Parallel I/O in Practice

Rob Ross
Mathematics and Computer Science Division
Argonne National Laboratory
rross@mcs.anl.gov

U.S. Department of Energy

UChicago ▶ Argonne LLC

Office of Science
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

... for a brighter future

Computational Science

IBM BG/L system.

- Use of computer simulation as a tool for greater understanding of the real world
  - Complements experimentation and theory
- Problems are increasingly computationally challenging
  - 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, portable interfaces
  - Understanding and tuning I/O
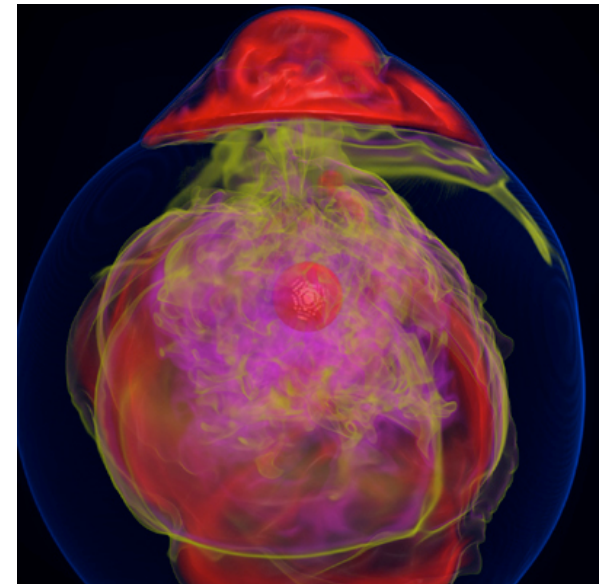
Visualization of entropy in Terascale Supernova Initiative application. Image from Kwan-Liu Ma's visualization team at UC Davis.

Large-Scale Data Sets

Application teams are beginning to generate 10s of Tbytes of data in a single simulation. For example, a recent GTC run on 29K processors on the XT4 generated over 54 Tbytes of data in a 24 hour period [1].

Data requirements for select 2008 INCITE applications at ALCF

PI	Project	On-Line Data	Off-Line Data
Lamb, Don	FLASH: Buoyancy-Driven Turbulent Nuclear Burning	75TB	300TB
Fischer, Paul	Reactor Core Hydrodynamics	2TB	5TB
Dean, David	Computational Nuclear Structure	4TB	40TB
Baker, David	Computational Protein Structure	1TB	2TB
Worley, Patrick H.	Performance Evaluation and Analysis	1TB	1TB
Wolverton, Christopher	Kinetics and Thermodynamics of Metal and Complex Hydride Nanoparticles	5TB	100TB
Washington, Warren	Climate Science	10TB	345TB
Tsigelny, Igor	Parkinson's Disease	2.5TB	50TB
Tang, William	Plasma Microturbulence	2TB	10TB
Sugar, Robert	Lattice QCD	1TB	44TB
Siegel, Andrew	Thermal Striping in Sodium Cooled Reactors	4TB	8TB
Roux, Benoit	Gating Mechanisms of Membrane Proteins	10TB	10TB

[1] S. Klasky, personal correspondence, June 19, 2008.

"There is no physics without I/O."

— Anonymous Physicist
SciDAC Conference
June 17, 2009

(I think he might have been kidding.)
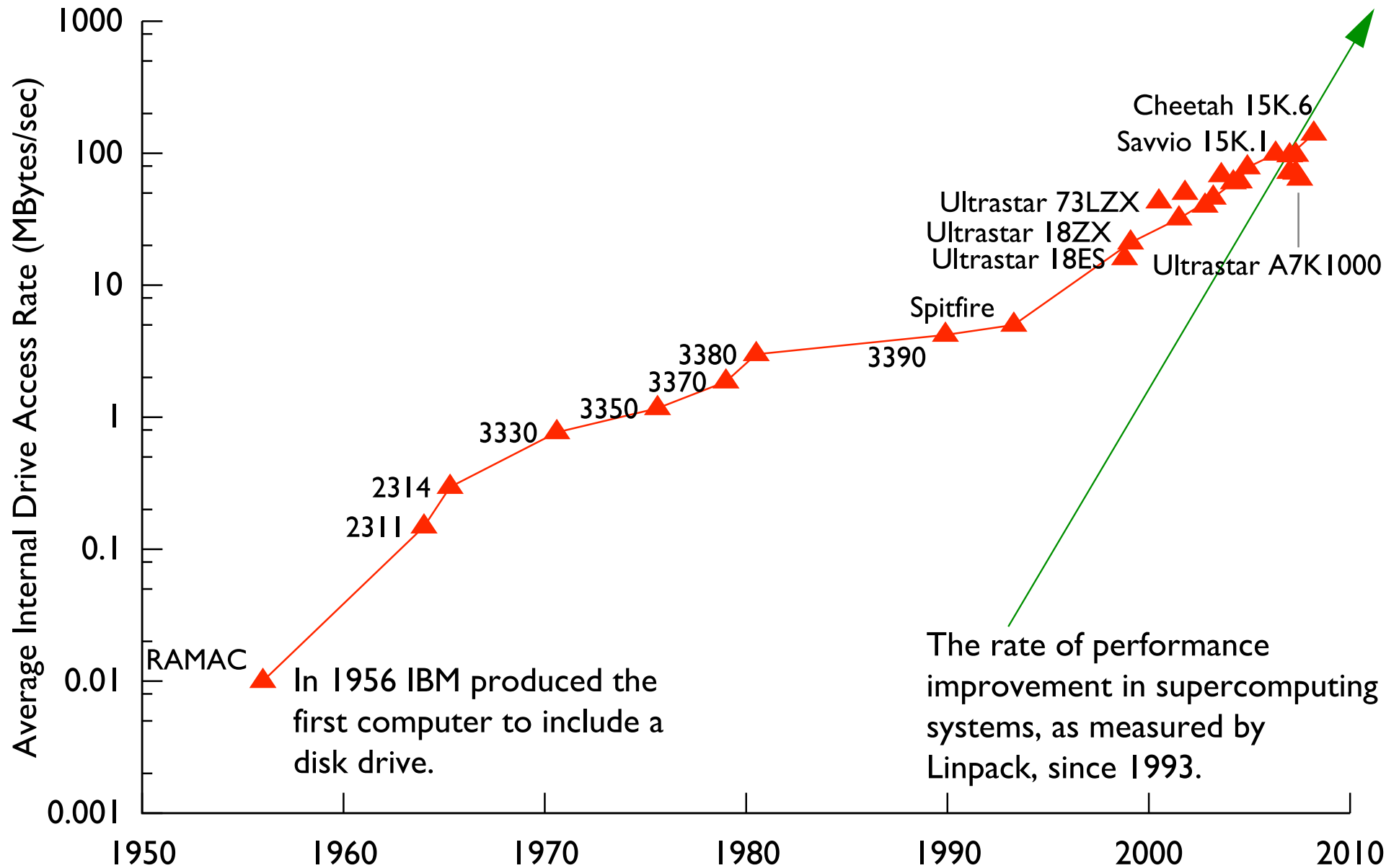
This Talk

Part 1 (Today)

- – Describe the lower layers of parallel I/O systems (storage and I/O middleware) and how these layers contribute to performance and reliability

- – Provide an understanding of how these pieces fit together to provide a resource for computational science applications

Part 2 (Tomorrow)

- – Detail the use of higher-level I/O libraries in computational science applications and how these fit with lower layers

- – Discuss how I/O can be tuned and tools to facilitate understanding of I/O in applications

Storage Hardware and Parallel File Systems

Disk Access Rates over Time

The rate of performance improvement in supercomputing systems, as measured by Linpack, since 1993.

In 1956 IBM produced the first computer to include a disk drive.

Thanks to R. Freitas of IBM Almaden Research Center for providing much of the data for this graph.

Blue Gene/P Parallel Storage System

BG/P Tree	Ethernet	InfiniBand	Serial ATA
6.8 Gbit/sec	10 Gbit/sec	16 Gbit/sec	3.0 Gbit/sec
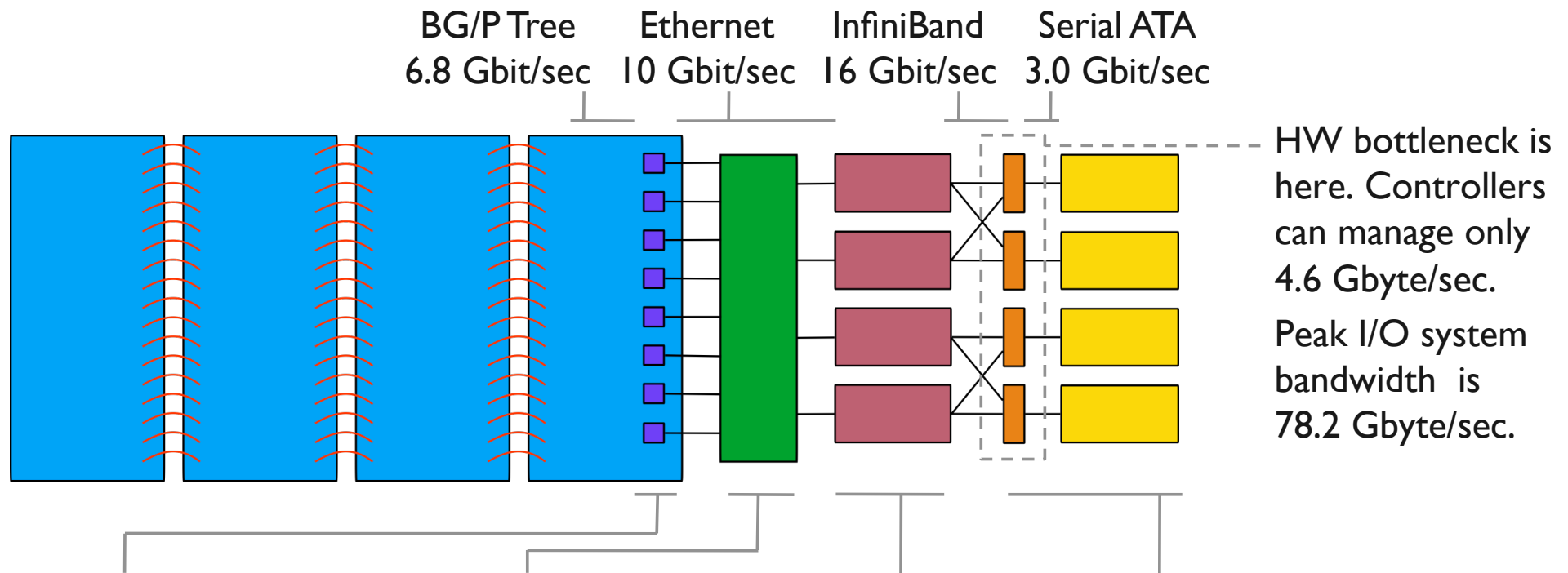
HW bottleneck is here. Controllers can manage only 4.6 Gbyte/sec.

Peak I/O system bandwidth is 78.2 Gbyte/sec.

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

640 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

Commodity network primarily carries storage traffic

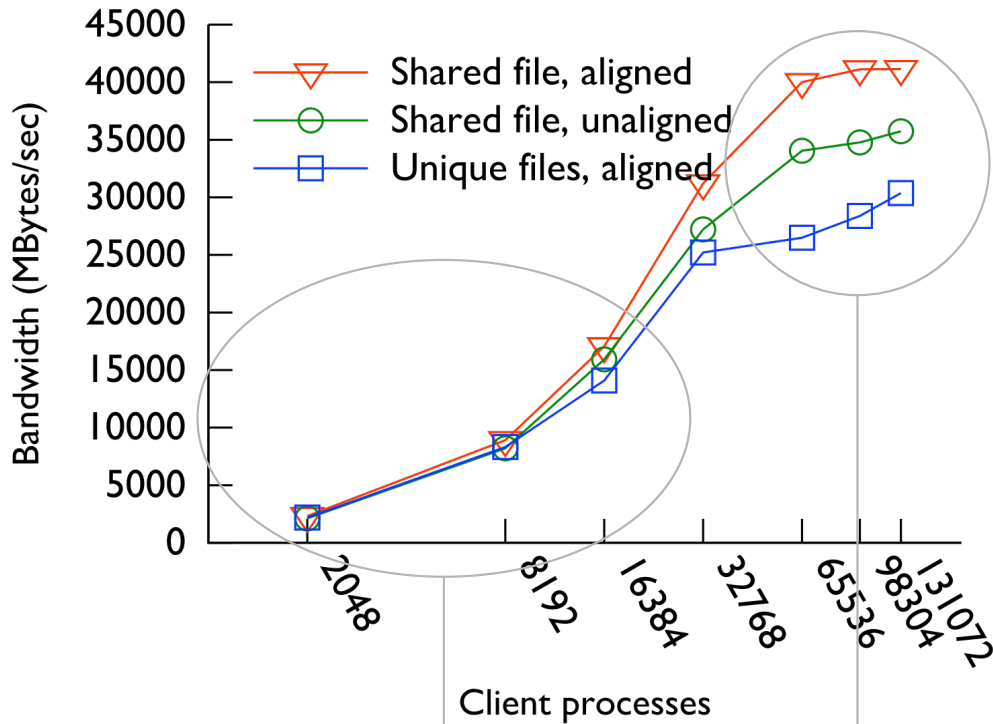*900+ port 10 Gigabit Ethernet Myricom switch complex*

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

136 two dual core Opteron servers with 8 Gbytes of RAM each

Enterprise storage controllers and large racks of disks connected via InfiniBand or Fibre Channel

17 DataDirect S2A9900 controller pairs with 480 1 Tbyte drives and 8 InfiniBand ports per pair

Architectural diagram of 557 TF Argonne Leadership Computing Facility Blue Gene/P I/O system.

Snapshot of Performance on Blue Gene/P

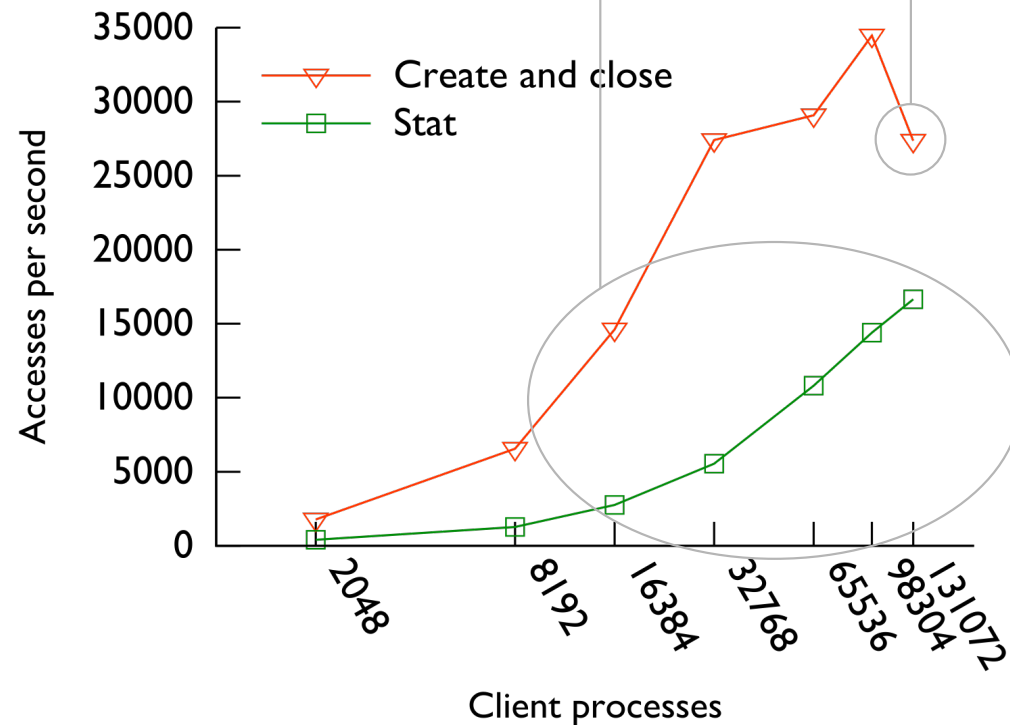## POSIX aggregate write performance (ior)

Legend:
- Shared file, aligned
- Shared file, unaligned
- Unique files, aligned

X-axis (Client processes): 2048, 8192, 16384, 32768, 65536, 98304, 131072
Y-axis (Bandwidth (MBytes/sec)): 0, 5000, 10000, 15000, 20000, 25000, 30000, 35000, 40000, 45000

Low stat performance relative to create may be due to poor choice of server-side cache size (256 Kbytes)?

We believe this drop is due to a disk going bad in a storage rack; waiting on repeat testing to confirm.

Aggregate metadata performance (metarates)

Legend:
- Create and close
- Stat

X-axis (Client processes): 2048, 8192, 16384, 32768, 65536, 98304, 131072
Y-axis (Accesses per second): 0, 5000, 10000, 15000, 20000, 25000, 30000, 35000

Maximum I/O rate of 300 Mbytes/sec per I/O forwarding node limits performance in this region.

Effective BW out of storage racks limits performance in this region (writing to /dev/null achieves around 65 Gbytes/sec).

Lang et. al, "I/O Performance Challenges at Leadership Scale", to appear in SC09, November 2009.

Parallel File Systems
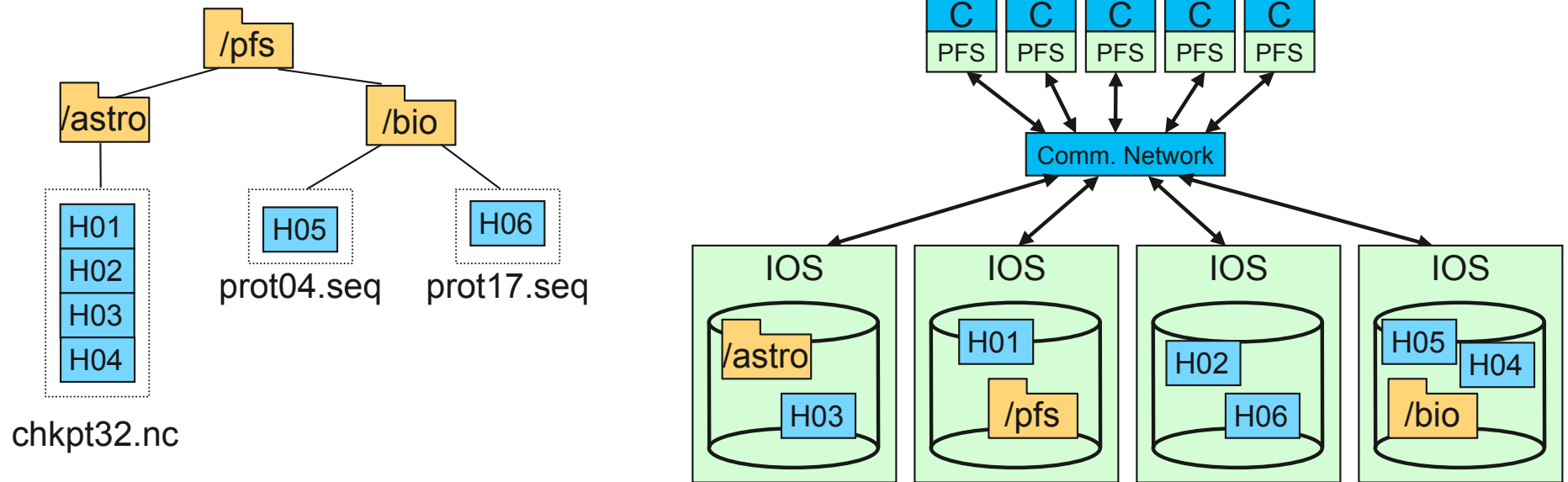
An example parallel file system, with large astrophysics checkpoints distributed across multiple I/O servers (IOS) while small bioinformatics files are each stored on a single IOS.

- Building block for HPC I/O systems
  - Present storage as a single, logical storage unit
  - Stripe files across disks and nodes for performance
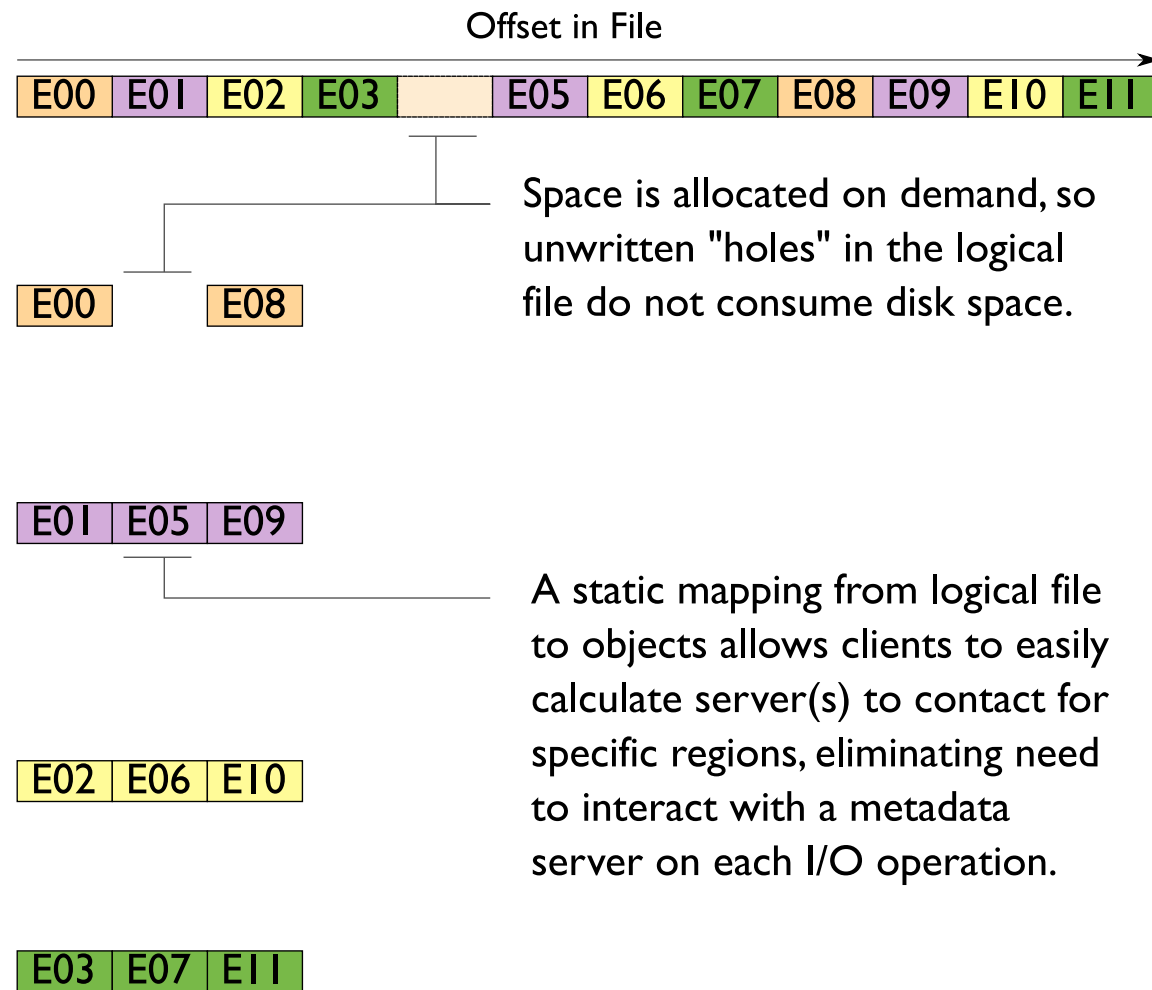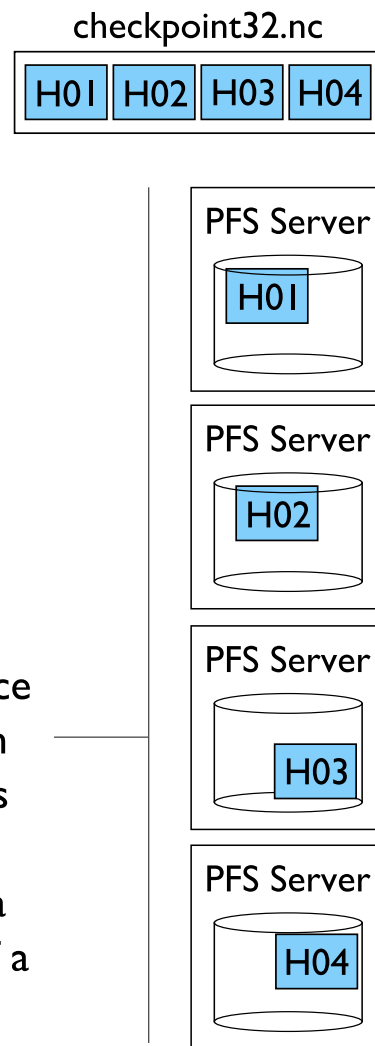 - Tolerate failures (in conjunction with other HW/SW)
- User interface is POSIX file I/O interface, not very good for HPC

Data Distribution in Parallel File Systems

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.

checkpoint32.nc

| H01 | H02 | H03 | H04 |

Offset in File

| E00 | E01 | E02 | E03 | | E05 | E06 | E07 | E08 | E09 | E10 | E11 |

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

PFS Server
H01

| E00 | | E08 |

PFS Server
H02

| E01 | E05 | E09 |

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.

PFS Server
H03

| E02 | E06 | E10 |

PFS Server
H04

| E03 | E07 | E11 |

Locking in Parallel File Systems

Most parallel file systems use **locks** to manage concurrent access to files

- 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.

Offset in File

Lock Boundary

Lock Unit
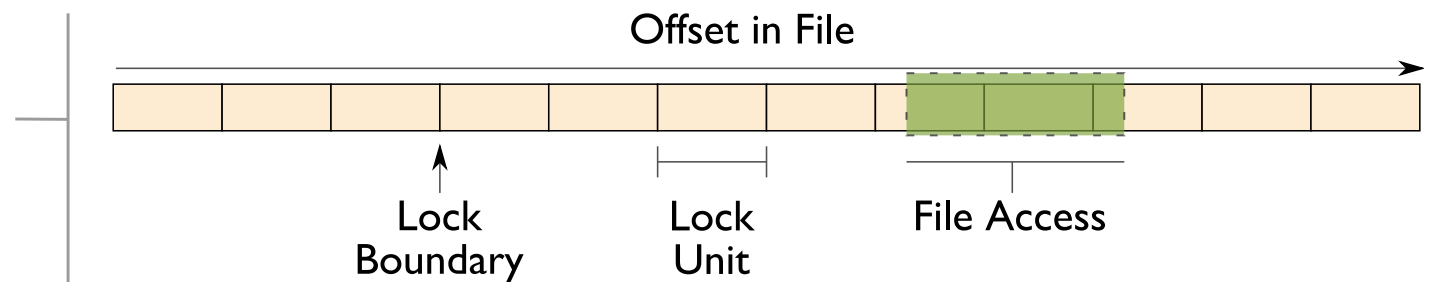
File Access

Locking and Concurrent Access

2D View of Data

Offset in File

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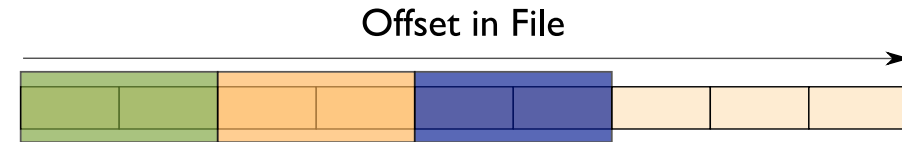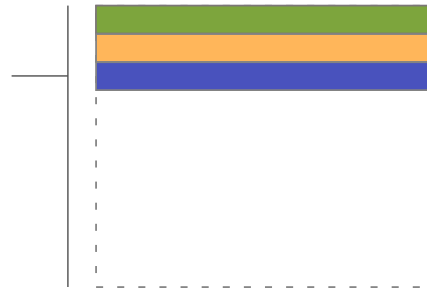
When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

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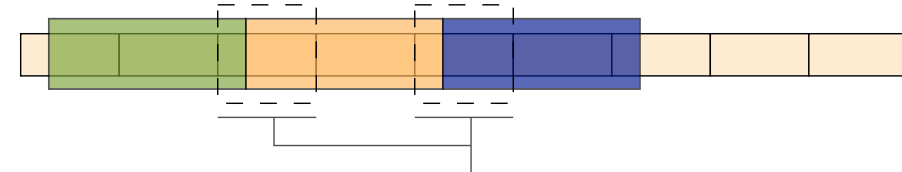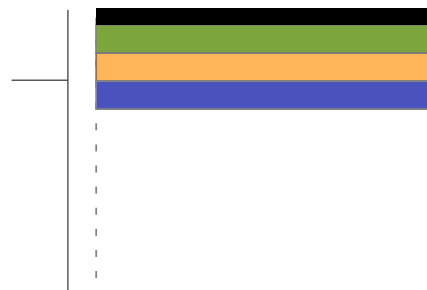
These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

In this example, processes exhibit a block-block access pattern (e.g. accessing a subarray). This results in many interleaved accesses in the file.
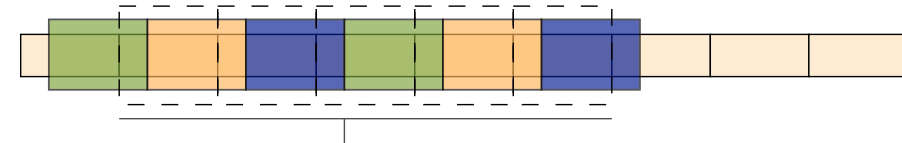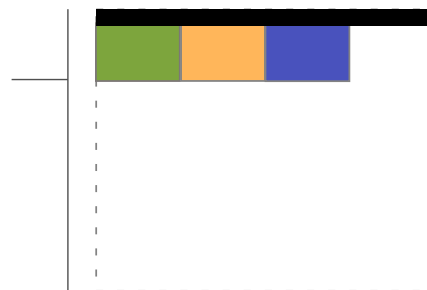
When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

Fault Tolerance and Parallel File Systems

Combination of hardware and software ensures continued operation in face of failures:

- RAID techniques hide disk failures

- Redundant controllers and shared access to storage

- Heartbeat software and quorum directs server failover
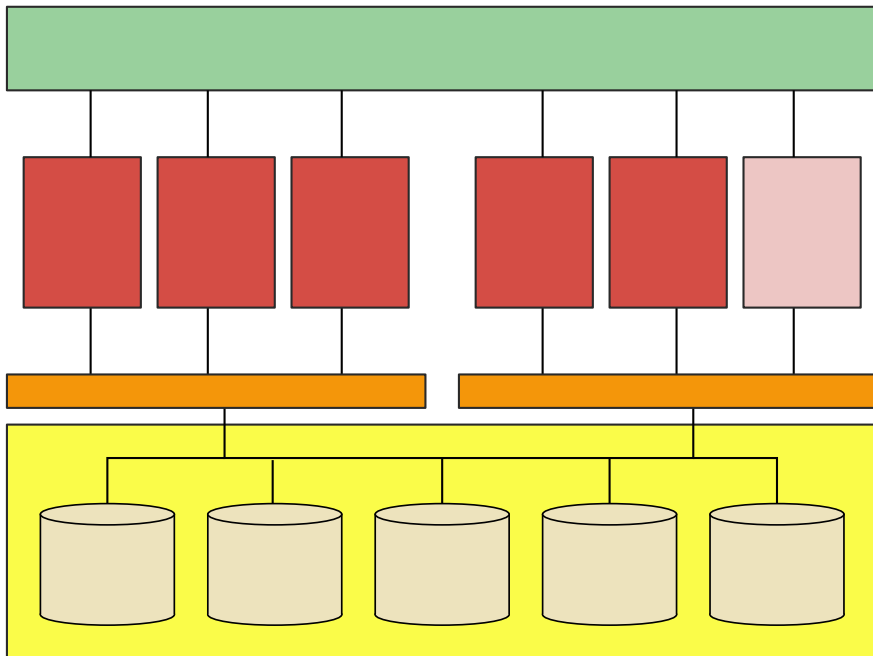
System network connects storage to compute resources.

Storage servers manage independent portions of a shared storage resource. In this diagram, five of six servers are active, while one is passive (a backup).

Storage controllers group individual drives into logical units (LUNs) and use RAID techniques to hide drive failures.

LUNs are accessible by all storage servers, but only one server accesses any LUN at one time.

Computational Science and Parallel I/O

Stressing the I/O System

- Computational science applications exhibit **complex I/O patterns** that are unique, and how we describe these patterns influences performance.
- Accessing from large numbers of processes has the potential to overwhelm the storage system. How we describe the **relationship between accesses** influences performance.
- In some cases we simply need to reduce the number of processes accessing the storage system in order to match number of servers or **limit concurrent access**.

Contiguous and Noncontiguous I/O

Contiguous | Noncontiguous in File | Noncontiguous in Memory | Noncontiguous in Both
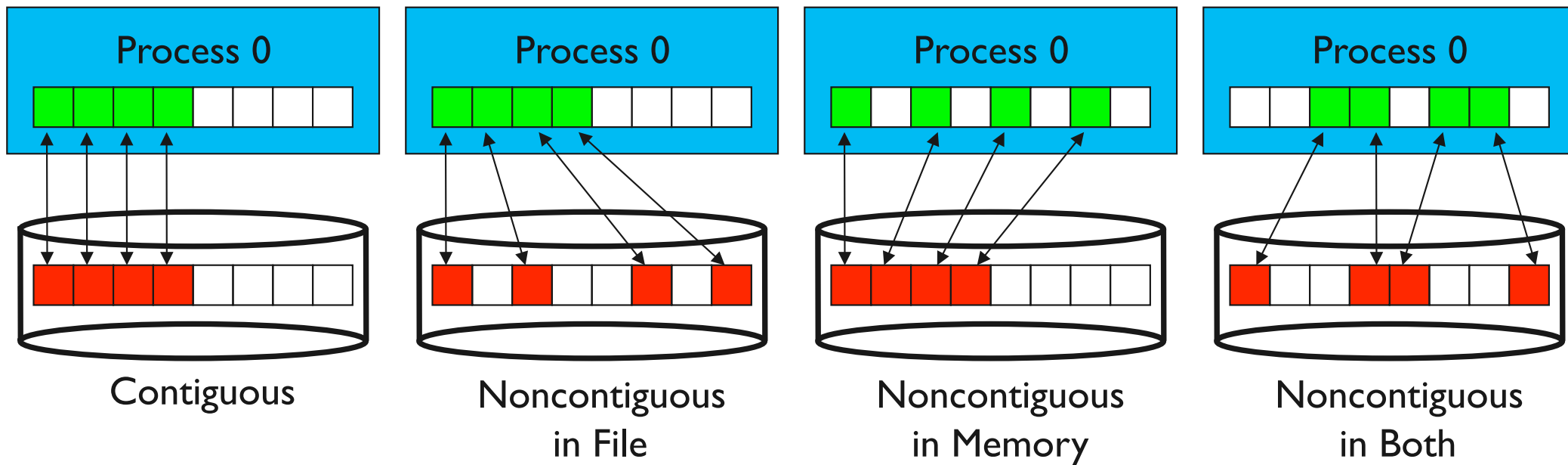
- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system

Independent and Collective I/O

Independent I/O

Collective I/O
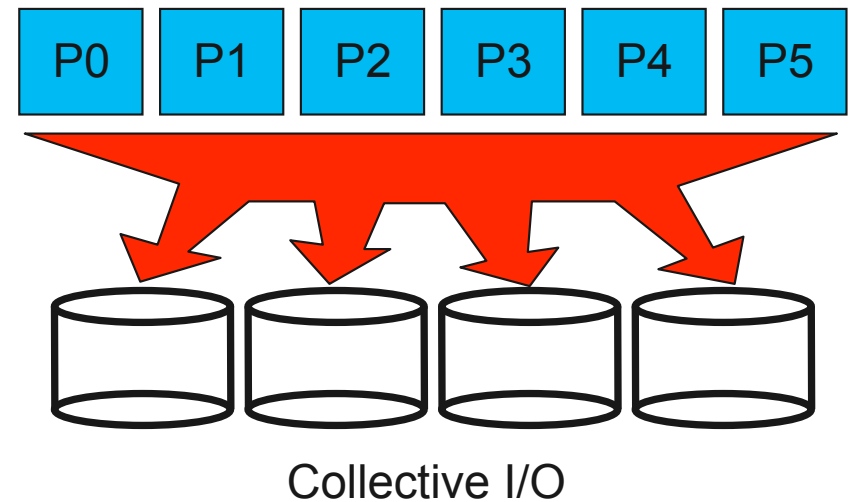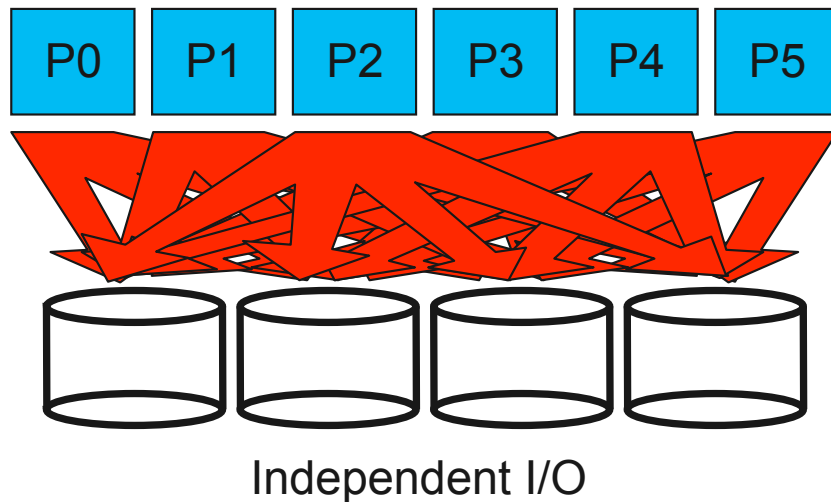
- Independent I/O operations specify only what a single process will do
  - Independent I/O calls obscure relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
- Collective I/O is coordinated access to storage by a group of processes
  - Collective I/O functions are called by all processes participating in I/O
  - Allows I/O layers to know more about access as a whole, more opportunities for optimization in lower software layers, better performance

I/O Aggregation

As the number of nodes on systems grows, the access patterns seen by underlying file systems appear increasingly chaotic. Reducing the apparent number of clients before hitting the file system layer can significantly improve performance.

In this case we have 2K nodes rendering and compositing a 2048^2 image on the Blue Gene/P.

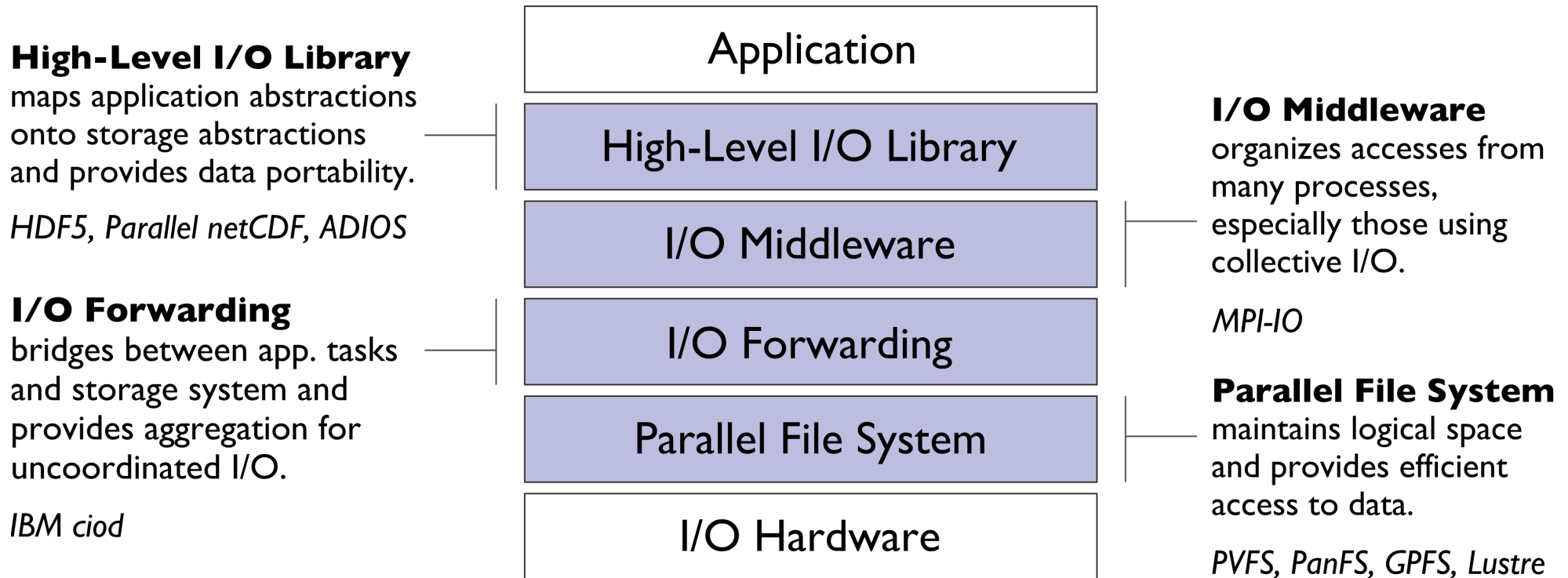There are 64 compute nodes per I/O node. 64 writers corresponds to 2 writers per I/O node.

Composite, Gather, Write Times for Varying Numbers of Writers

Legend:
- Writing time
- Gathering time
- Compositing time

Y-axis: Time (sec)
X-axis: Number of writers
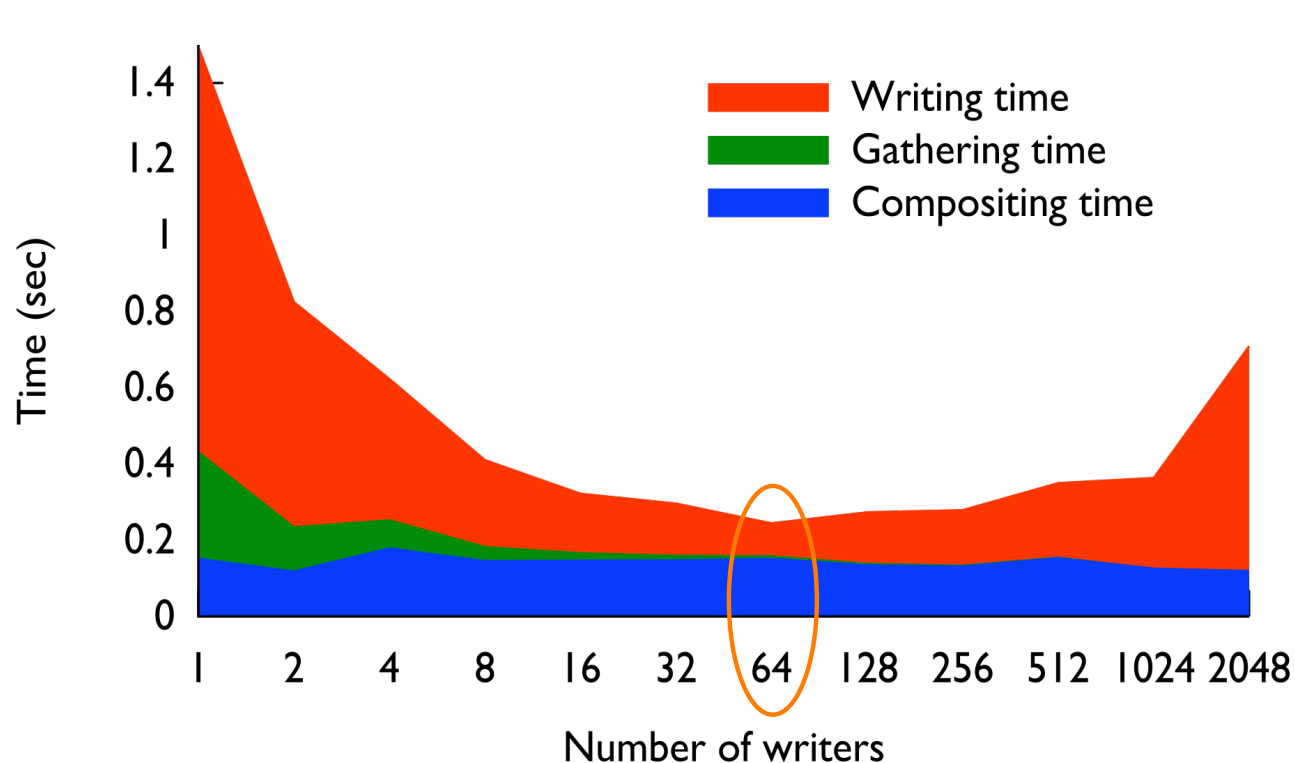
T. Peterka et al, "Assessing Improvements in the Parallel Volume Rendering Pipeline at Large Scale," SC08 Ultrascale Visualization Workshop, November 2008.

The I/O Software Stack

High-Level I/O Library
maps application abstractions onto storage abstractions and provides data portability.

HDF5, Parallel netCDF, ADIOS

I/O Forwarding
bridges between app. tasks and storage system and provides aggregation for uncoordinated I/O.

IBM ciod

Application
High-Level I/O Library
I/O Middleware
I/O Forwarding
Parallel File System
I/O Hardware

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

MPI-IO

Parallel File System
maintains logical space and provides efficient access to data.

PVFS, PanFS, GPFS, Lustre

Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.

I/O Middleware

Application
High-Level I/O Library
I/O Middleware
I/O Forwarding
Parallel File System
I/O Hardware

- Match the programming model (e.g. MPI)
- Facilitate concurrent access by groups of processes
  - Collective I/O
  - Atomicity rules
- Expose a generic interface
  - **Good building block for high-level libraries**
- Efficiently map middleware operations into PFS ones
  - Leverage any rich PFS access constructs, such as
   - *Scalable file name resolution*
    - *Rich I/O descriptions*
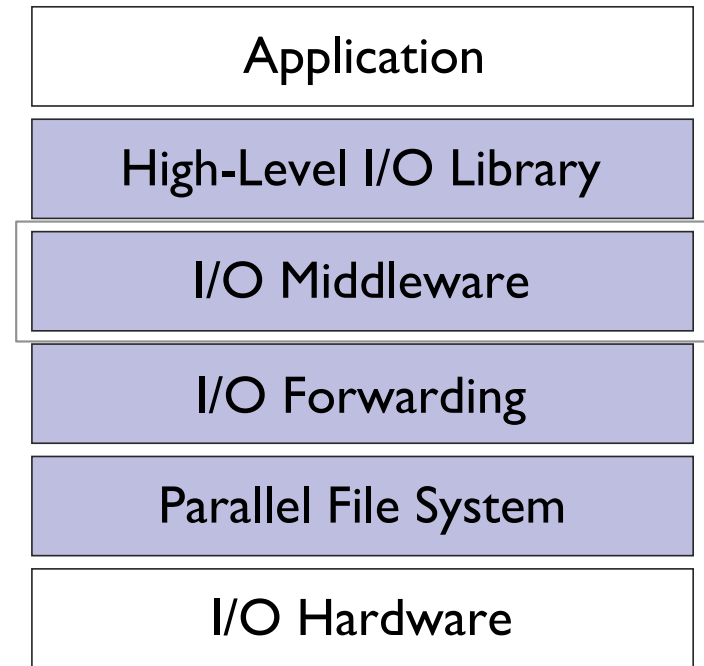
MPI-IO

MPI-IO

- I/O interface specification for use in MPI apps
- Data model is a stream of bytes in a file
  - Same as POSIX and stdio
- Features:
  - Noncontiguous I/O with MPI datatypes and file views
  - Collective I/O
  - Nonblocking I/O
  - Fortran bindings (and additional languages)

Example: Noncontiguous I/O in MPI with the subarray datatype
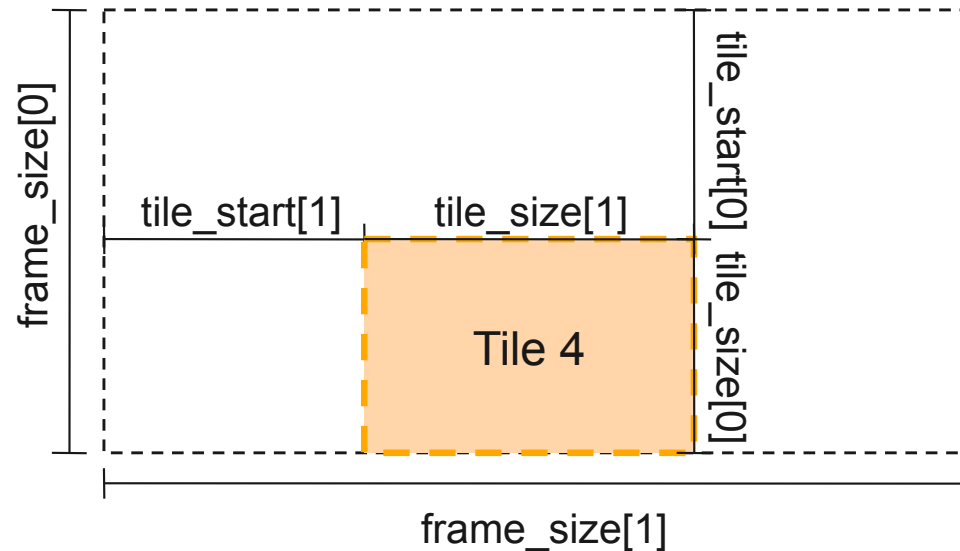
- MPI_Type_create_subarray can describe any N-dimensional subarray of an N-dimensional array
- In this case we use it to pull out a 2-D tile
- Tiles can overlap if we need them to
- Separate MPI_File_set_view call uses this type to select the file region
- More arbitrary structures can be described with MPI datatypes as well

Noncontiguous I/O: Data Sieving

Memory

Buffer
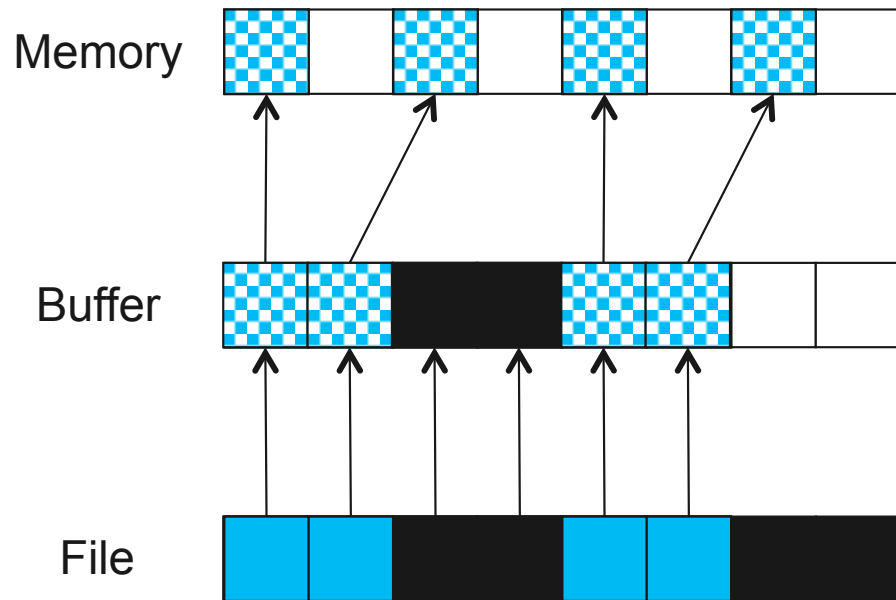
File

Data Sieving Read Transfers

- Data sieving is used to combine lots of small accesses into a single larger one
  - Remote file systems (parallel or not) tend to have high latencies
  - Reducing # of operations important
- Similar to how a block-based file system interacts with storage
- Generally very effective, but not as good as having a PFS that supports noncontiguous access

Data Sieving Write Operations

Memory
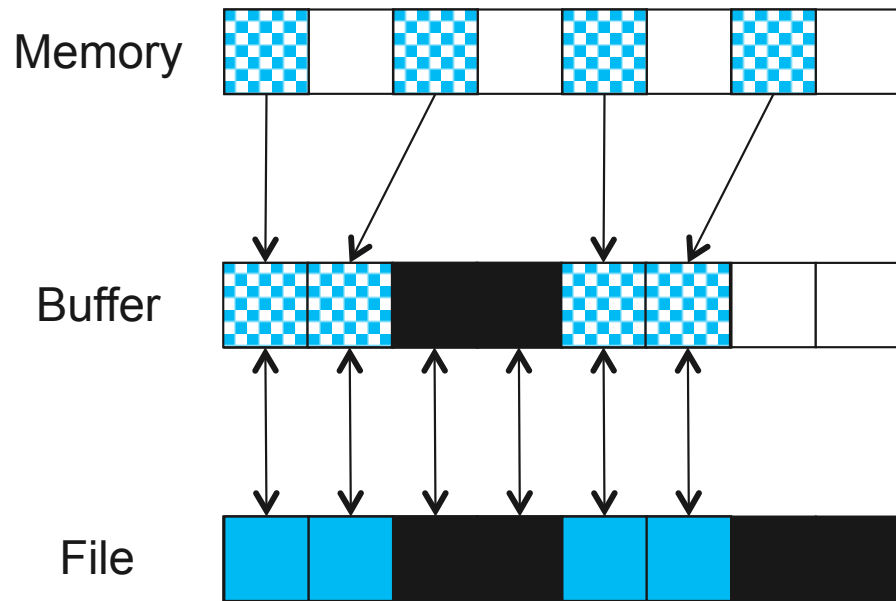
Buffer

File

Data Sieving Write Transfers

- Data sieving for writes is more complicated
  - Must read the entire region first
  - Then make changes in buffer
  - Then write the block back
- Requires locking in the file system
  - Can result in false sharing (interleaved access)
- PFS supporting noncontiguous writes is preferred

Collective I/O Optimization: Two-Phase I/O
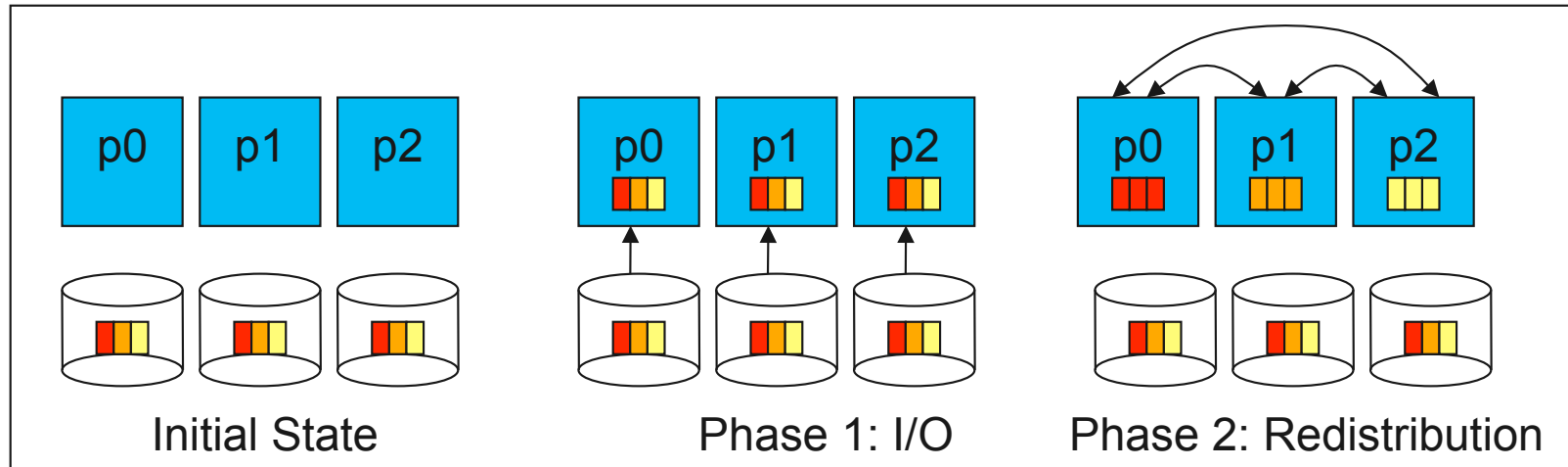
Two-Phase Read Algorithm

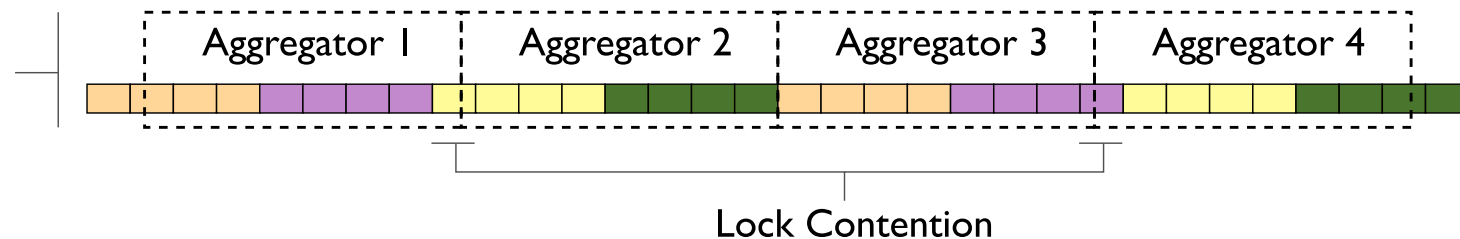- **Problems with independent, noncontiguous access**
  - Lots of small accesses
  - Independent data sieving reads lots of extra data, can exhibit false sharing
- **Idea: Reorganize access to match layout on disks**
  - Single processes use data sieving to get data for many
- **Second "phase" redistributes data to final destinations**
- **Two-phase writes operate in reverse (redistribute then I/O)**

Two-Phase I/O Algorithms

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

Offset in File

Stripe Unit   Lock Boundary   Extent of Accesses
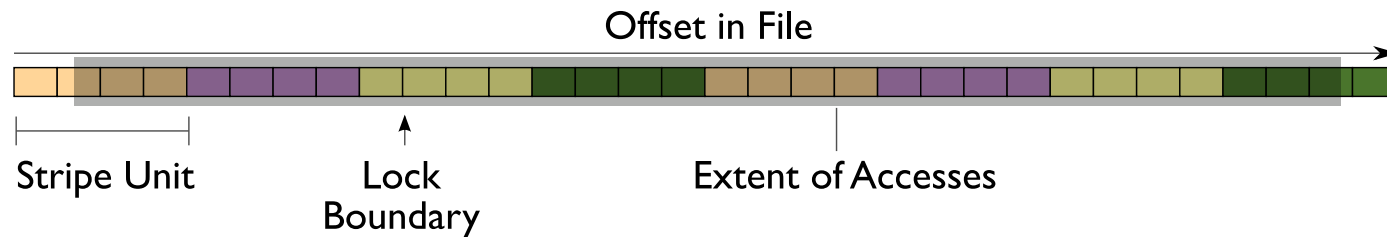
One approach is to evenly divide the region accessed across aggregators.

Aggregator 1   Aggregator 2   Aggregator 3   Aggregator 4

Lock Contention

Aligning regions with lock boundaries eliminates lock contention.

Aggregator 1   Aggregator 2   Aggregator 3   Aggregator 4
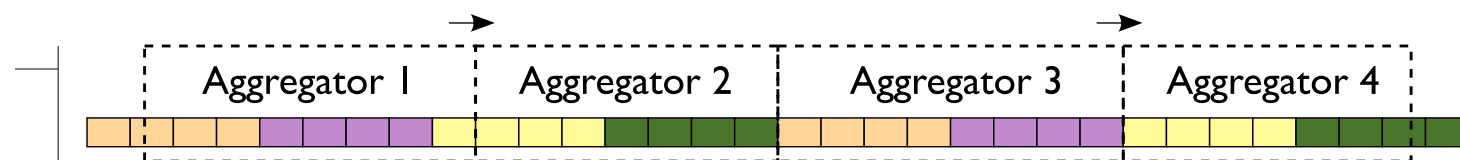
Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).

A1   A2   A3   A4   A1   A2   A3   A4
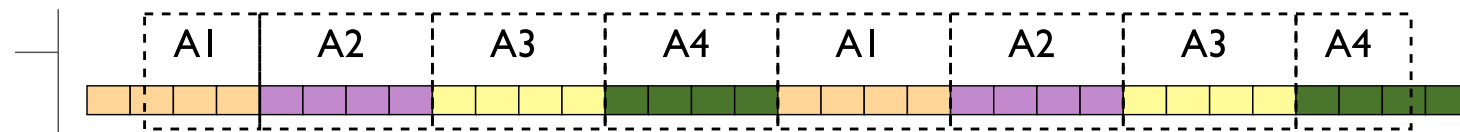
For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November, 2008.

Impact of Two-Phase I/O Algorithms

- This graph shows the performance for the S3D combustion code, writing to a single file.

- Aligning with lock boundaries doubles performance over default "even" algorithm.

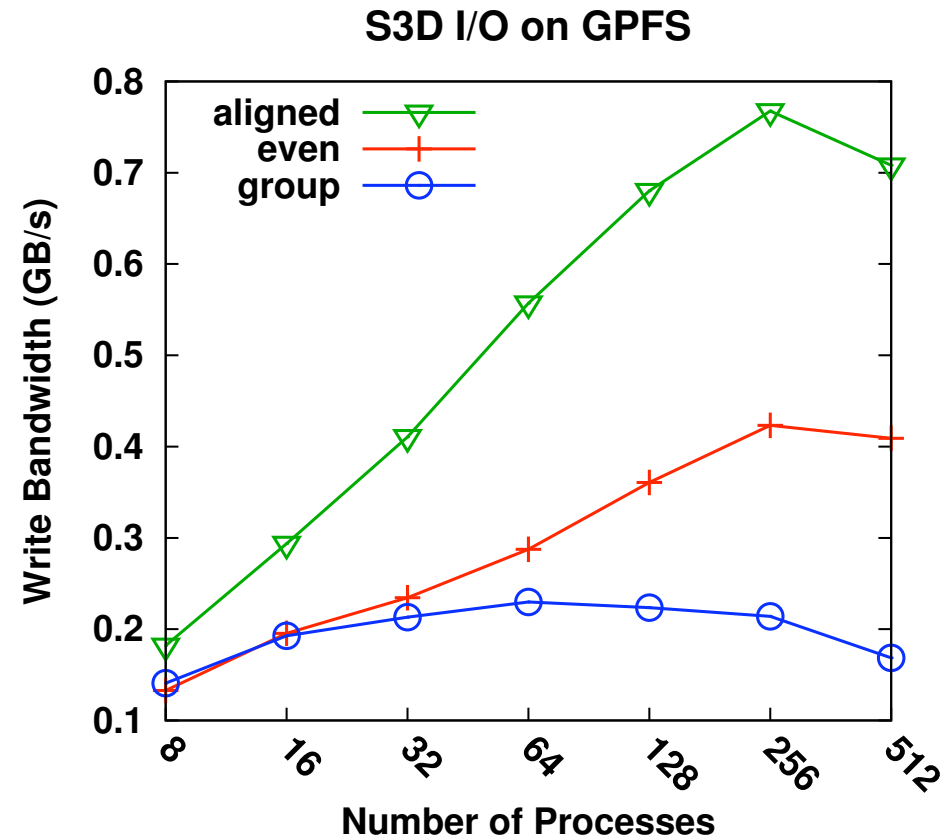- "Group" algorithm similar to server-aligned algorithm on last slide.

- Testing on Mercury, an IBM IA64 system at NCSA, with 54 servers and 512KB stripe size.

S3D I/O on GPFS

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.
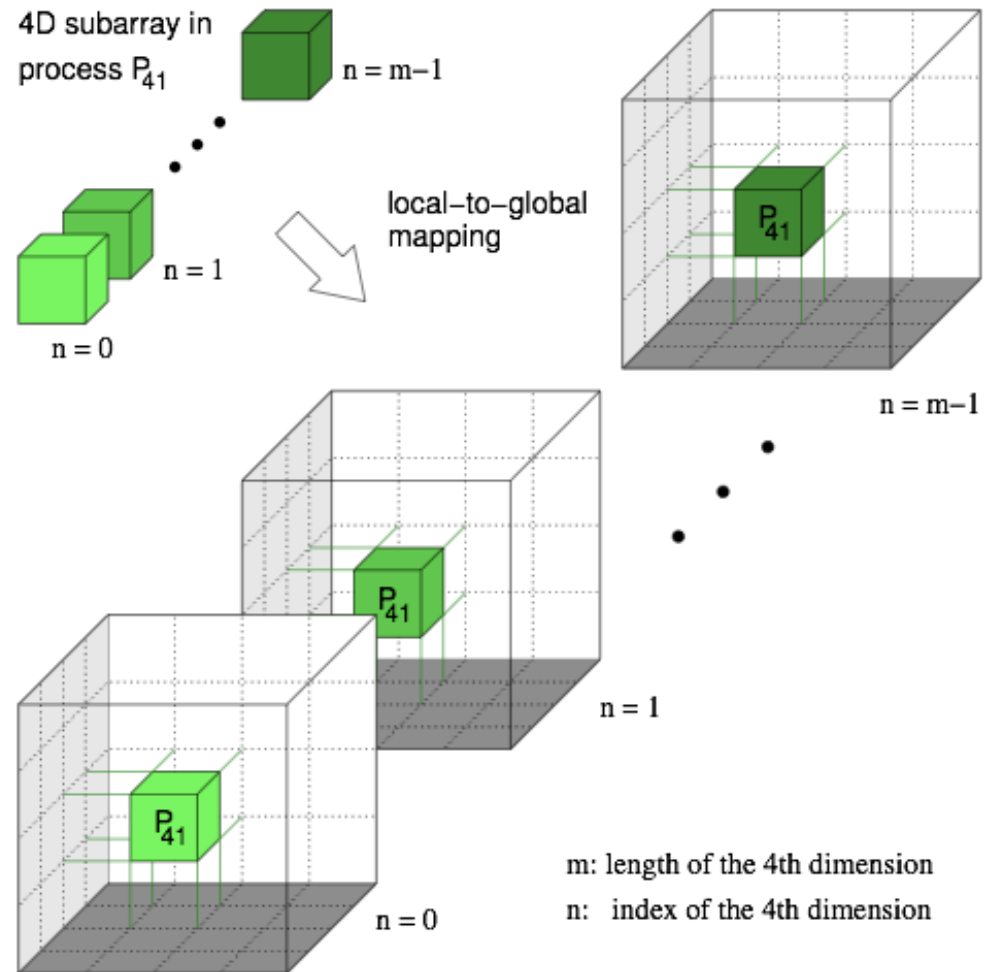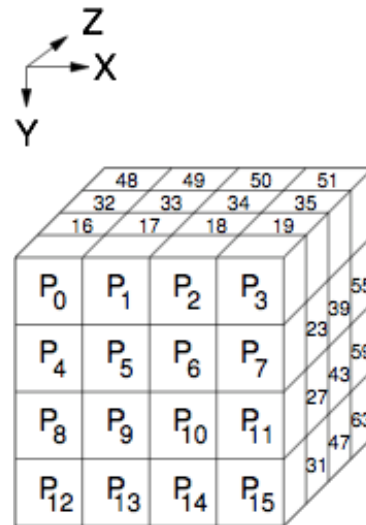
S3D Turbulent Combustion Code

- 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 P_{41}

$n = m-1$

$n = 1$

$n = 0$

local-to-global mapping

P_{41}

$n = m-1$

P_{41}

$n = 1$

P_{41}

$n = 0$

Z
X
Y

48	49	50	51
32	33	34	35
16	17	18	19

P_0	P_1	P_2	P_3	55
P_4	P_5	P_6	P_7	59
P_8	P_9	P_{10}	P_{11}	63
P_{12}	P_{13}	P_{14}	P_{15}	

39 23 43 27 47 31

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.

Impact of Optimizations 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

	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

Summarizing Part 1

- Storage systems combine very large numbers of devices together with software to create a logical unit on which scientific data may be stored.

- Computational science applications exhibit complex access patterns. How we describe those accesses can have a dramatic impact on performance.

- The I/O software stack implements optimizations designed to maximize performance, given enough information to apply the best optimization.

- Tomorrow we will look at high-level I/O libraries and their role in usability of these systems.

Parallel I/O in Practice (Part 2)

Rob Ross
Mathematics and Computer Science Division
Argonne National Laboratory
rross@mcs.anl.gov

Argonne

NATIONAL LABORATORY

... for a brighter future

U.S. Department of Energy

UChicago ▶ Argonne LLC

Office of Science
U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

Recap of Part 1

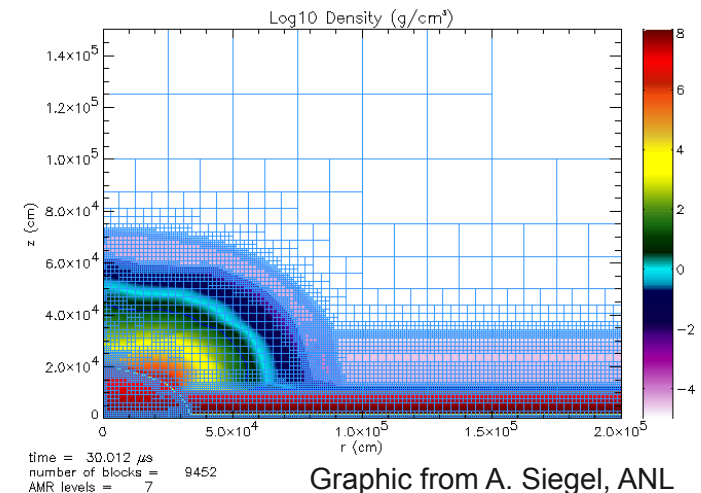- Storage systems combine very large numbers of devices together with software to create a logical unit on which scientific data may be stored.

- Computational science applications exhibit complex access patterns. How we describe those accesses can have a dramatic impact on performance.

- The I/O software stack implements optimizations designed to maximize performance, given enough information to apply the best optimization.

Application and Storage Data Models

- Applications have data models appropriate to domain
  - Multidimensional typed arrays, images composed of scan lines, variable length records
  - 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)
- High-level I/O libraries help map between these data models
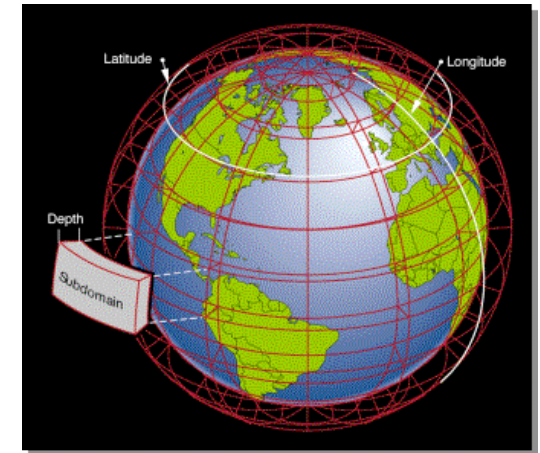
Graphic from J. Tannahill, LLNL

Graphic from A. Siegel, ANL

High-level I/O Interfaces

Application
High-Level I/O Library
I/O Middleware
I/O Forwarding
Parallel File System
I/O Hardware

- **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**
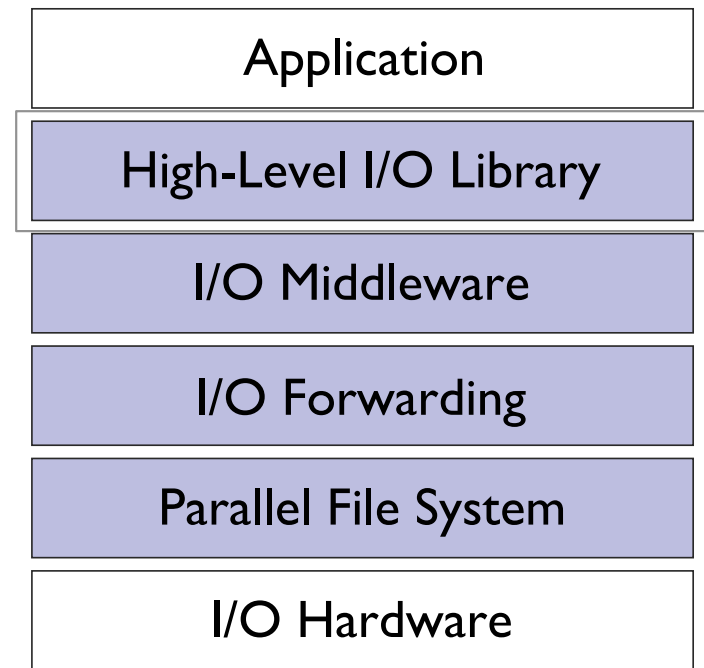
The Parallel netCDF Interface and File Format

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

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 and Fortran 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
- Unrelated to netCDF-4 work

Data Layout in netCDF Files

Application Data Structures

Double temp

1024

26

1024

Float surface_pressure

512

512
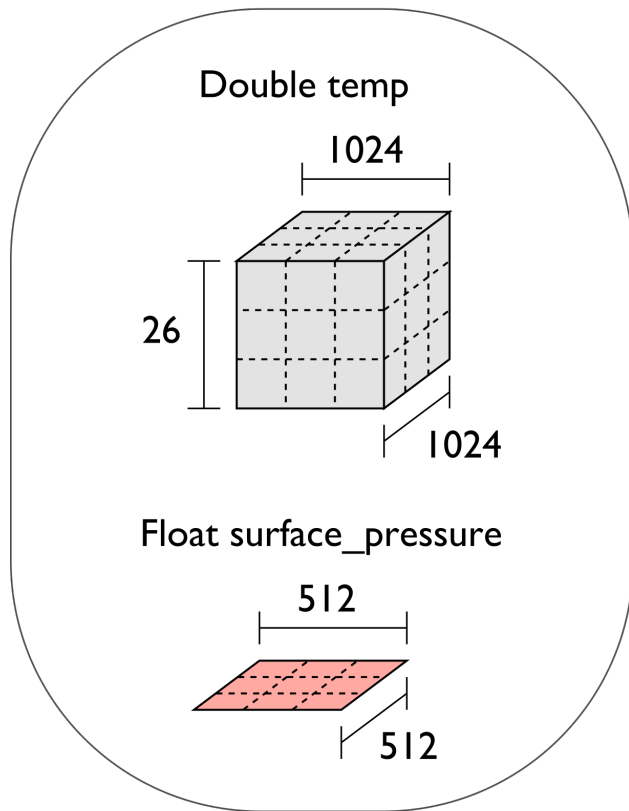
netCDF File "checkpoint07.nc"

Offset in File

```
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"}}
```

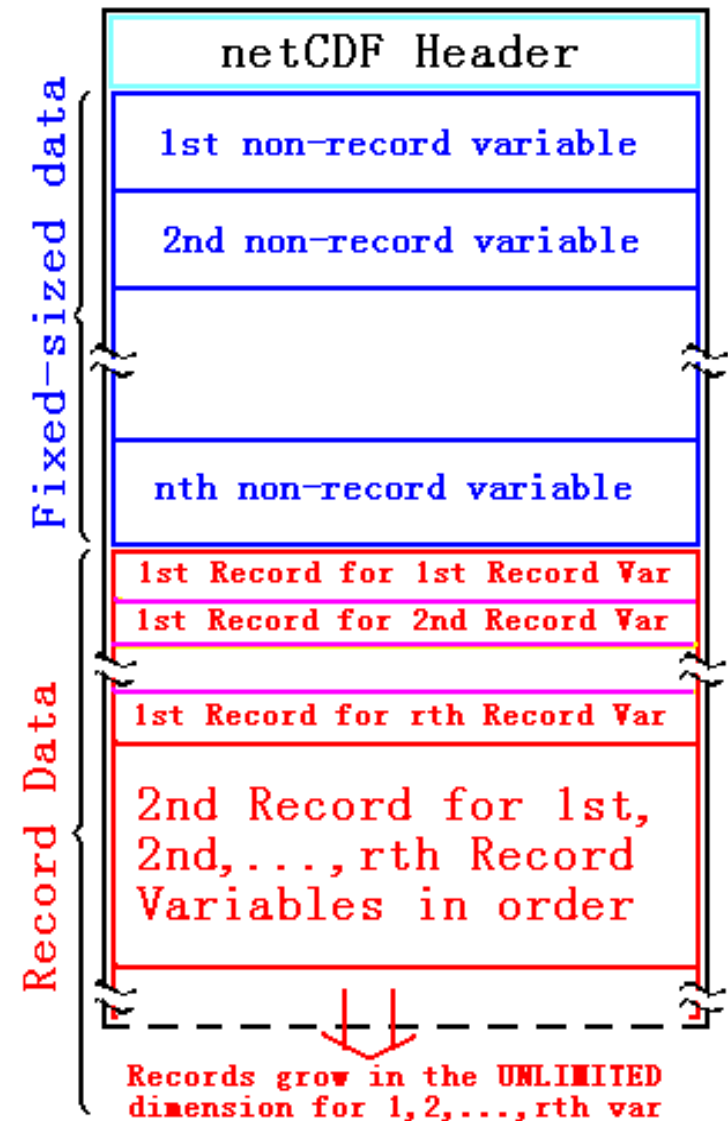< Data for "temp" >

< Data for "surface_pressure" >
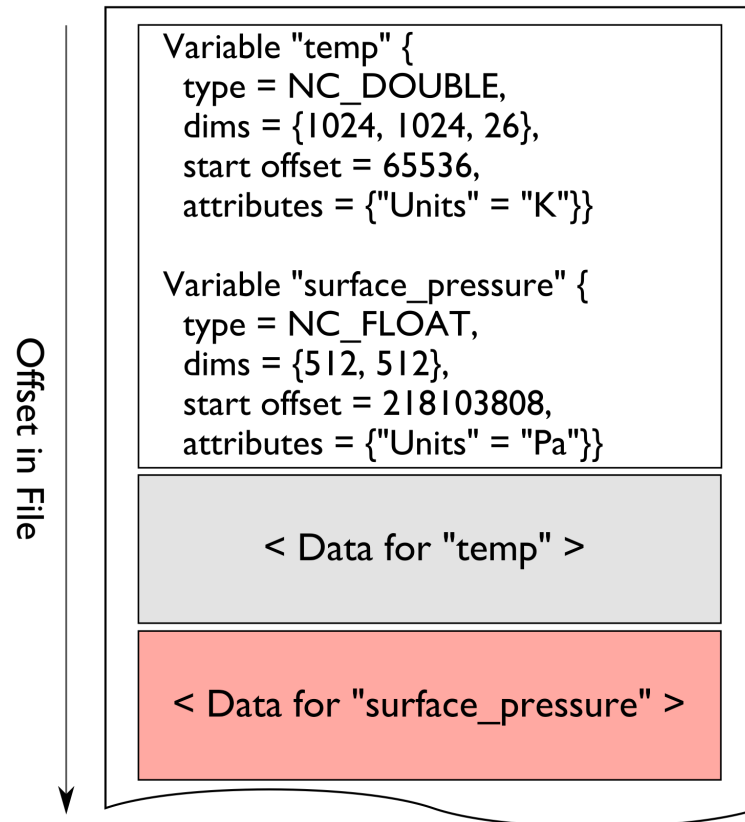
netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

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

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 tens of 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

Vars 0, 1, 2, 3, … 23

- Ghost cell
- Stored element
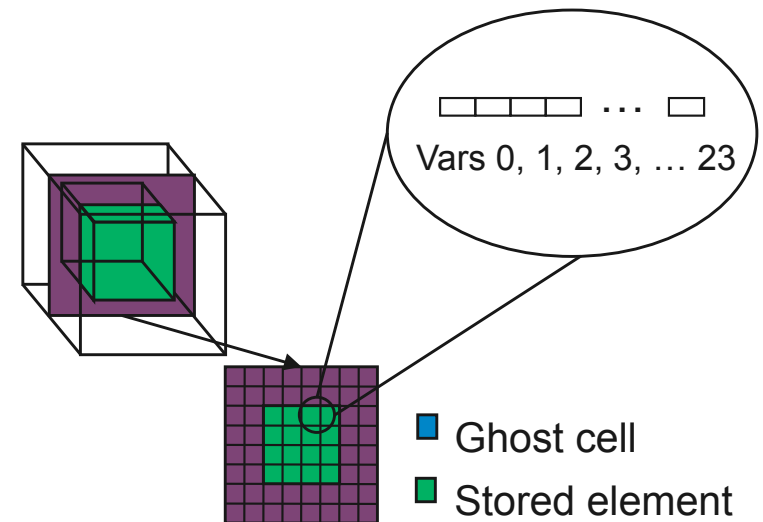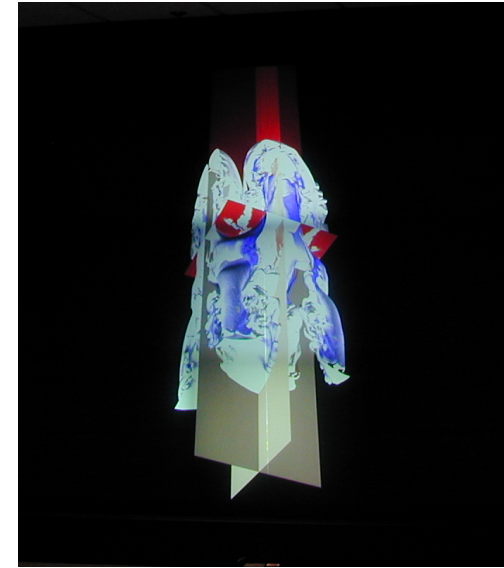
Example: FLASH with PnetCDF

- 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 (X, Y, Z, block)*
  - Store each FLASH variable in its own netCDF variable
    - *Skip ghost cells*
  - Record attributes describing run time, total blocks, etc.

Defining Variable 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
  - 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 file 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
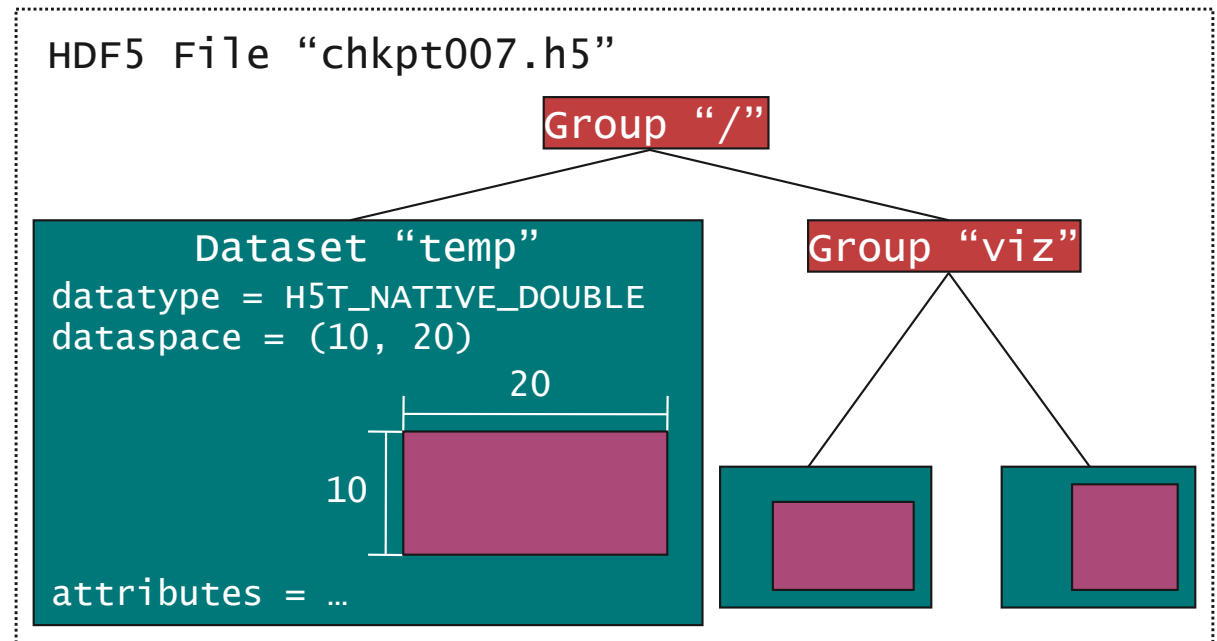
PnetCDF Wrap-Up

- 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 (CDF-2) file format:
  - Fixed-size variable: < 4 GiB
  - Per-record size of record variable: < 4 GiB
  - 2^{32} -1 records
- Work almost complete to relax these limits (CDF-5)

The HDF5 Interface and File Format

HDF5

- Hierarchical Data Format, from the HDF Group (formerly of NCSA)
- Data Model:
  - Hierarchical data organization in single file
  - Typed, multidimensional array storage
  - Attributes on dataset, data
- Features:
  - C, C++, and Fortran interfaces
  - Portable data format
  - Optional compression (not in parallel I/O mode)
  - Data reordering (chunking)
  - Noncontiguous I/O (memory and file) with hyperslabs

HDF5 Files

HDF5 File "chkpt007.h5"

Group "/"

Dataset "temp"
datatype = H5T_NATIVE_DOUBLE
dataspace = (10, 20)

20

10

attributes = …

Group "viz"

- **HDF5 files consist of groups, datasets, and attributes**
  - Groups are like directories, holding other groups and datasets
  - Datasets hold an array of typed data (what we think of as a variable)
    - *A datatype describes the type (not an MPI datatype)*
    - *A dataspace gives the dimensions of the array*
  - Attributes are small datasets associated with the file, a group, or another dataset
    - *Also have a datatype and dataspace*
    - *May only be accessed as a unit*

Example: FLASH Particle I/O with HDF5

- FLASH "Lagrangian particles" record location, characteristics of reaction
  - Passive particles don't exert forces; pushed along but do not interact
- Particle data included in checkpoints, but not in plotfiles; dump particle data to separate file
- One particle dump file per time step
  - i.e., all processes write to single particle file
- Output includes application info, runtime info in addition to particle data

```
Block=30;
Pos_x=0.65;
Pos_y=0.35;
Pos_z=0.125;
Tag=65;
Vel_x=0.0;
Vel_y=0.0;
vel_z=0.0;
```

Typical particle data

Storing Labels for Particles

```
int string_size = OUTPUT_PROP_LENGTH;
hsize_t dims_2d[2] = {npart_props, string_size};
hid_t dataspace, dataset, file_id, string_type;


/* store string creation time attribute */
string_type = H5Tcopy(H5T_C_S1);
H5Tset_size(string_type, string_size);
dataspace = H5Screate_simple(2, dims_2d, NULL);
dataset   = H5Dcreate(file_id, "particle names",
    string_type, dataspace, H5P_DEFAULT);
if (myrank == 0) {
    status = H5Dwrite(dataset, string_type, H5S_ALL,
    H5S_ALL,  H5P_DEFAULT, particle_labels);
}
```

get a copy of the string type and resize it (one way to deal with strings)
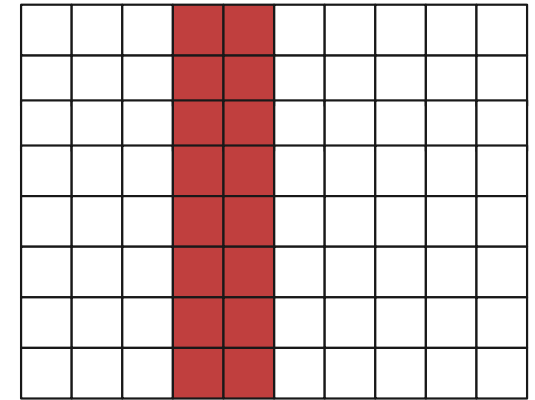
Write out all 8 labels in one call

Storing Particle Data with Hyperslabs (1 of 2)

```
hsize_t dims_2d[2];

/* Step 1: set up dataspace –
        describe global layout */

dims_2d[0] = total_particles;
dims_2d[1] = npart_props;

dspace = H5Screate_simple(2, dims_2d, NULL);
dset = H5Dcreate(file_id, "tracer particles",
    H5T_NATIVE_DOUBLE, dspace, H5P_DEFAULT);
```
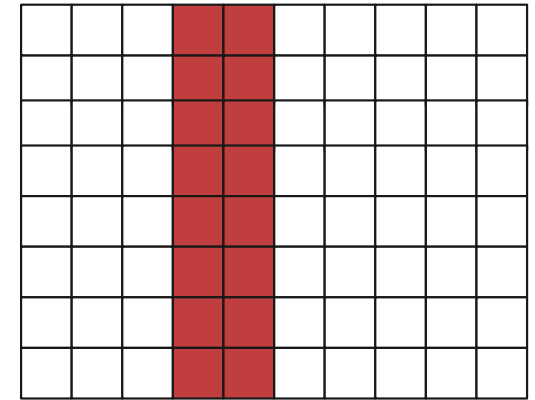
local_np = 2,
part_offset = 3,
total_particles = 10,
Npart_props = 8

Remember:
"S" is for dataspace,
"T" is for datatype,
"D" is for dataset!

Storing Particle Data with Hyperslabs (2 of 2)

```
hsize_t start_2d[2] = {0, 0},
       stride_2d[1] = {1, 1};
hsize_t count_2d[2] = {local_np,
                              npart_props};


/* Step 2: setup hyperslab for
   dataset in file */
```

local_np = 2,
part_offset = 3,
total_particles = 10,
Npart_props = 8

```
start_2d[0]  = part_offset; /* different for each process */
status = H5Sselect_hyperslab(dspace,
                    H5S_SELECT_SET,
                    start_2d, stride_2d, count_2d, NULL);
```

dataspace from
last slide

- Hyperslab selection similar to MPI-IO file view
- Selections don't overlap in this example (would be bad if writing!)
- `H5SSelect_none()` if no work for this process

Collectively Writing Particle Data

```
/* Step 1: specify collective I/O */
dxfer_template = H5Pcreate(H5P_DATASET_XFER);
ierr = H5Pset_dxpl_mpio(dxfer_template,
    H5FD_MPIO_COLLECTIVE);

/* Step 2: perform collective write */
status = H5Dwrite(dataset,
                  H5T_NATIVE_DOUBLE,
                  memspace,
                  dspace,
                  dxfer_template,
                  particles);
```

"P" is for property list; tuning parameters

dataspace describing memory, could also use a hyperslab

dataspace describing region in file, with hyperslab from previous two slides

Remember:
"S" is for dataspace,
"T" is for datatype,
"D" is for dataset!

Inside HDF5

- MPI_File_open used to open file
- Because there is no "define" mode, file layout is determined at write time
- In H5Dwrite:
  - Processes communicate to determine file layout
    - *Process 0 performs metadata updates*
  - Call MPI_File_set_view
  - Call MPI_File_write_all to collectively write
- Memory hyperslab could be used to define noncontig. region in memory
- In FLASH application, data is kept in native format and converted at read time (defers overhead)
  - Could store in some other format if desired
- At the MPI-IO layer:
  - Metadata updates at every write are a bit of a bottleneck
    - *MPI-IO from process 0 introduces some skew*

Additional I/O Libraries

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)
    - Multiple unlimited dimensions
    - Higher limits for file and variable sizes
    - Backwards compatibility with "classic" datasets
    - Groups
    - Compound types
    - Variable length arrays
    - Data chunking and compression (parallel reads only – serial writes)

Comparing PnetCDF and netCDF-4

- netCDF-4: parallel access through new function calls (_par)
  - Open, create take MPI hints (like PnetCDF)
  - Collective I/O by default (like PnetCDF)
  - Same routine can be either independent or collective depending on mode (like HDF5)
  - HDF5 tools understand netCDF-4 datasets

Parallel netCDF	netCDF-4
ncmpi_open	nc_open_par
ncmpi_create	nc_create_par
ncmpi_enddef	nc_enddef
ncmpi_def_dim	nc_def_dim
ncmpi_put_vara_float_all	nc_put_vara_float
ncmpi_begin_indep_data	nc_var_par_access

ADaptable IO System (ADIOS)

The goal of ADIOS is to create an easy and efficient I/O interface that hides the details of I/O from computational science applications:

- Operate across multiple HPC architectures and parallel file systems
  - Blue Gene, Cray, IB-based clusters
  - Lustre, PVFS2, GPFS, Panasas, PNFS
- Support many underlying file formats and interfaces
  - MPI-IO, POSIX, HDF5, netCDF
  - Facilitates switching underlying file formats to reach performance goals
- Cater to common I/O patterns
  - Restarts, analysis, diagnostics
  - Different combinations provide different levels of IO performance
- Compensate for inefficiencies in the current I/O infrastructures

ADIOS Binary Packed (BP) File Format

Defers translation into portable format to attain high performance at runtime.

Accelerates writing from large numbers of processes through a log-like storage format:

- **Each process writes independently**
- **Coordinate only twice**
  - Once at start to determine writing locations
  - Once at end for metadata collection
- **Move the "header" to the end to aid in alignment**
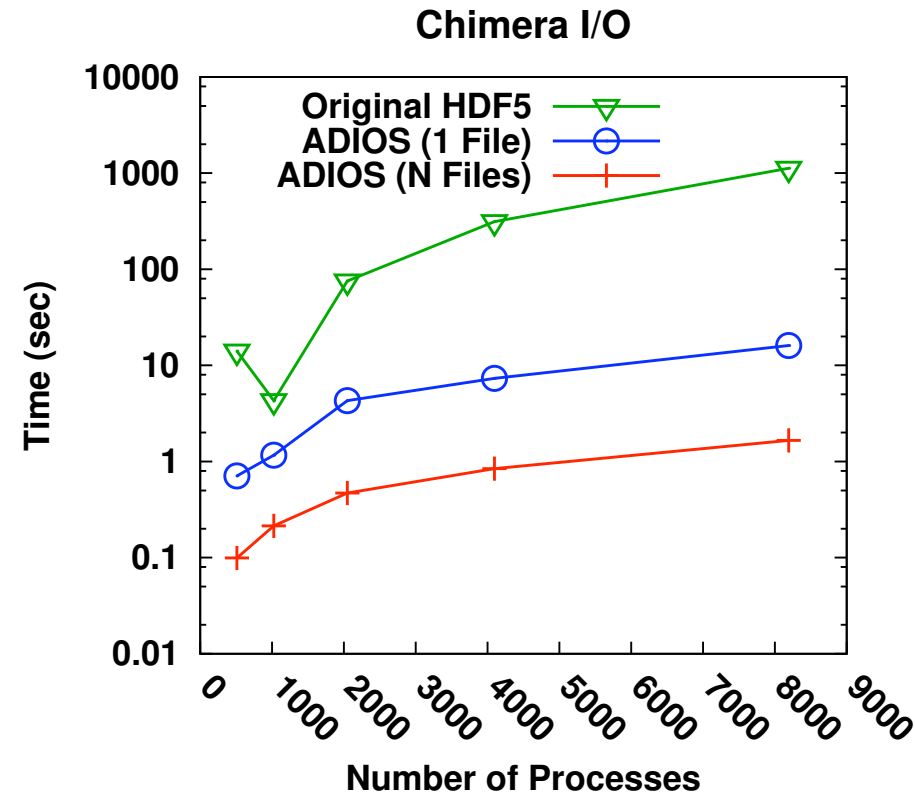
Chimera I/O

I/O times for Chimera astrophysics application on Cray XT at ORNL. "I File" results may benefit from Lustre optimizations that were not in place at time of testing.

Tuning Application I/O

Tuning Application I/O

- Find out what you have to work with
  - What is peak I/O rate?
  - What other testing has been done?
- Describe as much as possible to the I/O system
  - Use collective calls when available
  - Describe data movement with fewest possible operations
  - Open with appropriate mode
- Match file organization to process partitioning if possible
  - Order dimensions so relatively large blocks are contiguous with respect to data decomposition
- Know what you can control …

Controlling I/O Stack Behavior: Hints

- Most systems accept hints through one mechanism or another
  - Parameters to file "open" calls
  - Proprietary POSIX `ioctl` calls
  - MPI_Info
  - HDF5 property lists
- Allow the programmer to:
  - Explain more about the I/O pattern
  - **Control particular optimizations**
  - **Impose resource limitations**
- Generally pass information that is used only during a particular set of accesses (between open and close, for example)

MPI-IO Hints

- MPI-IO hints may be passed via:
  - `MPI_File_open`
  - `MPI_File_set_info`
  - `MPI_File_set_view`
- Hints are optional - implementations are guaranteed to ignore ones they do not understand
  - Different implementations, even different underlying file systems, support different hints
- `MPI_File_get_info` used to get list of hints

MPI-IO Hints: Data Sieving

- `ind_rd_buffer_size` - Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving reads

- `ind_wr_buffer_size` - Controls the size (in bytes) of the intermediate buffer used by ROMIO when performing data sieving writes

- `romio_ds_read` - Determines when ROMIO will choose to perform data sieving for reads (enable, disable, auto)

- `romio_ds_write` - Determines when ROMIO will choose to perform data sieving for writes

MPI-IO Hints: Collective I/O

- `cb_buffer_size` - Controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O
- `romio_cb_read` - Controls when collective buffering is applied to collective read operations
- `romio_cb_write` - Controls when collective buffering is applied to collective write operations
- `cb_nodes` - Controls the maximum number of aggregators to be used
- `cb_config_list` - Provides explicit control over aggregators (see ROMIO User's Guide)

MPI-IO Hints: File System Specific

- `striping_factor` - Controls the number of I/O devices to stripe across
- `striping_unit` - Controls the amount of data placed on one device before moving to next device (in bytes)
- `start_iodevice` - Determines what I/O device data will first be written to
- `direct_read` - Controls direct I/O for reads
- `direct_write` - Controls direct I/O for writes

Using MPI_Info

- Example: setting data sieving buffer to be a whole "frame" of a tiled display movie

```
char info_value[16];
MPI_Info info;
MPI_File fh;
MPI_Info_create(&info);
snprintf(info_value, 15, "%d", 3*1024 * 2*768 * 3);
MPI_Info_set(info, "ind_rd_buffer_size",
    info_value);
MPI_File_open(comm, filename, MPI_MODE_RDONLY,
    info, &fh);
MPI_Info_free(&info);
```

Hints and PnetCDF

- Uses `MPI_Info`, so almost identical
- For example, turning off two-phase writes, in case you're doing large contiguous collective I/O on Lustre:

```
MPI_Info info;
MPI_File fh;
MPI_Info_create(&info);
MPI_Info_set(info, "romio_cb_write", "disable");
ncmpi_open(comm, filename, NC_NOWRITE, info,
    &ncfile);
MPI_Info_free(&info);
```

Hints and HDF5

- HDF5 uses property lists and MPI_Info structures for passing hints

```
/* HDF5 data sieving buffer size using file access property list */
acc_template = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_sieve_buf_size(acc_template, 524288);

/* align objects larger than 512K on 256K boundaries */
H5Pset_alignment(acc_template, 524288, 262144);

/* pass in an MPI hint using file access property list */
MPI_Info_set(info, "access_style", "write_once");
H5Pset_fapl_mpio(acc_template, MPI_COMM_WORLD, info);

/* specify collective I/O using data xfer property list */
xfer_template = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(xfer_template, H5FD_MPIO_COLLECTIVE);
```

Characterizing Application I/O

How is this application performing I/O?
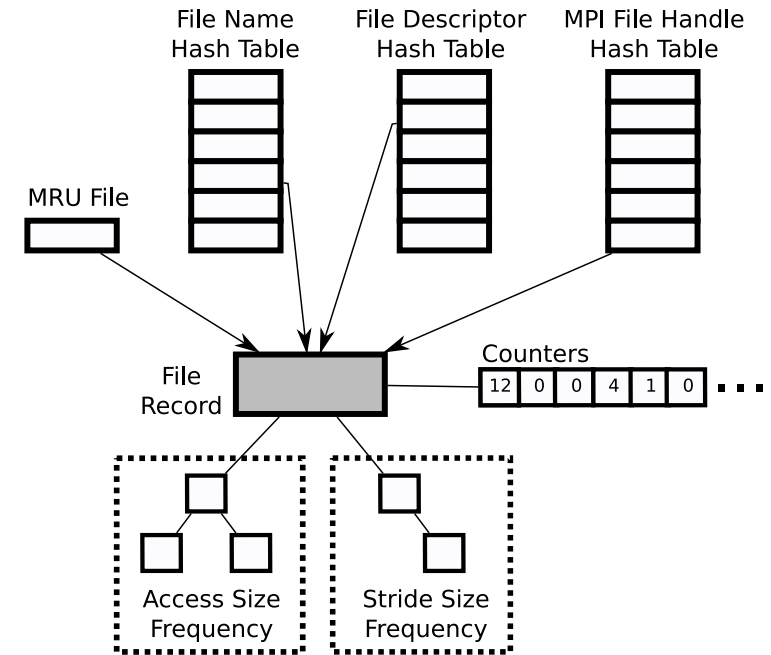## How successful is its approach?

Darshan (Sanskrit for "sight") is a tool we are developing for I/O characterization at extreme scale:

- No code changes (link time hooks into MPI-IO and POSIX calls)
- Small, tunable runtime memory footprint (~2MB default, 1024 distinct files)
- Relates accesses from multiple processes and to the same files
- Captures:
  - Counters for POSIX and MPI-IO operations
  - Counters for unaligned, sequential, consecutive, and strided access
  - Timing of opens, closes, first and last reads and writes
  - Histograms of access, stride, datatype, and extent size

P. Carns et. al, "24/7 Characterization of Petascale I/O Workloads", to appear in the Workshop on Interfaces and Abstractions for Scientific Data Storage, September, 2009.

Darshan Internals

- Characterization centers around per-file records
  - Falls back to aggregate (across files) mode if file limit is exceeded
- At output time, processes further reduce output size
  - Communicate to combine data on identical files accessed by all processes
  - Independently compress (gzip) remaining data
    - *32K processes writing a shared file leads to 203 bytes of compressed output*
   - *32K processes writing a total of 262,144 files leads to 13.3MB of output*

File Name Hash Table   File Descriptor Hash Table   MPI File Handle Hash Table

MRU File

File Record

Counters

| 12 | 0 | 0 | 4 | 1 | 0 | ··· |

Access Size Frequency   Stride Size Frequency

Multiple tables allow efficient location of file records by name, file descriptor, or MPI File handle.

Results from Chombo I/O Benchmark

■ Simulates I/O from a block-structured AMR code, writes a single HDF5 file

Chombo write size histogram, 512 procs

Chombo cumulative time per process
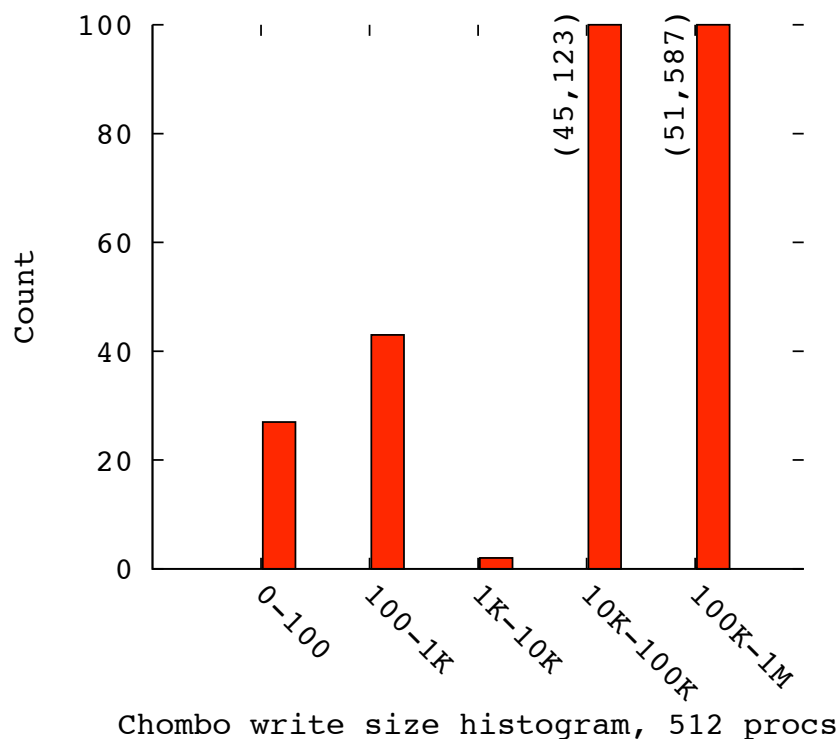
Benchmark results in nearly 100K individual I/O operations, none of which are larger than 1MByte in size. Darshan also tells us that 99.99% of these are unaligned in file.
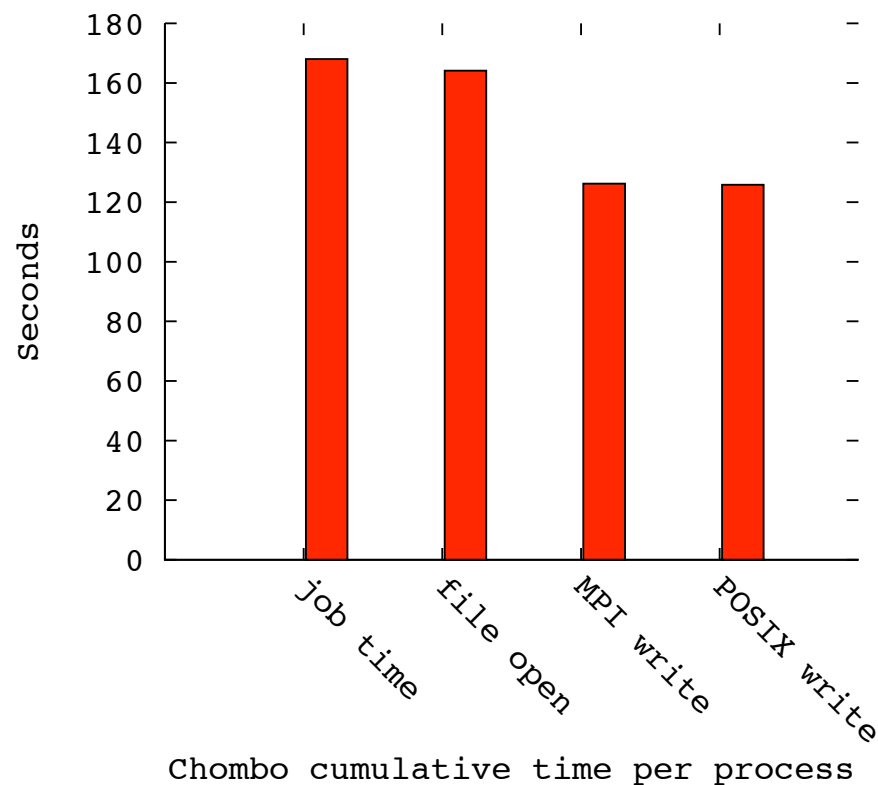
MPI and POSIX file write times are identical, indicating MPI-IO implementation isn't performing any optimization.

Conclusions

- Storage is a complex hardware/software system
- Lots of software is available to support computational science workloads at scale
  - Parallel file systems
  - I/O libraries
  - I/O tracing and characterization tools
- Using this software (correctly) can dramatically improve performance (execution time) and productivity (development time)
- Computer scientists are interested in helping with I/O challenges
  - SciDAC SDM center is one good point of contact

Printed References

- John May, <u>Parallel I/O for High Performance Computing</u>, Morgan Kaufmann, October 9, 2000.
  - Good coverage of basic concepts, some MPI-IO, HDF5, and serial netCDF
  - Out of print…
- William Gropp, Ewing Lusk, and Rajeev Thakur, <u>Using MPI-2: Advanced Features of the Message Passing Interface</u>, MIT Press, November 26, 1999.
  - In-depth coverage of MPI-IO API, including a very detailed description of the MPI-IO consistency semantics

On-Line References (1 of 4)

- netCDF and netCDF-4
  - http://www.unidata.ucar.edu/packages/netcdf/
  - http://www.unidata.ucar.edu/software/netcdf/netcdf-4
- PnetCDF
  - http://www.mcs.anl.gov/parallel-netcdf/
- ROMIO MPI-IO
  - http://www.mcs.anl.gov/romio/
- HDF5 and HDF5 Tutorial
  - http://www.hdfgroup.org/
  - http://hdf.ncsa.uiuc.edu/HDF5/
  - http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor/index.html
- POSIX I/O Extensions
  - http://www.opengroup.org/platform/hecewg/

On-Line References (2 of 4)

- PVFS

   http://www.pvfs.org/

- Panasas

   http://www.panasas.com/

- Lustre

   http://www.lustre.org/

- GPFS

   http://www.almaden.ibm.com/storagesystems/file_systems/GPFS/

On-Line References (3 of 4)

- LLNL I/O tests (IOR, fdtree, mdtest)
  - http://sourceforge.net/projects/ior-sio
- Parallel I/O Benchmarking Consortium (noncontig, mpi-tile-io, mpi-md-test)
  - http://www.mcs.anl.gov/pio-benchmark/
- FLASH I/O benchmark
  - http://www.mcs.anl.gov/pio-benchmark/
  - http://flash.uchicago.edu/~jbgallag/io_bench/ (original version)
- b_eff_io test
  - http://www.hlrs.de/organization/par/services/models/mpi/b_eff_io/
- mpiBLAST
  - http://www.mpiblast.org

On Line References (4 of 4)

- **NFS Version 4.1**
  - draft-ietf-nfsv4-minorversion1-26.txt
  - draft-ietf-nfsv4-pnfs-obj-09.txt
  - draft-ietf-nfsv4-pnfs-block-09.txt
- **pNFS Problem Statement**
  - Garth Gibson (Panasas), Peter Corbett (Netapp), Internet-draft, July 2004
  - http://www.pdl.cmu.edu/pNFS/archive/gibson-pnfs-problem-statement.html
- **Linux pNFS Kernel Development**
  - http://www.citi.umich.edu/projects/asci/pnfs/linux

Acknowledgments

Argonne National Laboratory