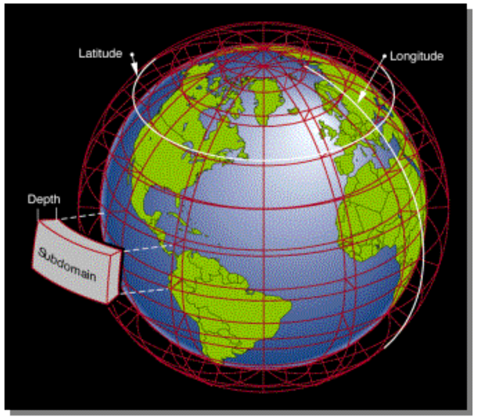
Making the Most of the I/O stack

Rob Latham
robl@mcs.anl.gov
Mathematics and Computer Science Division
Argonne National Laboratory

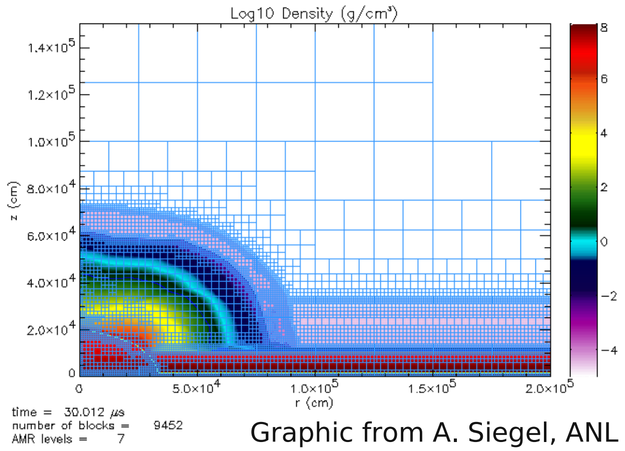
July 26, 2010

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

Large-Scale Data Sets

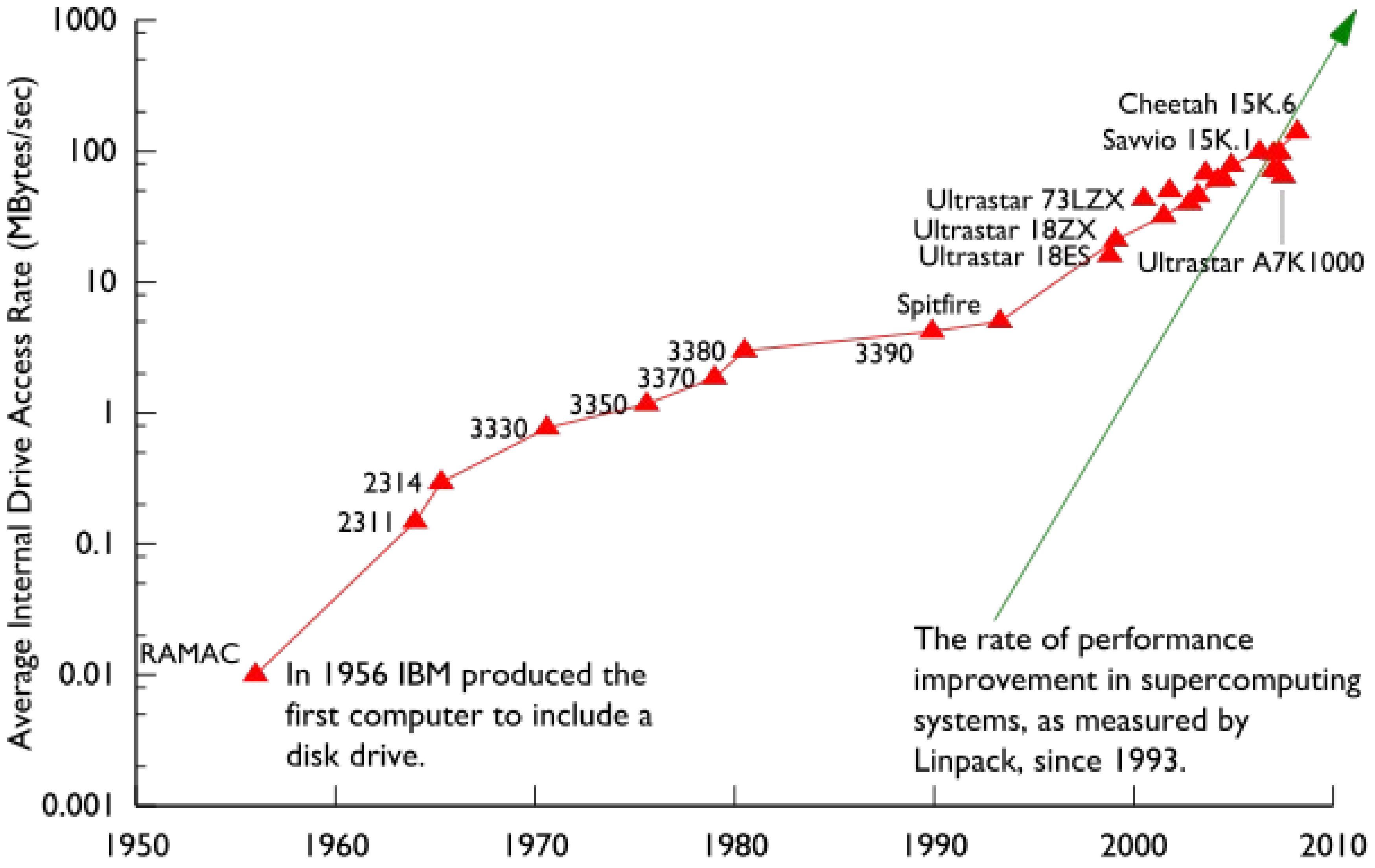
Application teams are beginning to generate 10s of Tbytes of data in a single simulation. For example, a recent GTC run on 29K processors on the XT4 generated over 54 Tbytes of data in a 24 hour period [1].

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.

Disk Access Rates over Time



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

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.

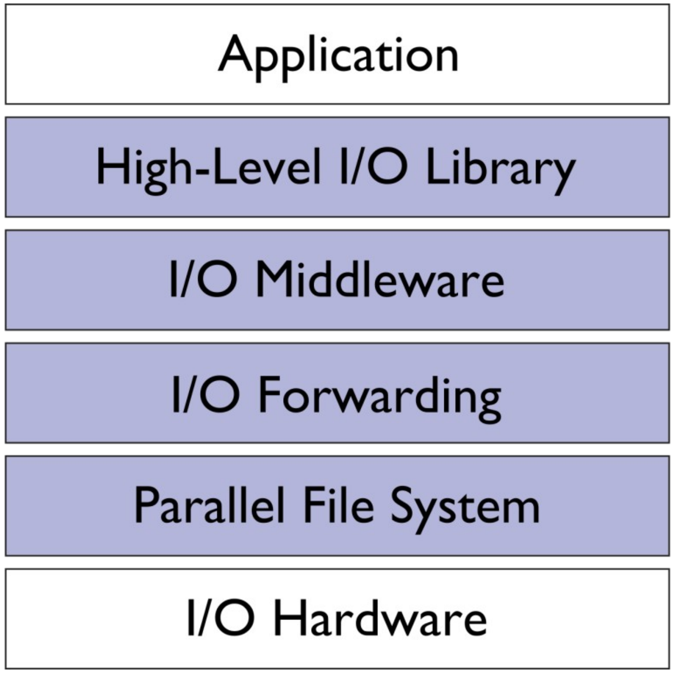
I/O for Computational Science

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.

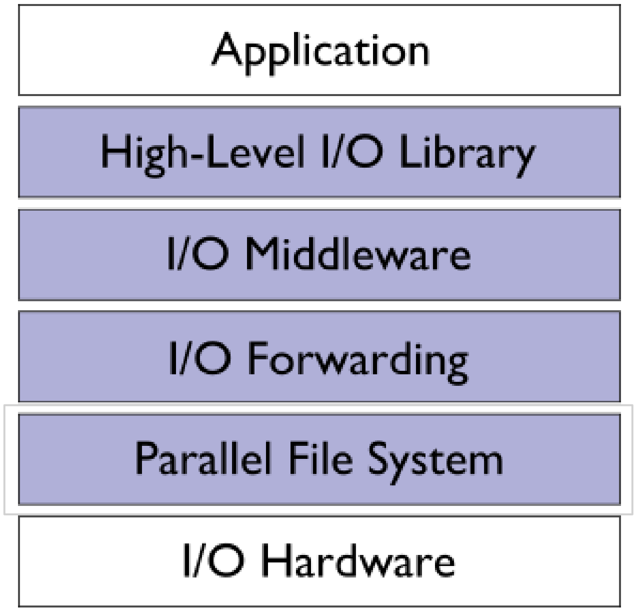
MPI-IO

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

PVFS, PanFS, GPFS, Lustre

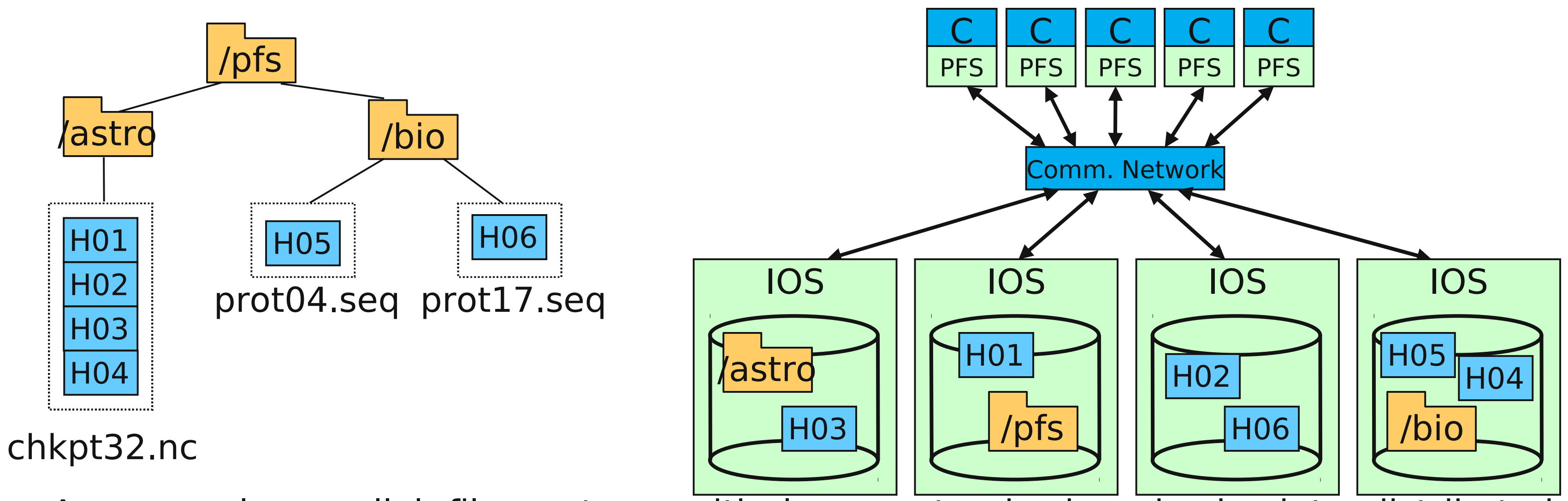
Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.

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

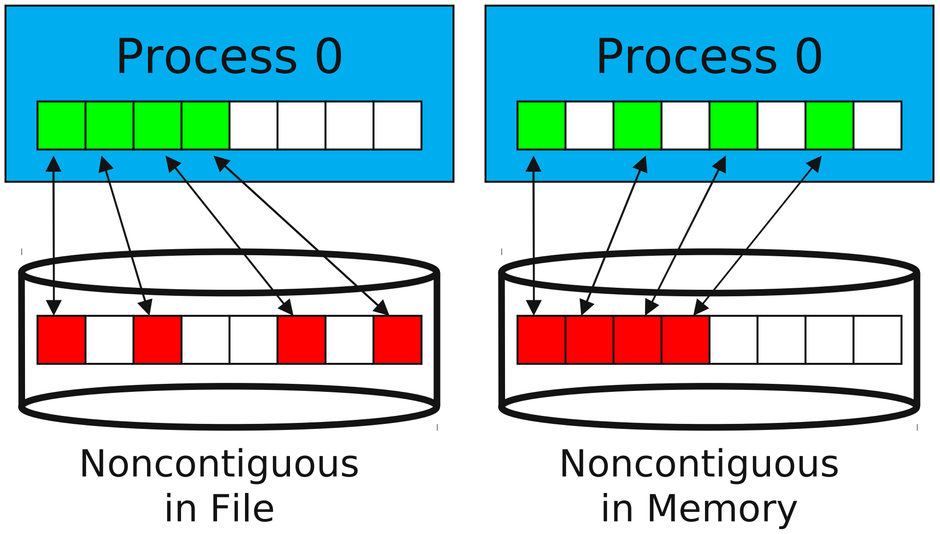
Parallel File Systems

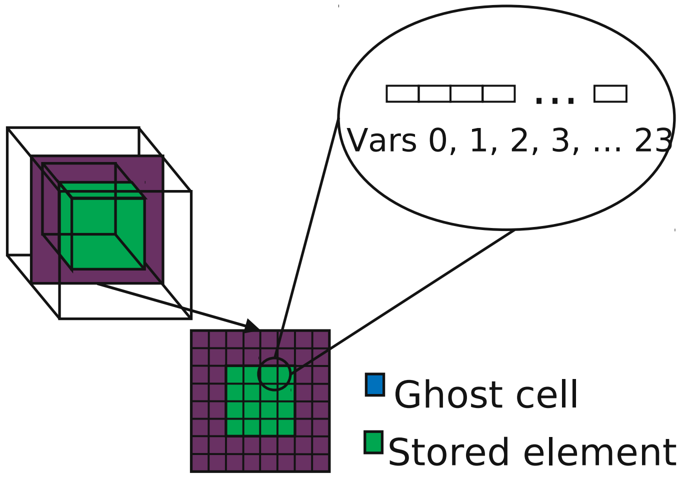


An example parallel file system, with large astrophysics checkpoints distributed across multiple I/O servers (IOS) while small bioinformatics files are each stored on a single IOS.

- 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

Contiguous and Noncontiguous I/O





Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O.

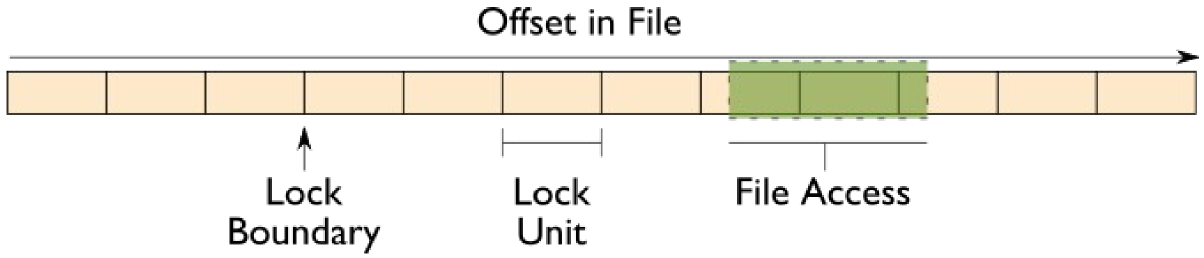
- Contiguous I/O moves data from a single memory block into a single file region
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous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

Locking in Parallel File Systems

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

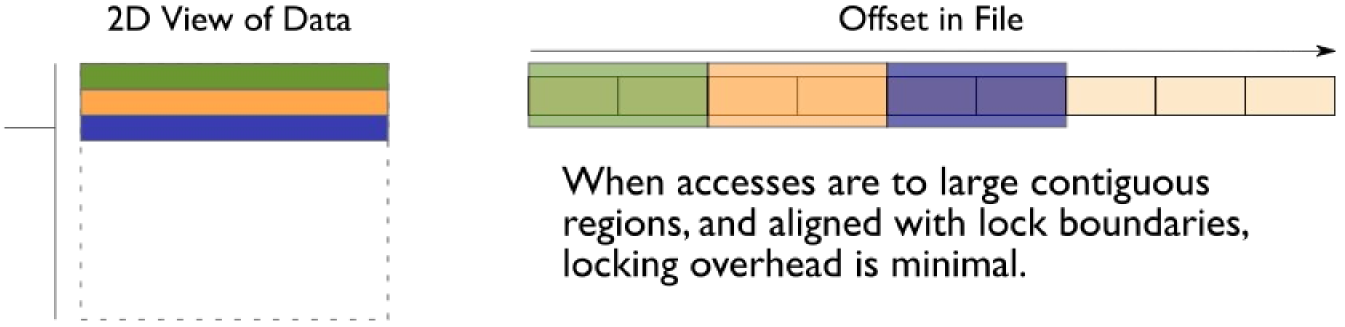
- Files are broken up into lock units
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

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



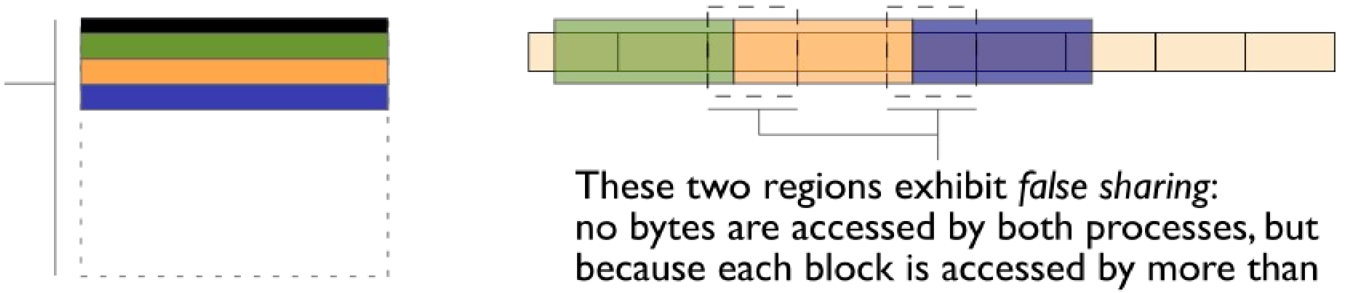
Locking and Concurrent Access

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.



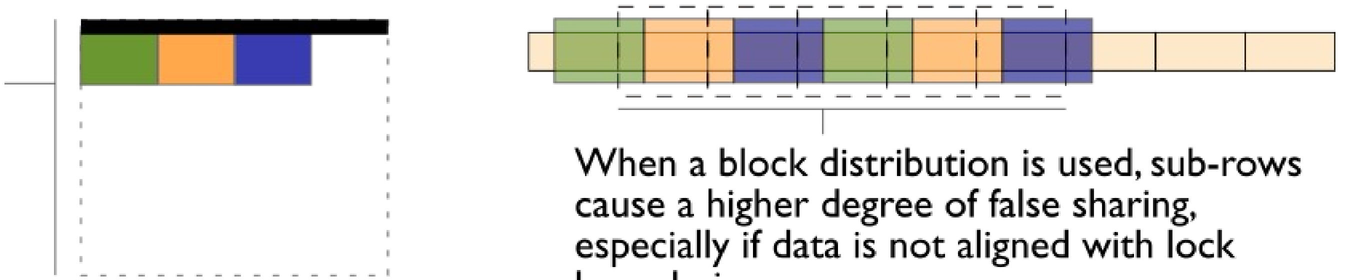
When accesses are to large contiguous regions, and aligned with lock boundaries, locking overhead is minimal.

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.



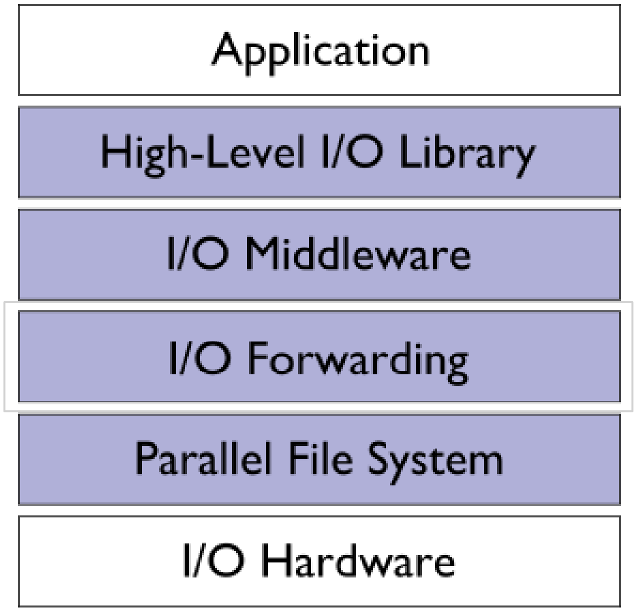
These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

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



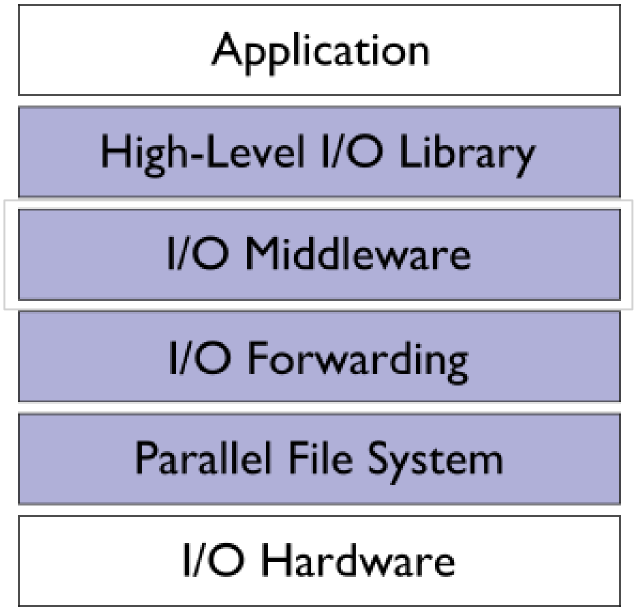
When a block distribution is used, sub-rows cause a higher degree of false sharing, especially if data is not aligned with lock boundaries.

I/O Forwarding



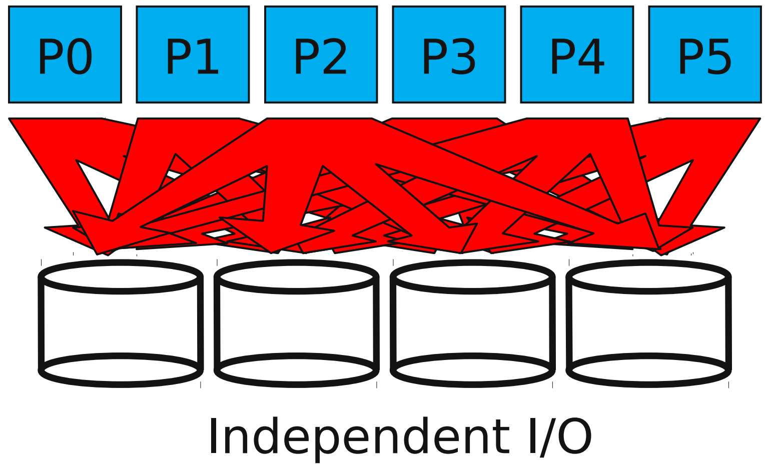
- Newest layer in the stack
  - Present in some of the largest systems
  - Provides bridge between system and storage in machines such as the Blue Gene/P
- Allows for a point of aggregation, hiding true number of clients from underlying file system
- Poor implementations can lead to unnecessary serialization, hindering performance

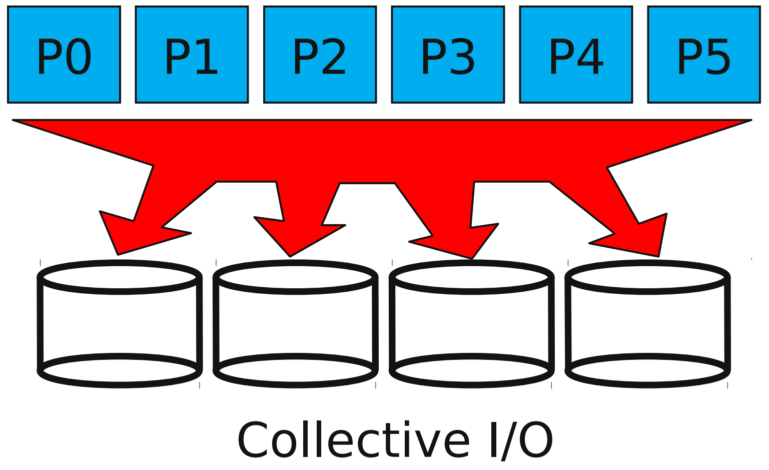
I/O Middleware



- 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

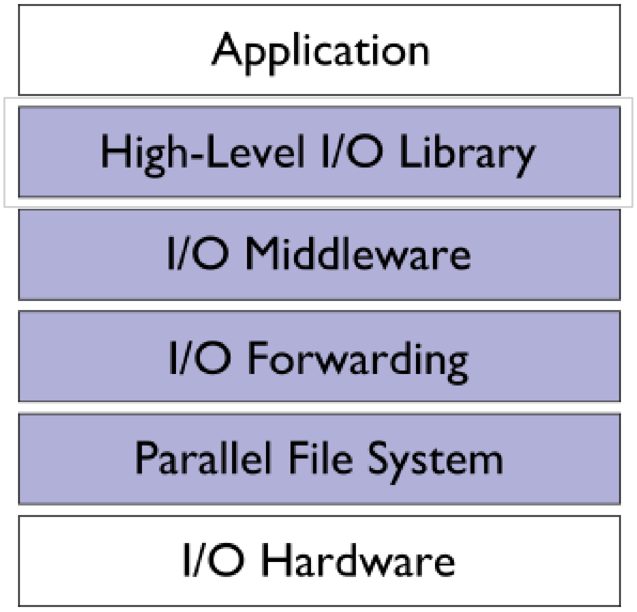
Independent and Collective I/O





- 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
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- 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

High Level Libraries



- Match storage abstraction to domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Provide self-describing, structured files
- Map to middleware interface
  - Encourage collective I/O
- Implement optimizations that middleware cannot, such as
  - Caching attributes of variables
  - Chunking of datasets

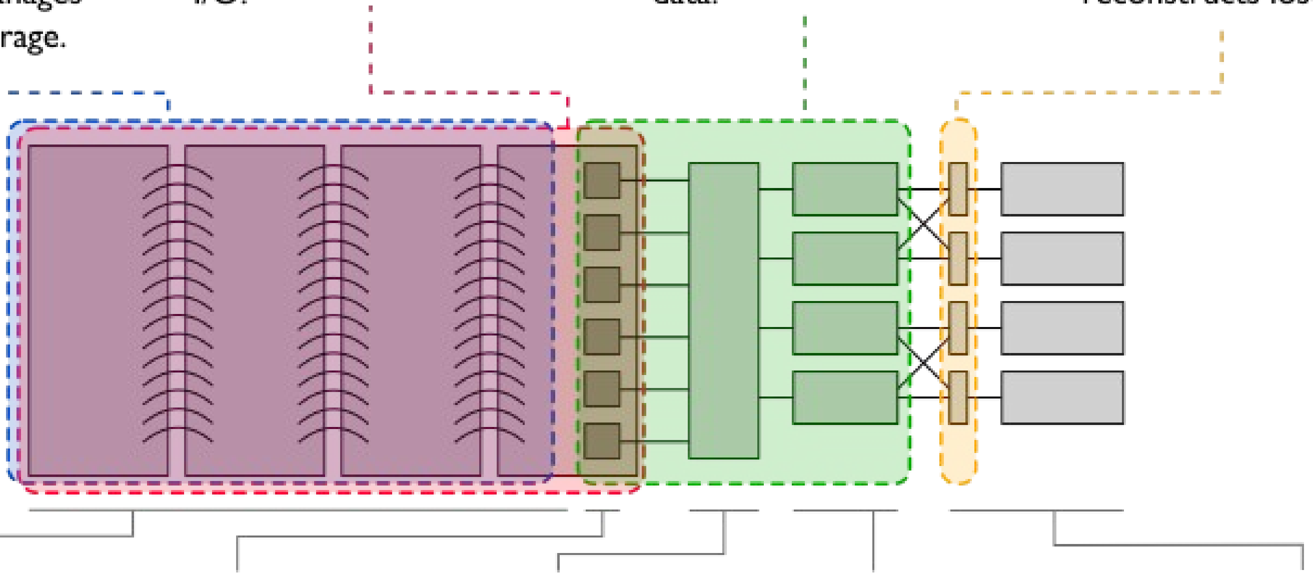
I/O Hardware and Software on Blue Gene/P

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

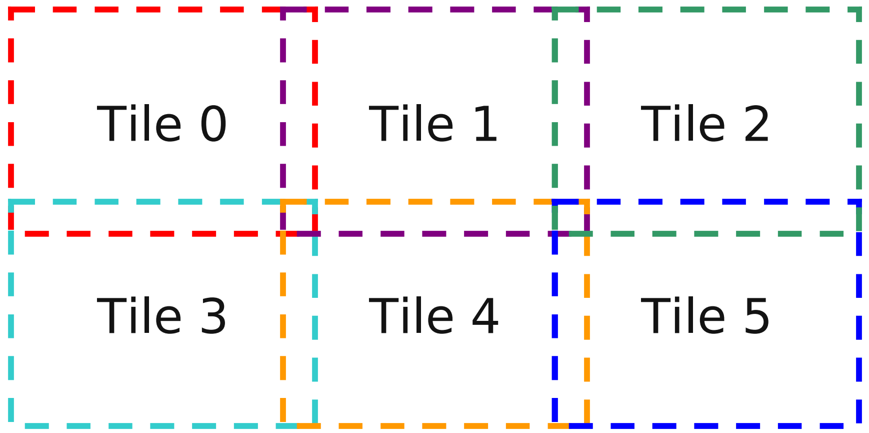
The MPI-IO Interface

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 (more later)

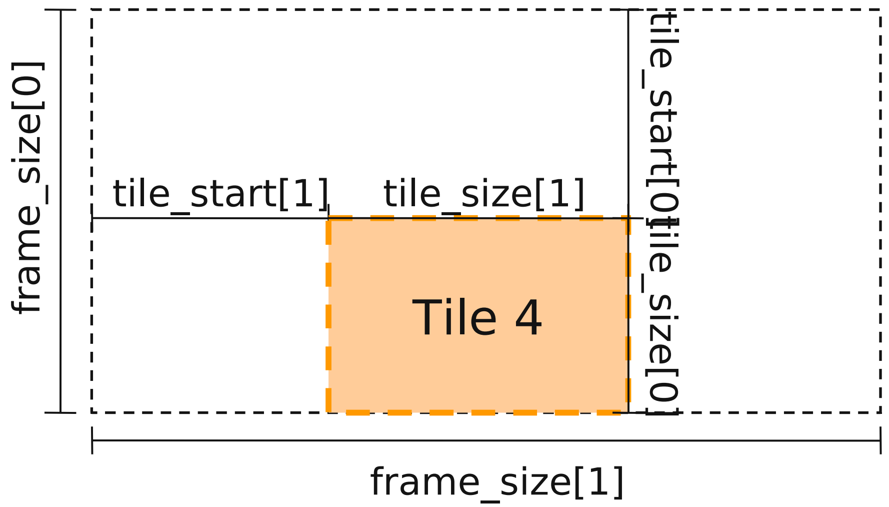
Example: Visualization Staging



- 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

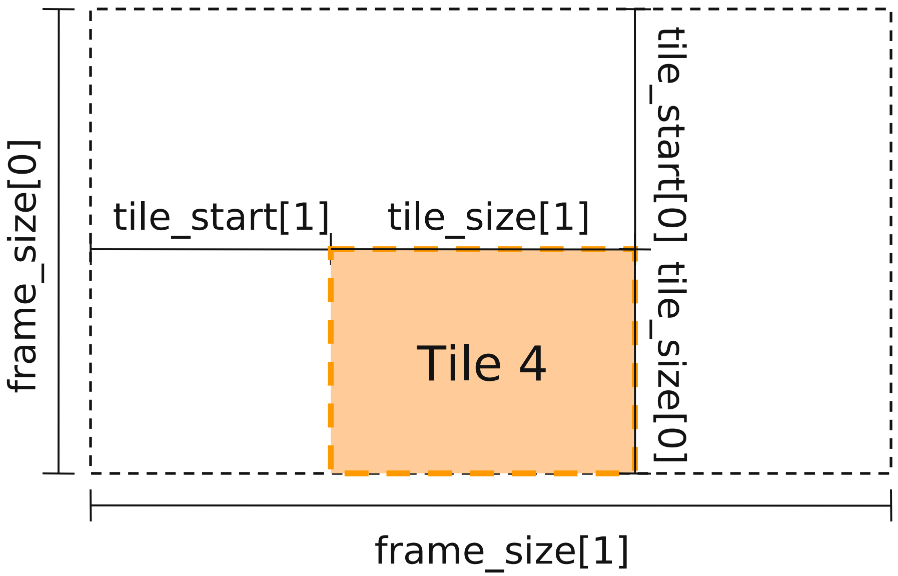


Opening the File, Defining RGB Type

```
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

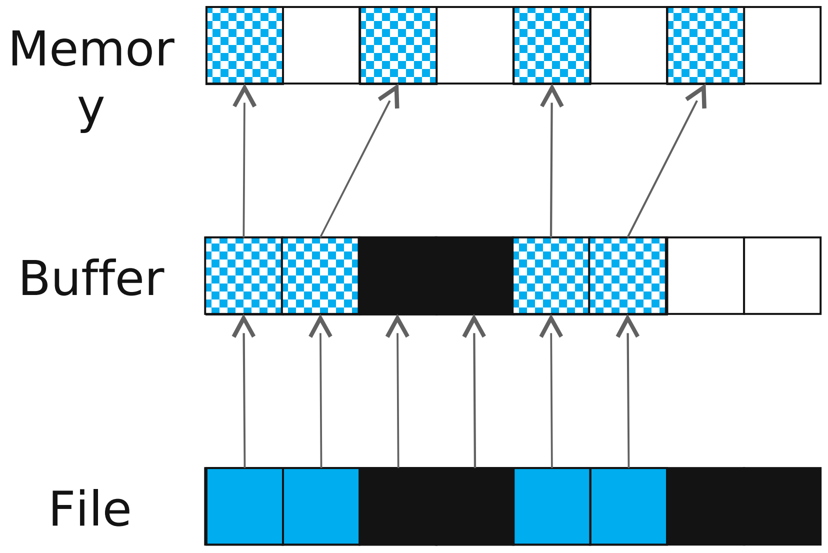
```
/* set file view, skipping header */
ret = MPI_File_set_view(filehandle,
  file_header_size, rgb, filetype, "native",
  MPI_INFO_NULL);
/* collectively read data */
ret = MPI_File_read_all(filehandle, buffer,
  tile_size[0] * tile_size[1], rgb, &status);
ret = MPI_File_close(&filehandle);
```

- 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)

Under the Covers of MPI-IO

- MPI-IO implementation given a lot of information in this example:
  - Collection of processes reading data
  - Structured description of the regions
- Implementation has some options for how to perform the data reads
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

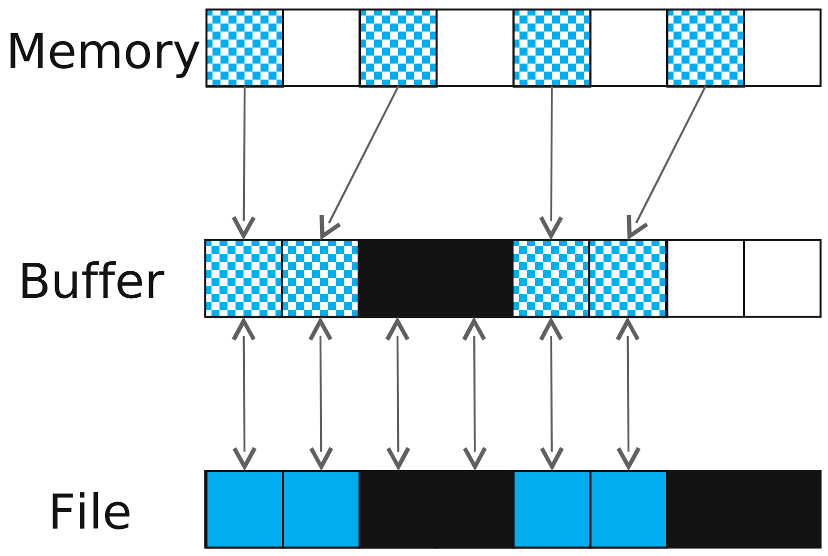
Noncontiguous I/O: Data Sieving



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

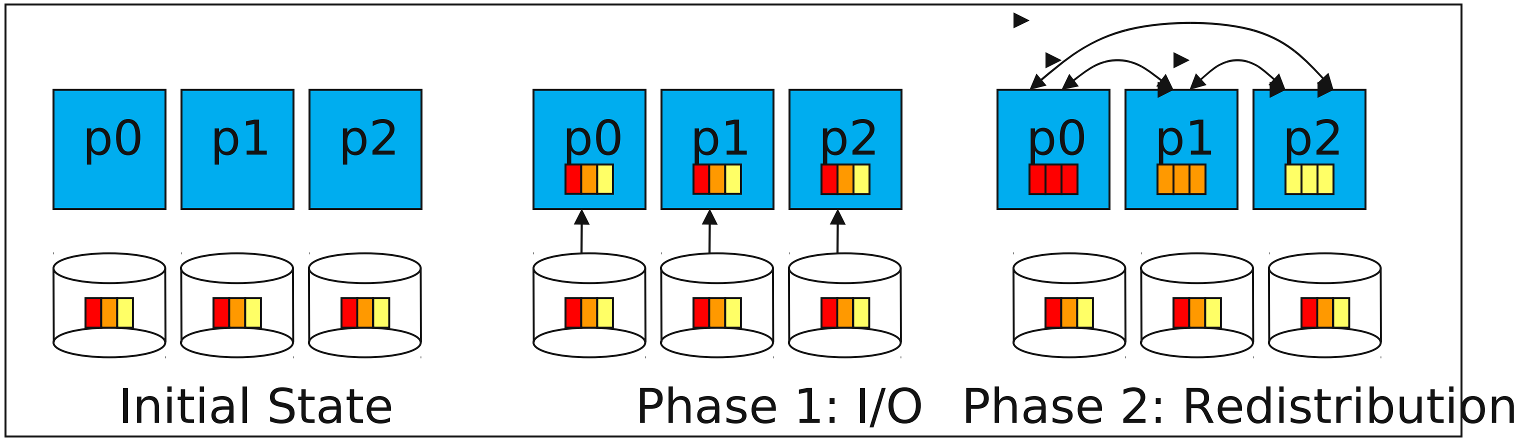
Data Sieving Write Operations



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 and Two-Phase I/O

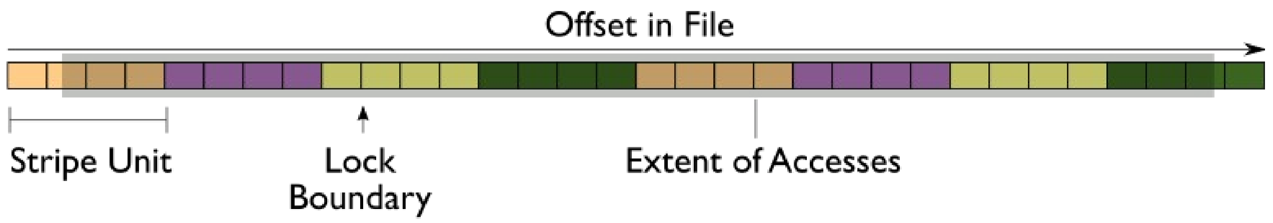


Two-Phase Read Algorithm

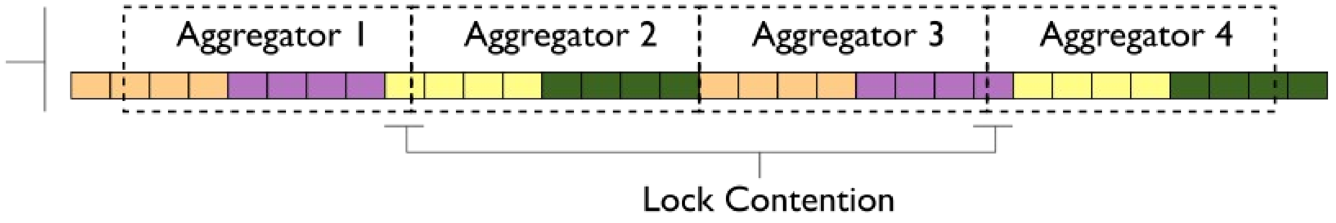
- Problems with independent, noncontiguous access
  - Lots of small accesses
  - Independent data sieving reads lots of extra data, can exhibit false sharing
- Idea: Reorganize access to match layout on disks
  - Single processes use data sieving to get data for many
  - Often reduces total I/O through 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

Two-Phase I/O Algorithms

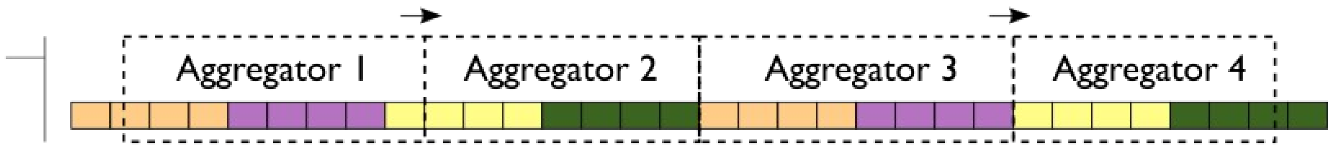
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



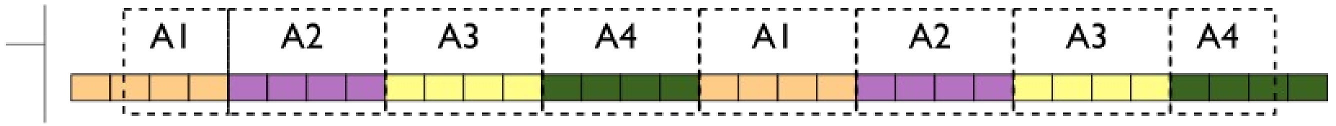
One approach is to evenly divide the region accessed across aggregators.



Aligning regions with lock boundaries eliminates lock contention.



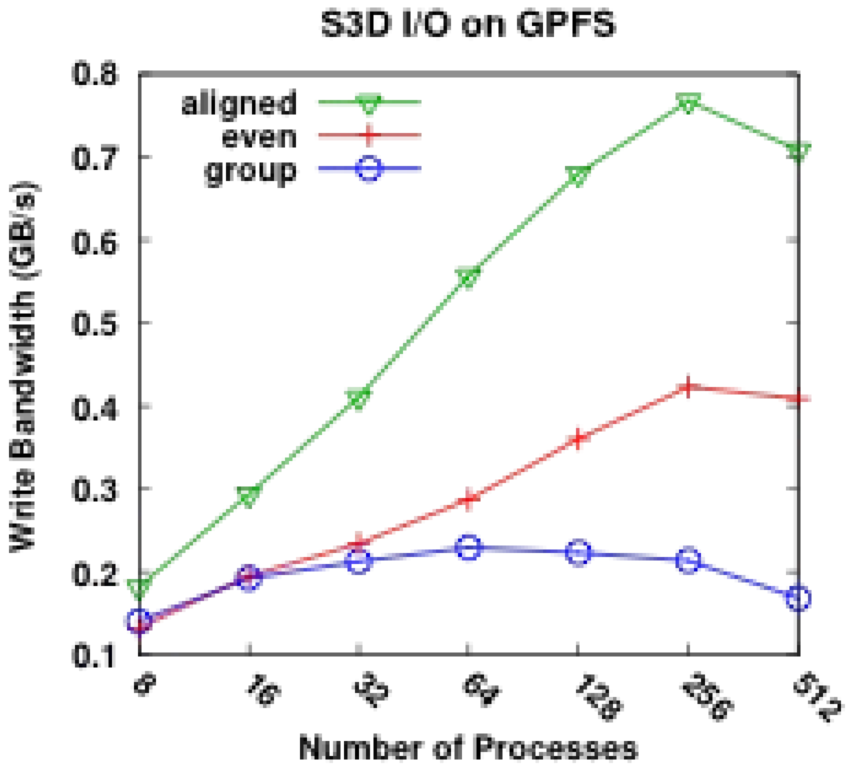
Mapping aggregators to servers reduces the number of concurrent operations on a single server and can be helpful when locks are handed out on a per-server basis (e.g., Lustre).



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.

Impact of Two-Phase I/O Algorithms

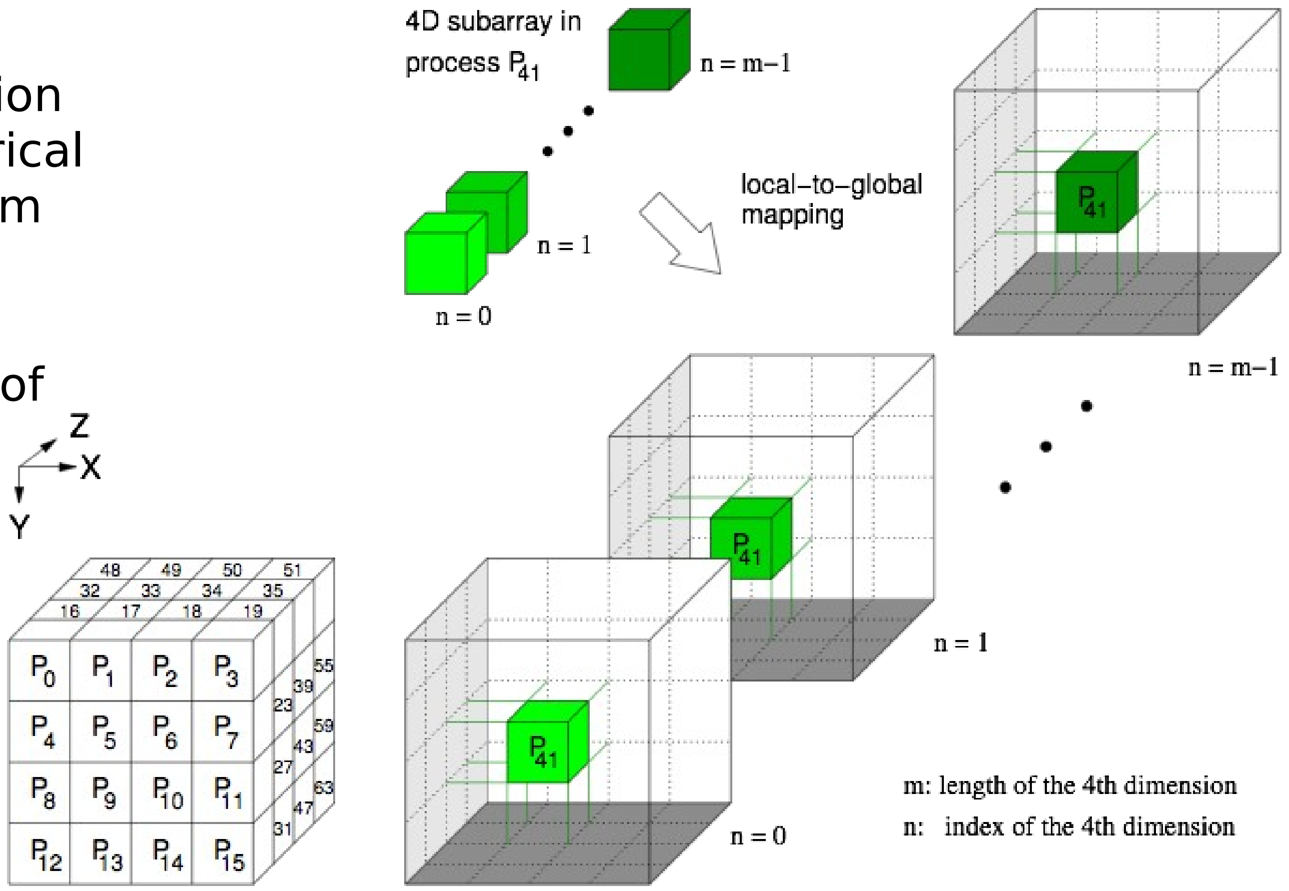
- This graph shows the performance for the S3D combustion code, writing to a single file.
- Aligning with lock boundaries doubles performance over default "even" algorithm.
- "Group" algorithm similar to server-aligned algorithm on last slide.
- Testing on Mercury, an IBM IA64 system at NCSA, with 54 servers and 512KB stripe size.



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

S3D Turbulent Combustion Code

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



Thanks to Jackie Chen (SNL), Ray Grout (SNL), and Wei-Keng Liao (NWU) for providing the S3D I/O benchmark, Wei-Keng Liao for providing this diagram.

Impact of Optimizations on S3D I/O

	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

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.

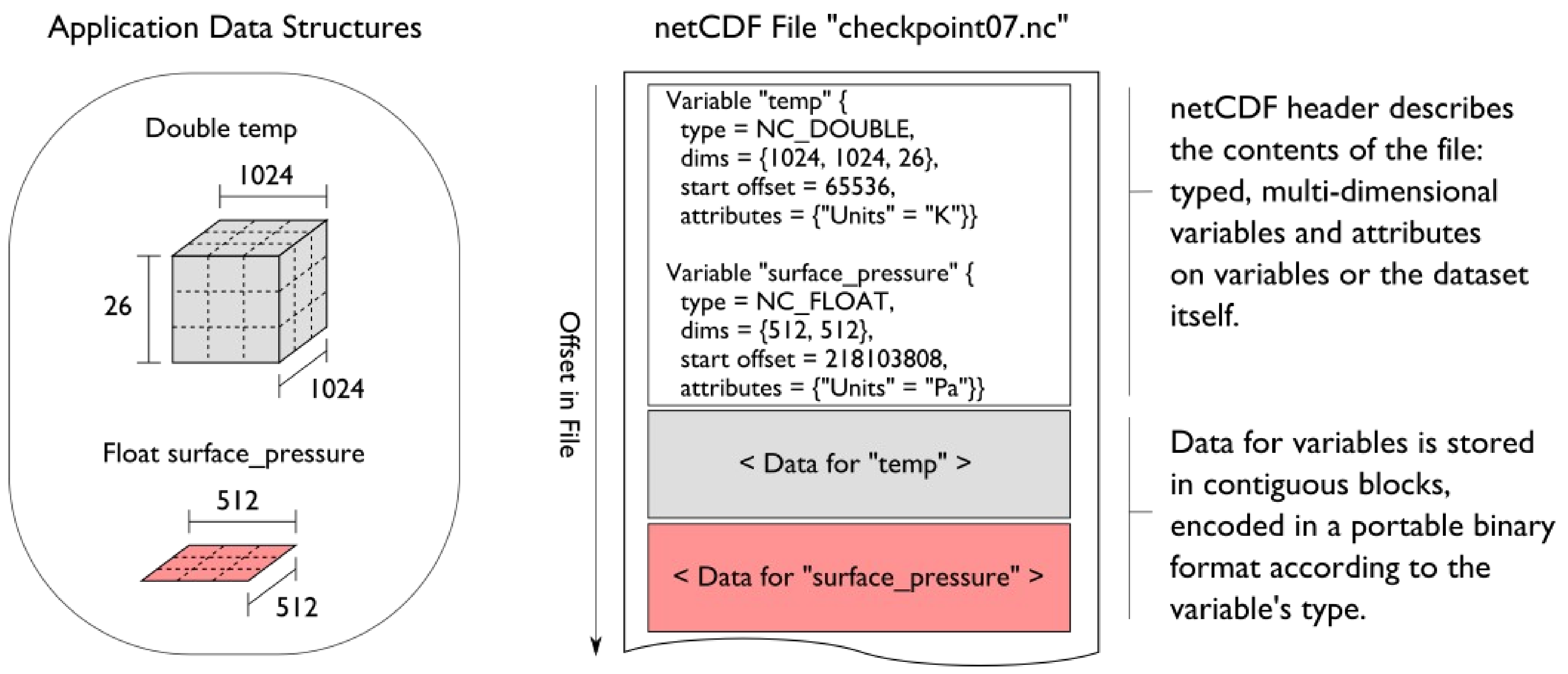
Higher Level I/O Interfaces

- 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 work (More about netCDF-4 later)

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
< Data for "temp" >
< Data for "surface_pressure" >

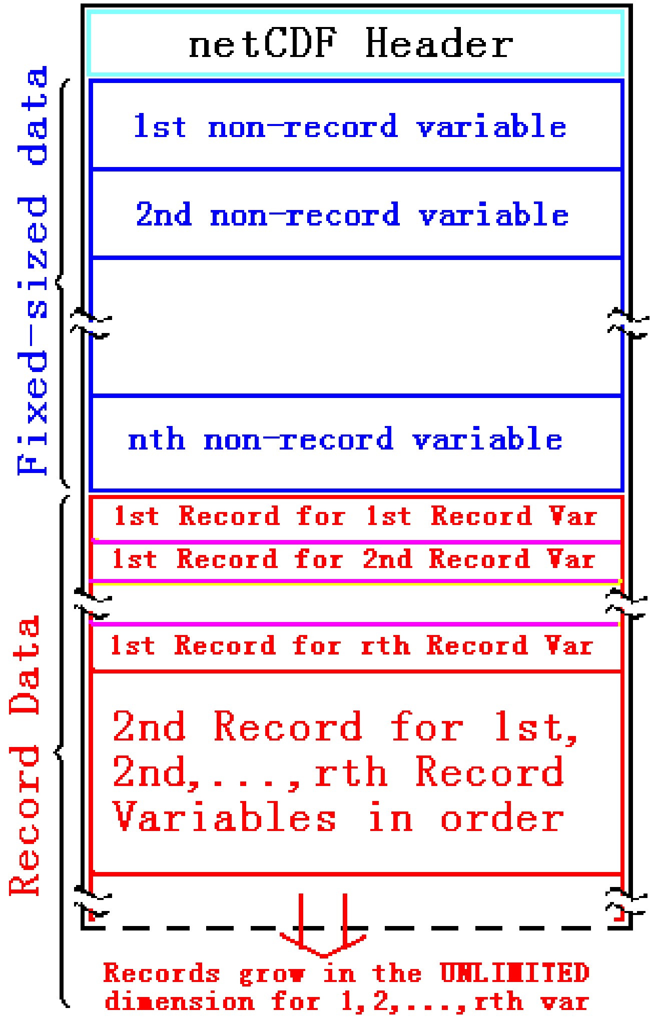
```
Variable "temp" {
  type = NC_DOUBLE,
  dims = {1024, 1024, 26},
  start offset = 65536,
  attributes = {"Units" = "K"}}

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.

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

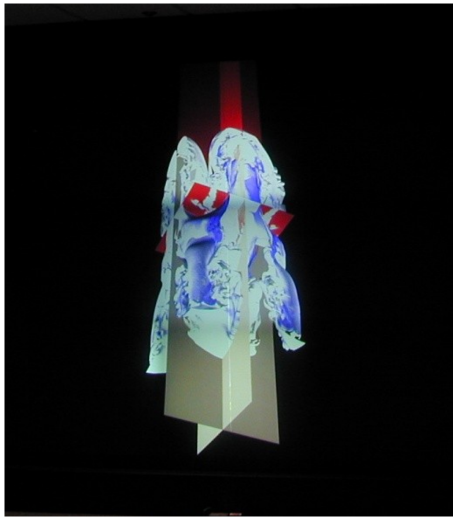

netCDF Header
Fixed-sized data
1st non-record variable
2nd non-record variable
nth non-record variable
Record Data
1st Record for 1st Record Var
1st Record for 2nd Record Var
1st Record for rth Record Var
2nd Record for 1st, 2nd,...,rth Record Variables in order
Records grow in the UNLIMITED dimension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

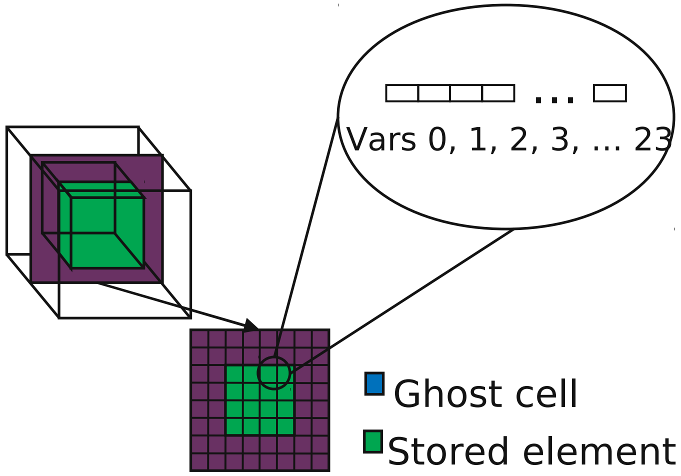
Storing Data in PnetCDF

- 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

PnetCDF Wrap-Up

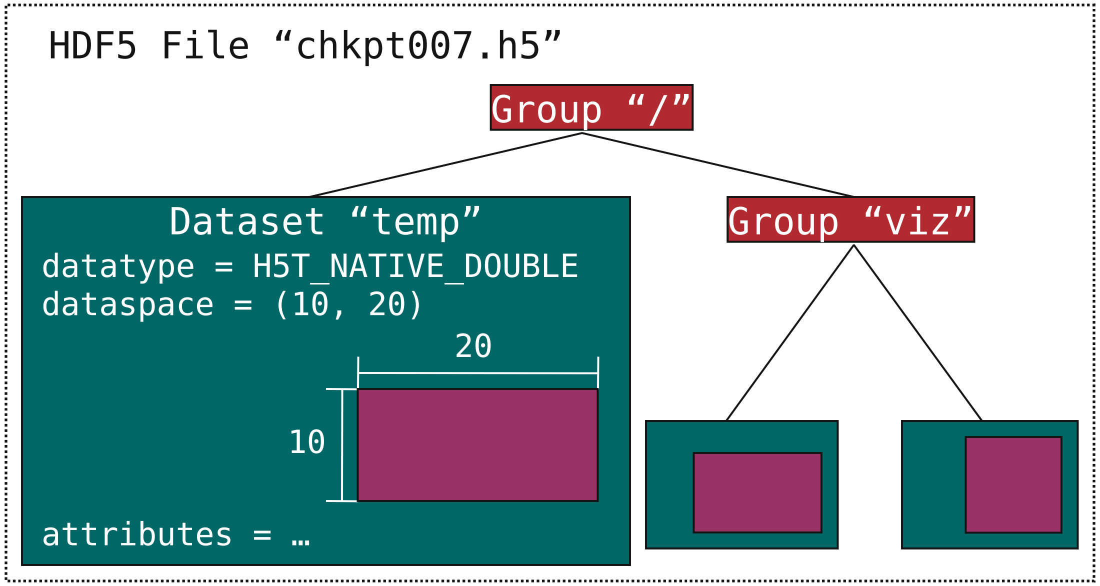
- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead
- Some limits on (CDF-2) file format:
  - Fixed-size variable: < 4 GiB
  - Per-record size of record variable: < 4 GiB
  - 2^{32} -1 records
  - Work completed to relax these limits (CDF-5): still need to port to serial netcdf

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 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

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

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

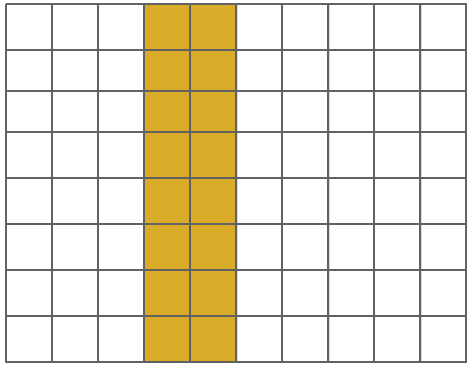
```
int string_size = OUTPUT_PROP_LENGTH;
hsize_t dims_2d[2] = {npart_props, string_size};
hid_t dataspace, dataset, file_id, string_type;

/* 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

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);
```

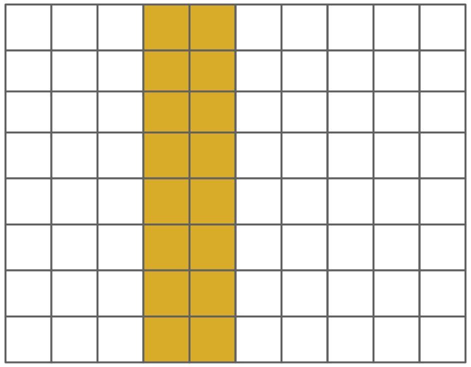


```
local_np = 2,
part_offset = 3,
total_particles = 10,
Npart_props = 8
```

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 */


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



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

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
    - Only if this was turned on (more later)
- Memory hyperslab could have been used to define noncontiguous 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

Other High-Level I/O libraries

- 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.

Lightweight Application Characterization with Darshan

Thanks to Phil Carns (carns@mcs.anl.gov) for providing background material on Darshan.

Darshan Goals

- Capture application-level behavior
  - Both POSIX and MPI-IO
  - Portable across file systems and hardware
- Transparent to users
  - Negligible performance impact
  - No source code changes
- Leadership-class scalability
  - 100,000+ processes

- Scalability tactics:
  - Bounded memory footprint
  - Minimize redundant information
  - Avoid shared resources at run time
  - Scalable algorithms to aggregate information

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

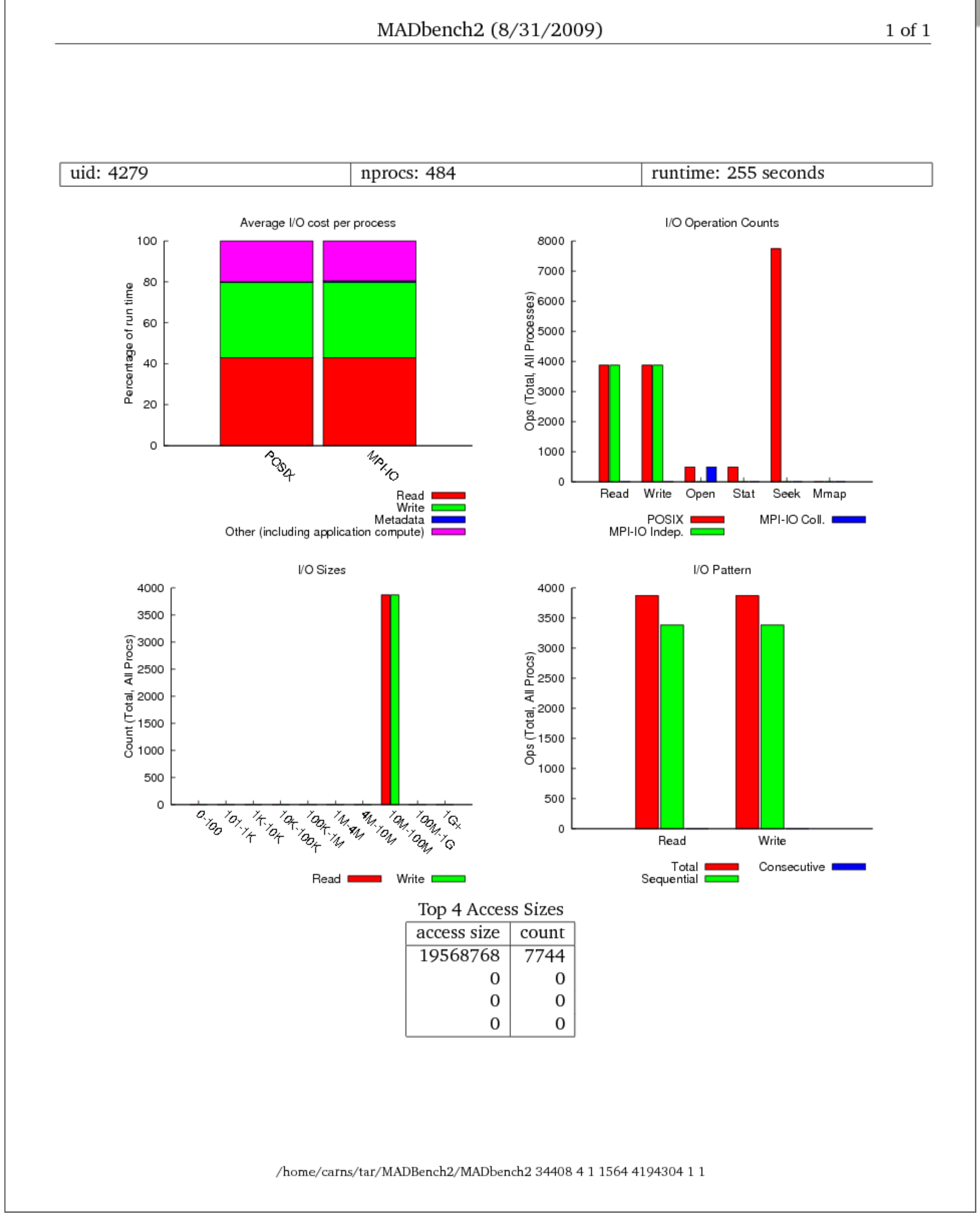
Example Statistics (per file)

- Counters:
  - POSIX open, read, write, seek, stat, etc.
  - MPI-IO nonblocking, collective, independent, 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.

Job Summary

- Job summary tool shows characteristics "at a glance"
- MADBench2 example
- 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



MADbench2 (8/31/2009) 1 of 1

uid: 4279	nprocs: 484	runtime: 255 seconds

Top 4 Access Sizes

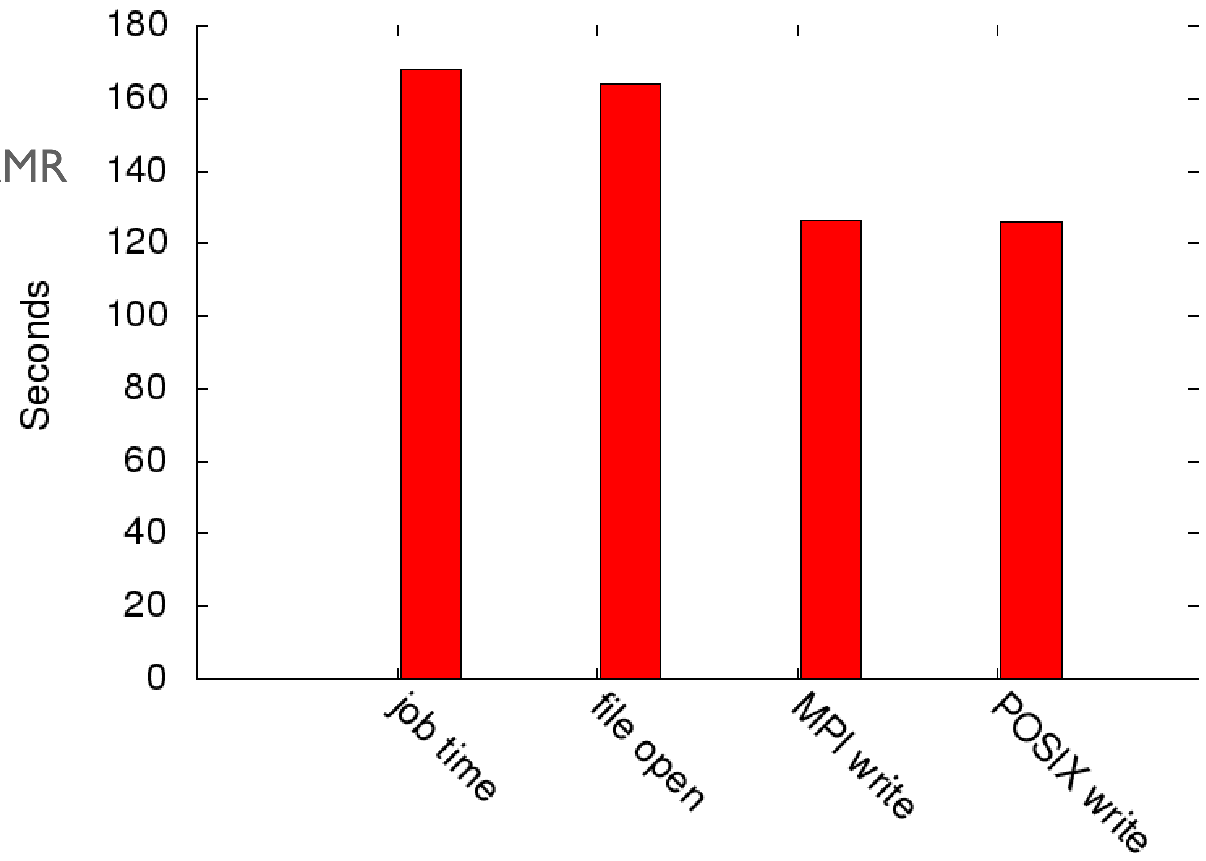
access size	count
19568768	7744
0	0
0	0
0	0

/home/carns/tar/MADBench2/MADbench2 34408 4 1 1564 4194304 1 1

Chombo I/O Benchmark

- Why does the I/O take so long in this case?
- Why isn't it busy writing data the whole time?

- Checkpoint writes from AMR framework
- Uses HDF5 for I/O
- Code base is complex
- 512 processes
- 18.24 GB output file

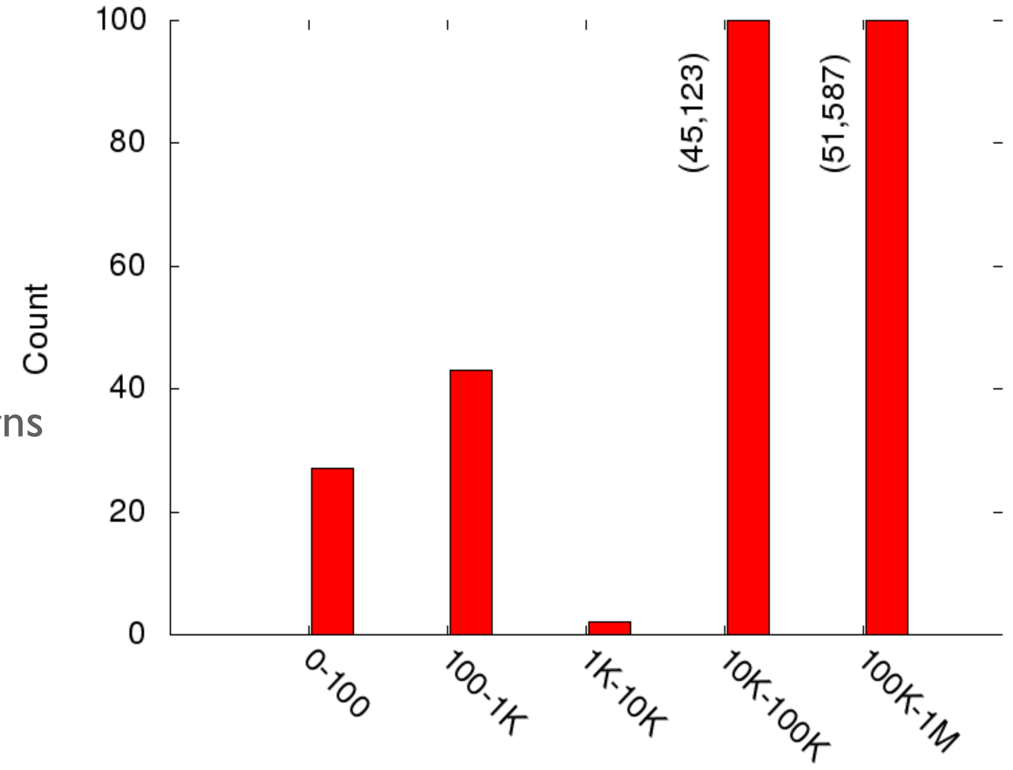


Chombo cumulative time per process

Chombo I/O Benchmark

- Many write operations, with none over 1 MB in size
- Most common access size is 28,800 (occurs 15622 times)
- No MPI datatypes or collectives
- All processes frequently seek forward between writes

- Consecutive: 49.25%
- Sequential: 99.98%
- Unaligned in file: 99.99%
- Several recurring regular stride patterns



Chombo write size histogram, 512 procs

Darshan Summary

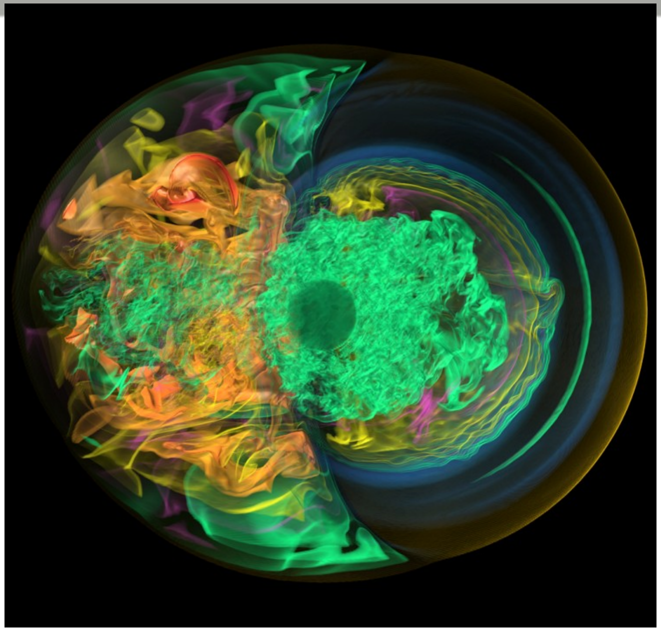
- Scalable tools like Darshan can yield useful insight
  - Identify characteristics that make applications successful
  - Identify problems to address through I/O research
- Petascale performance tools require special considerations
  - Target the problem domain carefully to minimize amount of data
  - Avoid shared resources
  - Use collectives where possible
- For more information:

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

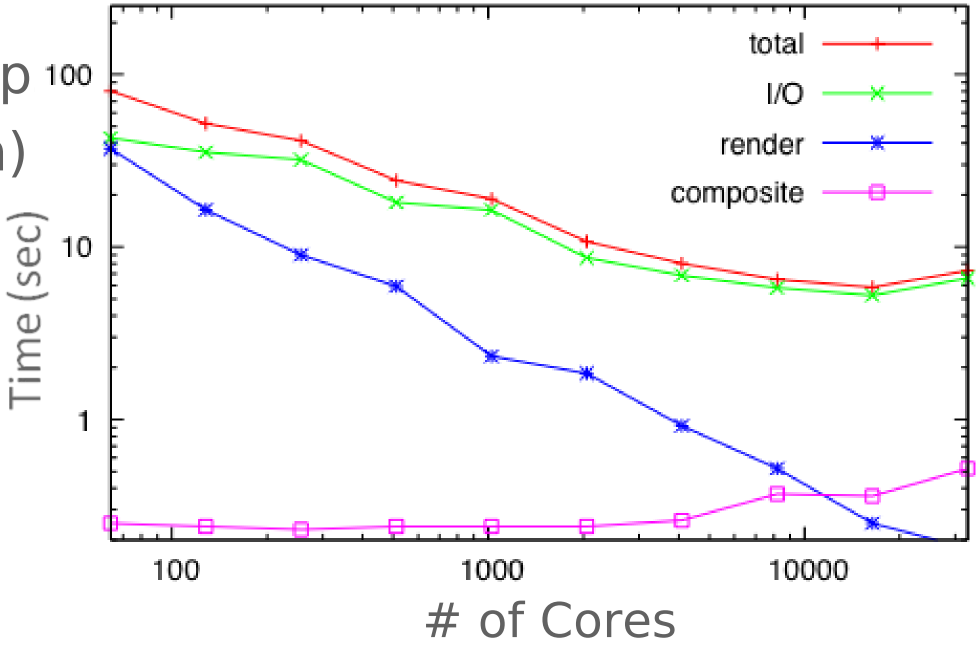
I/O in Parallel Volume Rendering

Thanks to Tom Peterka (ANL) and Hongfeng Yu and Kwan-Liu Ma (UC Davis) for providing the code on which this material is based.

Parallel Volume Rendering



- Supernova model with focus on core collapse
- Parallel rendering techniques scale to 16k cores on Argonne Blue Gene/P
- Produce a series of time step
- 1120^3 elements (~1.4 billion)
- Structured grid
- Simulated and rendered on multiple platforms, sites
- I/O time now largest component of runtime



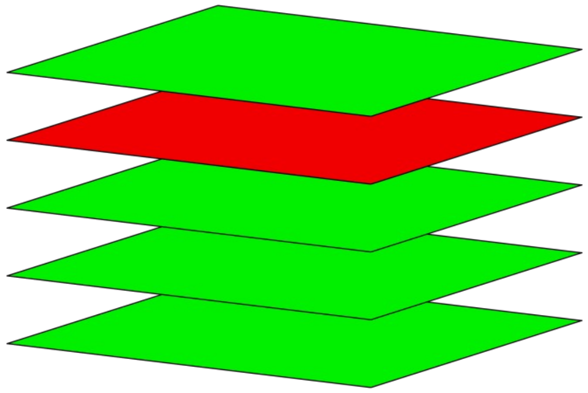
The I/O Code (essentially):

```
MPI_Init(&argc, &argv);
ncmpi_open(MPI_COMM_WORLD, argv[1], NC_NOWRITE,
   info, &ncid));
ncmpi_inq_varid(ncid, argv[2], &varid);
buffer =calloc(sizes[0]*sizes[1]*sizes[2],sizeof(float));
for (i=0; i<blocks; i++) {
   decompose(rank, nprocs, ndims, dims, starts, sizes);
   ncmpi_get_vara_float_all(ncid, varid,
        starts, sizes, buffer);
}
ncmpi_close(ncid));
MPI_Finalize();
```

- Read-only workload: no switch between define/data mode
- Omits error checking, full use of inquire (ncmpi_inq_*) routines
- Collective I/O of noncontiguous (in file) data
- “black box” decompose function:
  - divide 1120^3 elements into roughly equal mini-cubes
  - “face-wise” decomposition ideal for I/O access, but poor fit for volume rendering algorithms

Volume Rendering and pNetCDF

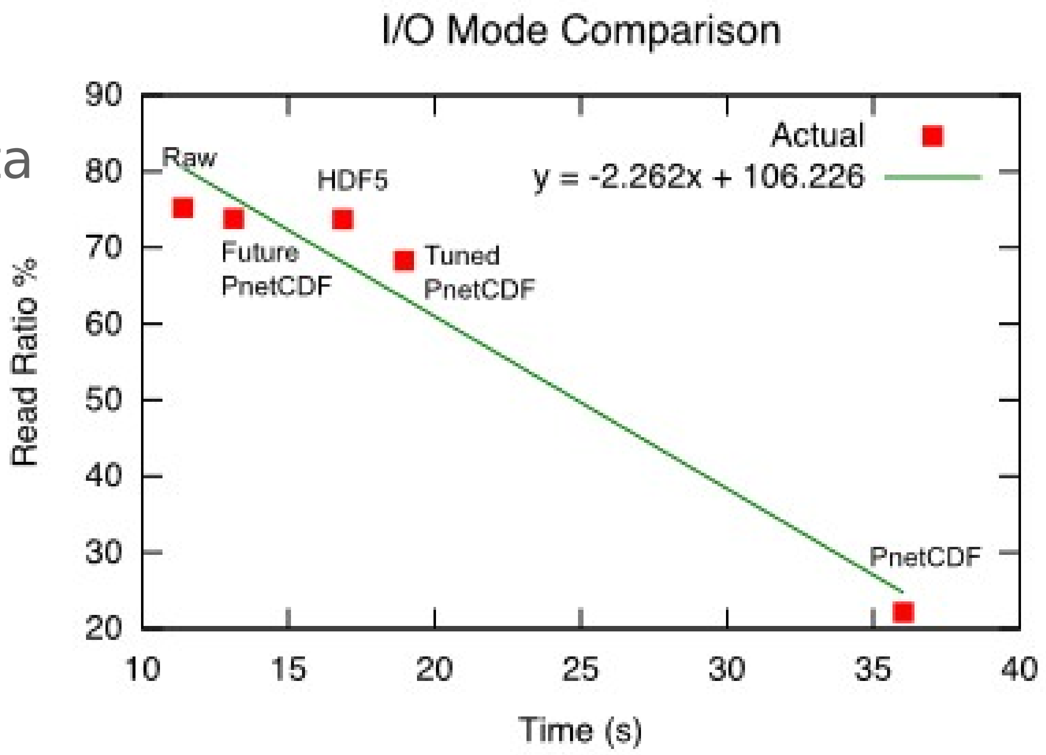
- Original data: netCDF formatted
- Two approaches for I/O
  - Pre-processing: extract each variable to separate file
    - Lengthy, duplicates data
  - Native: read data in parallel, on-demand from dataset
    - Skip preprocessing step but slower than raw
- Why so slow?
  - 5 large "record" variables in a single netcdf file
    - Interleaved on per-record basis
  - Bad interaction with default MPI-IO parameters



Record variable interleaving is performed in N-1 dimension slices, where N is the number of dimensions in the variable.

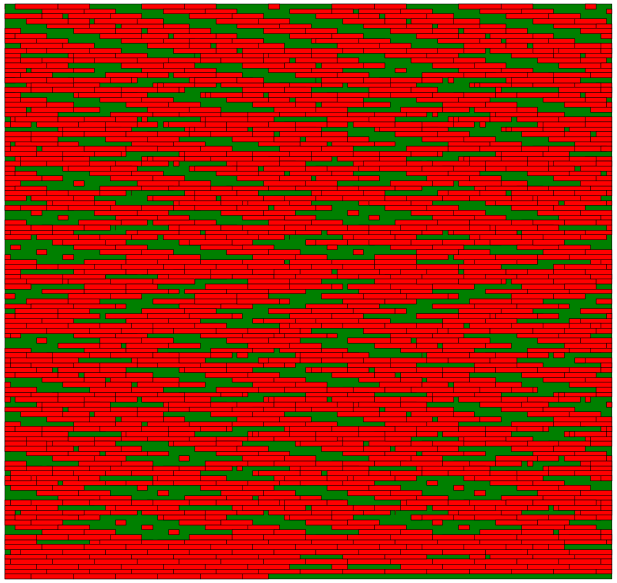
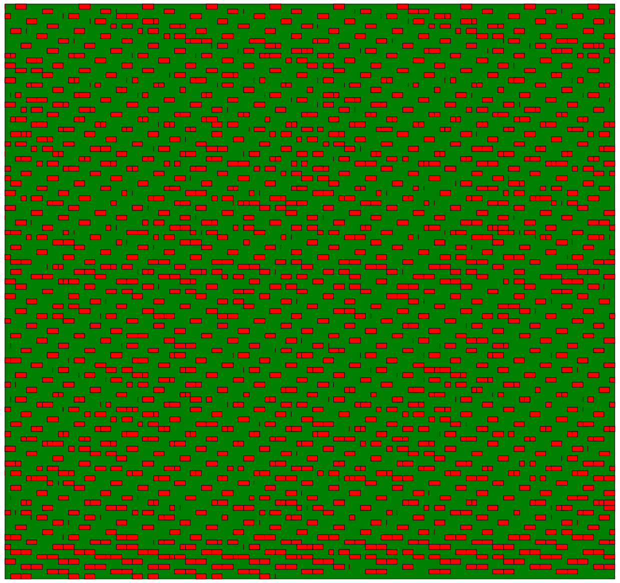
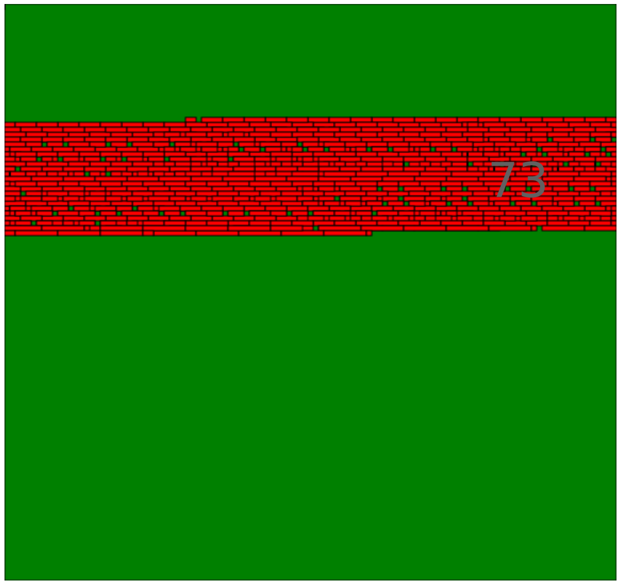
Access Method Comparison

- MPI-IO hints matter
- HDF5: many small metadata reads
- Interleaved record format: bad news



API	time (s)	accesses	read data (MB)	efficency
MPI (raw data)	11.388	960	7126	75.20%
PnetCDF (no hints)	36.030	1863	24200	22.15%
PnetCDF (hints)	18.946	2178	7848	68.29%
HDF5	16.862	23450	7270	73.72%
PnetCDF (beta)	13.128	923	7262	73.79%

File Access Three Ways

No hints: reading in way too much data

With tuning: no wasted data; file layout not ideal

HDF5 & new pnetcdf: no wasted data; larger request sizes

I/O in FLASH

- Worked with FLASH folks to identify ways to make I/O even better
- Change file format
  - Lump all FLASH variables (4-D) into one bigger (5-D) variable in file
  - Build up great big MPI datatype (or HDF5 hyperslab) describing all variables
  - Dump I/O with a single call
  - Good for I/O, but change in file format requires change in analysis and viz tools

- Try out parallel-netCDF non-blocking API
  - Not asynchronous.
  - Instead, combines operations at “wait”

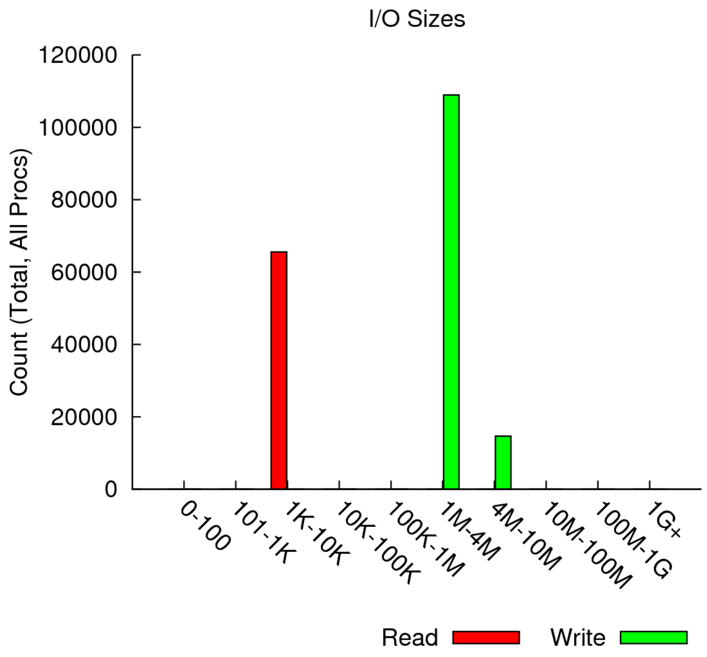
Parallel-netCDF Nonblocking API

```
data[0] = rank + 1000;
ncmpi_iput_vara_int_all(ncfile, varid1, &start, &count,
                        &(data[0]), count,&(requests[0]));

data[1] = rank + 10000;
/* Note: cannot touch buffer until wait completed */
ncmpi_iput_vara_int_all(ncfile, varid2, &start, &count,
                        &(data[1]), count,&(requests[1]));

ncmpi_wait_all(ncid, 2, requests statuses);
```

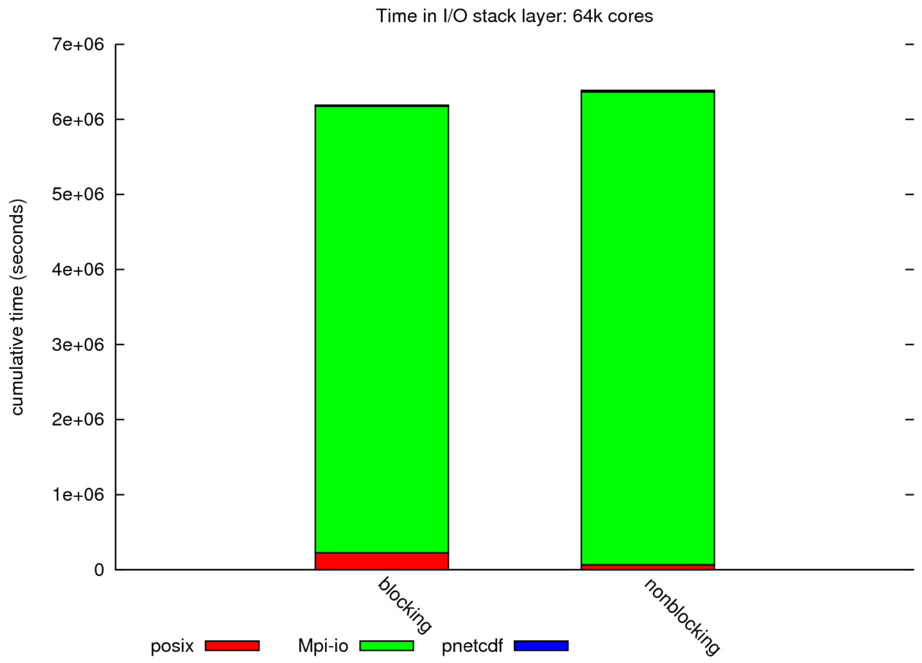
- Similar rules as MPI non-blocking routines
- All work happens in wait

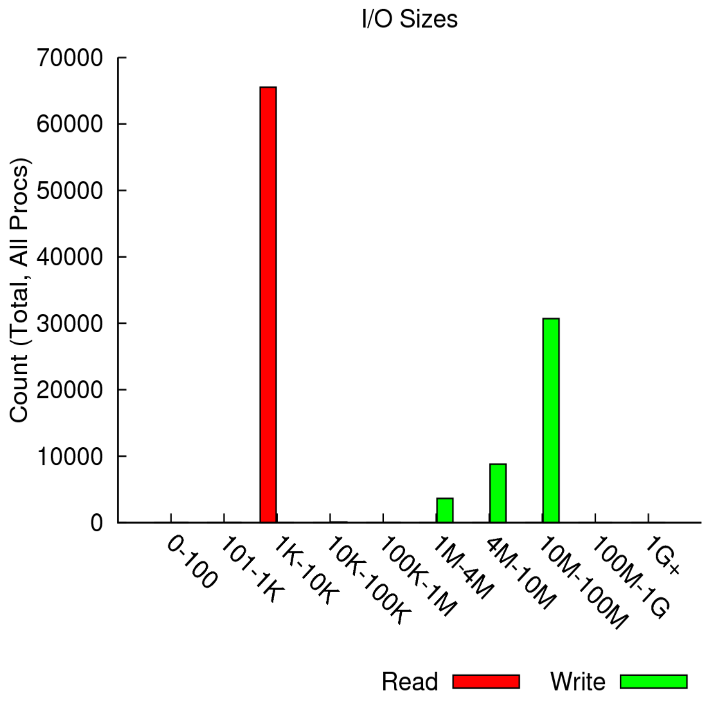
FLASH with traditional (blocking) API

- 64k cores
- 16k nodes
- 4 Intrepid (ANL BlueGene/P) racks
- 92 GB checkpoint file
  - 10 variables
  - Double precision
- 14 GB plotfile
  - 3 variables
  - Single precision
- Plotfile: 48.19 seconds (1.16 GB/sec)
- Checkpoint: 179.1 seconds (2.05 GB/sec)



FLASH with non-blocking API





- Better application performance despite more (total) time in MPI-IO
- Parallel-netcdf overhead is bupkis
- Fewer but larger operations
- Plotfile rate: 1.4 GB/sec; Checkpoint: 6.8

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 (1 of 4)

- 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://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/
- Darshan I/O Characterization Tool
  - http://www.mcs.anl.gov/research/projects/darshan

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://www.llnl.gov/icc/lc/siop/downloads/download.html
- 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

Acknowledgements

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 Rajeev Thakur (ANL) and Bill Loewe (Panasas) for their help in creating this material and presenting this tutorial in prior years.