



Argonne
NATIONAL
LABORATORY

... for a brighter future

Parallel I/O for High Performance and Scalability

or

It's Not *Your* Job (...it's mine)

Rob Latham

Mathematics and Computer Science Division

Argonne National Laboratory

robl@mcs.anl.gov



U.S. Department
of Energy

UChicago ►
Argonne_{LLC}

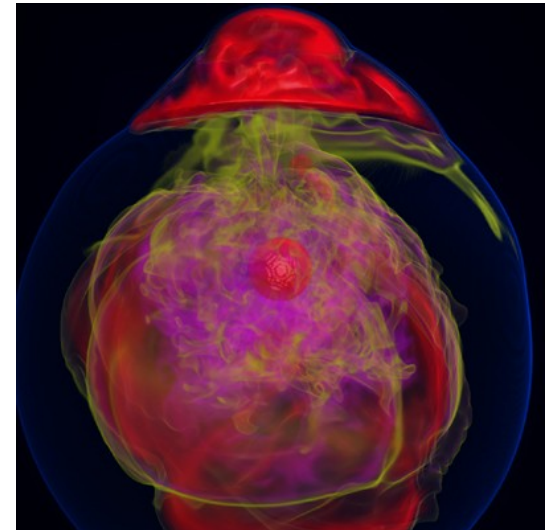


Computational Science

- Use of computer simulation as a tool for greater understanding of the real world
- Complements experimentation and theory
- As our simulations become ever more complicated
 - Large parallel machines needed to perform calculations
 - Leveraging parallelism becomes more important
- Managing code complexity bigger issue as well
 - Use of libraries increases (e.g. MPI, BLAS)
- Data access is a huge challenge
 - Using parallelism to obtain performance
 - Providing usable and efficient interfaces



IBM BG/L system.



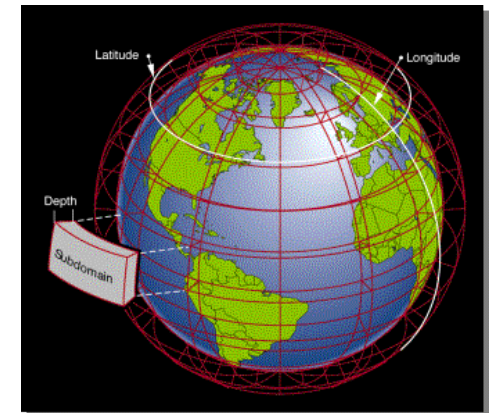
Visualization of entropy in Terascale Supernova Initiative application. Image from Kwan-Liu Ma's visualization team at UC Davis.

Outline

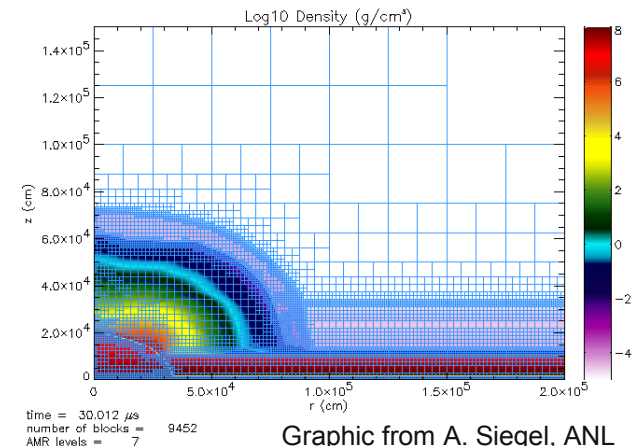
- Big Picture
 - Introduction
 - I/O software stacks
- Libraries (both low-level and high-level)
 - MPI-IO
 - Parallel netCDF
 - NetCDF-4
- Tying it all together
 - Tips, tricks, and general suggestions
 - Wrapping up

Application 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

Common Approaches to Application I/O

- Root performs I/O
 - Pro: trivially simple for “small” I/O
 - Con: bandwidth limited by rate one client can sustain
 - Con: may not have enough memory on root to hold all data
- All processes access their own file
 - Pro: no communication or coordination necessary between processes
 - Pro: avoids some file system quirks (e.g. false sharing)
 - Con: for large process counts, lots of files created
 - Con: data often must be post-processed to recreate canonical dataset
 - Con: uncoordinated I/O from all processes may swamp I/O system
- All processes access one file
 - Pro: only one file (per timestep etc.) to manage: fewer files overall
 - Pro: data can be stored in canonical representation, avoiding post-processing
 - Con: can uncover inefficiencies in file systems (e.g. false sharing)
 - Con: uncoordinated I/O from all processes may swamp I/O system

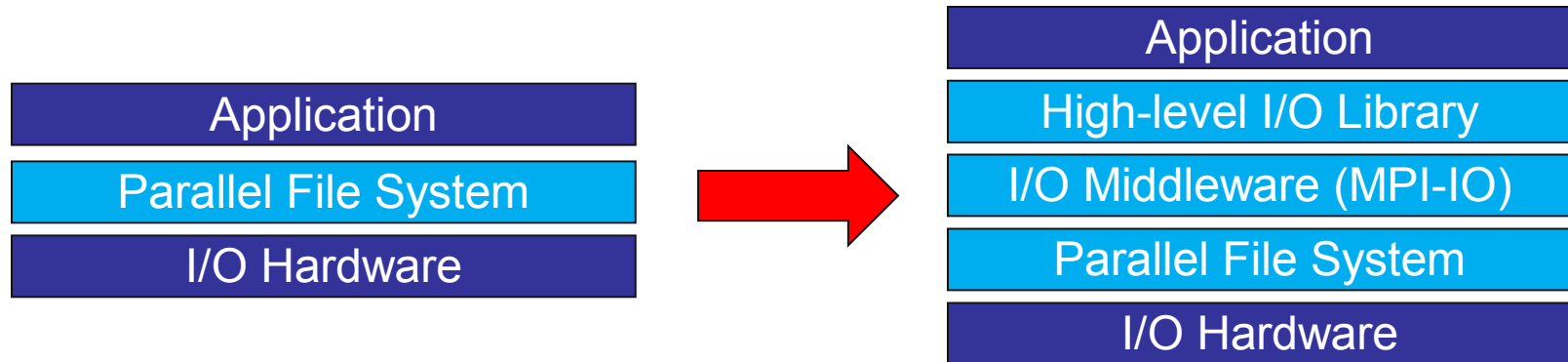
Challenges in Application I/O

- 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
- Avoiding file system quirks

- 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

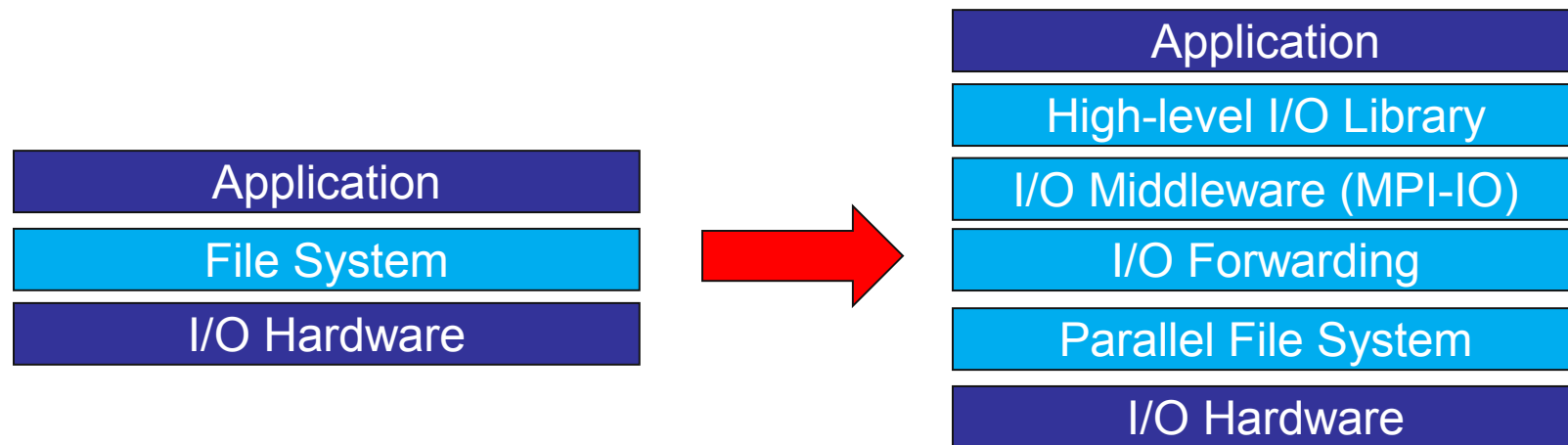
- Computer science teams that are experienced in parallel I/O have developed software to tackle all of these problems
 - **Not your job.**

I/O for Computational Science



- Applications require more software than just a parallel file system
- Break up support into multiple layers with distinct roles:
 - **Parallel file system** maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)
 - **Middleware layer** deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
 - **High level I/O library** maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF)

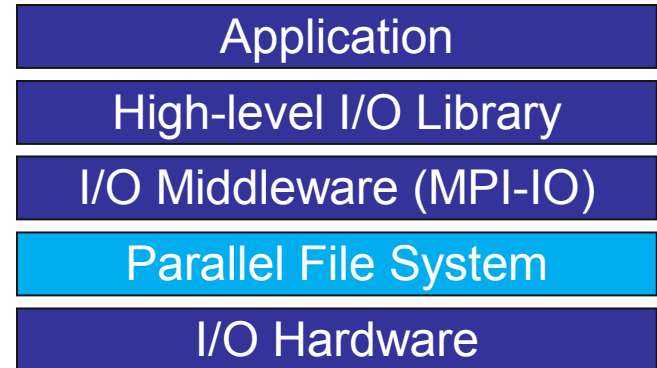
Software for Parallel I/O in HPC



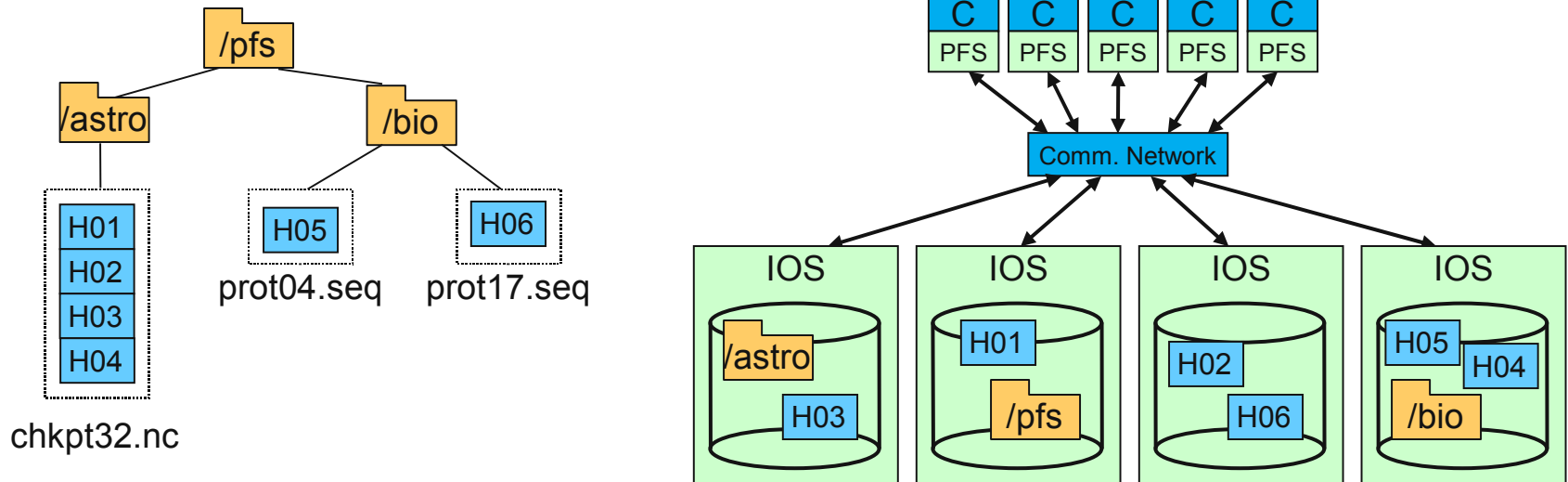
- Applications require more software than just a parallel file system
- Support provided via multiple layers with distinct roles:
 - **Parallel file system** maintains logical space, provides efficient access to data (e.g. PVFS, GPFS, Lustre)
 - **I/O Forwarding** found on largest systems to assist with I/O scalability
 - **Middleware** layer deals with organizing access by many processes (e.g. MPI-IO, UPC-IO)
 - **High level I/O library** maps app. abstractions to a structured, portable file format (e.g. HDF5, Parallel netCDF)
- Goals: scalability, parallelism (high bandwidth), and usability

Parallel File System

- Manage storage hardware
 - Present single view
 - Stripe files for performance
- In the context of 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*
 - Knowledge of collective I/O usually very limited



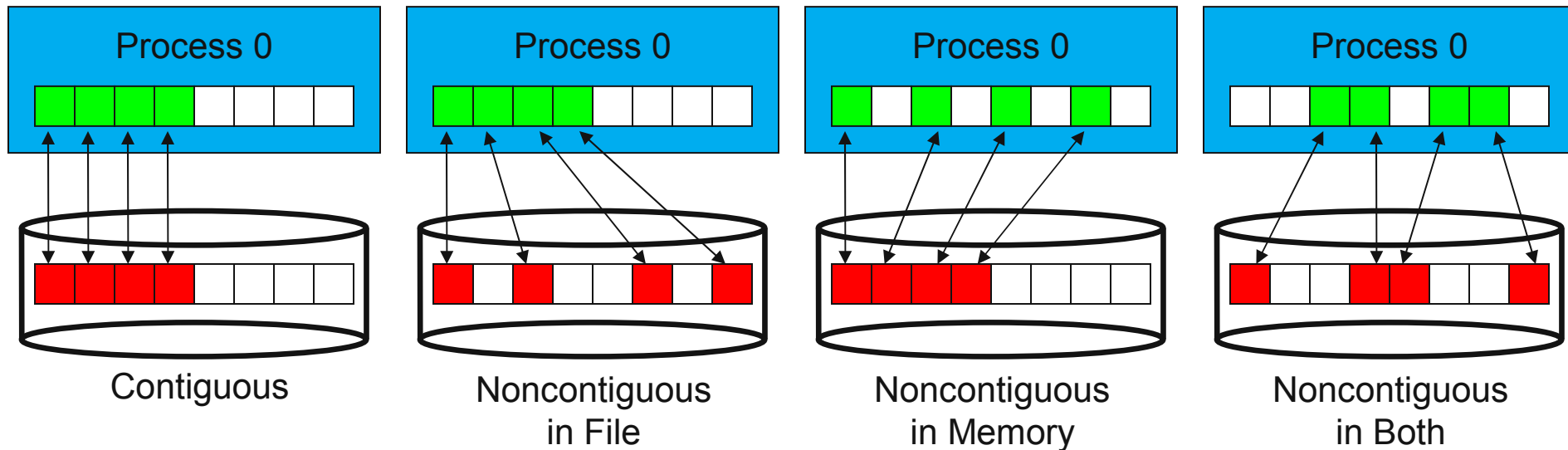
Parallel File Systems



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.

- Block-based or region-oriented accesses
- Stripe data across multiple resources
 - Simultaneous use of multiple servers, disks, and network links
- Tradeoffs between performance and consistency
 - POSIX: strict consistency hurts performance
 - NFS: consistency too weak: much time spent flushing buffers

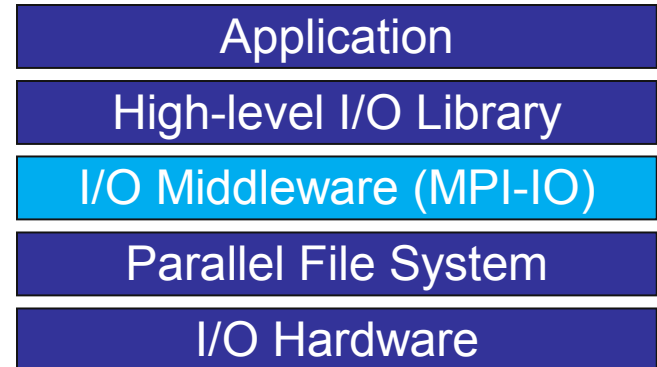
Contiguous and Noncontiguous I/O



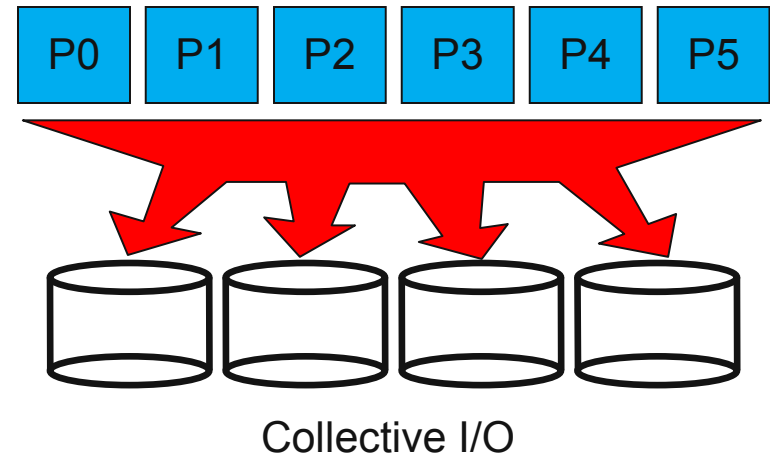
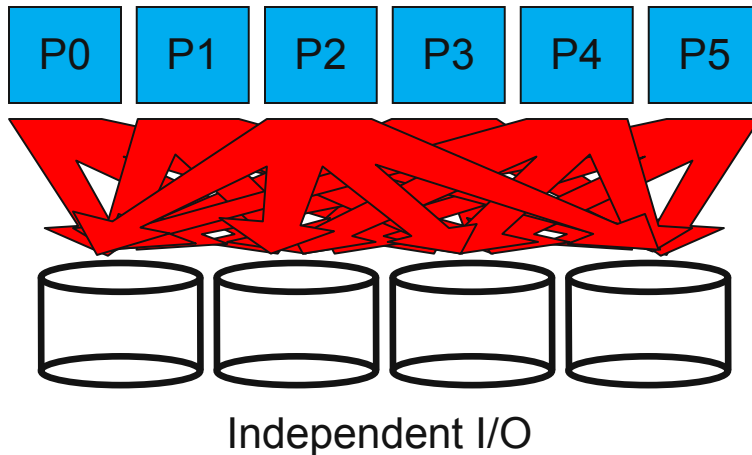
- **Contiguous I/O** moves data from a single memory block into a single file region
- **Noncontiguous I/O** has three forms:
 - Noncontiguous in memory, noncontiguous in file, or noncontiguous in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- **Describing noncontiguous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

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*

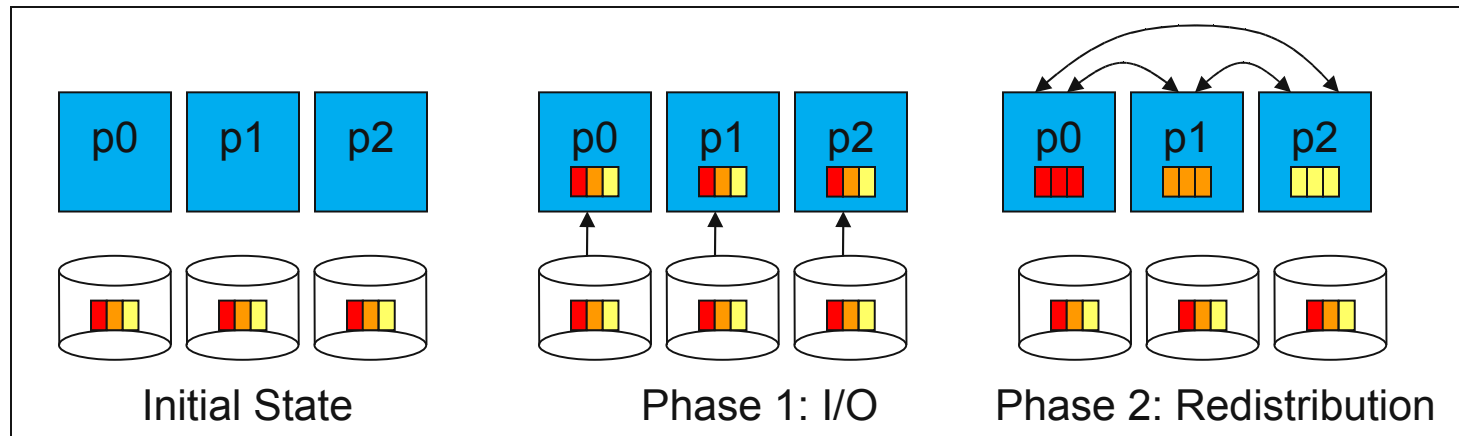


Independent and Collective I/O



- **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**

The Two-Phase I/O Optimization

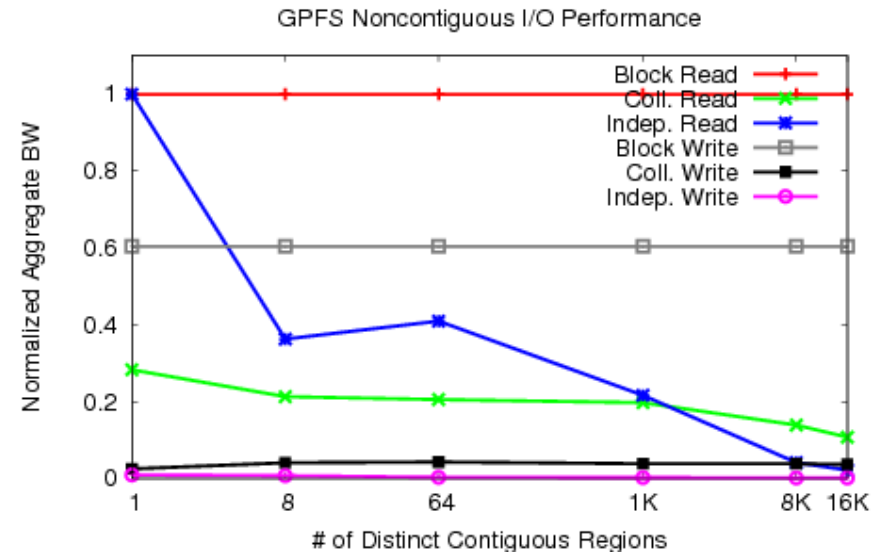
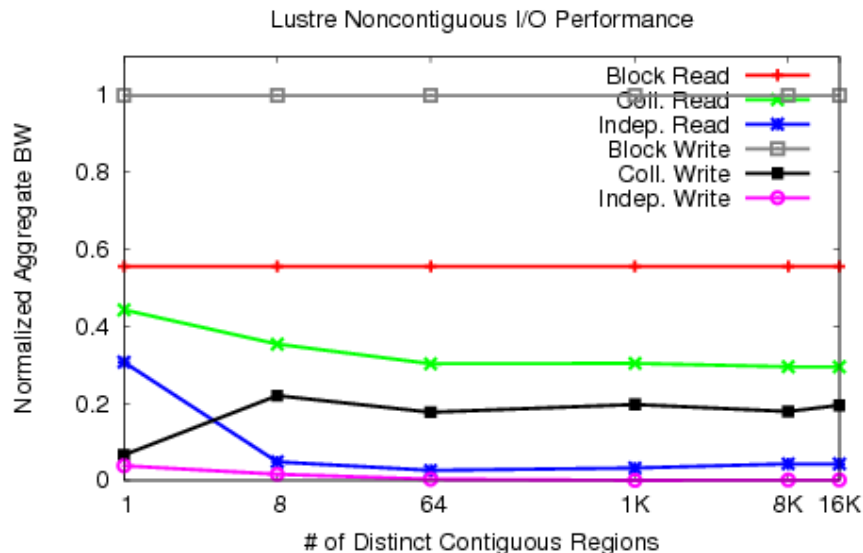


Two-Phase Read Algorithm

- 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
- Aggregating to fewer nodes as part of this process is trivial (and implemented!)

noncontig Collective I/O Results

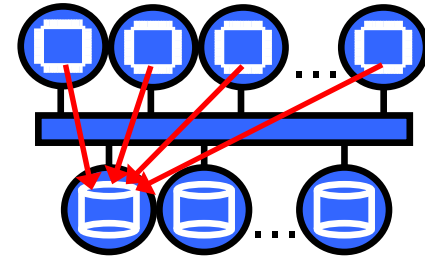
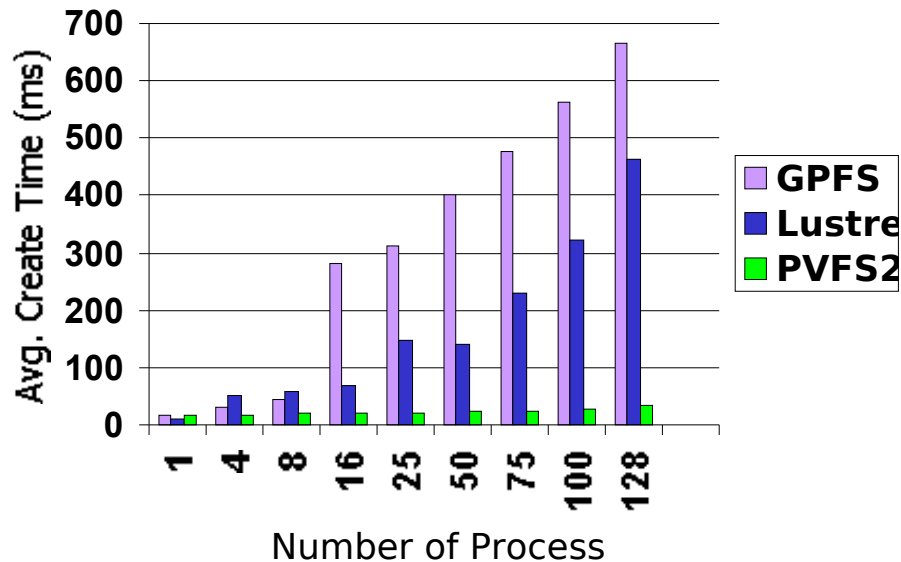
- Benchmark that tests file system performance with increasingly small contiguous regions (keeping total size same)
- All file systems benefit from collective I/O optimizations for all but the most contiguous patterns
 - Collective I/O optimizations can be absolutely critical to performance



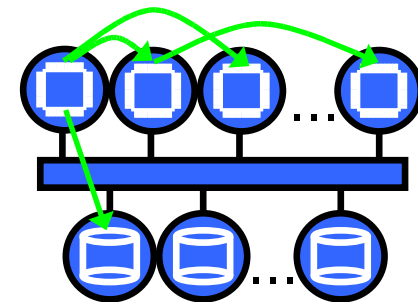
Creating Files Efficiently

- File create rates can actually have a significant performance impact
- Improving the file system interface improves performance for computational science
 - Leverage communication in MPI-IO layer

Time to Create Files Through MPI-IO



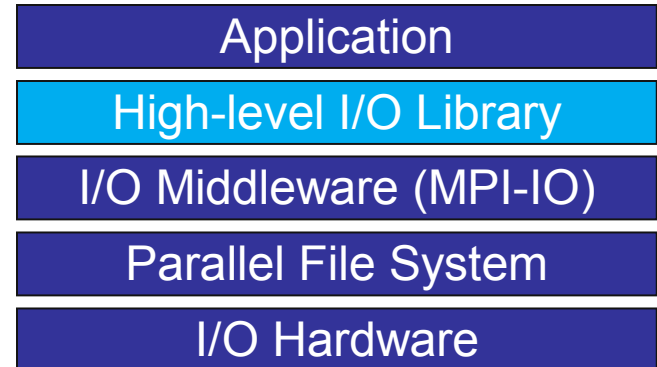
File system interfaces force all processes to open a file, causing a storm of system calls.



MPI-IO can leverage other interfaces, avoiding this behavior.

High Level Libraries

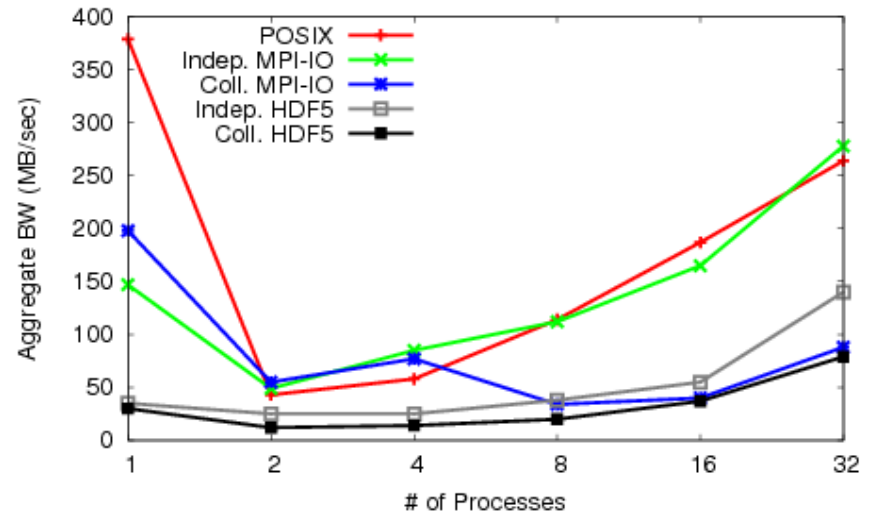
- Match storage abstraction to domain
 - Multidimensional datasets
 - Typed variables
 - Attributes
- Provide self-describing, structured files
- Map to middleware interface
 - Encourage collective I/O
- Implement optimizations that middleware cannot, such as
 - Caching attributes of variables
 - Chunking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



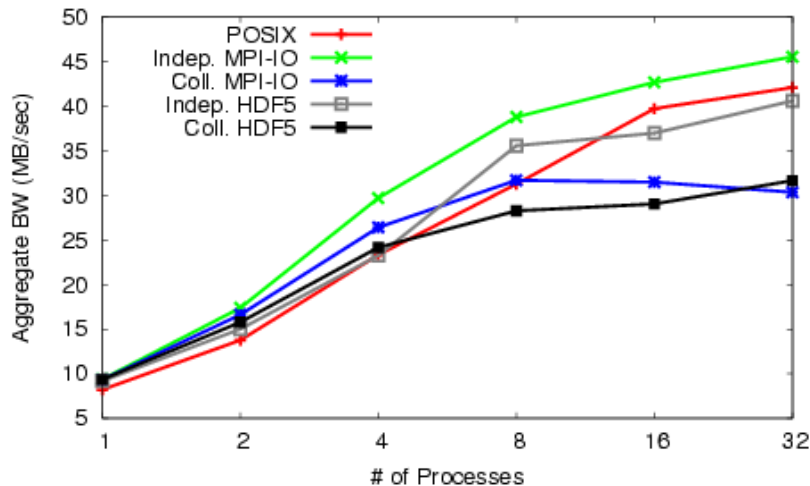
H5perf Write Results

- Performance of high-level I/O libraries can approach that of well-formed POSIX and MPI-IO, but doesn't always
 - Complexities of HLL storage formats can cause some performance degradation
 - Obviously developers are sensitive to this potential

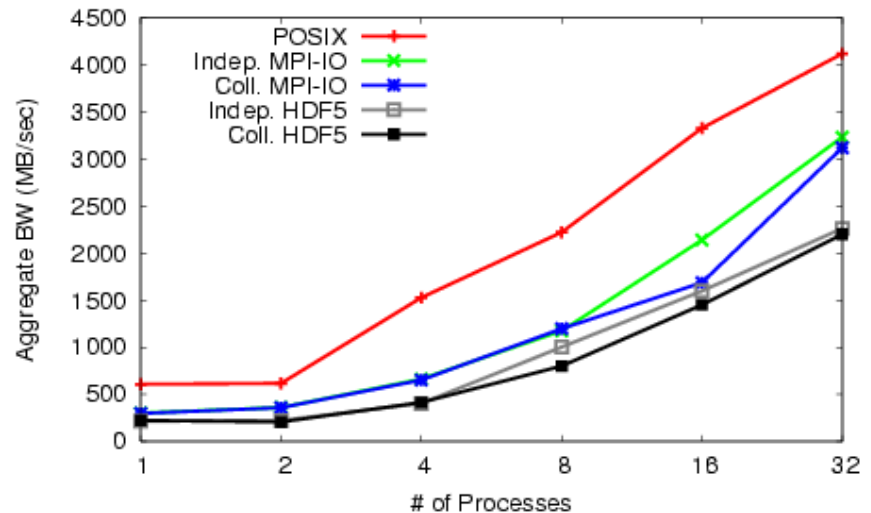
Lustre H5perf Write Performance



PVFS H5perf Write Performance



GPFS H5perf Write Performance



What we've said so far...

- Application scientists have basic goals for interacting with storage
 - Keep productivity high (meaningful interfaces)
 - Keep efficiency high (performant systems)
- 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

Why All This Software?

“All problems in computer science can be solved by another level of indirection.” -- David Wheeler

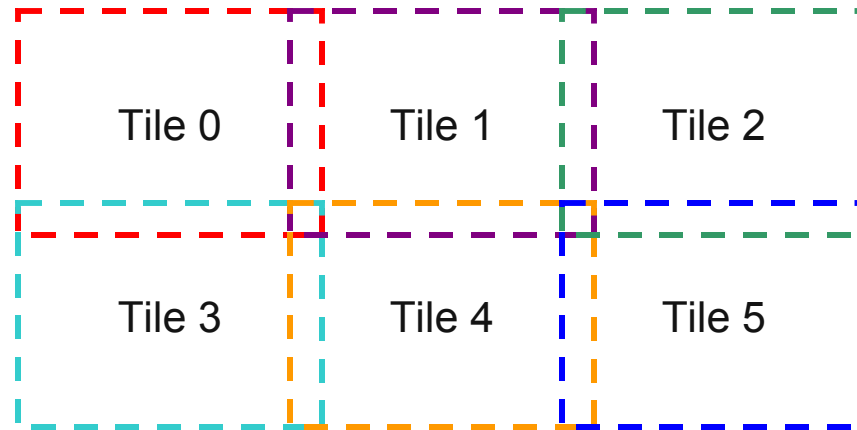
- Parallel file systems must be general purpose to be viable products
 - Many workloads for parallel file systems still include serial codes
 - Most of our tools still operate on the UNIX “byte stream” file model
- I/O forwarding addresses HW constraints and helps us leverage existing file system implementations at greater (unintended?) scales
- Programming model developers are not (usually) file system experts
 - Implementing programming model optimizations on top of common file system APIs provides flexibility to move to new file systems
 - Again, trying to stay as general purpose as possible
- High level I/O libraries mainly provide convenience functionality on top of existing APIs
 - Specifically attempting to cater to specific data models
 - Enable code sharing between applications with similar models
 - Standardize how **contents** of files are stored

MPI-IO Interface

MPI-IO

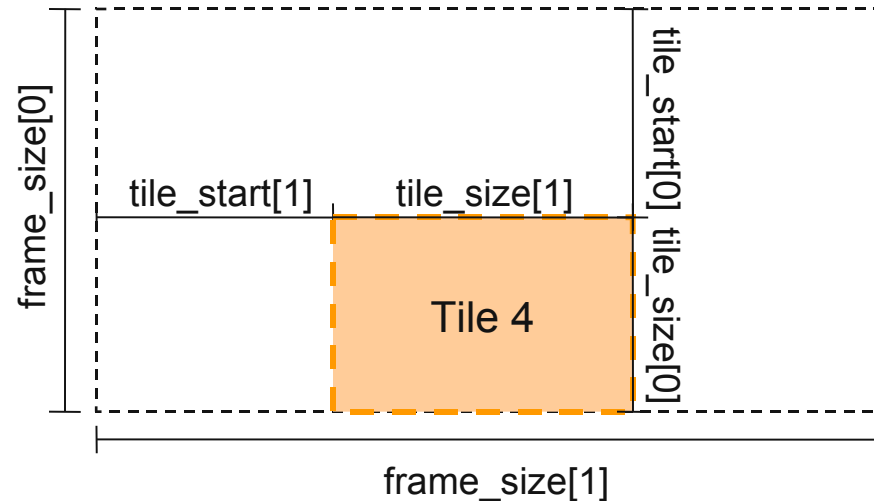
- I/O interface **specification** for use in MPI apps
- Data Model:
 - Stream of bytes in a file
 - Portable data format (external32)
 - *Not self-describing - just a well-defined encoding of types*
- Features:
 - Collective I/O
 - Noncontiguous I/O with MPI datatypes and file views
 - Nonblocking I/O
 - Fortran bindings (and additional languages)
- Implementations available on most platforms

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;
```

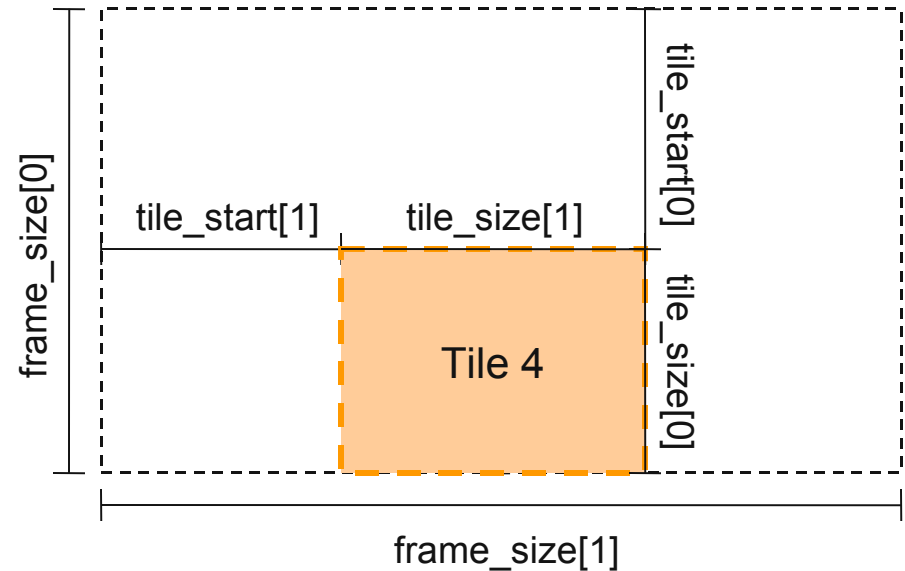
```
tile_size[0] = 768;
```

```
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!

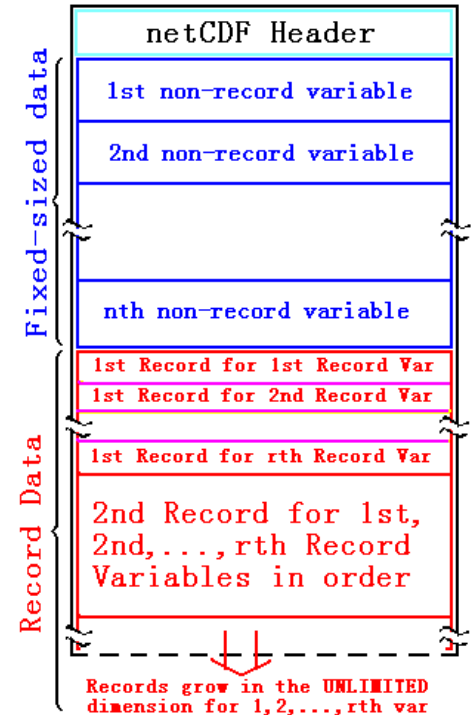
PnetCDF Interface and File Format

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

netCDF/PnetCDF Files

- PnetCDF files consist of three regions
 - Header
 - Non-record variables (all dimensions specified)
 - Record variables (ones with an unlimited dimension)
- Record variables are interleaved on a per-record basis
 - using more than one in a file is likely to result in worse performance due to noncontiguous accesses
- Data is always written in a big-endian format

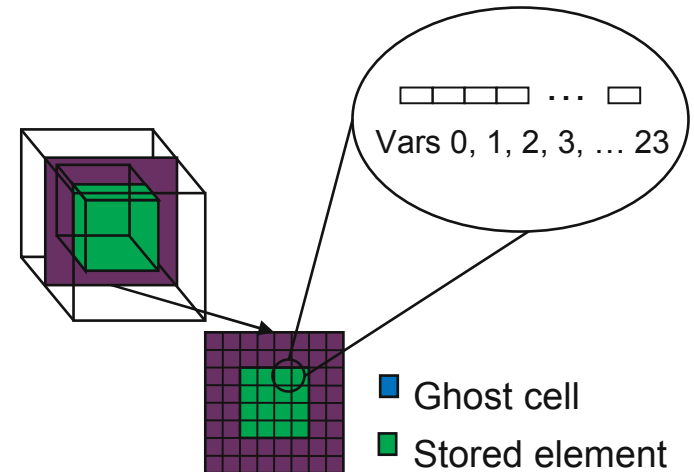
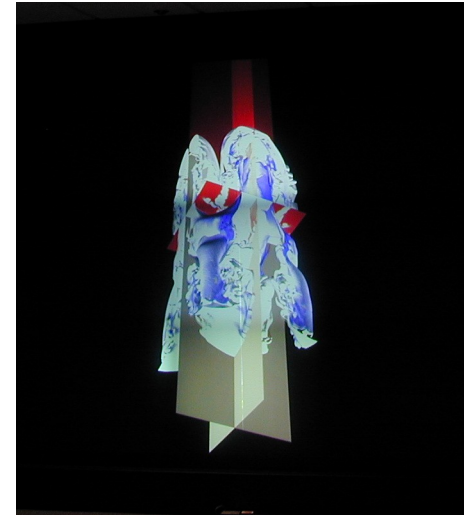


Storing Data in PnetCDF

- 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 1D 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 < NVAR; 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
- Some restrictions on variable size
 - Work under way to eliminate those
 - Viable workarounds exist in meantime.
- If PnetCDF meets application needs, it is likely to give good performance
 - Type conversion to portable format does add overhead
 - Tuning required to efficiently deal with record variables.

The netCDF-4 Effort

Thanks to Quincey Koziol (HDF group), Russ Rew (UCAR), and Ed Hartnett (UCAR) for helping ensure the accuracy of this material.

netCDF-4

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

Prepare Dataset (1/3)

```
#include <mpi.h>
#include <netcdf.h>

int main(int argc, char **argv) {
    int ret, ncfiler, nprocs, rank, dimid, varid, ndims=1;
    size_t start, count=1;
    char buf[13] = "Hello World\n";

    MPI_Init(&argc, &argv);

    ret = nc_create_par("demo", NC_MPIIO|NC_NETCDF4,
        MPI_COMM_WORLD, MPI_INFO_NULL, &ncfile);
```

Define mode (2/3)

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

ret = nc_def_dim(ncfile, "d1", nprocs, &dimid);
ret = nc_def_var(ncfile, "v1", NC_INT, ndims, &dimid, &varid);
ret = nc_put_att_text(ncfile, NC_GLOBAL, "string", 13, buf);
ret = nc_enddef(ncfile);

/* all this is nearly identical to serial NetCDF */
```

Write Variables (3/3)

```
/* demonstrative: collective access is the default */
ret = nc_var_par_access(ncfile, varid, NC_COLLECTIVE);
start = rank;
ret = nc_put_vara_int(ncfile, varid, &start, &count, &rank);
ret = nc_close(ncfile);
MPI_Finalize();
return 0;

/* - Very similar to serial NetCDF, but note no "send to master"
step
- Like pnetcdf, decompose variable access into distinct
regions*/
```

Opening dataset, reading metadata (1/2)

```
#include <mpi.h>
#include <netcdf.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int ret, ncfile, rank, varid, value;
    size_t start, count=1;
    char buf[13];

    MPI_Init(&argc, &argv);

    ret = nc_open_par("demo", NC_MPIIO|NC_NETCDF4,
        MPI_COMM_WORLD, MPI_INFO_NULL, &ncfile);
    ret=nc_get_att_text(ncfile, NC_GLOBAL, "string", buf);
    ret = nc_inq_varid(ncfile, "v1", &varid);
```

Read Variable (2/2)

```
/* demonstrative: collective access is the default */
ret = nc_var_par_access(ncfile, varid, NC_COLLECTIVE);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
start = rank;
ret=nc_get_vara_int(ncfile, varid, &start, &count, &value);
printf("rank: %d variable: %d att: %s", rank, value, buf);

ret = nc_close(ncfile);
MPI_Finalize();

return 0;
}
/* small amount of data in this example: still parallel I/O */
```

Compiling and Running

```
$ /usr/local/hdf5-1.8.0-beta3/bin/h5pcc -Wall -g \  
-I/usr/local/netcdf-4.0-beta1/include netcdf4-write.c -o netcdf4-write \  
\   
-L/usr/local/netcdf-4.0-beta1/lib -lnetcdf  
$ ## (reader similar. omitted for space)  
$ /usr/local/mpich2/bin/mpixec -np 4 ./netcdf4-write  
$ /usr/local/mpich2/bin/mpixec -np 4 ./netcdf4-read  
... ## (output omitted for space)  
$ /usr/local/netcdf-4.0-beta1/bin/ncdump demo.nc  
netcdf demo.nc {  
dimensions:  
    d1 = 4 ;  
variables:  
    int v1(d1) ;  
// global attributes:  
    :string = "Hello World\n",  
    "" ;  
data:  
    v1 = 0, 1, 2, 3 ;  
}
```

netCDF-4 wrapup

- No official release yet
 - Current API likely to be final one.
 - In beta release now.
- Similarities to both HDF5 and Parallel-netCDF
 - HDF5: additional routine to toggle collective vs. independent
 - PnetCDF: takes MPI_Comm and MPI_Info as part of open/create calls
 - HDF5 tools understand netCDF-4 datasets
- More information:
 - <http://www.unidata.ucar.edu/software/netcdf/netcdf-4/>
 - Muqun Yang, “Performance Study of Parallel NetCDF4 in ROMS.”, NCSA HDF group, June 30th, 2006

I/O Best Practices

How do I choose an API?

- Your programming model will limit choices
 - Domain might too
 - e.g. Climate community has substantial existing netCDF data and tools to manipulate that data
- Find something that matches your data model
- Avoid APIs with lots of features you won't use
 - Potential for overhead costing performance is high
- Maybe the right API isn't available?
 - Get I/O people interested, consider designing a new library

Summary of API Capabilities

	POSIX	MPI-IO	PNetCDF	HDF5	NetCDF4
Noncontig Memory	Yes	Yes	Yes	Yes	Yes
Noncontig File	Limited	Yes	Yes	Yes	Yes
Collective I/O		Yes	Yes	Yes	Yes
Portable Format		Yes	Yes	Yes	Yes
Self Describing			Yes	Yes	Yes
Attributes			Yes	Yes	Yes
Chunking				Yes	Yes
Hierarchical File				Yes	Yes

Tuning Application I/O (1 of 2)

- Have realistic goals:
 - What is peak I/O rate?
 - What other testing has been done?
- Describe as much as possible to the I/O system:
 - Open with appropriate mode
 - Use collective calls when available
 - Describe data movement with fewest possible operations
- Match file organization to process partitioning if possible
 - Order dimensions so relatively large blocks are contiguous with respect to data decomposition

Tuning Application I/O (2 of 2)

- Know what you can control:
 - What I/O components are in use?
 - What hints are accepted?
- Consider system architecture as a whole:
 - Is storage network faster than communication network?
 - Do some nodes have better storage access than others?

Do's and Don'ts

- PFSs are not optimized for metadata, instead for moving data
 - Don't use 'ls -l' or 'du' on millions of files
 - ***Certainly not to check application progress!***
 - Use your own subdirectory to avoid contention with others
- Keep file creates, opens, and closes to a minimum
 - Open once, close once
 - Use shared files or at least a subset of tasks
- Aggregate writes – PFSs are not databases, they need large transfers (at least 64K)
 - Contiguous data patterns utilize prefetching and write-behind far better than noncontiguous patterns
 - Collective I/O can aggregate for you, transform accesses into contiguous ones
- Avoid overlapped write regions if file systems rely on locks
 - Attempt to use block-aligned data
- Check error codes!

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 transfer templates
- Allow the programmer to:
 - Explain more about the I/O pattern
 - Specify 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: Collective I/O

- **cb_buffer_size** - Controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O
- **cb_nodes** - Controls the maximum number of aggregators to be used
- **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_config_list** - Provides explicit control over aggregators (see ROMIO User's Guide)

MPI-IO Hints: FS-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”

```
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, reducing I/O to a smaller number of processors (aggregators):

```
MPI_Info info;  
MPI_File fh;  
MPI_Info_create(&info);  
MPI_Info_set(info, "cb_nodes", "16");  
MPI_info_set(info, "bgl_nodes_pset" 4);  
ncmpi_open(comm, filename, NC_NOWRITE, info, &ncfile);  
MPI_Info_free(&info);
```

Helping I/O Experts Help You

- Scenarios
 - Explaining logically what you are doing
 - Separate the conceptual structures from their representation on storage
 - Common vs. infrequent patterns
 - Possible consistency management simplifications
- Application I/O kernels
 - Simple codes exhibiting similar I/O behavior
 - Easier for I/O group to work with
 - **Useful for acceptance testing!**
 - Needs to be pretty close to the real thing...

Concluding Remarks

Wrapping Up

- Computer scientists have developed solutions to many common computational science I/O problems
 - In most cases, these solutions will lead to high efficiency with minimal effort
 - Knowing how these components work will lead you to better performance
- I/O systems will continue to get more complicated, but hopefully easier to use at the same time!
 - Remote access to data
 - More layers to I/O stack
 - Domain-specific application interfaces

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
- 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 3)

- netCDF
<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/HDF5>
<http://hdfgroup.org/HDF5/Tutor>

On-Line References (2 of 3)

- PVFS

<http://www.pvfs.org/>

- Lustre

<http://www.lustre.org/>

- GPFS

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

On-Line References (3 of 3)

- 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>

Acknowledgments

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 W-31-109-ENG-38.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.