Parallel I/O in Practice

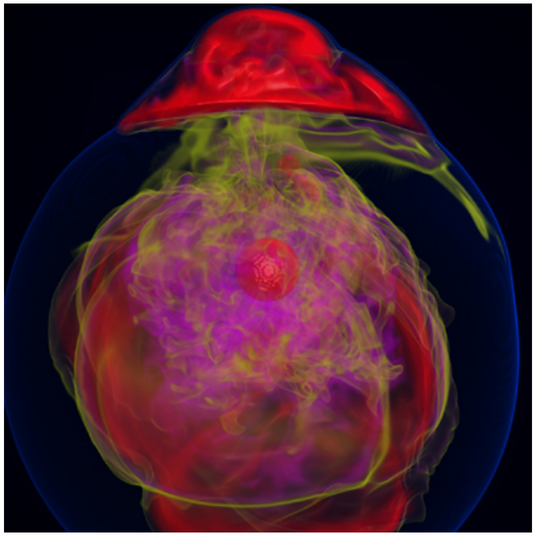
Tom Peterka
Mathematics and Computer Science Division
Argonne National Laboratory

July 26, 2012

Computational Science

- 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
  - Data analysis and visualization are also increasingly bound by data access (both read and write)

IBM Blue Gene/P system at Argonne National Laboratory.



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

Large-Scale Data Sets

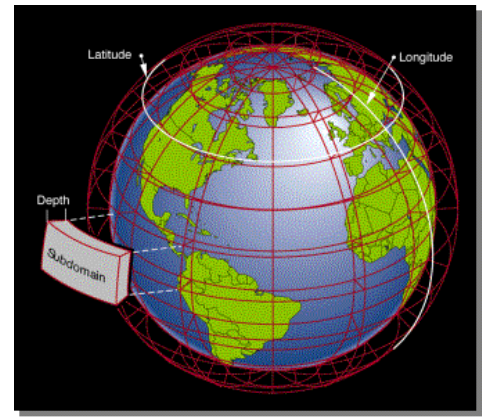
Application teams are beginning to generate 10s of Tbytes of data in a single simulation. Keeping 100s of TBs online is common.

Data requirements for select 2011 INCITE applications at ALCF

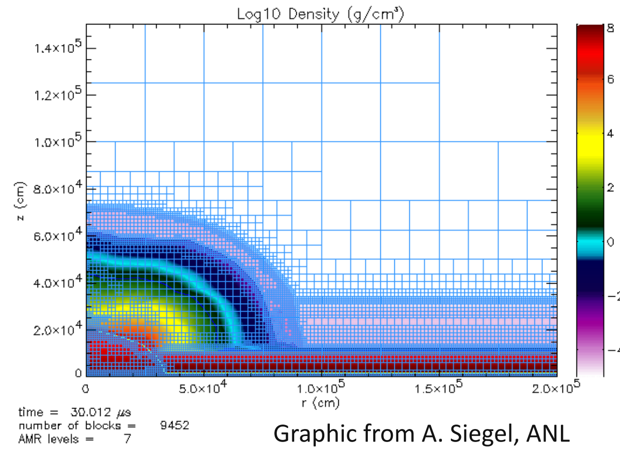
PI	Projec	On-line Data (TBytes)	Off-line Data (TBytes)
Khokhlov	Combustion in Gaseous Mixtures	1	17
Baker	Protein Structure	1	2
Hinkel	Laser-Plasma Interactions	60	60
Lamb	Type Ia Supernovae	75	300
Vary	Nuclear Structure and Reactions	6	15
Fischer	Fast Neutron Reactors	100	100
Mackenzie	Lattice Quantum Chromodynamics	300	70
Vashishta	Fracture Behavior in Materials	12	72
Moser	Engineering Design of Fluid Systems	3	200
Lele	Multi-material Mixing	215	100
Kurien	Turbulent Flows	10	20
Jordan	Earthquake Wave Propagation	1000	1000
Tang	Fustion Reactor Design	50	100

Applications, Data Models, and I/O

- 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)
- Someone has to map from one to the other!



Graphic from J. Tannahill, LLNL



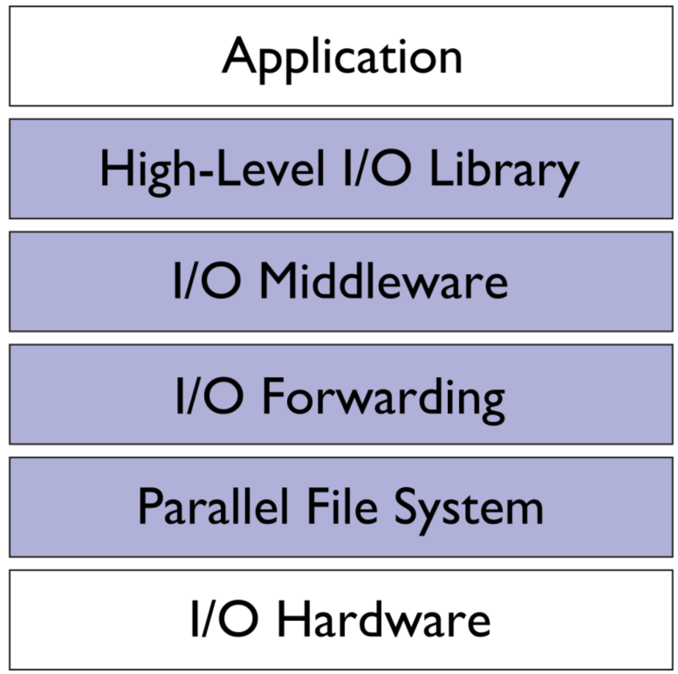
Graphic from A. Siegel, ANL

Challenges in Application I/O

- Leveraging aggregate communication and I/O bandwidth of clients
  - …but not overwhelming a resource limited I/O system with uncoordinated accesses!
- Limiting number of files that must be managed
  - Also a performance issue
- Avoiding unnecessary post-processing
- Often application teams spend so much time on this that they never get any further:
  - Interacting with storage through convenient abstractions
  - Storing in portable formats

Parallel I/O software is available to address all of these problems, when used appropriately.

I/O for Computational Science



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

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.

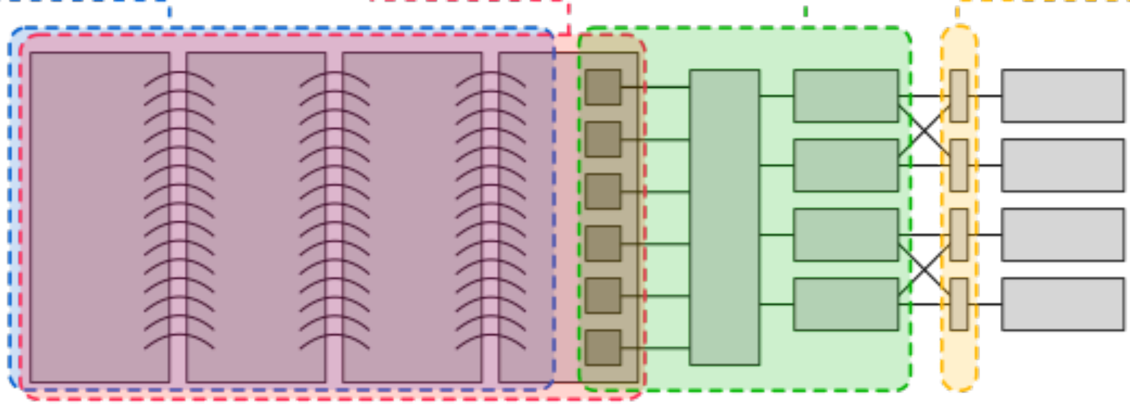
I/O Hardware and Software on Blue Gene/P

High-level I/O libraries execute on compute nodes, mapping application abstractions into flat files, and encoding data in portable formats.
I/O middleware manages collective access to storage.

I/O forwarding software runs on compute and gateway nodes, bridges networks, and provides aggregation of independent I/O.

Parallel file system code runs on gateway and storage nodes, maintains logical storage space and enables efficient access to data.

Drive management software or firmware executes on storage controllers, organizes individual drives, detects drive failures, and reconstructs lost data.



Compute nodes
40,960 Quad core PowerPC 450 nodes with 2 Gbytes of RAM each

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

Commodity network
900+ port 10 Gigabit Ethernet Myricom switch complex

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

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

Architectural diagram of the 557 TFlop IBM Blue Gene/P system at the Argonne Leadership Computing Facility.

What we've said so far…

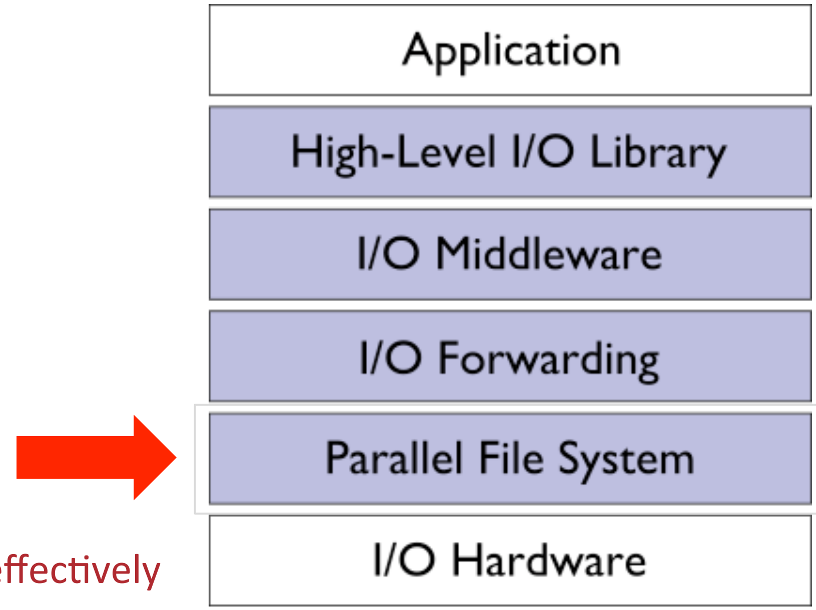
- Application scientists have basic goals for interacting with storage
  - Keep productivity high (meaningful interfaces)
  - Keep efficiency high (extracting high performance from hardware)
- Many solutions have been pursued by application teams, with limited success
  - This is largely due to reliance on file system APIs, which are poorly designed for computational science
- Parallel I/O teams have developed software to address these goals
  - Provide meaningful interfaces with common abstractions
  - Interact with the file system in the most efficient way possible

Parallel File Systems

Thanks to Rob Ross (ANL)

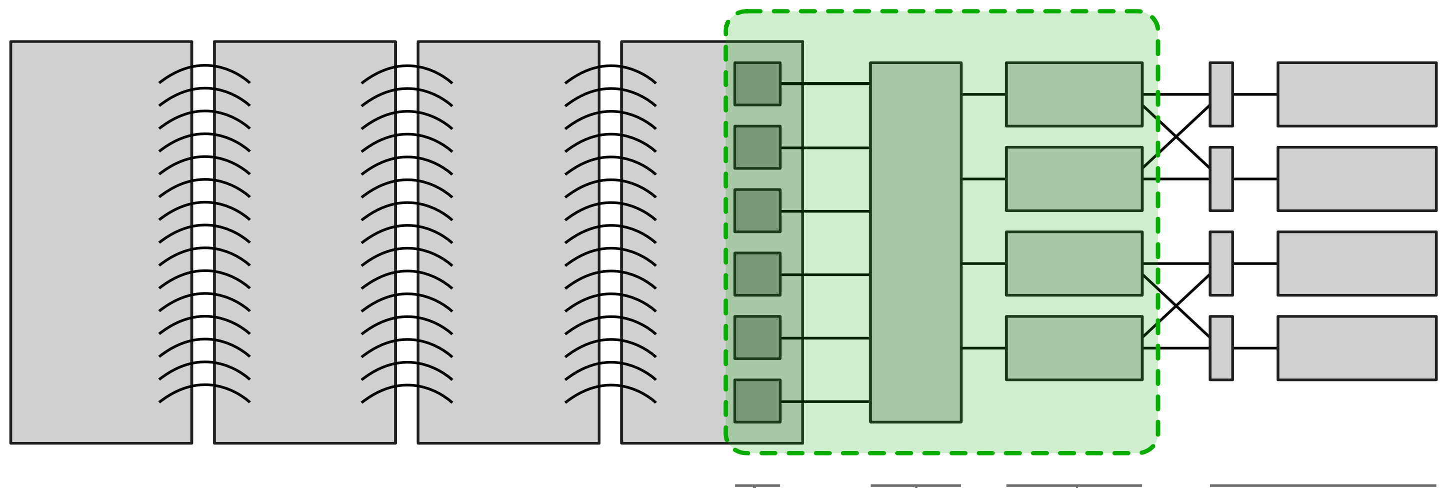
Parallel File System

- Manage storage hardware
  - Present single view
  - Stripe files for performance
- In the I/O software stack
  - Focus on concurrent, independent access
  - Publish an interface that middleware can use effectively
    - Rich I/O language
    - Relaxed but sufficient semantics



Parallel File System Software

PVFS code runs on gateway and storage nodes, maintains logical storage space, and enables efficient access to data.



Gateway nodes

run parallel file system client software

Commodity network

primarily carries storage traffic

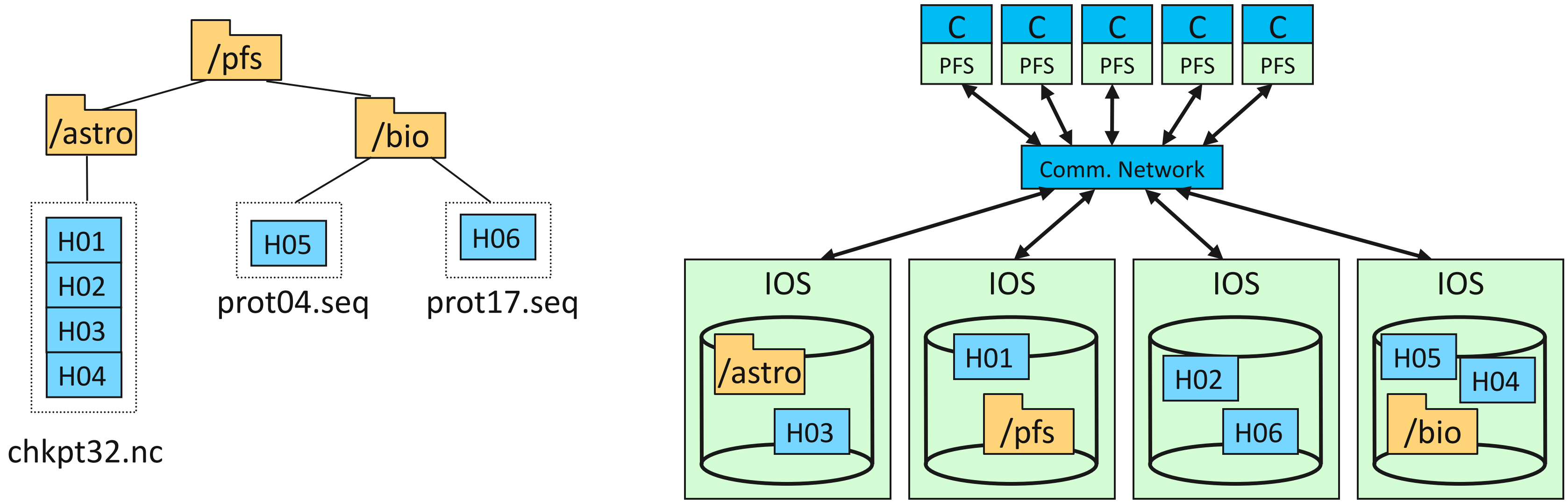
Storage nodes

run parallel file system server software and manage incoming FS traffic

Enterprise storage

accept block device requests from file server and manage logical units (LUNs)

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.

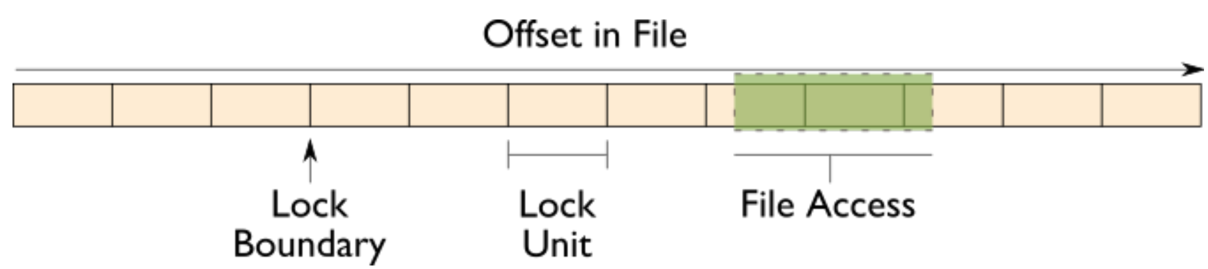
- Building block for HPC I/O systems
  - Present storage as a single, logical storage unit
  - Stripe files across disks and nodes for performance
  - Tolerate failures (in conjunction with other HW/SW)
- User interface is often POSIX file I/O interface, not very good for HPC

Locking in Parallel File Systems

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

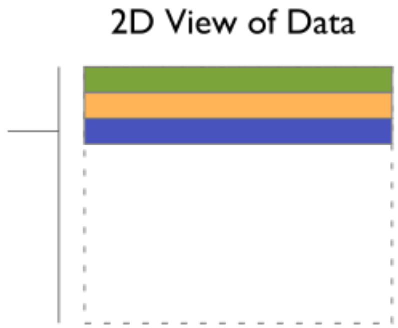
- Files are broken up into lock units
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

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



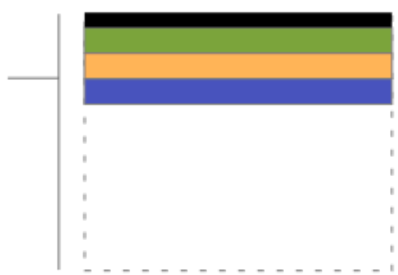
Locking and Concurrent Access

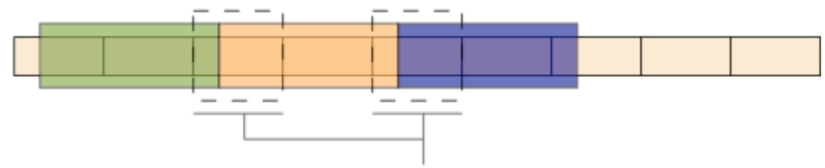
The left diagram shows a row-block distribution of data for three processes. On the right we see how these accesses map onto locking units in the file.



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

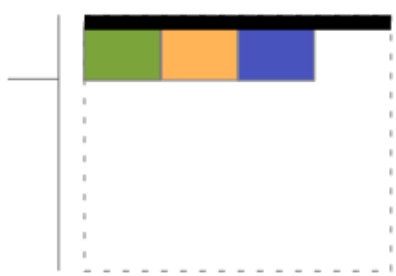
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

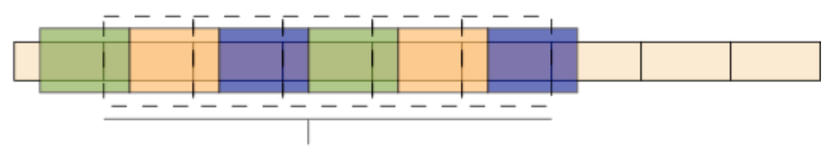




These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

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.

Parallel File Systems Recap

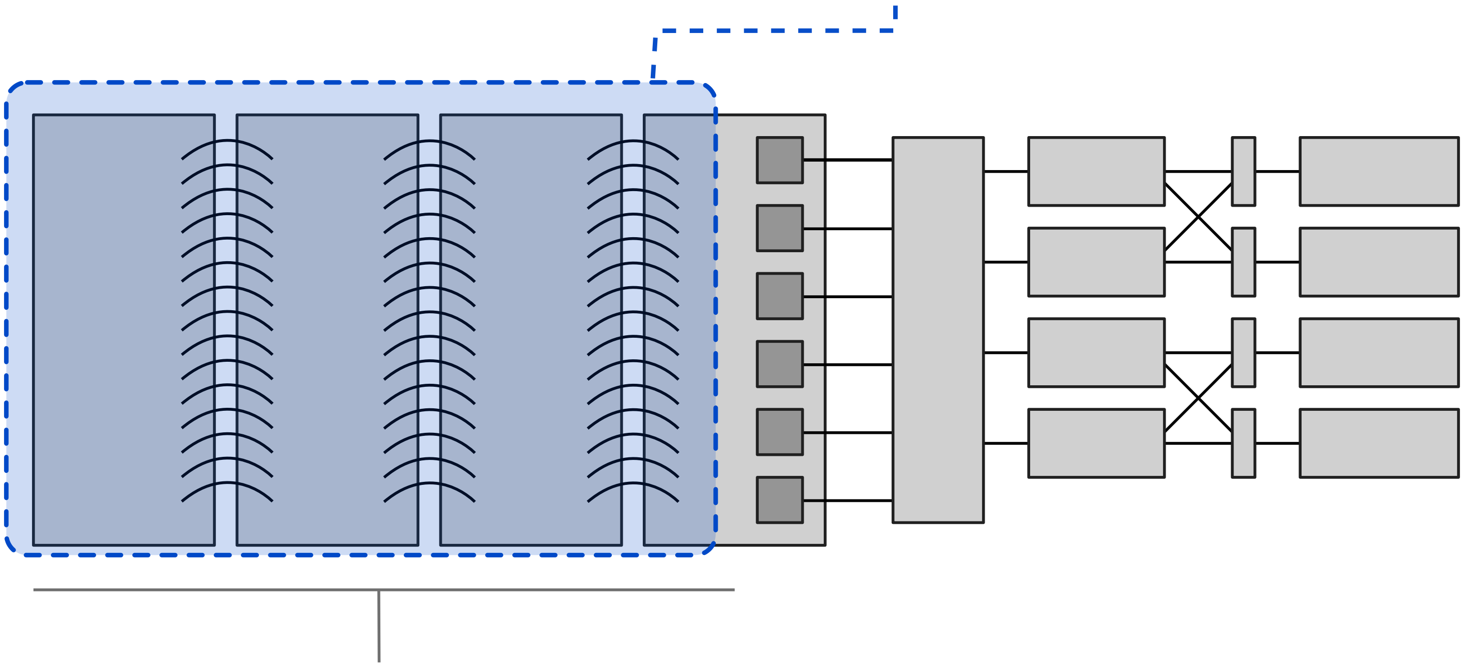
- Manage storage hardware for programmer productivity and performance
- Expose API to next higher level in software stack
- Make striped files look like one file to the programmer
- Manage metadata (directories, file names, stripe locations)
- Manage concurrent access (usually locks)

The MPI-IO Interface

Thanks to Rob Latham (ANL)

High-level Libraries and MPI-IO Software

High-level I/O libraries and **MPI-IO** execute on compute nodes and organize accesses before the I/O system sees them.

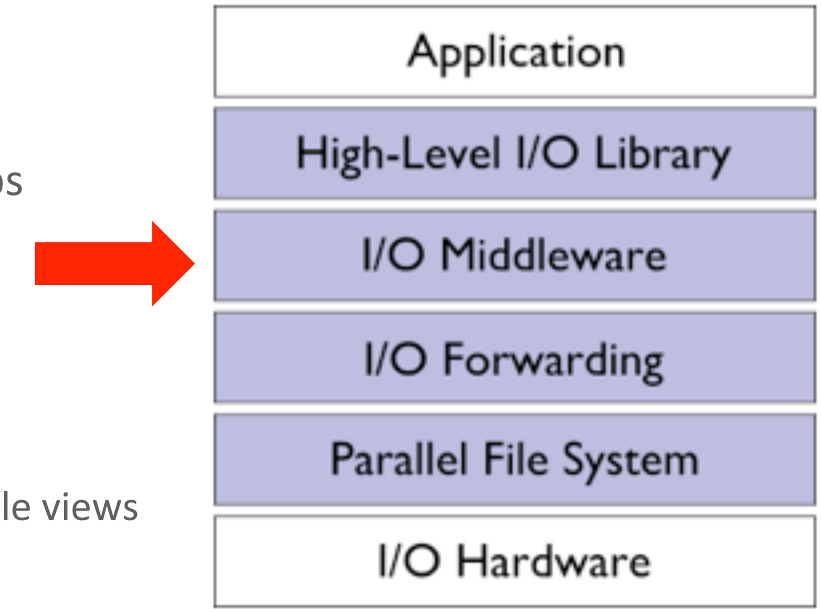


Compute nodes

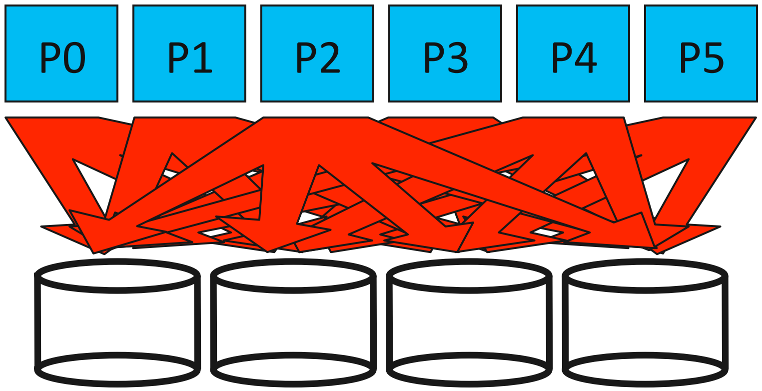
run application codes with high-level I/O libraries and MPI-IO. I/O libraries make I/O calls to I/O forwarding system

MPI-IO

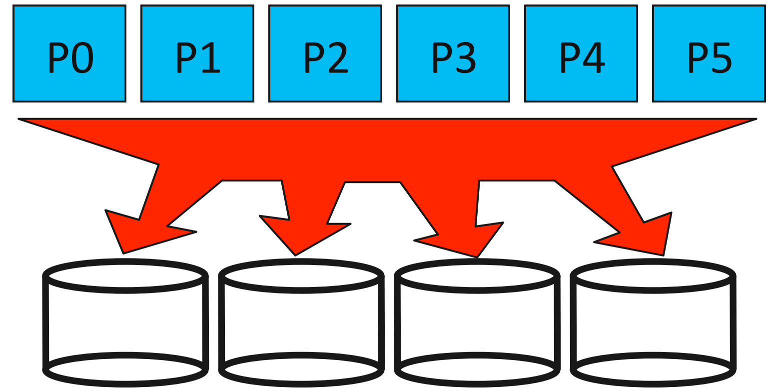
- I/O interface specification for use in MPI apps
- Data model is same as POSIX
  - Stream of bytes in a file
- Features:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
  - System for encoding files in a portable format (external32)
    - Not self-describing - just a well-defined encoding of types
- Implementations available on most platforms



Independent and Collective I/O



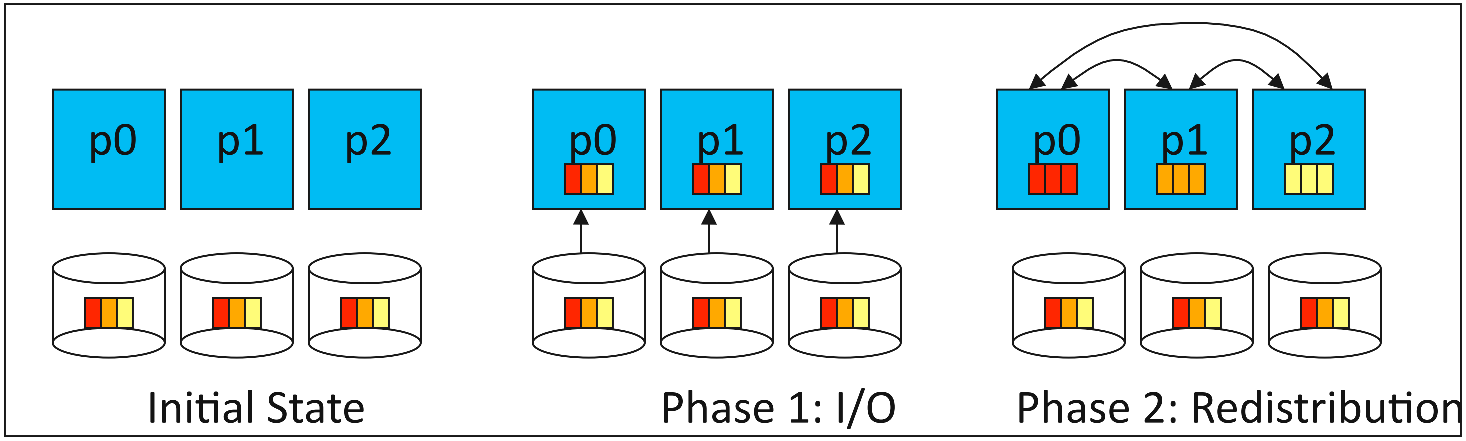
Independent I/O



Collective I/O

- Independent I/O operations specify only what a single process will do
  - Independent I/O calls do not pass on relationships between I/O on other processes
- 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

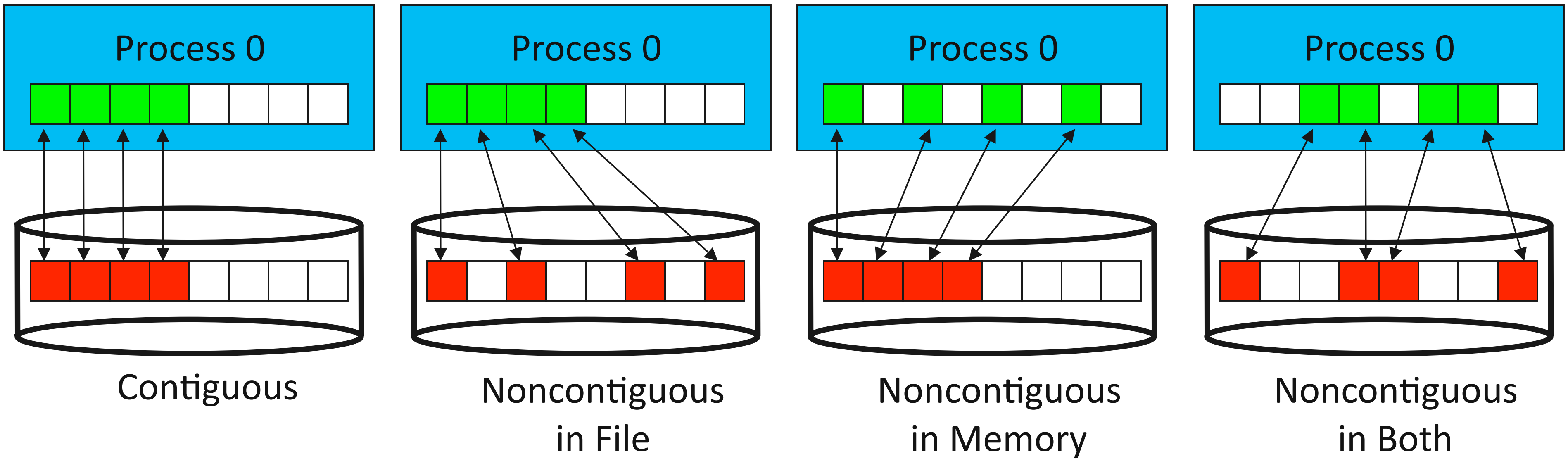
Collective I/O and Two-Phase I/O



Two-Phase Read Algorithm

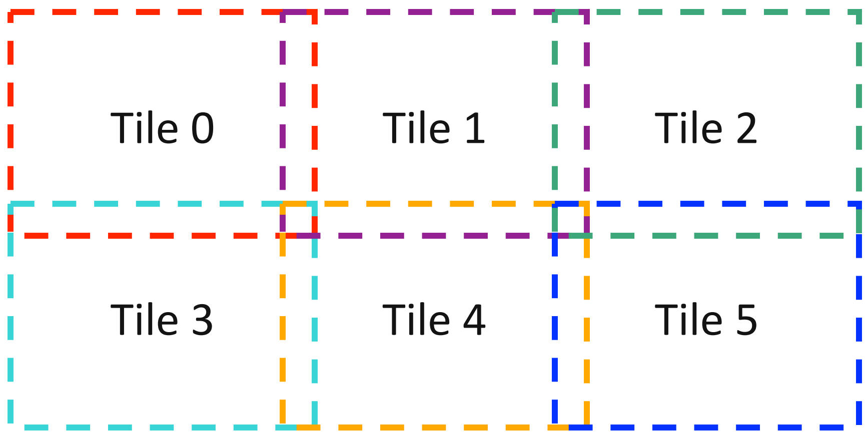
- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data, can exhibit false sharing
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through sharing of common blocks
- Second "phase" redistributes data to final destinations
- Two-phase writes operate in reverse (redistribute then I/O)
  - Typically read/modify/write (like data sieving)
  - Overhead is lower than independent access because there is little or no false sharing
- Note that two-phase is usually applied to file regions, not to actual blocks

Contiguous and Noncontiguous I/O



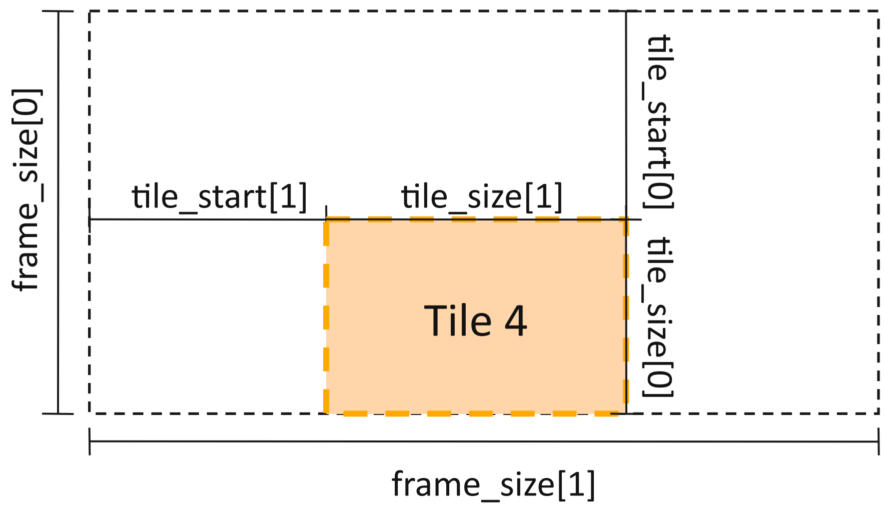
- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous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

Example: Visualization Staging



- Often large frames must be preprocessed before display on a tiled display
- First step in process is extracting "tiles" that will go to each projector
  - Perform scaling, etc.
- Parallel I/O can be used to speed up reading of tiles
  - One process reads each tile
- We're assuming a raw RGB format with a fixed-length header

MPI 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

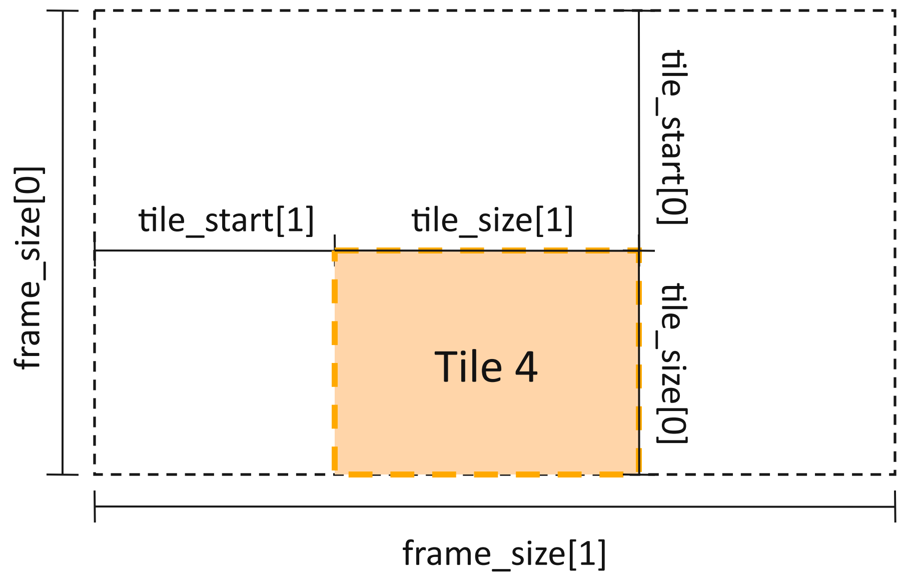
Opening the File, Defining RGB Type

```
MPI_Datatype rgb, filetype;
MPI_File filehandle;
ret = MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

/* collectively open frame file */
ret = MPI_File_open(MPI_COMM_WORLD, filename,
  MPI_MODE_RDONLY, MPI_INFO_NULL, &filehandle);

/* first define a simple, three-byte RGB type */
ret = MPI_Type_contiguous(3, MPI_BYTE, &rgb);
ret = MPI_Type_commit(&rgb);
/* continued on next slide */
```

Defining Tile Type Using Subarray




```
/* in C order, last array
 * value (X) changes most
 * quickly
 */
frame_size[1] = 3*1024;
frame_size[0] = 2*768;
tile_size[1] = 1024;
tile_size[0] = 768;
tile_start[1] = 1024 * (myrank % 3);
tile_start[0] = (myrank < 3) ? 0 : 768;
ret = MPI_Type_create_subarray(2, frame_size,
   tile_size, tile_start, MPI_ORDER_C, rgb,
   &filetype);
ret = MPI_Type_commit(&filetype);
```

Reading Noncontiguous Data

```
/* set file view, skipping header */
ret = MPI_File_set_view(filehandle,
  file_header_size, rgb, filetype, "native",
  MPI_INFO_NULL);
/* collectively read data */
ret = MPI_File_read_all(filehandle, buffer,
  tile_size[0] * tile_size[1], rgb, &status);
ret = MPI_File_close(&filehandle);
```

- MPI_File_set_view is the MPI-IO mechanism for describing noncontiguous regions in a file
  - In this case we use it to skip a header and read a subarray
- Using file views, rather than reading each individual piece, gives the implementation more information to work with (more later)
- Likewise, using a collective I/O call (MPI_File_read_all) provides additional information for optimization purposes (more later)

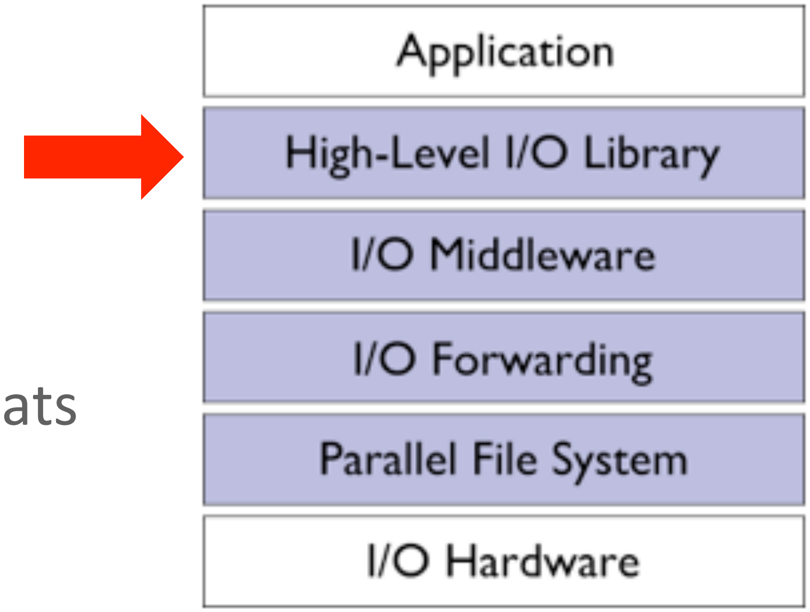
MPI-IO Wrap-Up

- MPI-IO provides a rich interface allowing us to describe
  - Noncontiguous accesses in memory, file, or both
  - Collective I/O
- This allows implementations to perform many transformations that result in better I/O performance
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

The Parallel netCDF Interface and File Format

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

Higher Level I/O Interfaces

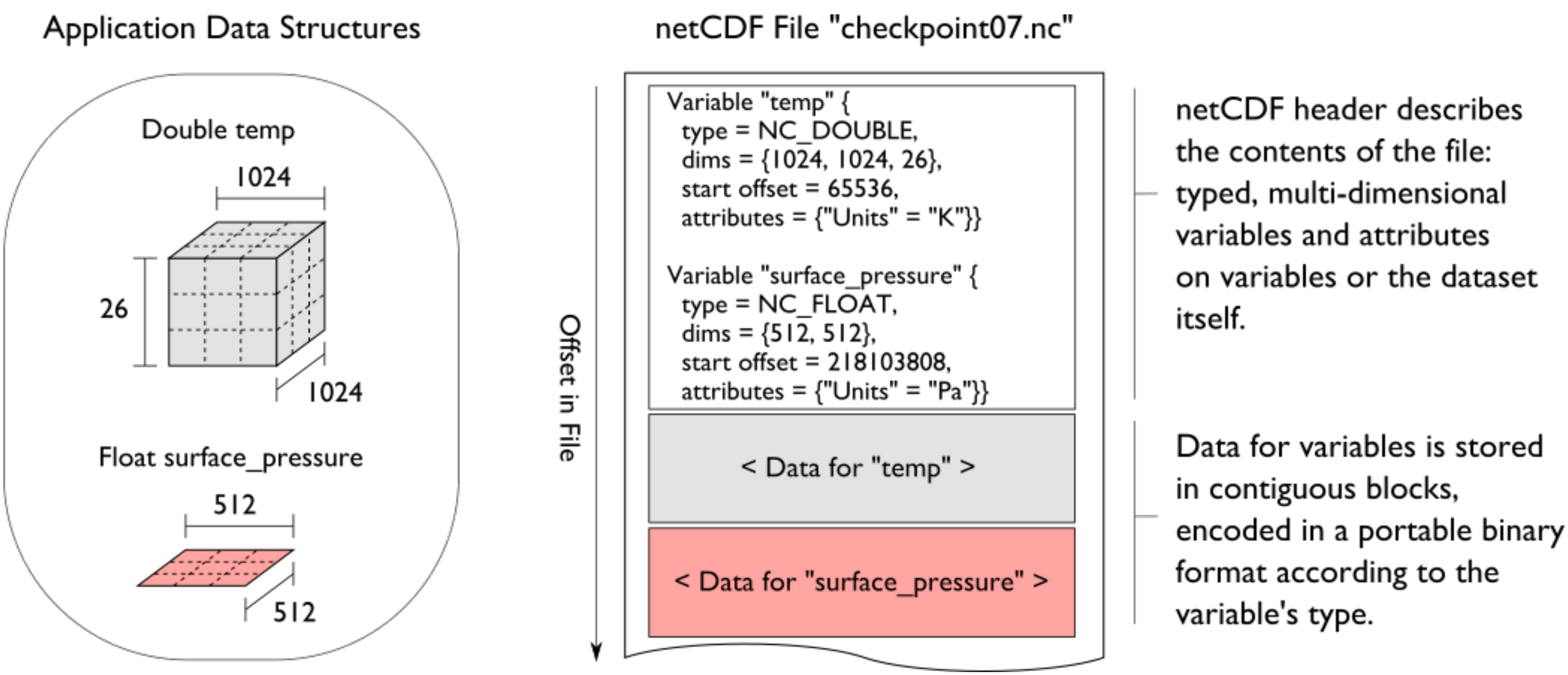


- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents
- Present APIs more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays 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 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

Data Layout in netCDF Files



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

PnetCDF Wrap-Up

- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
  - Nonblocking option
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead

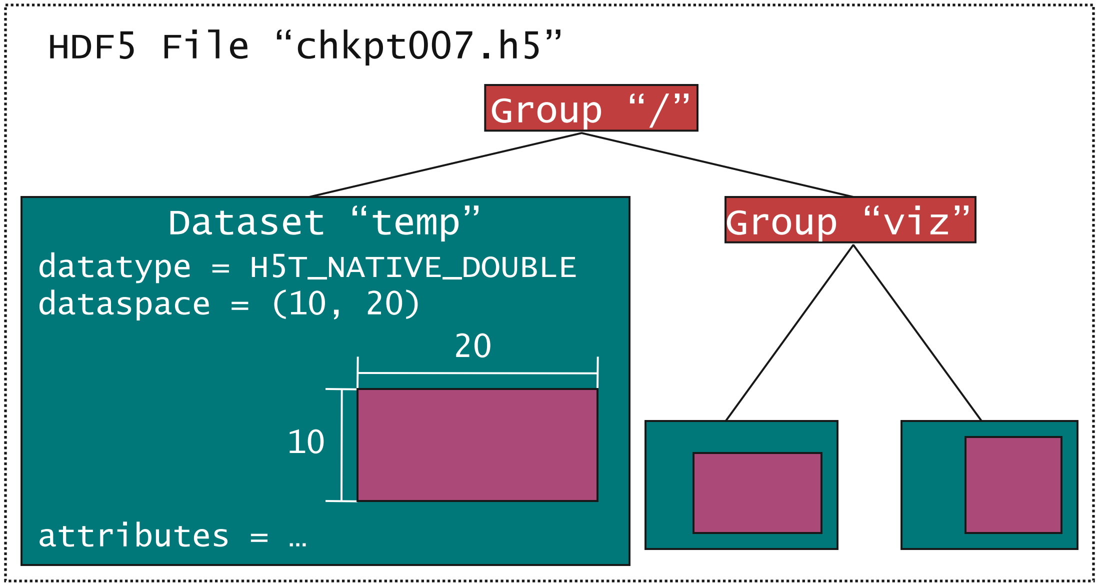
The HDF5 Interface and File Format

Thanks to Quincey Koziol (HDF group)

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 files consist of groups, datasets, and attributes
  - Groups are like directories, holding other groups and datasets
  - Datasets hold an array of typed data
    - 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

HDF5 Data Chunking

- Apps often read subsets of arrays (subarrays)
- Performance of subarray access depends in part on how data is laid out in the file
  - e.g. column vs. row major
- Apps also sometimes store sparse data sets
- Chunking describes a reordering of array data
  - Subarray placement in file determined lazily
  - Can reduce worst-case performance for subarray access
  - Can lead to efficient storage of sparse data
- Dynamic placement of chunks in file requires coordination
  - Coordination imposes overhead and can impact performance

The ADaptable IO System (ADIOS)

Thanks to Scott Klasky (ORNL) for providing background material on ADIOS.

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 Philosophy (End User)

- Simple API very similar to standard Fortran or C POSIX IO calls.
  - As close to identical as possible for C and Fortran API
  - open, read/write, close is the core
  - set_path, end_iteration, begin/end_computation, init/finalize are the auxiliaries
- No changes in the API for different transport methods.
- Metadata and configuration defined in an external XML file parsed once on startup.
  - Describe the various IO groupings including attributes and hierarchical path structures for elements as an adios-group
  - Define the transport method used for each adios-group and give parameters for communication/writing/reading
  - Change on a per element basis what is written
  - Change on a per adios-group basis how the IO is handled

ADIOS and File Formats

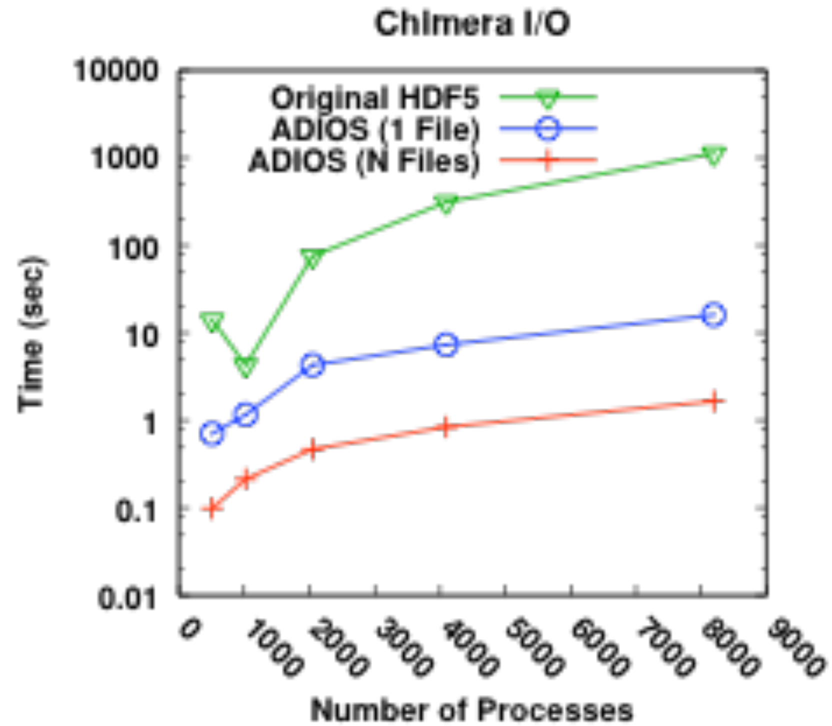
- netCDF and HDF-5 are excellent, mature file formats
- APIs can have trouble scaling to petascale and beyond
  - metadata operations bottleneck at MDS
  - coordination among all processes takes time
  - MPI Collective writes/reads add additional coordination
  - Non-stripe-sized writes impact performance
  - Read/write mode is slower than write only
  - Replicate some metadata for resilience
- ADIOS provides a custom file format for accelerating large-scale write operations

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



I/O times for Chimera astrophysics application on Cray XT at ORNL. “1 File” results may benefit from Lustre optimizations that were not in place at time of testing.

Other High-Level I/O libraries

- NetCDF-4: http://www.unidata.ucar.edu/software/netcdf/netcdf-4/
  - netCDF API with HDF5 back-end
- 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.

Lightweight Application Characterization with Darshan

Thanks to Phil Carns (ANL)

Characterizing Application I/O

How are 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 POSIX and MPI-IO operations
  - 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

http://www.mcs.anl.gov/darshan/

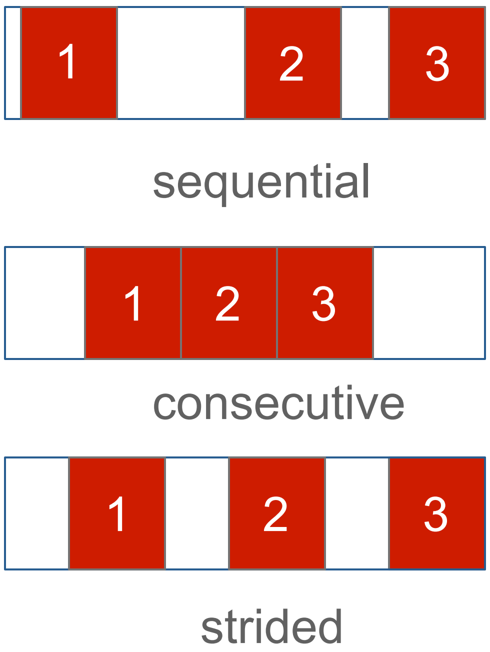
P. Carns et al, "24/7 Characterization of Petascale I/O Workloads," IASDS Workshop, held in conjunction with IEEE Cluster 2009, September 2009.

The Darshan Approach

- 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

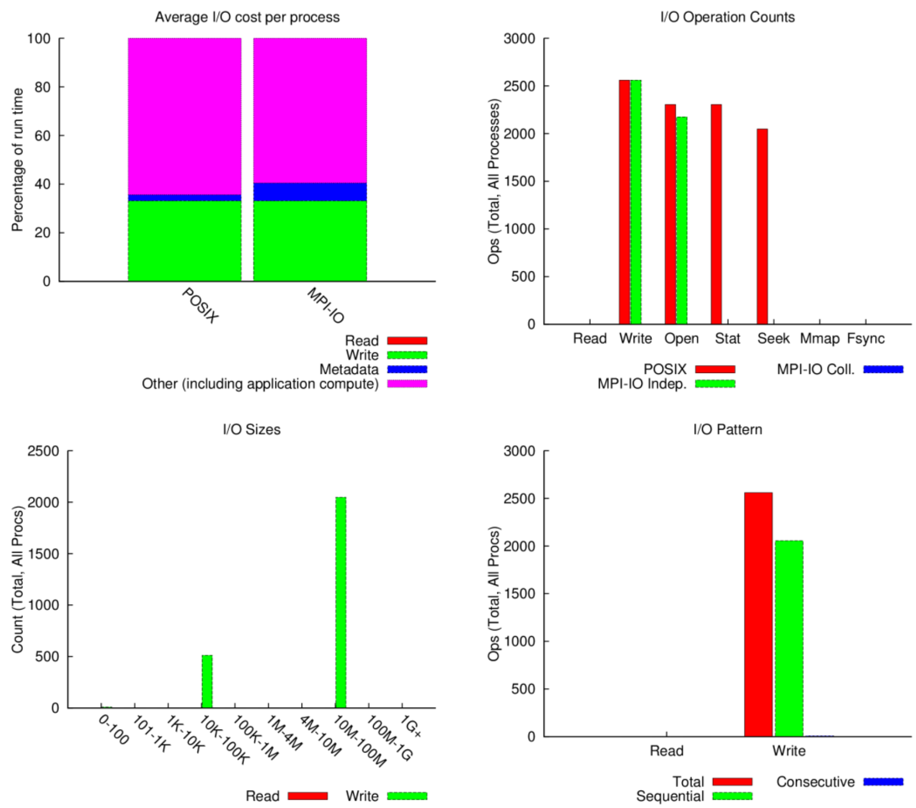
Example Statistics (per file)

- Counters:
  - POSIX open, read, write, seek, stat, etc.
  - MPI-IO nonblocking, collective, indep., etc.
  - Unaligned, sequential, consecutive, strided access
  - MPI-IO datatypes and hints
- Histograms:
  - access, stride, datatype, and extent sizes
- Timestamps:
  - open, close, first I/O, last I/O
- Cumulative bytes read and written
- Cumulative time spent in I/O and metadata operations
- Most frequent access sizes and strides
- Darshan records 150 integer or floating point parameters per file, plus job level information such as command line, execution time, and number of processes.



Darshan Job Summary

jobid: 2	uid:	nprocs: 4096	runtime: 175 seconds



Most Common Access Sizes

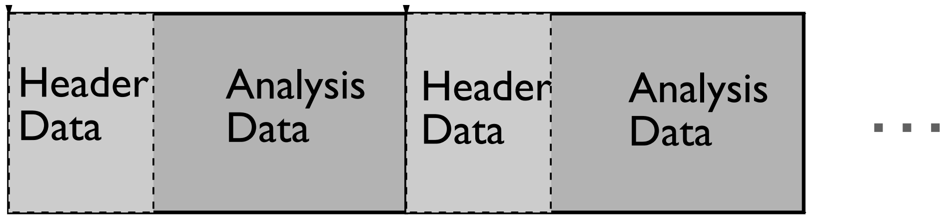
access size	count
67108864	2048
41120	512
8	4
4	3

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”; available to all users
- Shows time spent in read, write, and metadata
- Operation counts, access size histogram, and access pattern
- 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

A Data Analysis I/O Example



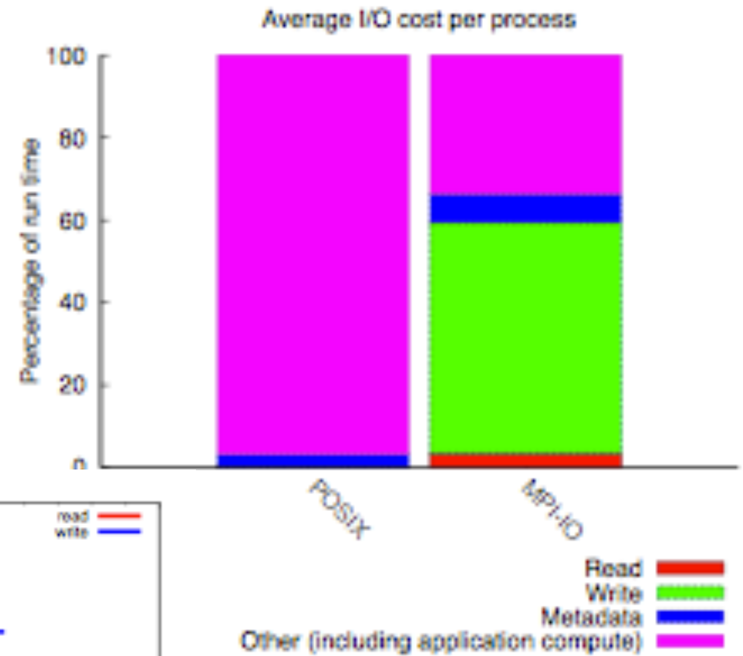
- Variable size analysis data requires headers to contain size information
- Original idea: all processes collectively write headers, followed by all processes collectively write analysis data
- Use MPI-IO, collective I/O, all optimizations
- 4 GB output file (not very large)

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

- Why does the I/O take so long in this case?

A Data 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.

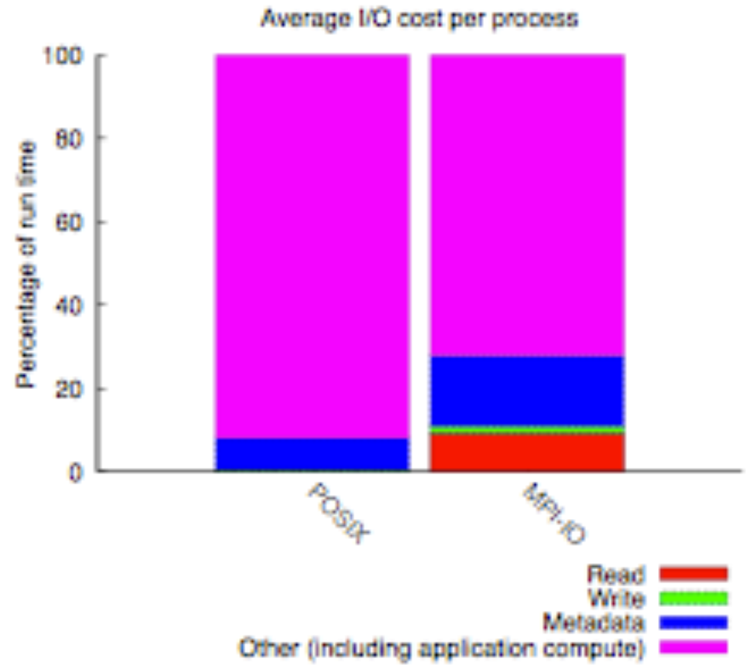


- What we expected to see, read data followed by write analysis:

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

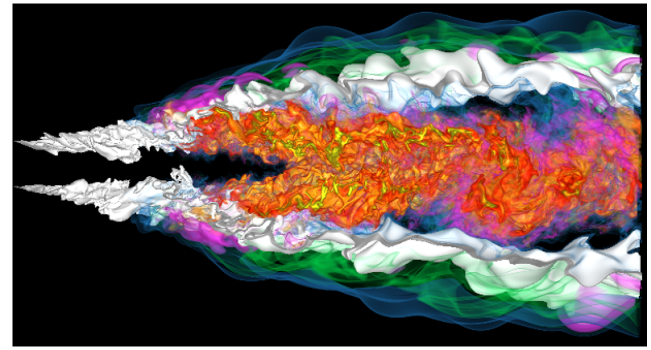
A Data Analysis I/O Example (continued)

- 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. Also fix miscellaneous I/O bugs; both problems were identified using Darshan.
- 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. Also used similar output strategy for cosmology checkpointing, further leveraging the lessons learned.

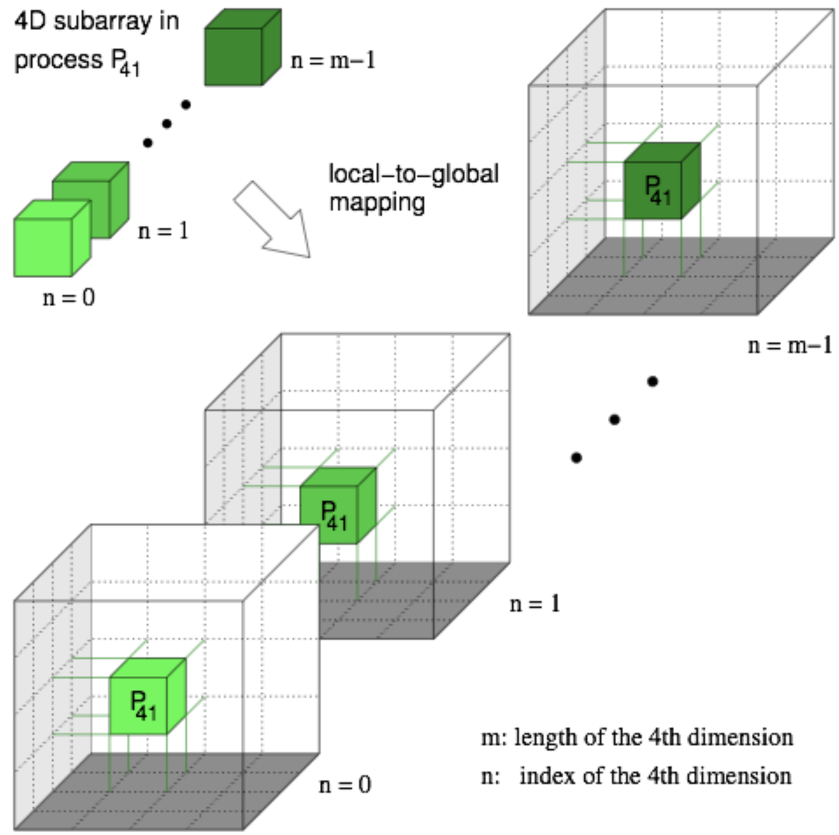


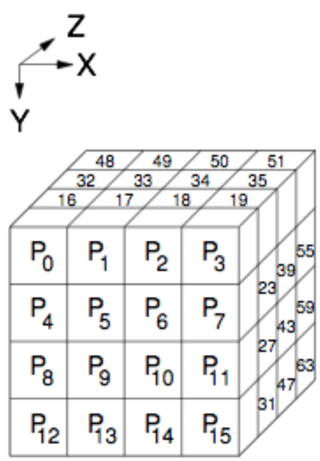
Processes	I/O Time (s)	Total Time (s)
8,192	7	60
16,384	6	40
32,768	7	33

S3D Turbulent Combustion Code



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





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

Impact of 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
  - Collective I/O optimization (collective buffering, a.k.a. two-phase I/O) enabled

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

Darshan Summary

- Scalable tools like Darshan can yield useful insight
  - Identify characteristics that make applications successful
  - 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:

  http://www.mcs.anl.gov/research/projects/darshan

Summary

Wrapping Up

- We've covered a lot of ground in a short time
  - Very low-level, serial interfaces
  - High-level, hierarchical file formats
- Storage is a complex hardware/software system
- There is no magic in high performance I/O
  - Lots of software is available to support computational science workloads at scale
  - Knowing how things work will lead you to better performance
- Using this software (correctly) can dramatically improve performance (execution time) and productivity (development time)

Printed References

- John May, Parallel I/O for High Performance Computing, 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, Using MPI-2: Advanced Features of the Message Passing Interface, 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

- netCDF and netCDF-4
  - http://www.unidata.ucar.edu/packages/netcdf/
- 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://www.hdfgroup.org/HDF5/
  - http://www.hdfgroup.org/HDF5/Tutor
- Darshan I/O Characterization Tool
  - http://www.mcs.anl.gov/research/projects/darshan
- Assorted ALCF-Specific suggestions:
  - https://wiki.alcf.anl.gov/index.php/I_O_Tuning
  - http://wiki.mcs.anl.gov/Darshan/index.php/Documentation_for_ALCF_users

Parallel I/O in Practice

Tom Peterka

Mathematics and Computer Science Division

Argonne National Laboratory

July 26, 2012

This work is supported in part by U.S. Department of Energy Grant DE-FC02-01ER25506, by National Science Foundation Grants EIA-9986052, CCR-0204429, and CCR-0311542, and by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

Thanks to Rob Latham (ANL), Rob Ross (ANL), Phil Carns (ANL), Scott Klasky (ORNL), Wei-keng Liao (NWU), Alok Choudhary (NWU), and Quincey Koziol (HDF Group) for this material.