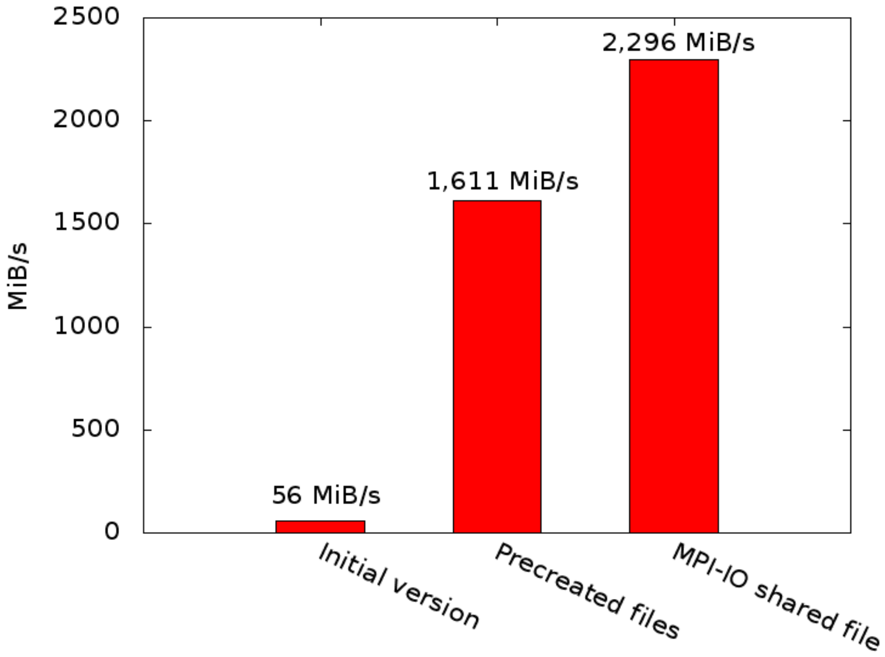
- With help from ALCF catalysts and Darshan instrumentation, we developed an I/O strategy that used MPI-IO collectives and a new file layout to reduce metadata overhead
- **Impact: 41X improvement in I/O throughput for production application**

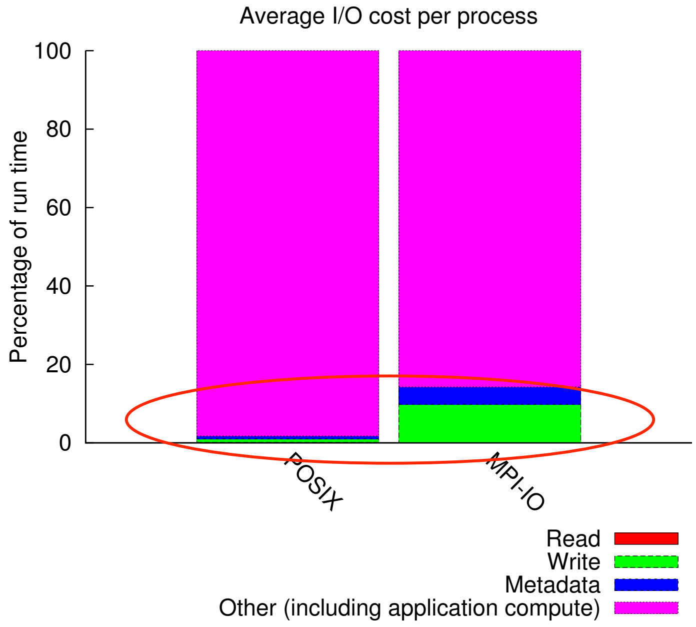


File Count Summary

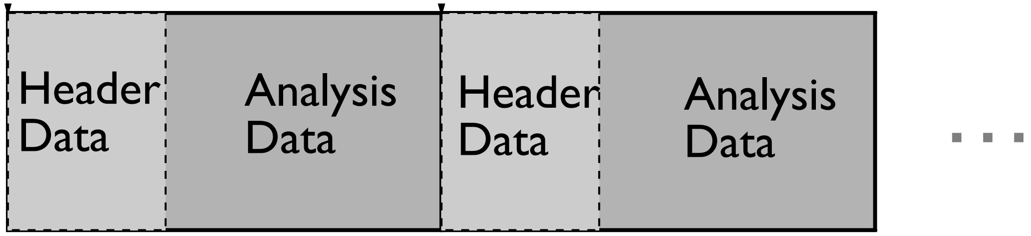
type	number of files	avg. size	max size
total opened	8	515M	2.0G
read-only files	2	2.2K	3.7K
write-only files	6	686M	2.0G
read/write files	0	0	0
created files	6	686M	2.0G

HSCD I/O performance with 32,768 cores





Performance Debugging: An Analysis I/O Example



Header Data

Analysis Data

Header Data

Analysis Data

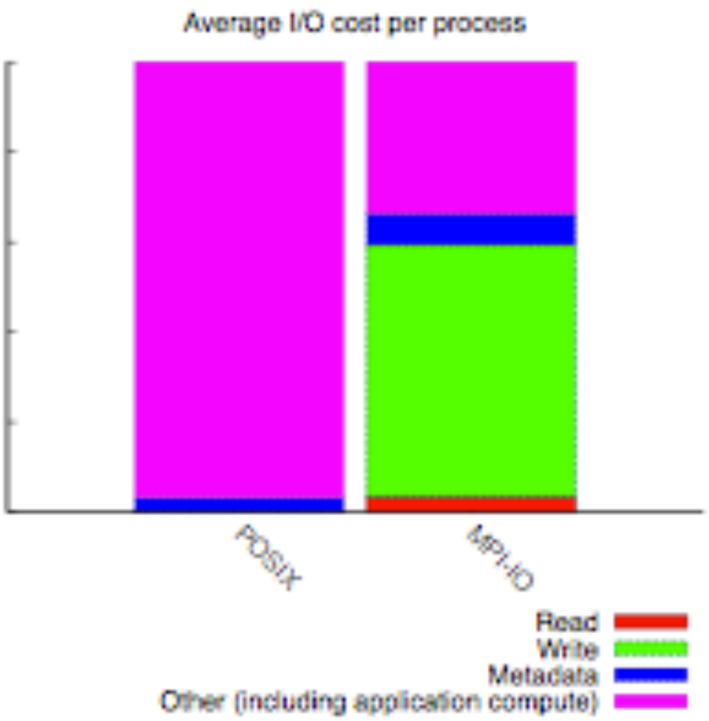
. . .

- 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?

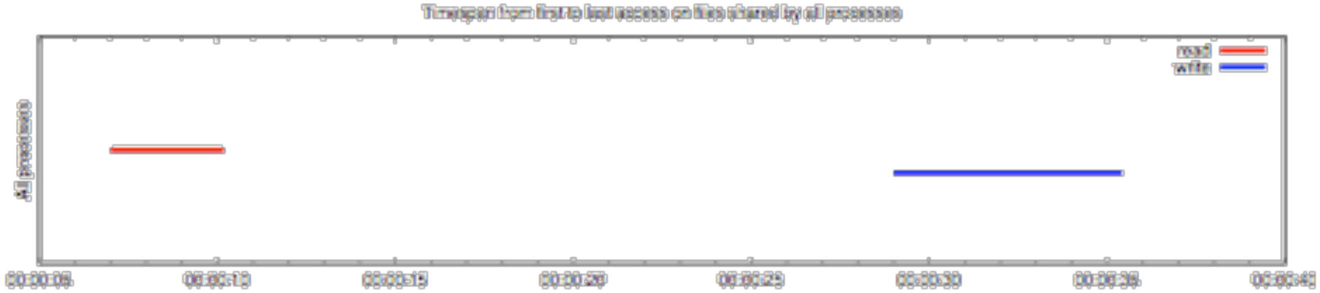
Processes	I/O Time (s)	Total Time (s)
8,192	8	60
16,384	16	47
32,768	32	57

An Analysis I/O Example (continued)

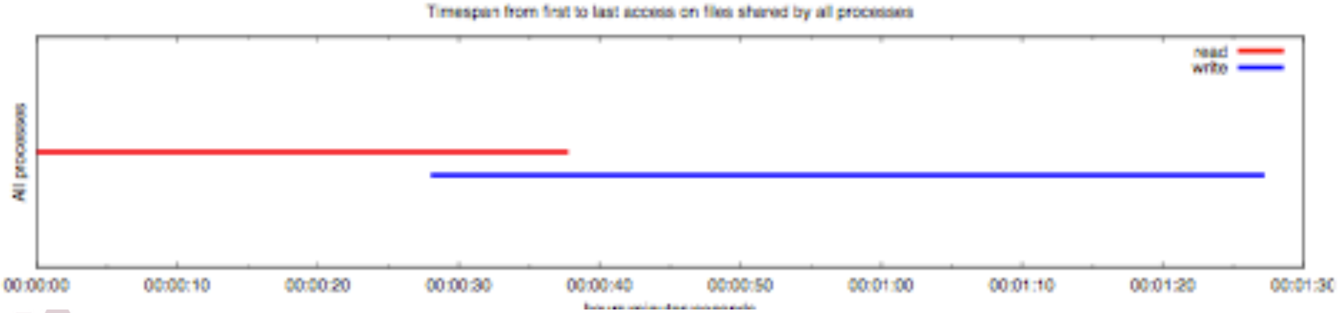
- **Problem**: More than 50% of time spent writing output at 32K processes. Cause: Unexpected RMW pattern, difficult to see at the application code level, was identified from Darshan summaries.



- What we expected to see, read data followed by write analysis:

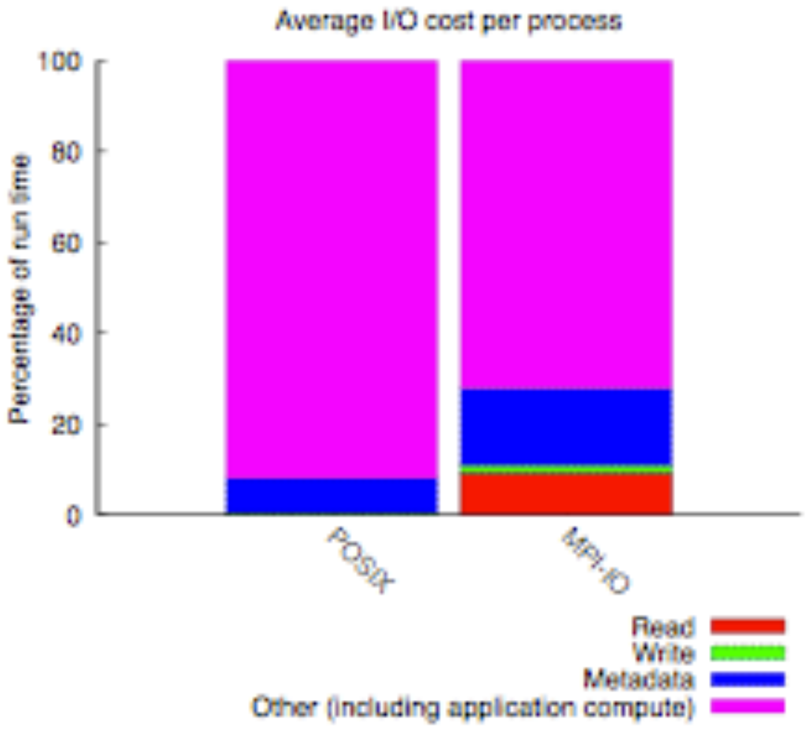


- What we saw instead: RMW during the writing shown by overlapping red (read) and blue (write), and a very long write as well.



An Analysis I/O Example (continued)

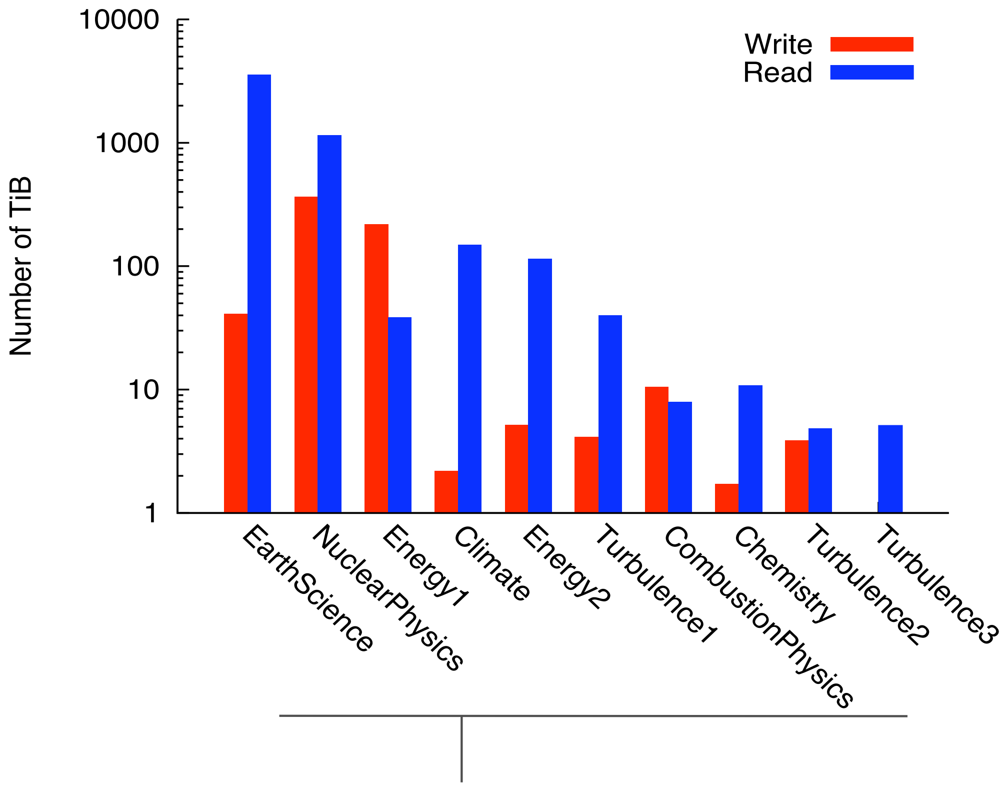
- **Solution**: Reorder operations to combine writing block headers with block payloads, so that "holes" are not written into the file during the writing of block headers, to be filled when writing block payloads
- **Result**: Less than 25% of time spent writing output, output time 4X shorter, overall 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

Two Months of Application I/O on ALCF Blue Gene/P

- Data captured from late January through late March of 2010
- Darshan captured data on 6,480 jobs (27%) from 39 projects (59%)
- Simultaneously captured data on servers related to storage utilization



Top 10 data producers and/or consumers shown. Surprisingly, most "big I/O" users read more data during simulations than they wrote.

Application I/O on ALCF Blue Gene/P

Application	Mbytes/ sec/CN*	Cum. MD	Files/ Proc	Creates/ Proc	Seq. I/O	Mbytes/ Proc
EarthScience	0.69	95%	140.67	98.87	65%	1779.48
NuclearPhysics	1.53	55%	1.72	0.63	100%	234.57
Energy1	0.77	31%	0.26	0.16	87%	66.35
Climate	0.31	82%	3.17	2.44	97%	1034.92
Energy2	0.44	3%	0.02	0.01	86%	24.49
Turbulence1	0.54	64%	0.26	0.13	77%	117.92
CombustionPhysics	1.34	67%	6.74	2.73	100%	657.37
Chemistry	0.86	21%	0.20	0.18	42%	321.36
Turbulence2	1.16	81%	0.53	0.03	67%	37.36
Turbulence3	0.58	1%	0.03	0.01	100%	40.40

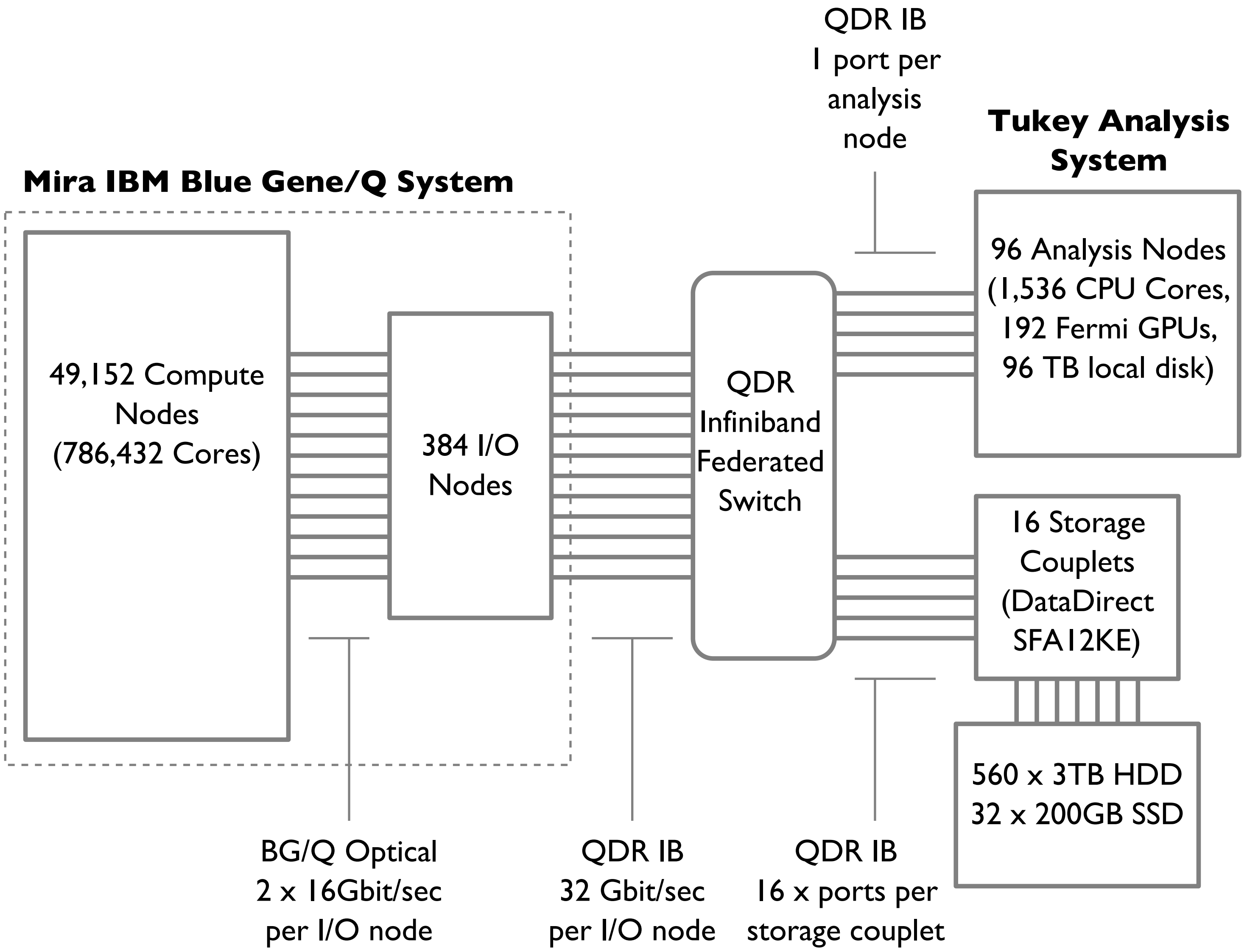
* Synthetic I/O benchmarks (e.g., IOR) attain 3.93 - 5.75 Mbytes/sec/CN for modest job sizes, down to approximately 1.59 Mbytes/sec/CN for full-scale runs.

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

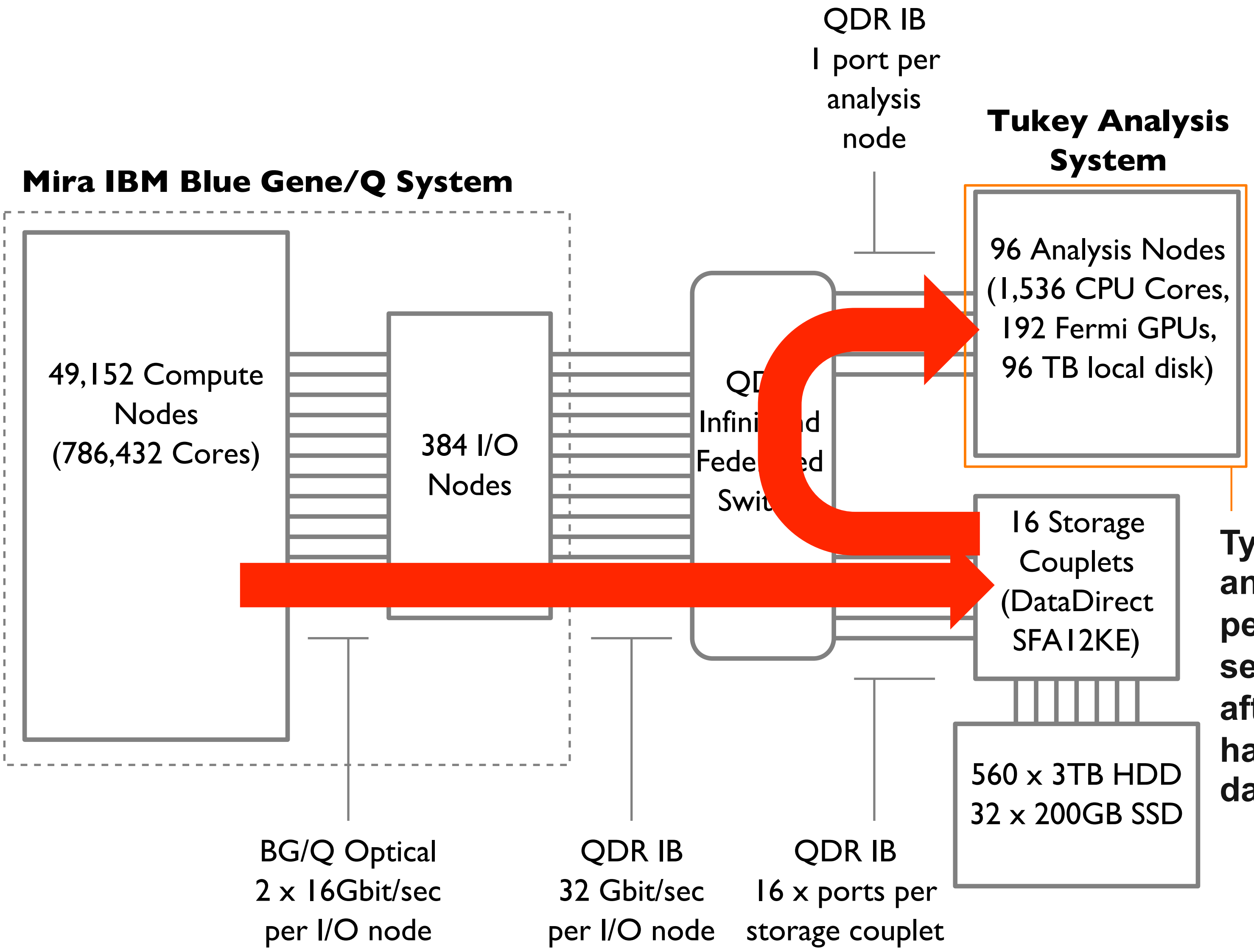
Changes in Data Analysis Workflows

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



Mira IBM Blue Gene/Q System

49,152 Compute Nodes (786,432 Cores)

384 I/O Nodes

QD
Infini nd
Fede ed
Swit

Tukey Analysis System

96 Analysis Nodes (1,536 CPU Cores, 192 Fermi GPUs, 96 TB local disk)

16 Storage Couplets (DataDirect SFA12KE)

560 x 3TB HDD
32 x 200GB SSD

QDR IB
1 port per analysis node

BG/Q Optical
2 x 16Gbit/sec per I/O node

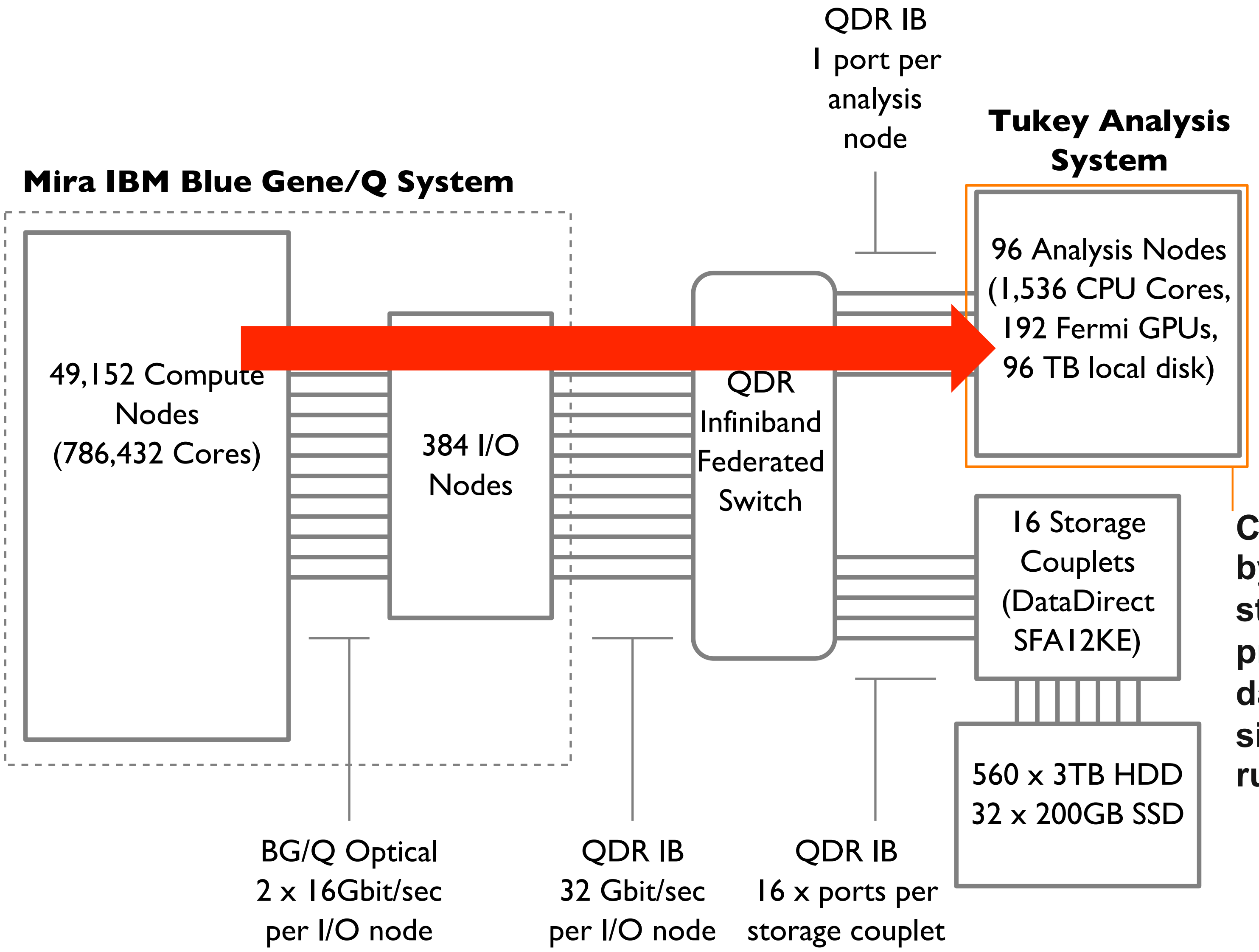
QDR IB
32 Gbit/sec per I/O node

QDR IB
16 x ports per storage couplet

Typically analysis is performed on a separate cluster, after simulation has written to data to disk.

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

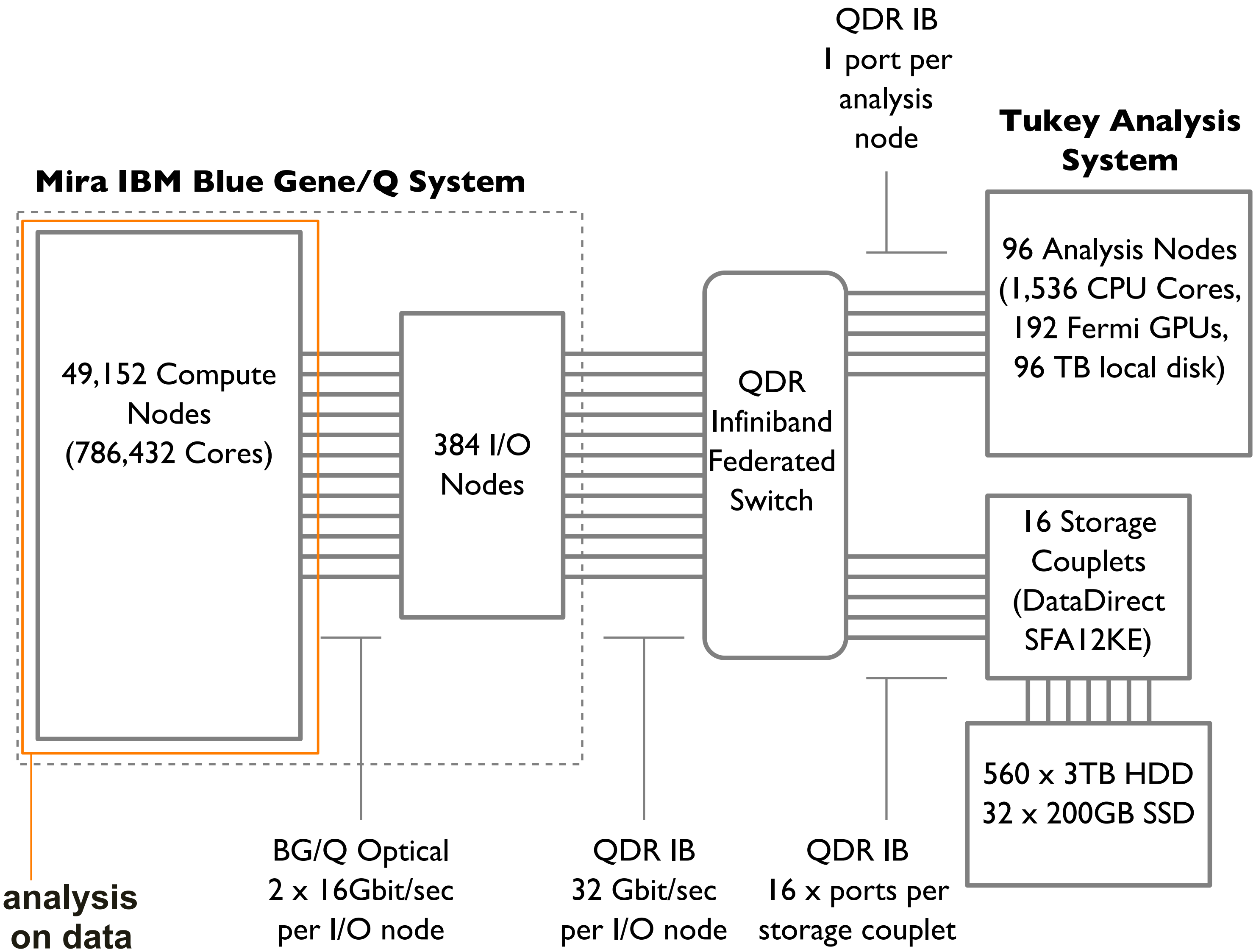
Analyzing Data: Co-Analysis



Co-analysis bypasses storage and processes data while simulation runs.

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

Analyzing Data: In Situ Analysis

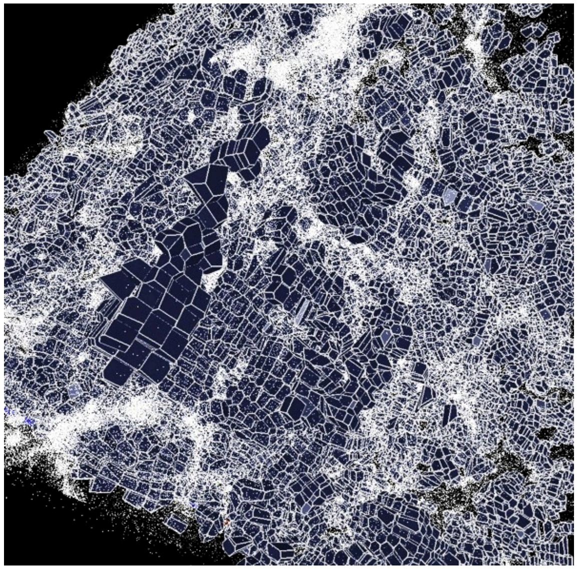

Mira IBM Blue Gene/Q System
49,152 Compute Nodes (786,432 Cores)
384 I/O Nodes
QDR Infiniband Federated Switch
Tukey Analysis System
96 Analysis Nodes (1,536 CPU Cores, 192 Fermi GPUs, 96 TB local disk)
16 Storage Couplets (DataDirect SFA12KE)
560 x 3TB HDD
32 x 200GB SSD
QDR IB
1 port per analysis node
BG/Q Optical
2 x 16Gbit/sec per I/O node
QDR IB
32 Gbit/sec per I/O node
QDR IB
16 x ports per storage couplet

"In situ" analysis operates on data before it leaves the compute nodes.

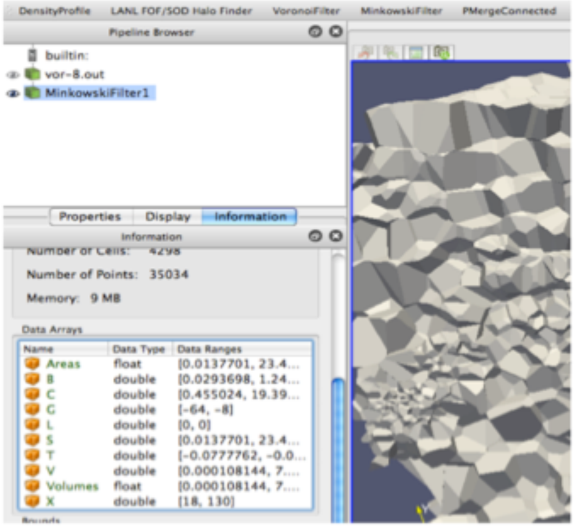
High-level diagram of 10 Pflop IBM Blue Gene/Q system at Argonne Leadership Computing Facility

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.



Number of Cells: 4298

Number of Points: 35034

Memory: 9 MB

Name	Data Type	Data Ranges
Areas	float	[0.0137701, 23.4...
B	double	[0.0293698, 1.24...
C	double	[0.455024, 19.39...
G	double	[-64, -8]
L	double	[0, 0]
S	double	[0.0137701, 23.4...
T	double	[-0.0777762, -0.0...
V	double	[0.000108144, 7....
Volumes	float	[0.000108144, 7....
X	double	[18, 130]

ParaView plugin provides interactive feature exploration.

Collaboration with Kitware and U. of Tennessee

In-System Storage

Many thanks to:

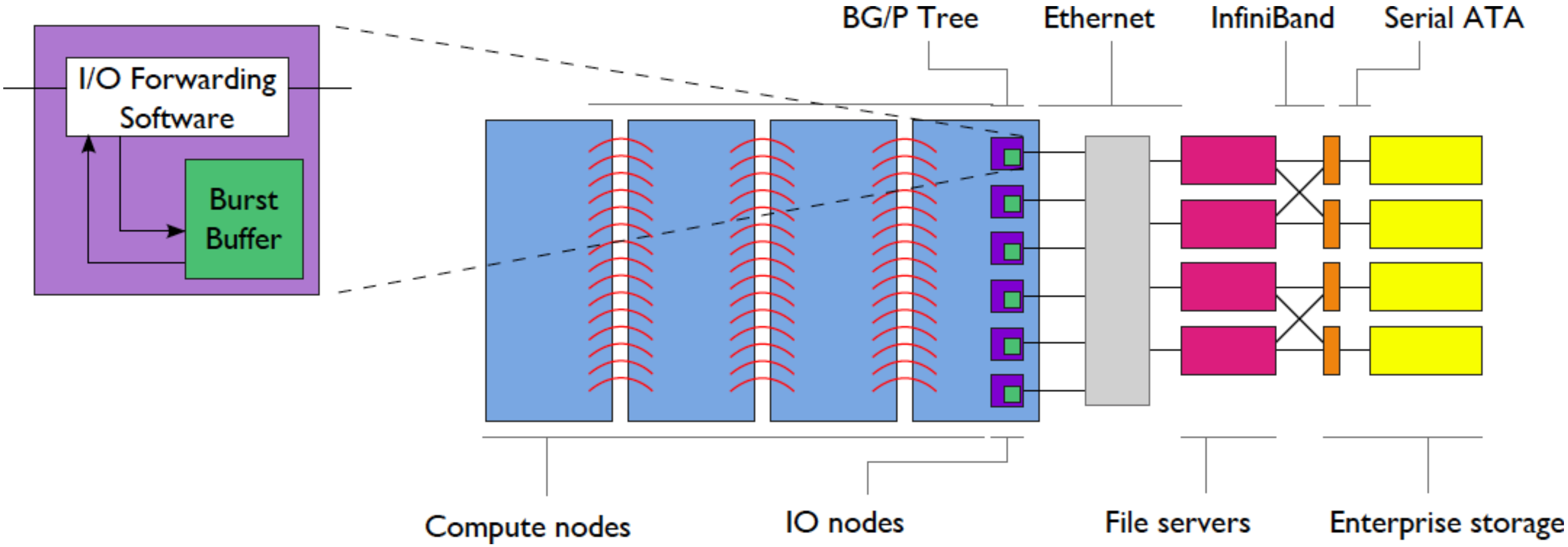
Ning Liu
Illinois Institute of Technology

Jason Cope
DataDirect Networks

Chris Carothers
Rensselear Polytechnic Institute

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".

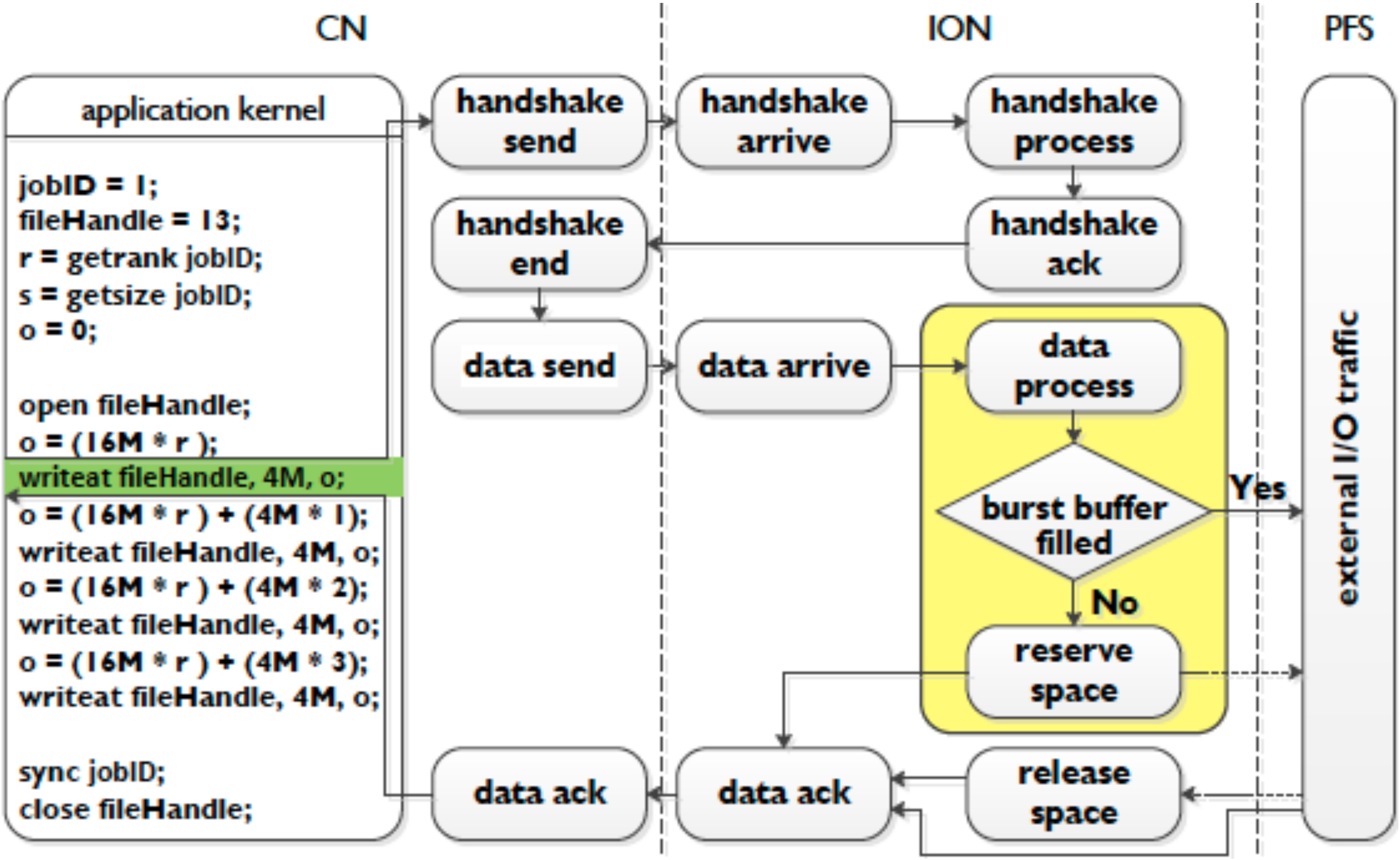


What’s a Burst?

- 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	Run Time	Avg. Size and Subsequent Idle Time for Write Bursts>1 GiB				
			Written	(hours)	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

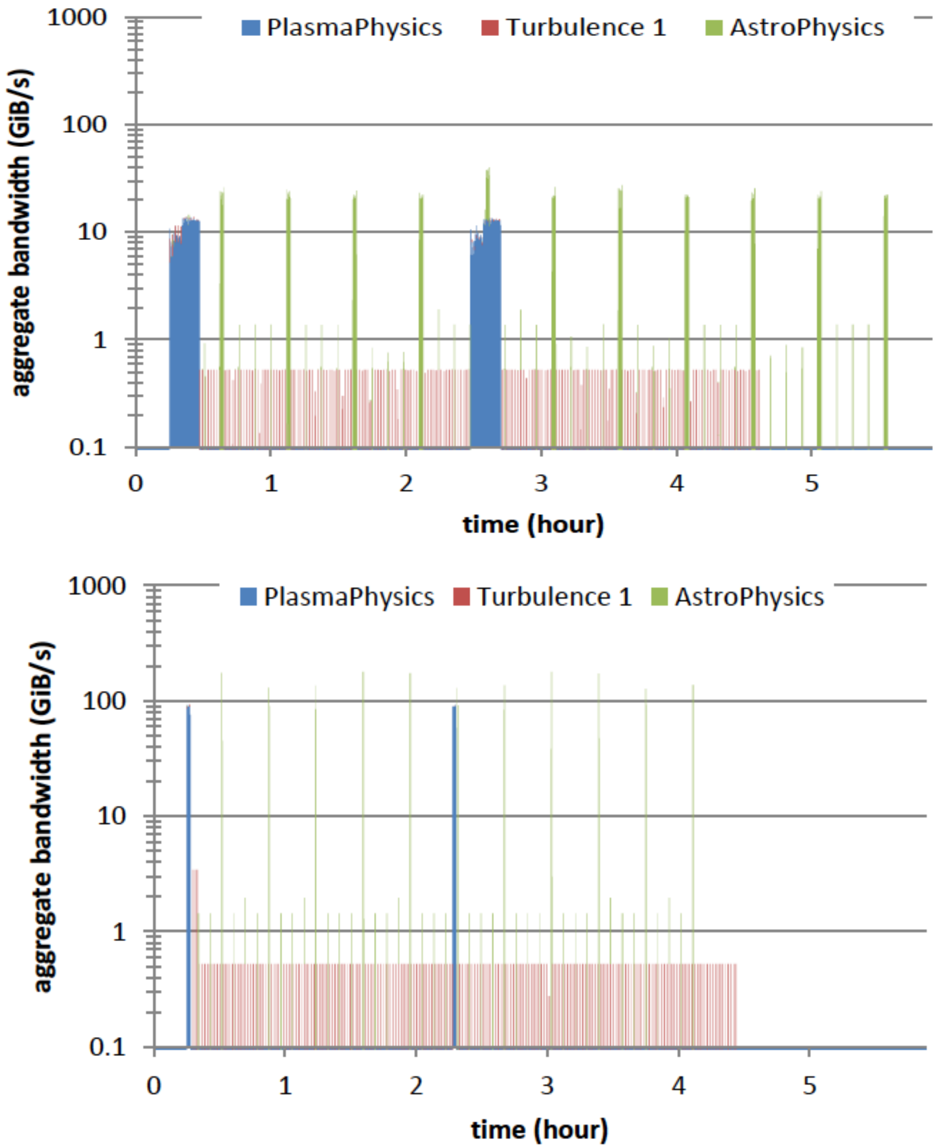
Studying Burst Buffers with Parallel Discrete Event Simulation



```
jobID = 1;
fileHandle = 13;
r = getrank jobID;
s = getsize jobID;
o = 0;

open fileHandle;
o = (16M * r );
writeat fileHandle, 4M, o;
o = (16M * r ) + (4M * 1);
writeat fileHandle, 4M, o;
o = (16M * r ) + (4M * 2);
writeat fileHandle, 4M, o;
o = (16M * r ) + (4M * 3);
writeat fileHandle, 4M, o;

sync jobID;
close fileHandle;
```

Burst Buffers Work for Multi-application Workloads

- 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



Application perceived I/O rates, with no burst buffer (top), burst buffer (bottom).

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 our example)
  - On some/all compute nodes?

Replacing the File Storage Model

Many thanks to:

Dave Goodell
Cisco Systems

Shawn Kim
Penn State University

Mahmut Kandemir
Penn State University

The Problem with the File Storage Model

- The POSIX file model gives us a single stream to work with
- HPC applications create complex output that is naturally multi-stream
  - Structured datasets (e.g., PnetCDF, HDF5)
  - Log-based datasets (e.g., PLFS, ADIOS BP)
- Dilemma
  - Do I create lots of files to hold all these streams?
  - Do I map all the streams into a single file?

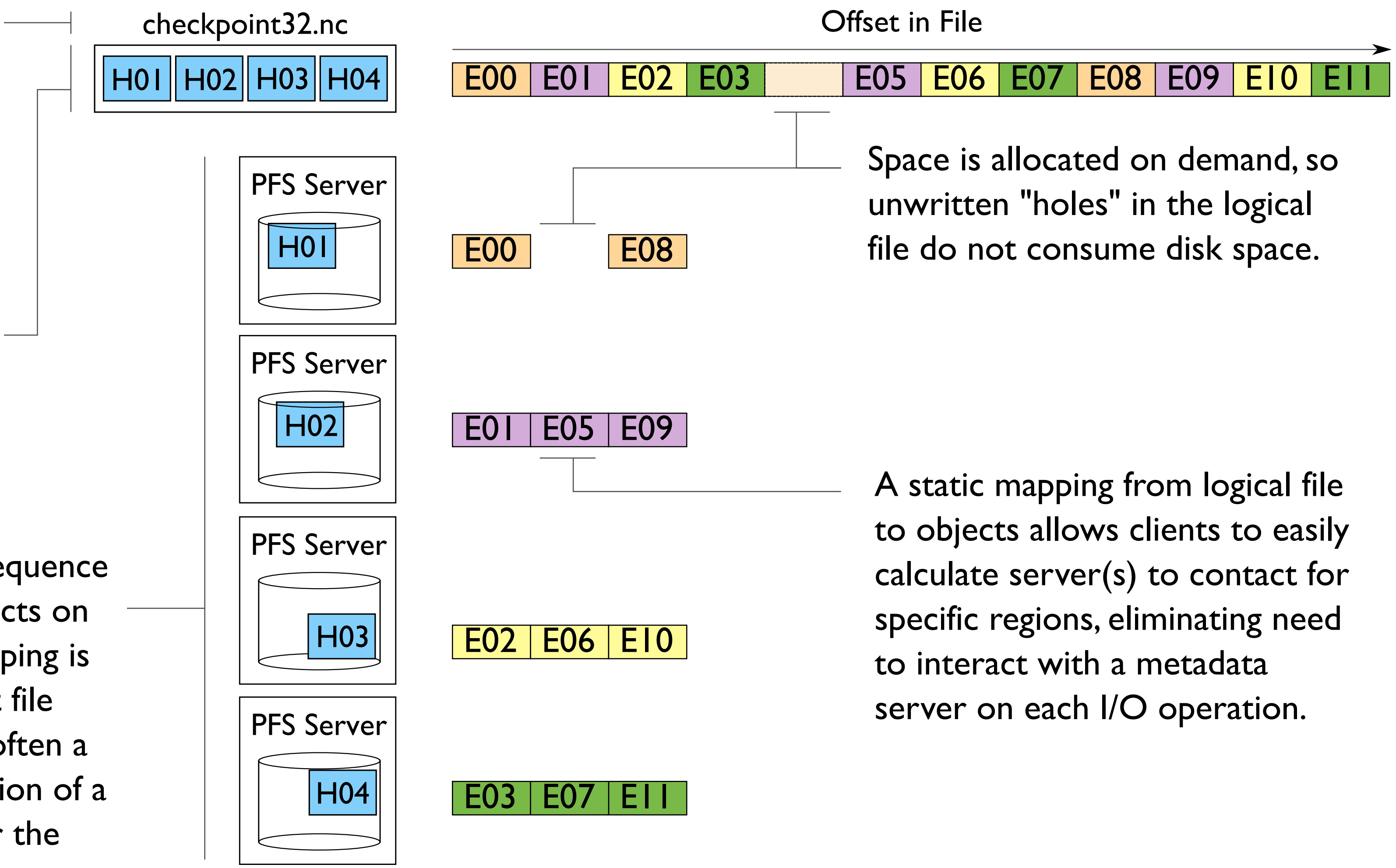
Recall: Data Distribution in Parallel File Systems

Modern parallel file systems internally manage multiple data streams; they just aren’t exposed to the user.

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.



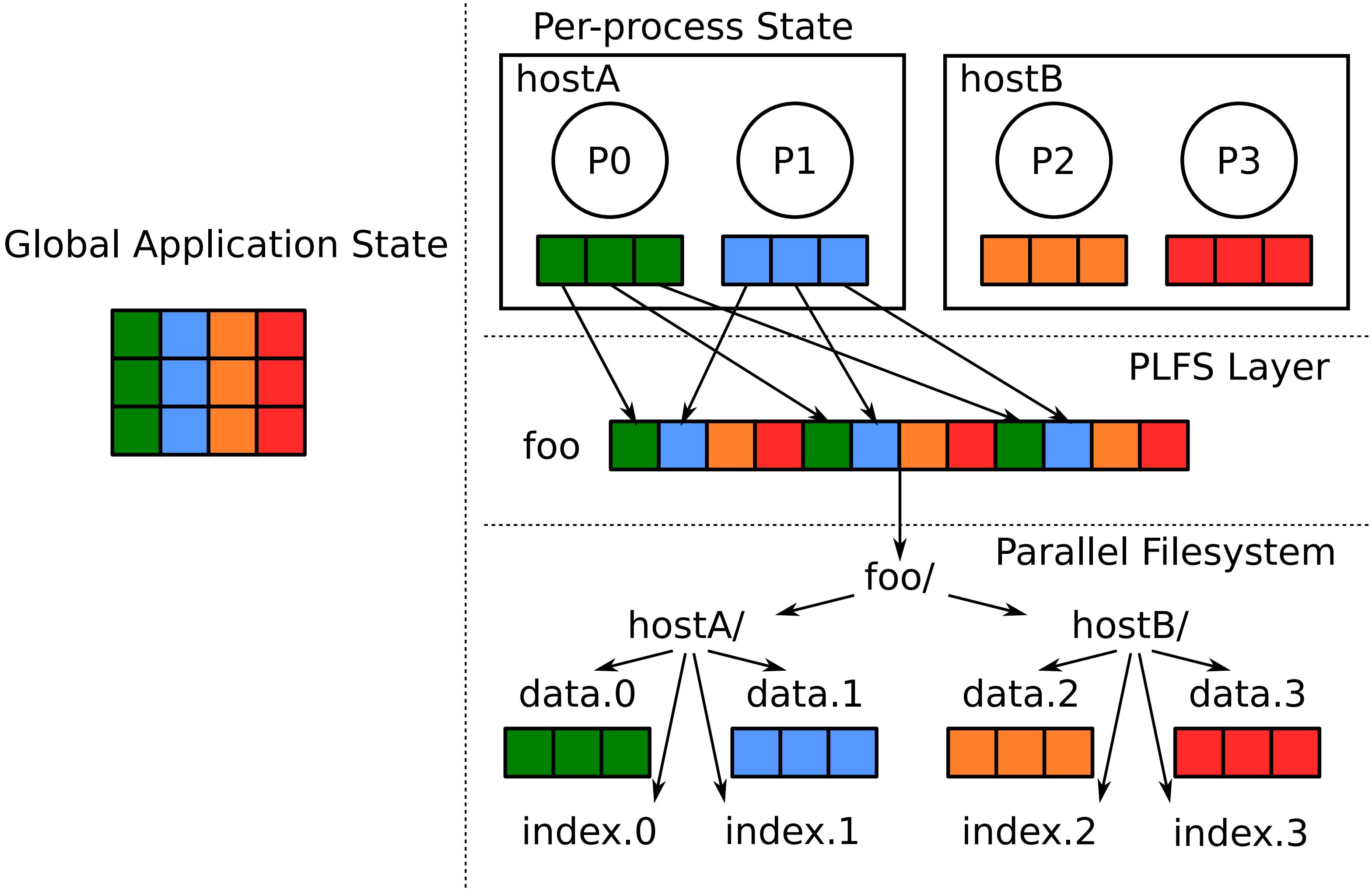
Space is allocated on demand, so unwritten "holes" in the logical file do not consume disk space.

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.

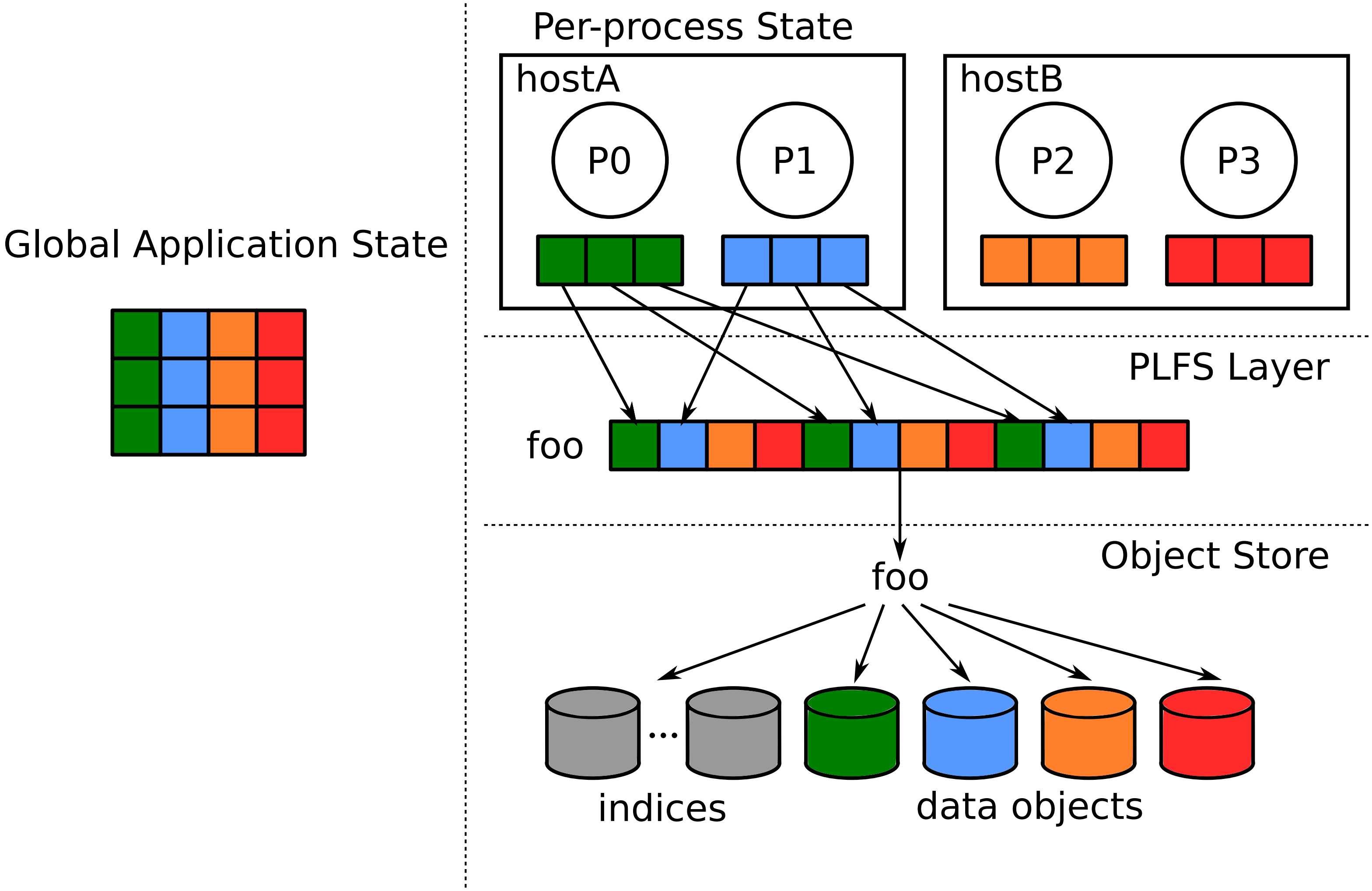
An Alternative Storage Model

- Expose individual object streams for use by users and I/O libraries
  - Users/libraries become responsible for mapping of data to objects
- Keep the name space as it is
  - Directories, file names, permissions
- General approach is being pursued by the Intel Fast Forward team (Intel, EMC, HDF5 Group) and also by ANL/Penn State

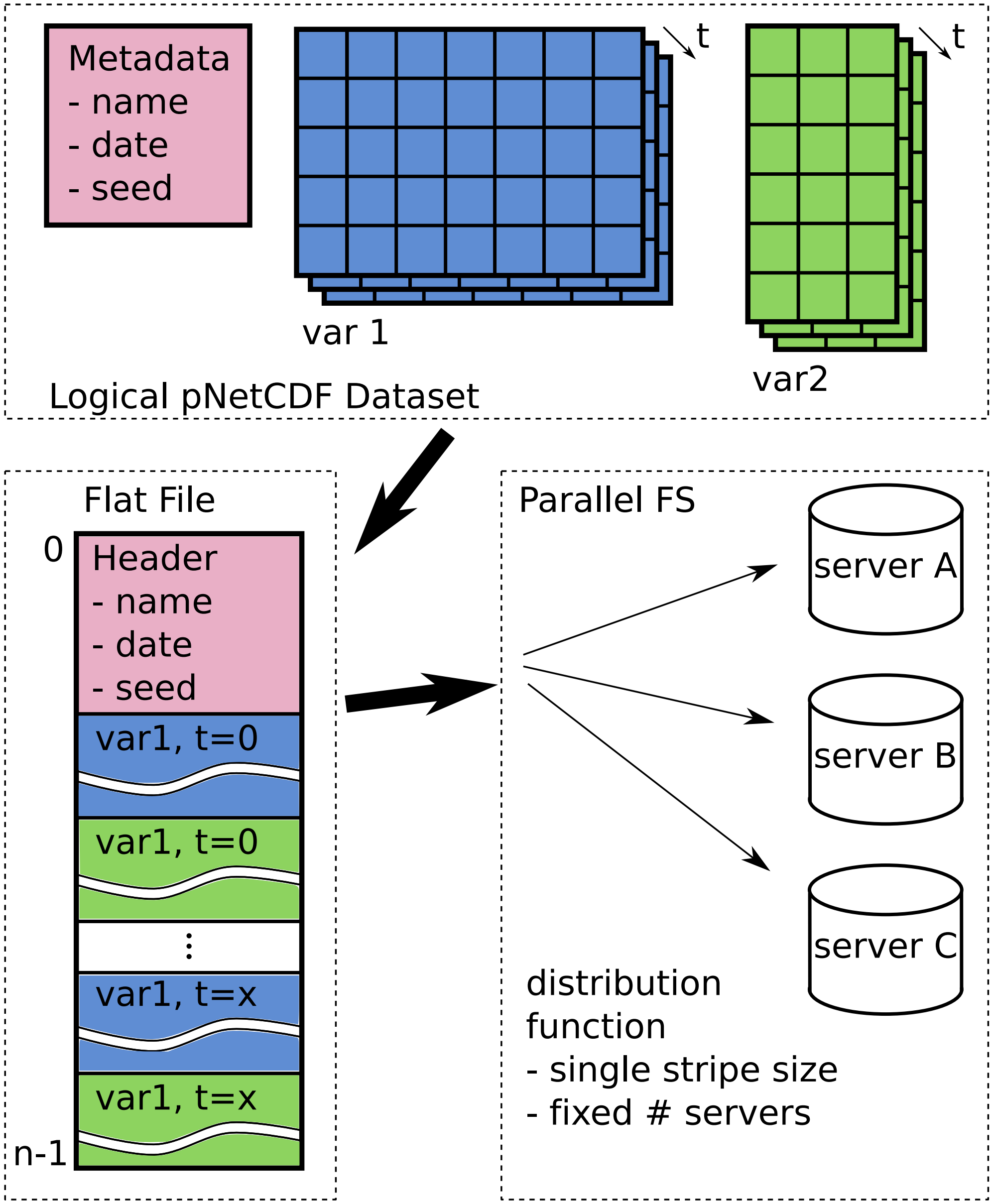
PLFS Mapping to POSIX Files



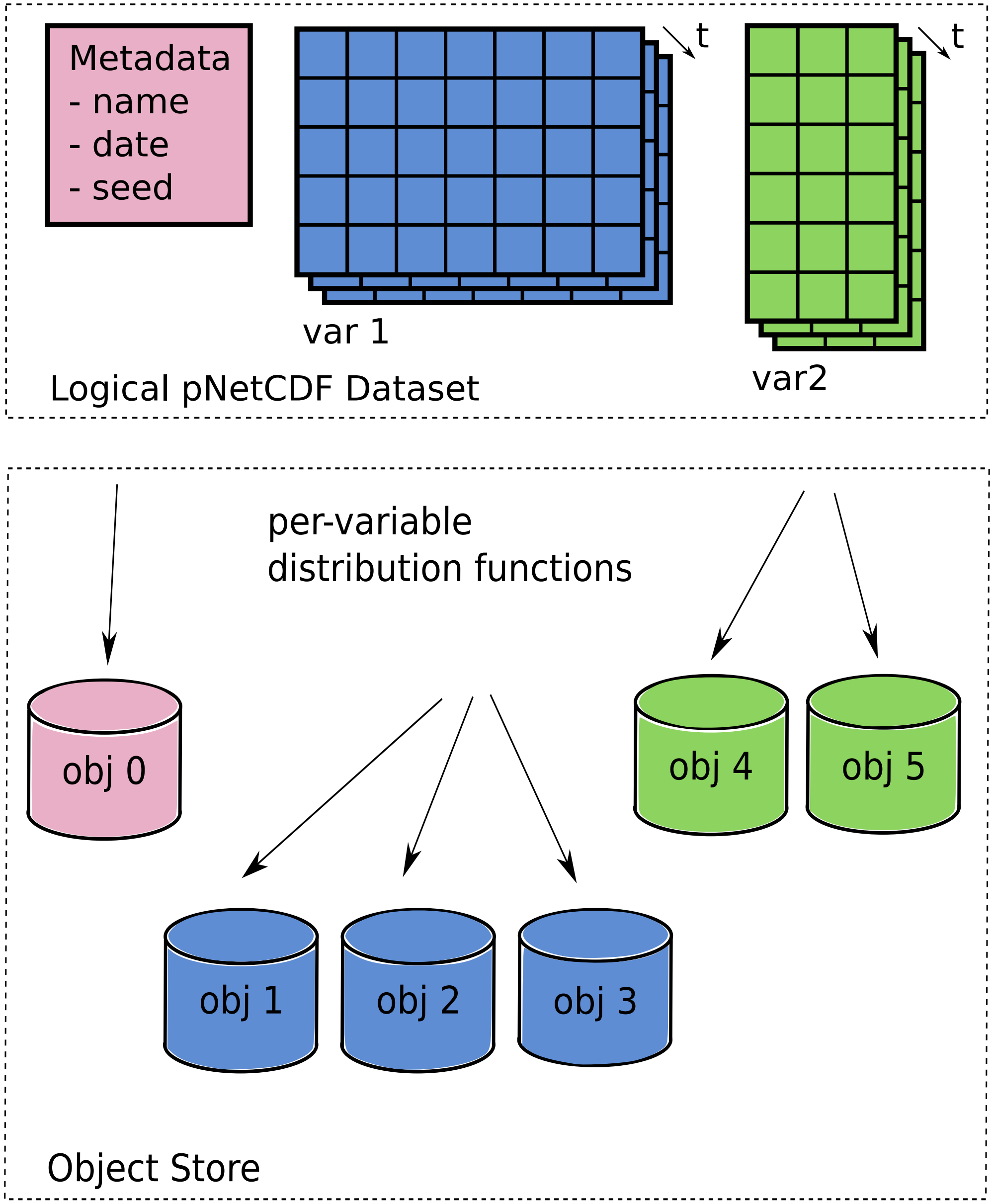
PLFS Mapping to Alternative Storage Model



PnetCDF Mapping to POSIX Files



PnetCDF Mapping to Alternative Storage Model



Other Interesting Ideas

- Lots of alternatives being kicked around in various contexts:
  - Record-oriented storage
  - Forks
  - Search / alternative name spaces
  - Versioning storage
- Our hope is that we see a standard replacement emerge for shared block storage and the file system model

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

Thanks for spending the day with us!

- See you after dinner for more discussion, maybe some hacking?

Acknowledgments

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357, in part through the Scientific Discovery through Advanced Computing (SciDAC) Institute of Scalable Data Management, Analysis and Visualization.

This work used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.