Five orders of magnitude between compute and I/O capacity on Titan Cray system at ORNL

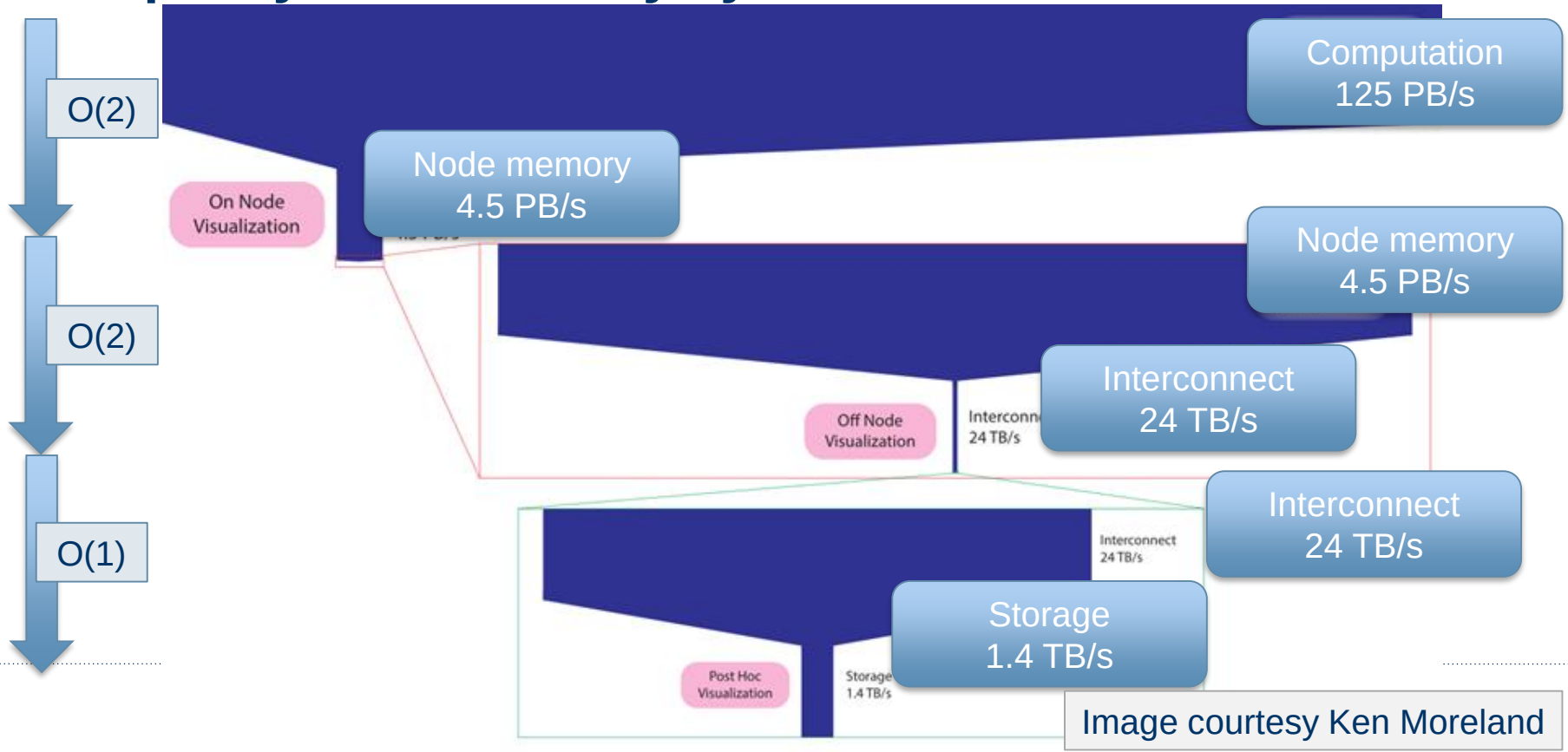


Image courtesy Ken Moreland

The problem is not going away

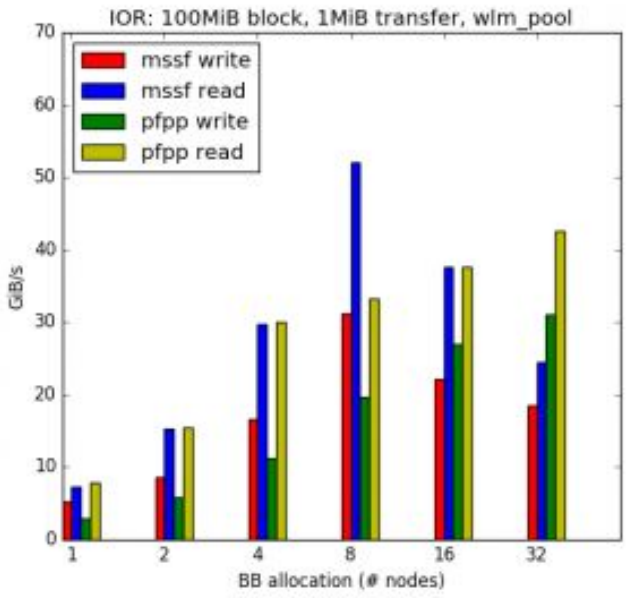
How does Summit compare to Titan

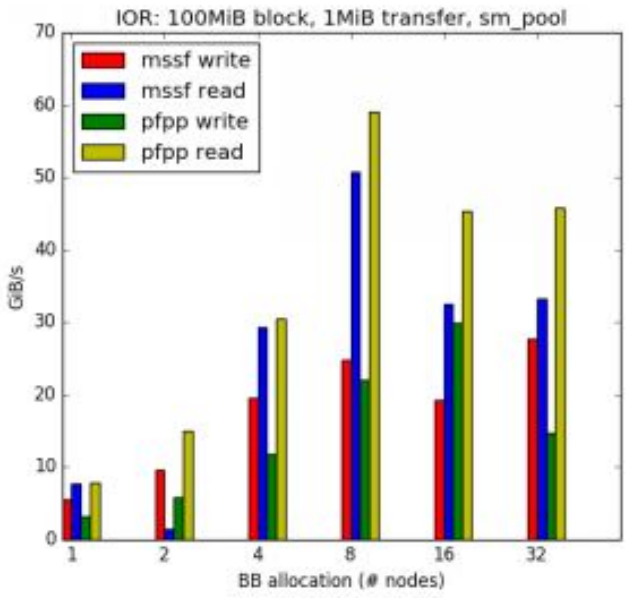
Feature	Summit	Titan
Application Performance	5-10x Titan	Baseline
Number of Nodes	~3,400	18,688
Node performance	> 40 TF	1.4 TF
Memory per Node	>512 GB (HBM + DDR4)	38GB (GDDR5+DDR3)
NVRAM per Node	800 GB	0
Node Interconnect	NVLink (5-12x PCIe 3)	PCIe 2
System Interconnect (node injection bandwidth)	Dual Rail EDR-IB (23 GB/s)	Gemini (6.4 GB/s)
Interconnect Topology	Non-blocking Fat Tree	3D Torus
Processors	IBM POWER9™ NVIDIA Volta™	AMD Opteron™ NVIDIA Kepler™
File System	120 PB, 1 TB/s, GPFS™	32 PB, 1 TB/s, Lustre®
Peak power consumption	10 MW	9 MW

Data courtesy A. Geist (ORNL)

Performance tips

- **Stripe your files across multiple BB servers**
  - To obtain good scaling, need to drive IO with sufficient compute - scale up # BB nodes with # compute nodes





جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Shaheen II Supercomputer

Compute	Node	Processor type: Intel Haswell	2 CPU sockets per node @2.3GHz 16 processor cores per CPU
		6174 nodes	197,568 cores
		128 GB of memory per node	Over 790 TB total memory
	Power	Up to 3.5MW	Water cooled
	Weight/Size	More than 100 metrics tons	36 XC40 Compute cabinets, disk, blowers, management nodes
	Speed	7.2 Peta FLOPS peak performance	5.53 Peta FLOPS sustained LINPACK and ranked 15th in the latest Top500 list
	Network	Cray Aries interconnect with Dragonfly topology	57% of the maximum global bandwidth between the 18 groups of two cabinets
Storage	Storage	Sonexion 2000 Lustre appliance	17.6 Peta Bytes of usable storage Over 500 GB/s bandwidth
	Burst Buffer	DataWarp	Intel Solid Sate Devices (SDD) fast data cache Over 1.5 TB/s bandwidth
	Archive	Tiered Adaptive Storage (TAS)	Hierarchical storage with 200 TB disk cache and 20 PB of tape storage, using a spectra logic tape library (Upgradable to 100 PB)

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Introduction to parallel I/O

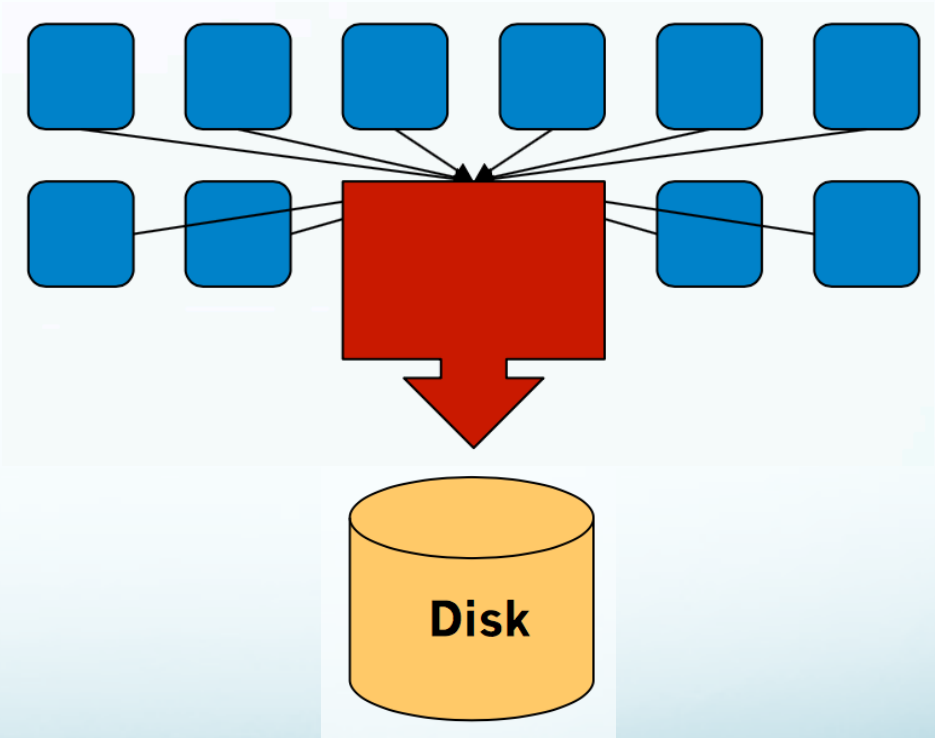
- I/O can create bottlenecks
  - I/O components are much slower than the compute parts of a supercomputer
  - If the bandwidth is saturated, larger scale of execution can not improve the I/O performance
- Parallel I/O is needed to
  - Do more science than waiting files to be read/written
  - No waste of resources
  - Not stressing the file system, thus affecting other users

I/O Performance

- There is no magic solution
- I/O performance depends on the pattern
- Of course a bottleneck can occur from any part of an application
- Increasing computation and decreasing I/O is a good solution but not always possible

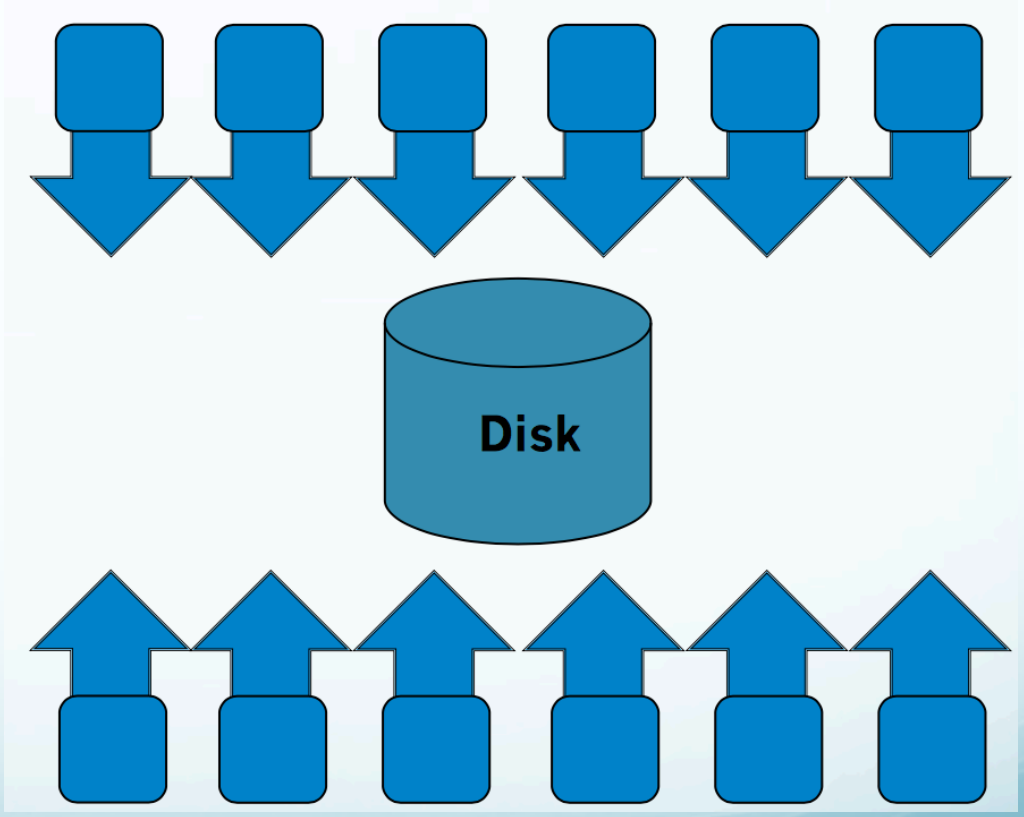
Serial I/O

- Only one process performs I/O (default option for WRF)
  - Data Aggregation or Duplication
  - Limited by single I/O process
- Simple solution but does not scale
- Time increases with amount of data and also with number of processes



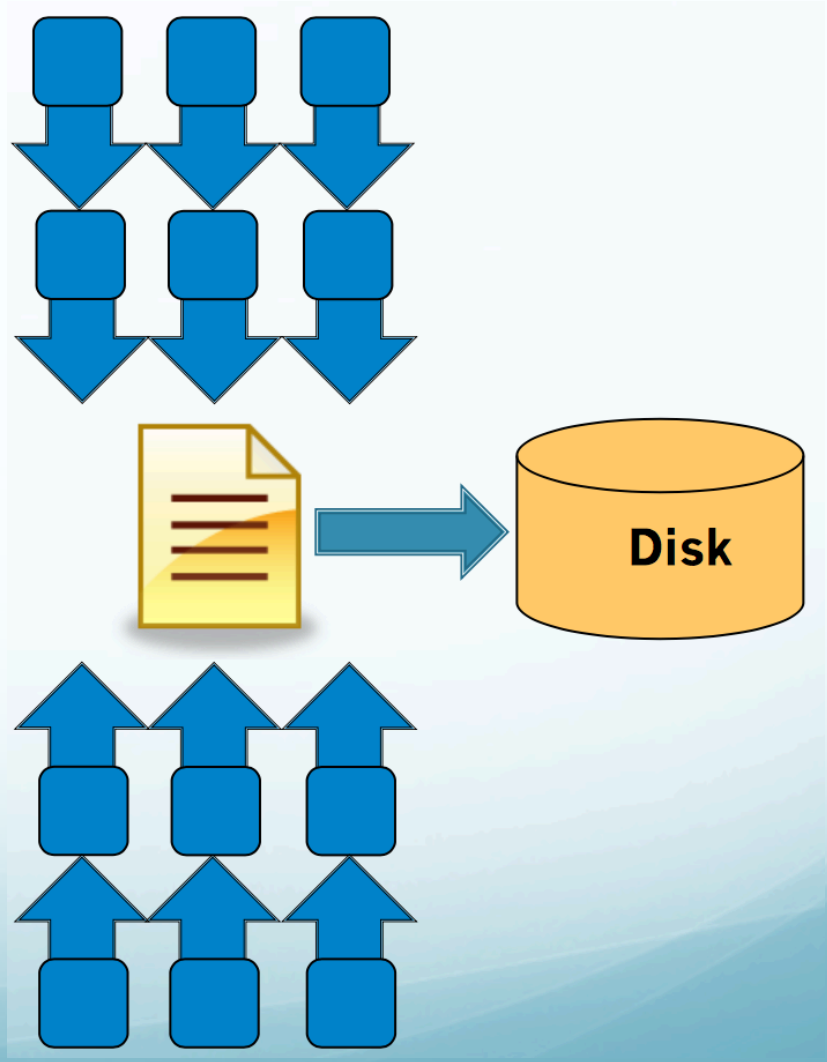
Parallel I/O: File-per-Process

- All processes read/write their own separate file
  - The number of the files can be limited by file system
  - Significant contention can be observed



Parallel I/O: Shared File

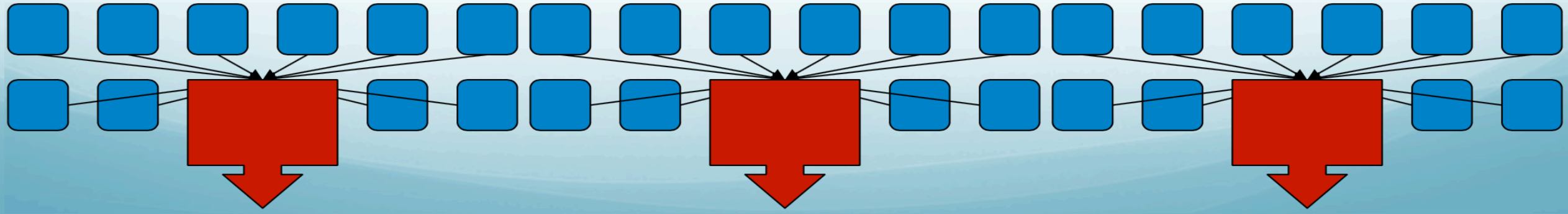
- Shared File
  - One file is accessed from all the processes
  - The performance depends on the data layout
  - Large number of processes can cause contention



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Pattern Combinations

- Subset of processes perform I/O
  - **Aggregation** of a group of processes data
  - I/O process may access independent files
  - Group of processes perform parallel I/O to a shared file



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 that can address all of these problems, when used appropriately.

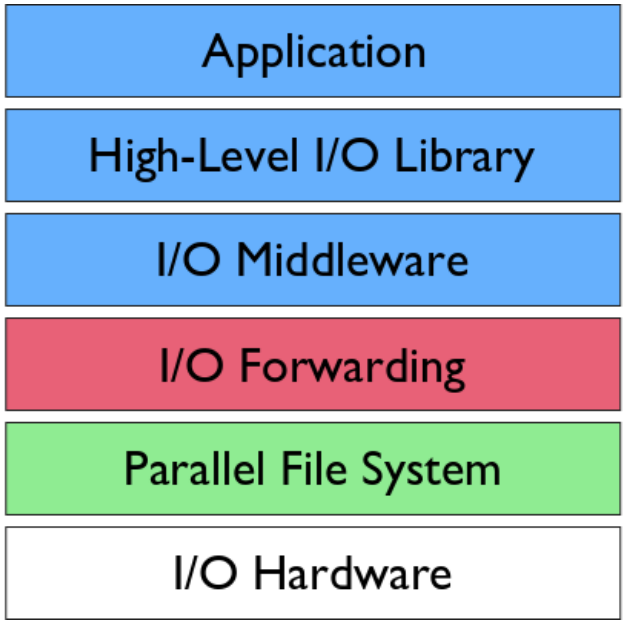
I/O for Computational Science

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, IOFSL, Cray DVS



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

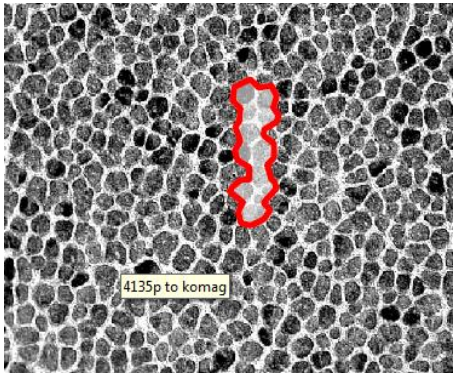
MPI-IO

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

PVFS, PanFS, GPFS, Lustre

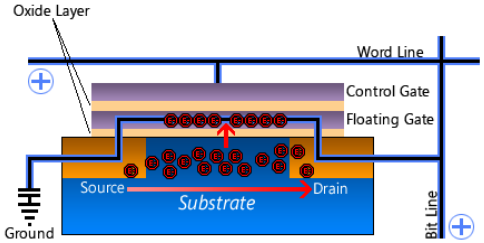
Additional I/O software provides improved performance and usability over directly accessing the parallel file system. Reduces or (ideally) eliminates need for optimization in application codes.

I/O Hardware



Magnetic or Solid State storage bits

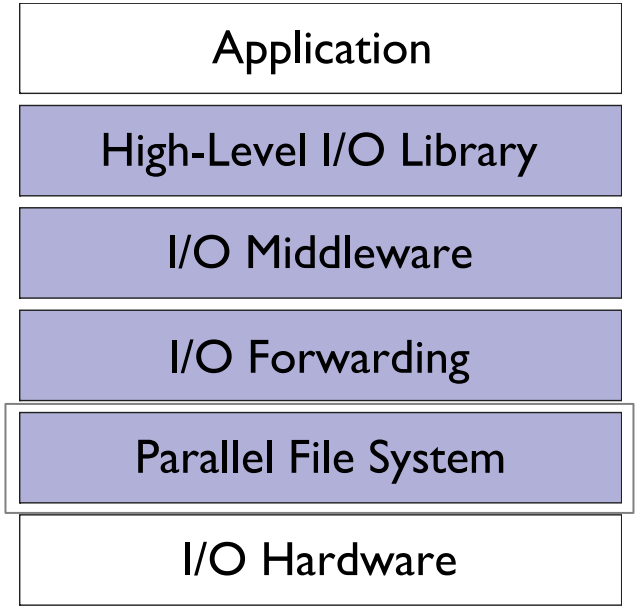
Storage Devices



Characteristics of Storage Devices affect performance, reliability, and system design

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

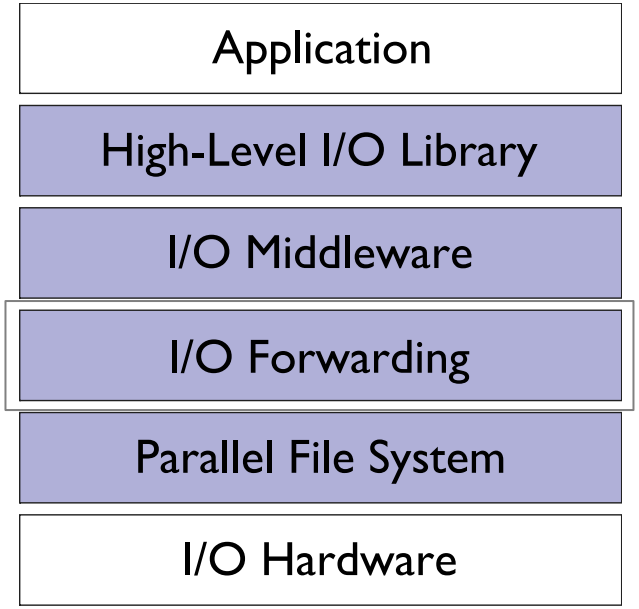
Parallel File System



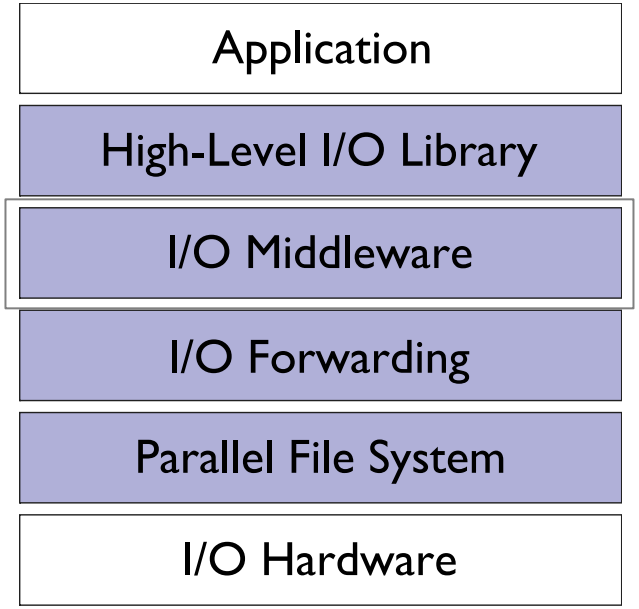
- Manage storage hardware
  - Present single view
  - Stripe files for performance
- In the I/O software stack
  - Focus on concurrent, independent access
  - Publish an interface that middleware can use effectively
    - Rich I/O language
    - Relaxed but sufficient semantics

I/O Forwarding

- 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
  - Also allows in-situ processing
- Poor implementations can lead to unnecessary serialization, hindering performance

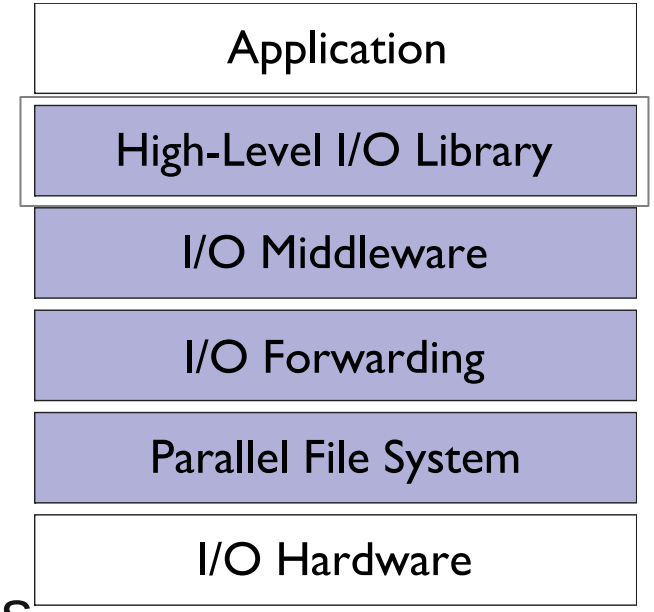


I/O Middleware



- Match the programming model (e.g. MPI)
- Facilitate concurrent access by groupsof 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

High Level Libraries



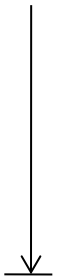
- Match storage abstraction to domain
  - Multidimensional datasets
  - Typed variables
  - Attributes
- Provide self-describing, structured files
- Map to middleware interface
  - Encourage collective I/O
- Implement optimizations that middleware cannot, such as
  - Caching attributes of variables
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

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

Capacity vs Bandwidth

- Areal density increases by 40% per year
  - Per drive capacity increases by 50% to 100% per year
  - 2008: **500 GB**
  - 2009: **1 TB**
  - 2010: **2 TB**
  - 2011: **3 TB**
  - 2012: **4 TB**
  - 2014: **6 TB**
  - 2015: **8 TB**
  - 2016, 2017**: 10 TB**

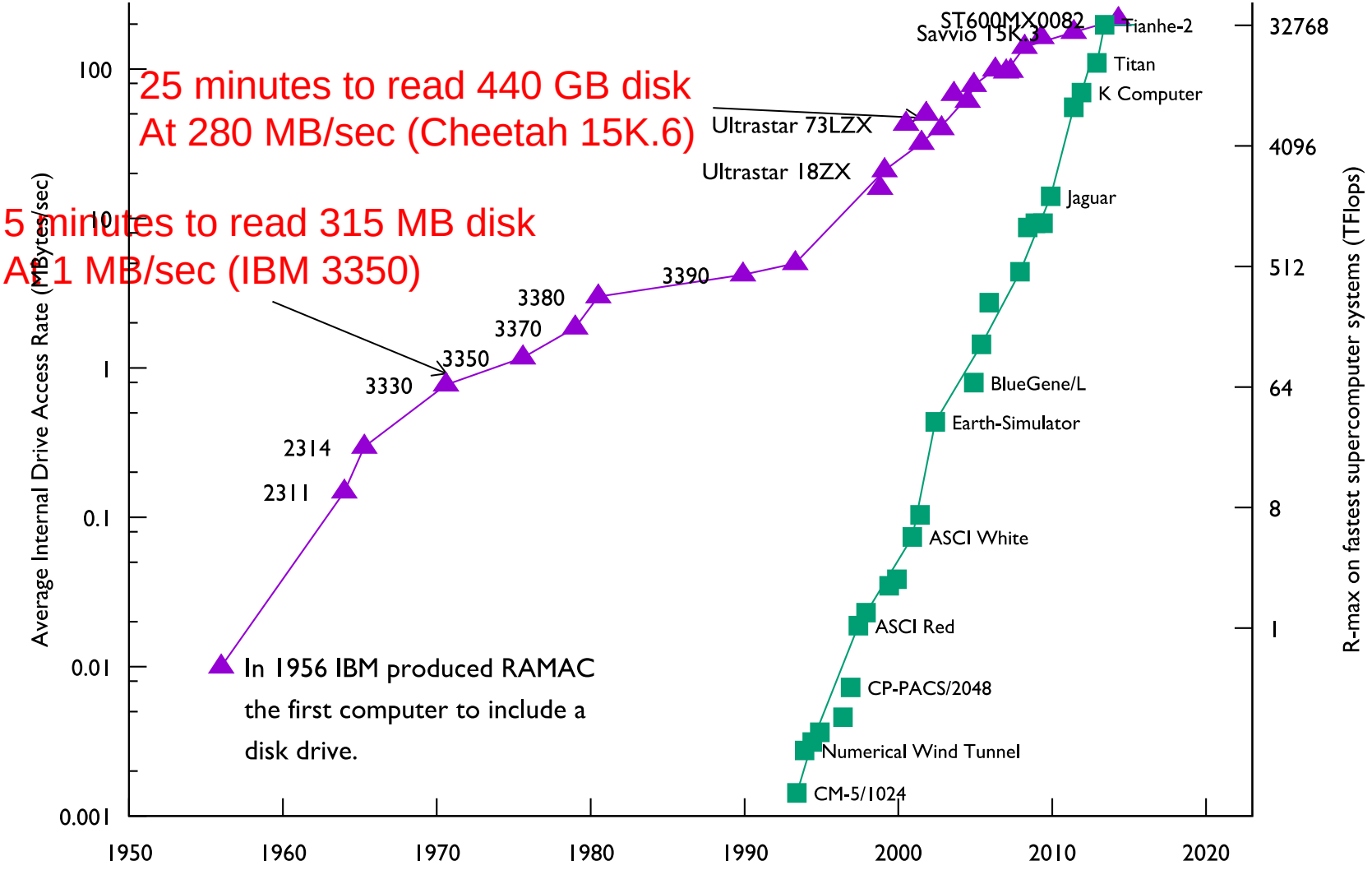


Takes longer and longer to completely read each new generation of drive

- Drive interface speed increases by 15-20% per year
  - 2008: 500 GB disk (WD RE2): **98 MB/sec**
  - 2009: 1 TB disk (WD RE3): **113 MB/sec** (+15%)
  - 2010: 2 TB disk (WD RE4): **138 MB/sec** (+22%)
  - 2013: 4 TB disk (WD SAS): **150 MB/sec** (+ 8%)

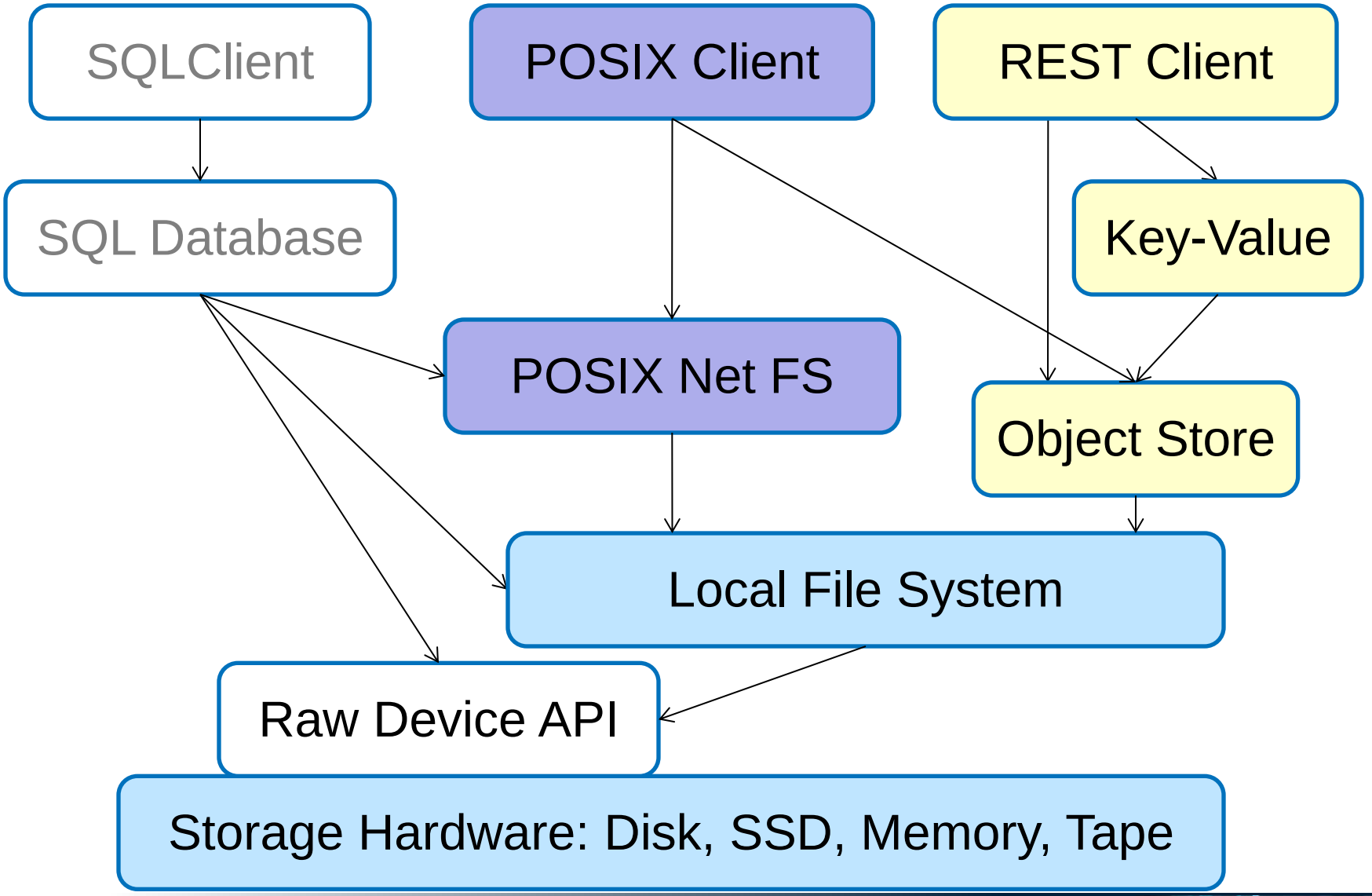
Disk Transfer Rates over Time

7.4 hours to read 4 TB SATA
At 150 MB/sec



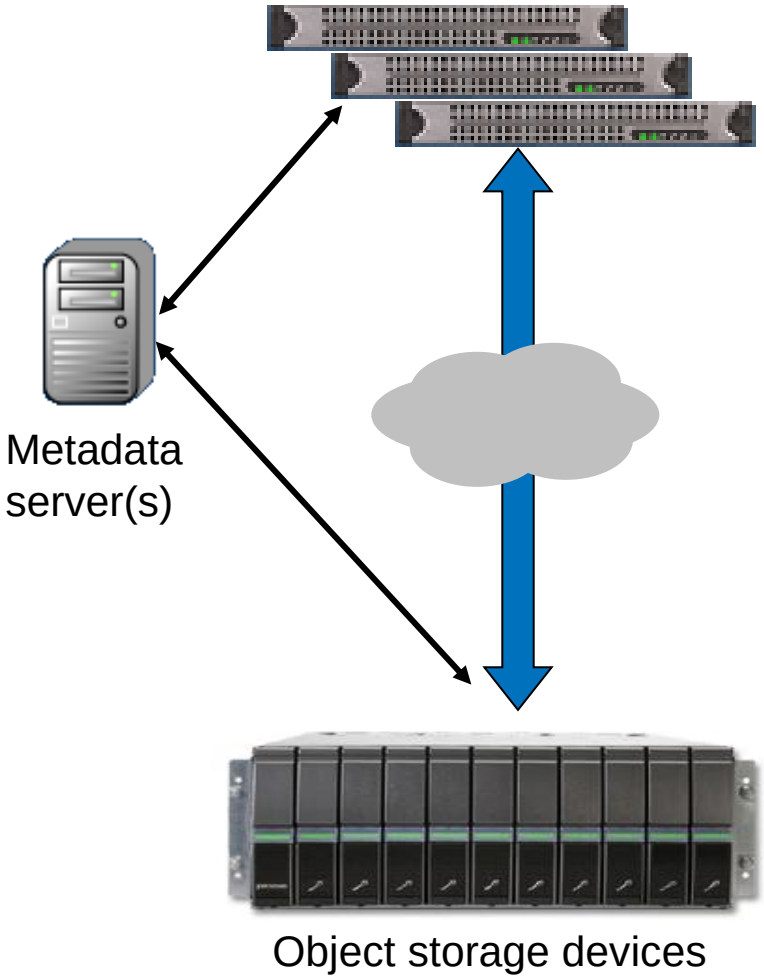
Thanks to R. Freitas of IBM Almaden Research Center for providing much of the data for this graph.

Storage Stack Overview



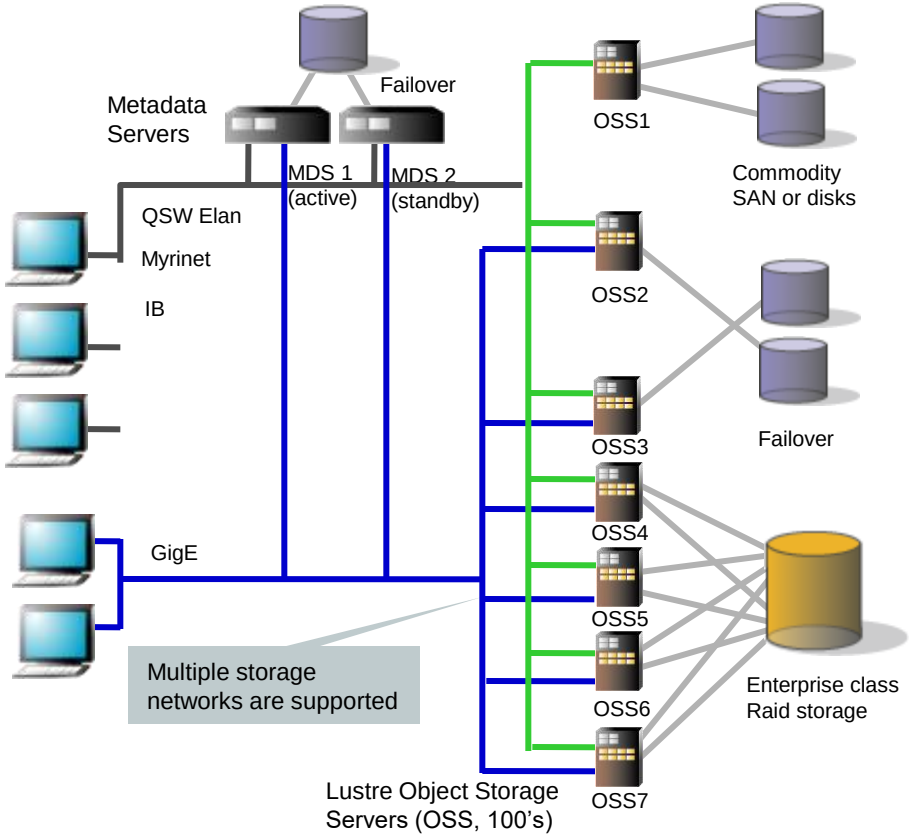
Object-based Storage Clusters

- Lustre, PanFS, Ceph, PVFS
- File system layered over objects
  - Details of block management hidden by the object interface
  - Metadata server manages namespace, access control, and data striping over objects
  - Data transfer directly between OSDs and object-aware clients
- High performance through clustering
  - Scalable to thousands of clients
  - 100+ GB/sec demonstrated to single filesystem



Lustre

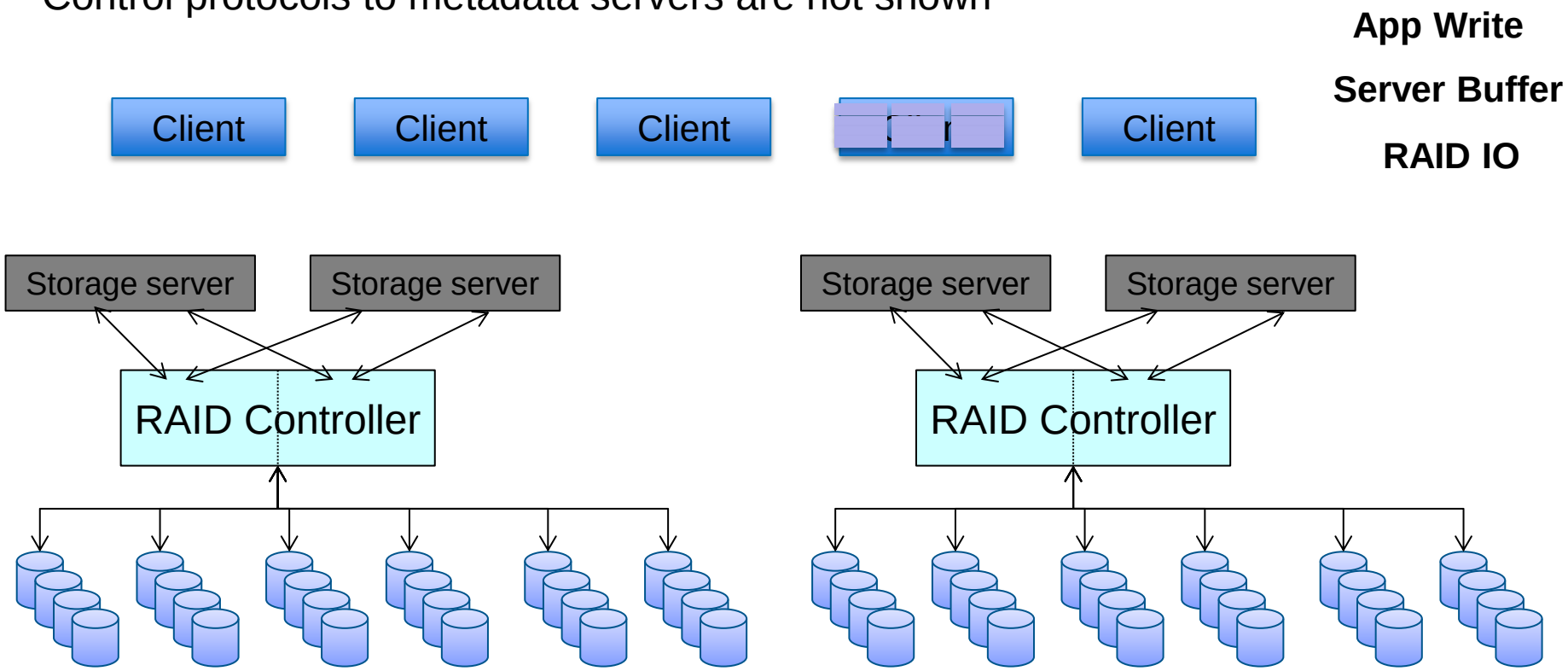
- Open source object-based parallel file system
  - Based on CMU NASD architecture
  - Lots of file system ideas from Coda and InterMezzo
  - ClusterFS acquired by Sun, 9/2007
  - Sun acquired by Oracle 4/2009
  - Whamcloud aquired by Intel, 2012
- Originally Linux-based; Sun ported to Solaris
- Asymmetric design with separate metadata server
- Proprietary RPC network protocol between client & MDS/OSS
- Distributed locking with client-driven lock recovery



Lustre and GPFS Data Path

Lustre clients stripe data across Object Storage Servers (OSS), which in turn write data through a RAID controller to Object Storage Targets (OST). OST hides local file system data structures

GPFS has different metadata model but a similar data path
Control protocols to metadata servers are not shown



Comparing Parallel File Systems

- Block Management
- How Metadata is stored
- What is cached, and where
- Fault tolerance mechanisms
- Management/Administration

Designer cares about

- Performance
- Reliability
- Manageability
- Cost

Customer cares about

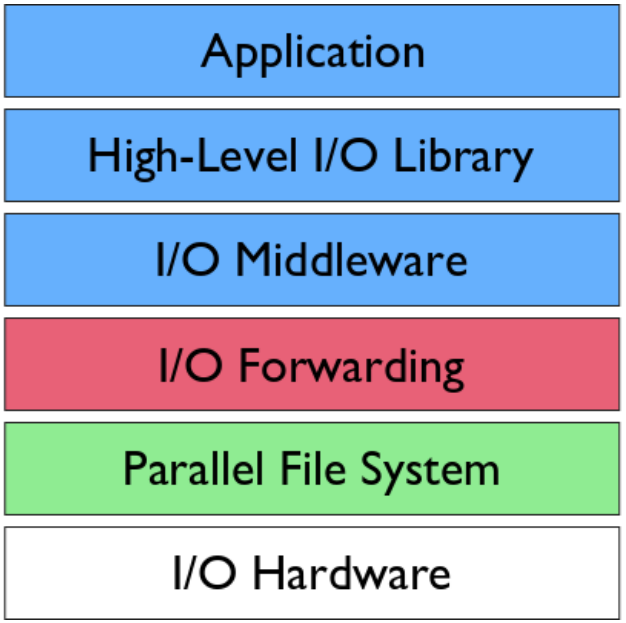
I/O for Computational Science

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, IOFSL, Cray DVS



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.

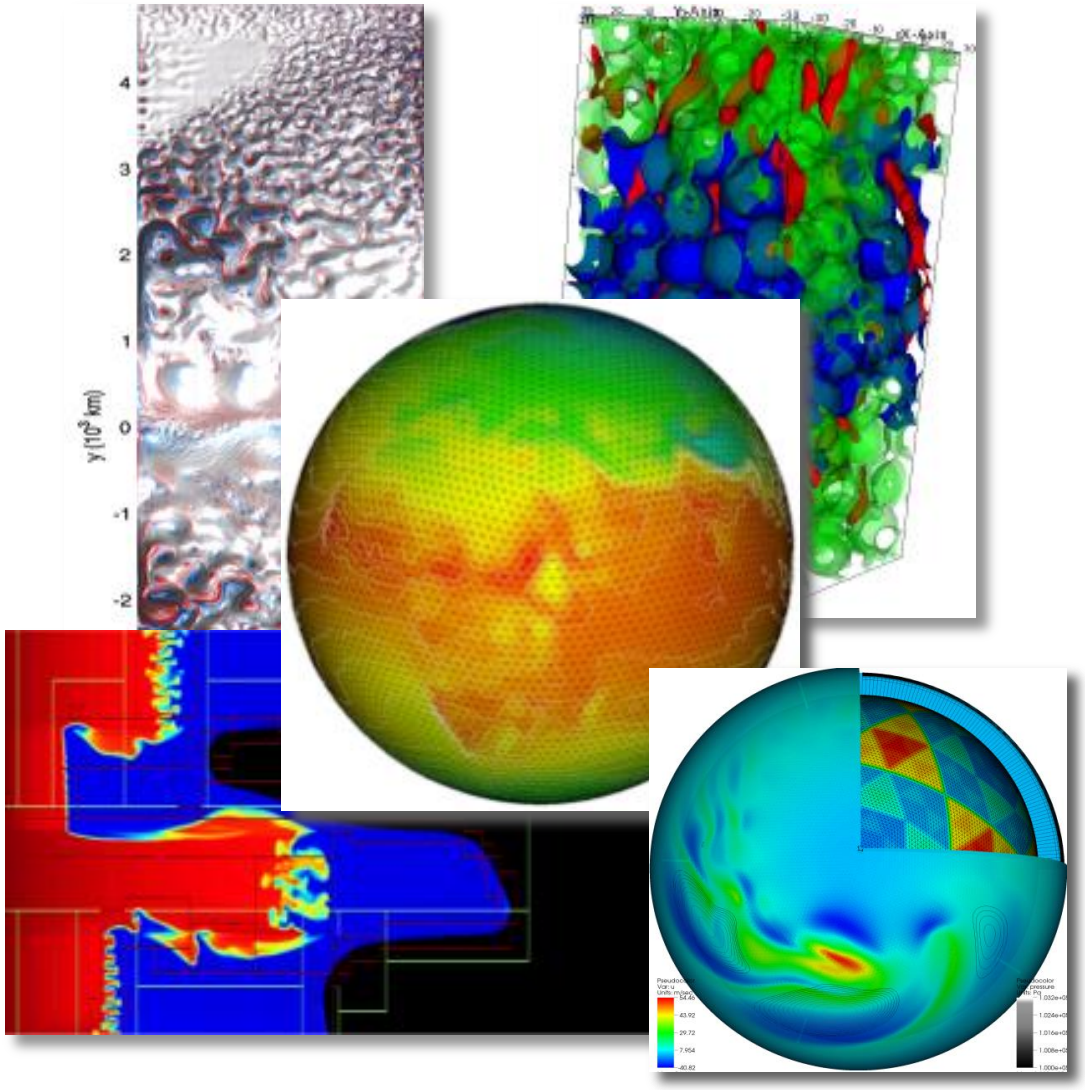
I/O Architectures: Similarities

I/O architectures at large scale have converged to a common model.

- Compute nodes with indirect access to storage
- I/O nodes that form a bridge to the storage system
  - Lnet, DVS, ciod
- External network fabric connecting HPC system to storage
- Collection of storage servers and enterprise storage hardware providing reliable, persistent storage

Scientific I/O: more than hard drives and file systems

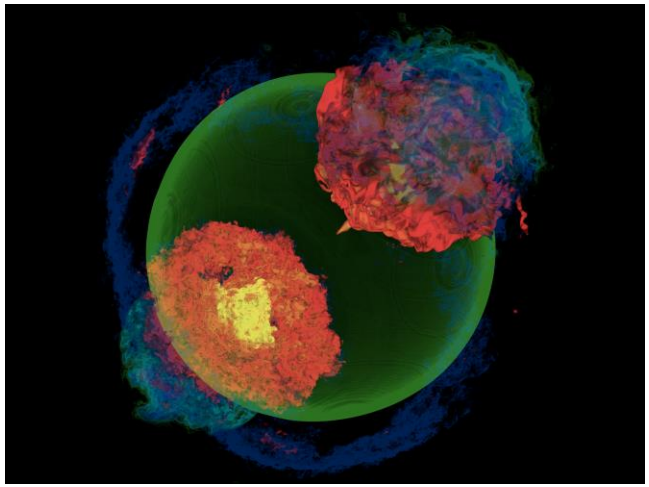
- Scientists think about data in terms of their science problem: molecules, atoms, grid cells, particles
- Ultimately, physical disks store bytes of data
- Layers in between, the application and physical disks are at various levels of sophistication

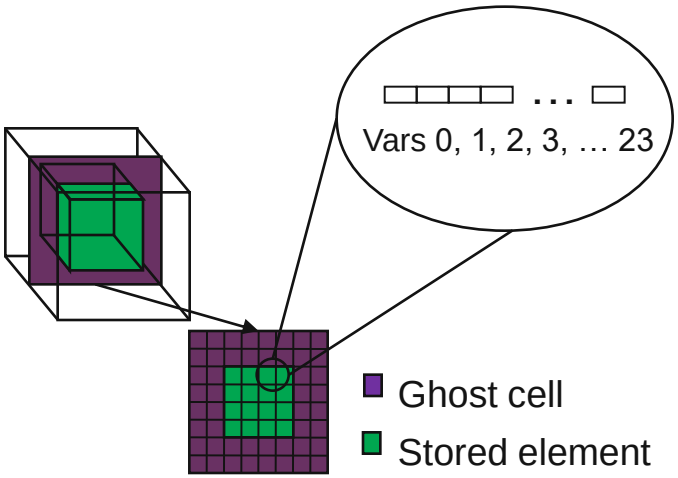


Images from David Randall, Paola Cessi, John Bell, T Scheibe

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's HDF5 requirements

- FLASH AMR structures do not map directly to HDF5 multidimensional arrays
- Must create mapping of the in-memory FLASH data structures into a representation in HDF5 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 HDF5 variable
    - Skip ghost cells
  - Record attributes describing run time, total blocks, etc.

FLASH HDF5 Usage

- Annotations describing data, experiment

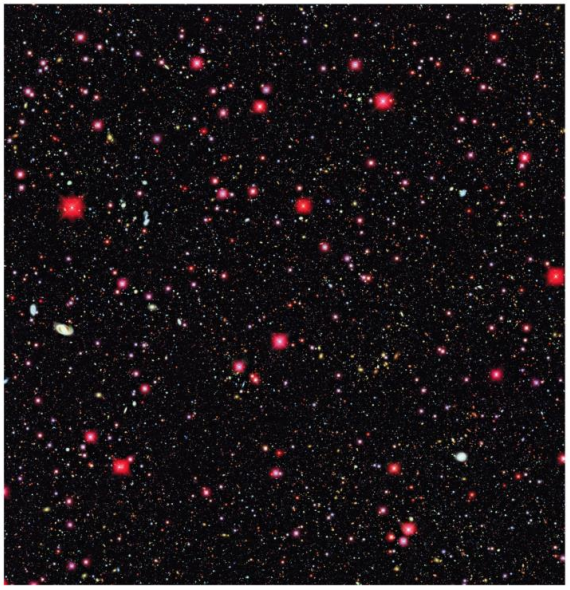
```
attribute_id = H5Acreate(group_id,
  “iteration“,H5T_NATIVE_INT, dataspace_id, H5P_DEFAULT);
status = H5Awrite(attribute_id, H5T_NATIVE_INT, temp);
```

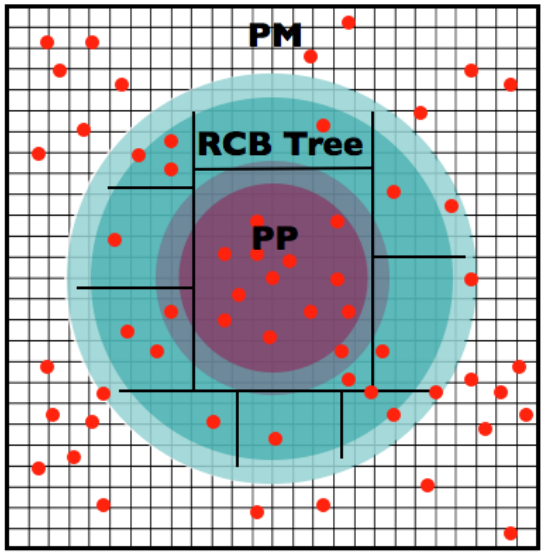
- HDF5 variables for each FLASH variable

```
ierr = H5Sselect_hyperslab(dataspace, H5S_SELECT_SET,
  start_4d, stride_4d, count_4d, NULL);
status = H5Dwrite(dataset, H5T_NATIVE_DOUBLE, memspace,
  dataspace, dxfer_template, unknowns);
```

HACC: understanding cosmos via simulation

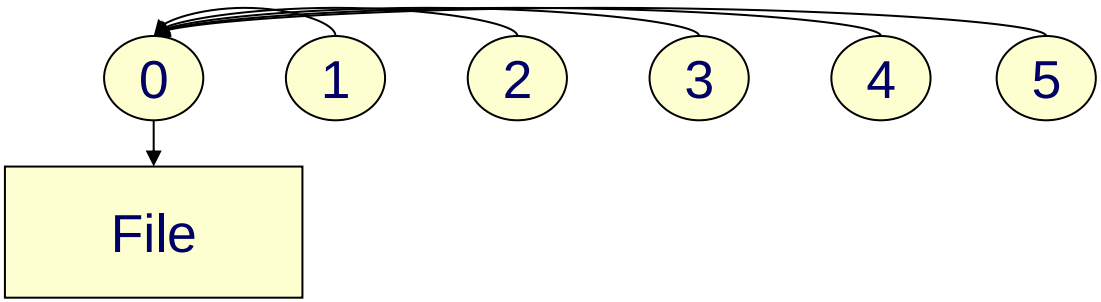
- "Cosmology = Physics + Simulation" (Salman Habib)
- Sky surveys collecting massive amounts of data
  - (~100 PB)
- Understanding of these massive datasets rests on modeling distribution of cosmic entities
- Seed simulations with initial conditions
- Run for 13 billion (simulated) years
- Comparison with observed data validates physics model.
- I/O challenges:
  - Checkpointing
  - analysis



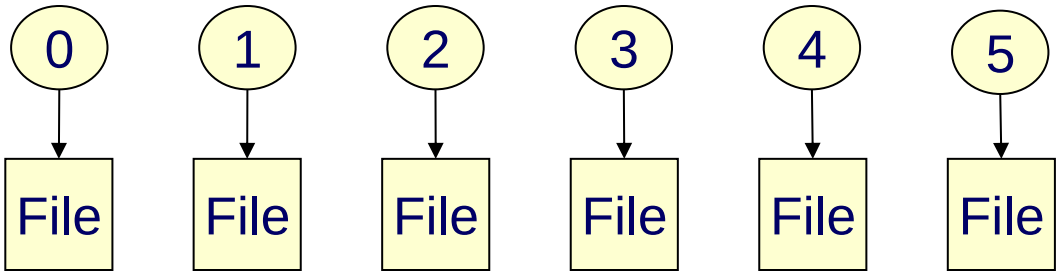


Serial, multi-file parallel and shared file parallel I/O

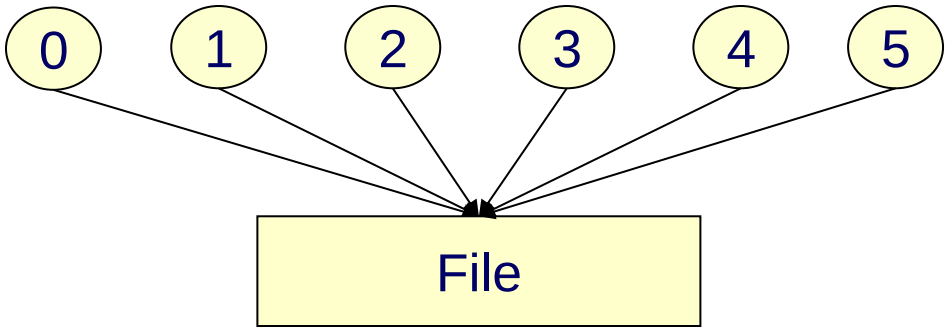
Serial I/O



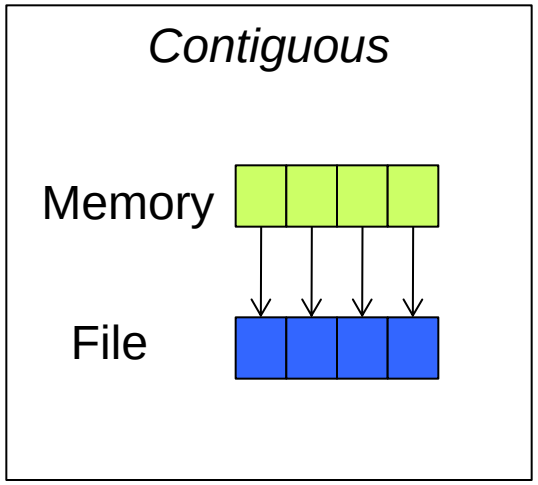
Parallel Multi-file I/O

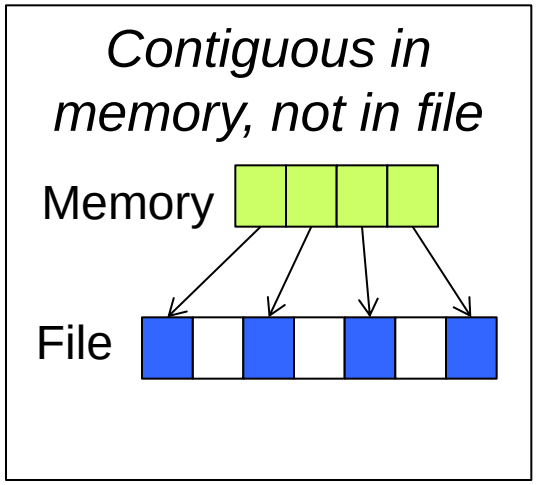


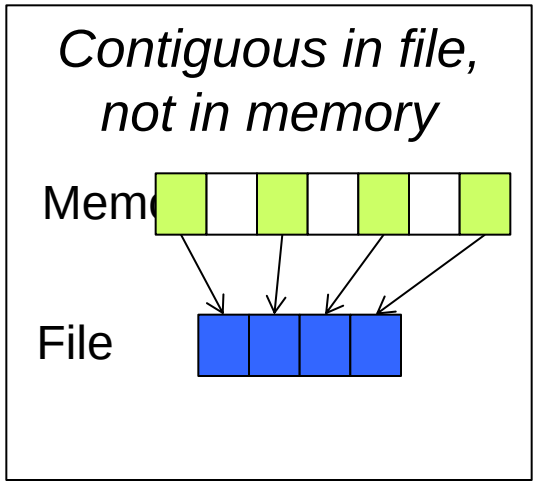
Parallel Shared-file I/O

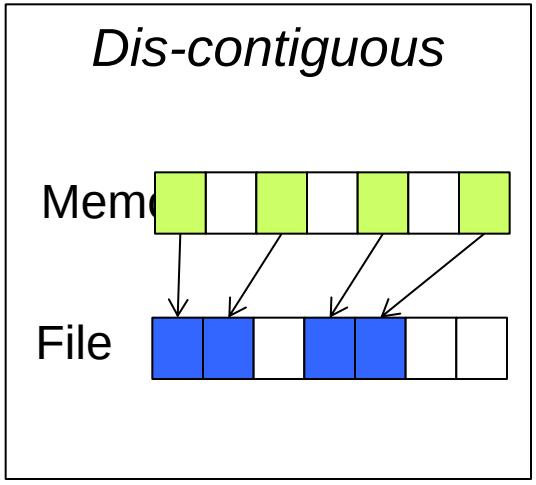


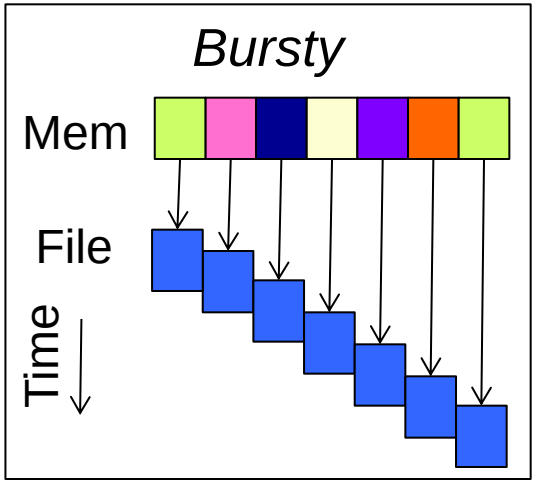
Access Patterns

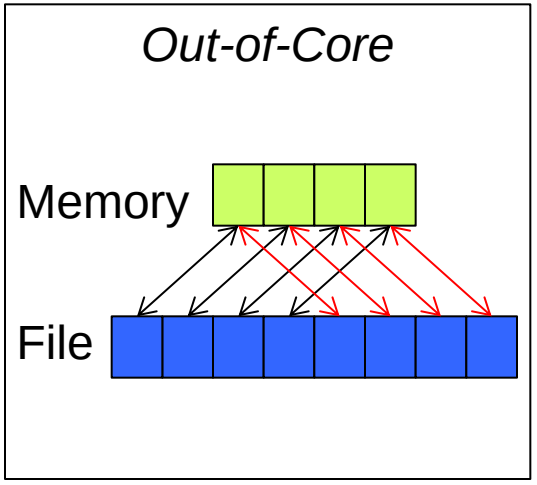












The MPI-IO Interface

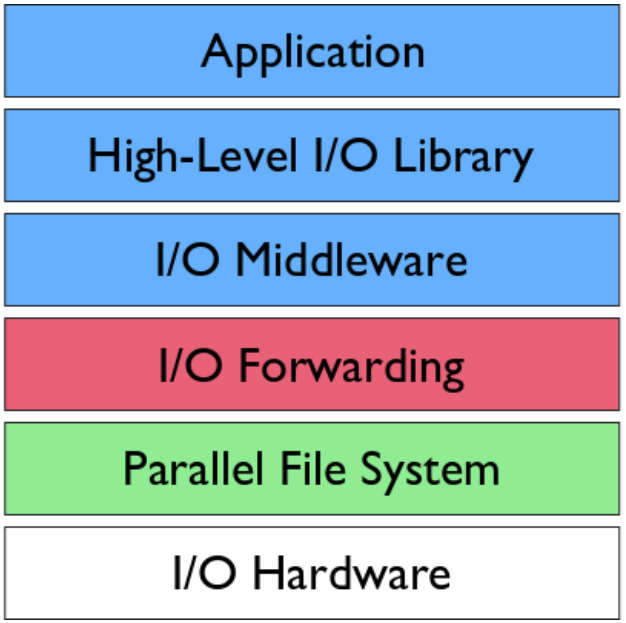
I/O for Computational Science

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, IOFSL, Cray DVS



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.

MPI-IO

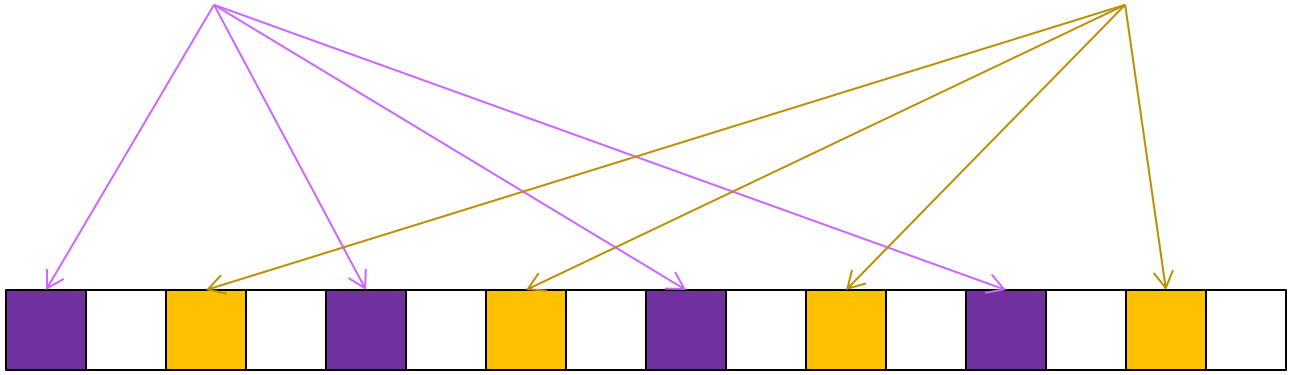
- I/O interface specification for use in MPI apps
- Data model is same as POSIX
  - Stream of bytes in a file
- Features:
  - Collective I/O
  - Noncontiguous I/O with MPI datatypes and file views
  - Nonblocking I/O
  - Fortran bindings (and additional languages)
  - System for encoding files in a portable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

Simple MPI-IO

- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype* passed to write

```
MPI_File_open(COMM, name, mode,
    info, fh);
MPI_File_set_view(fh, disp, etype,
    filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
    datatype, status);
```

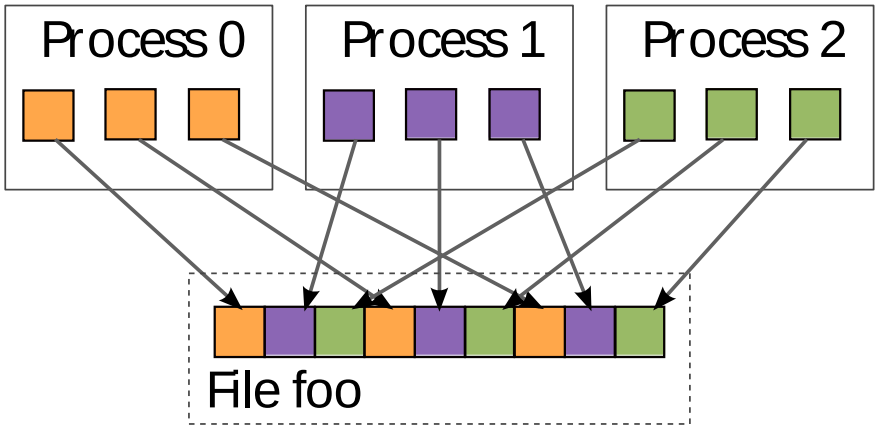
```
MPI_File_open(COMM, name, mode,
    info, fh);
MPI_File_set_view(fh, disp, etype,
    filetype, datarep, info);
MPI_File_write_all(fh, buf, count,
    datatype, status);
```



I/O Transformations

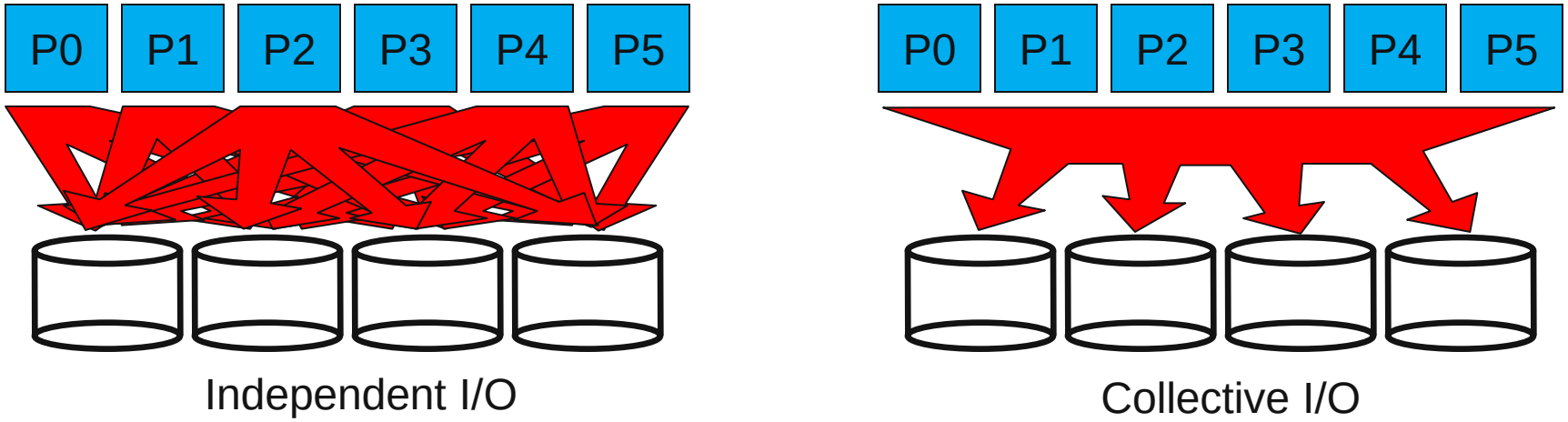
Software between the application and the PFS performs transformations, primarily to improve performance.

- Goals of transformations:
  - Reduce number of operations to PFS (avoiding latency)
  - Avoid lock contention (increasing level of concurrency)
  - Hide number of clients (more on this later)
- With “transparent” transformations, data ends up in the same locations in the file
  - i.e., the file system is still aware of the actual data organization



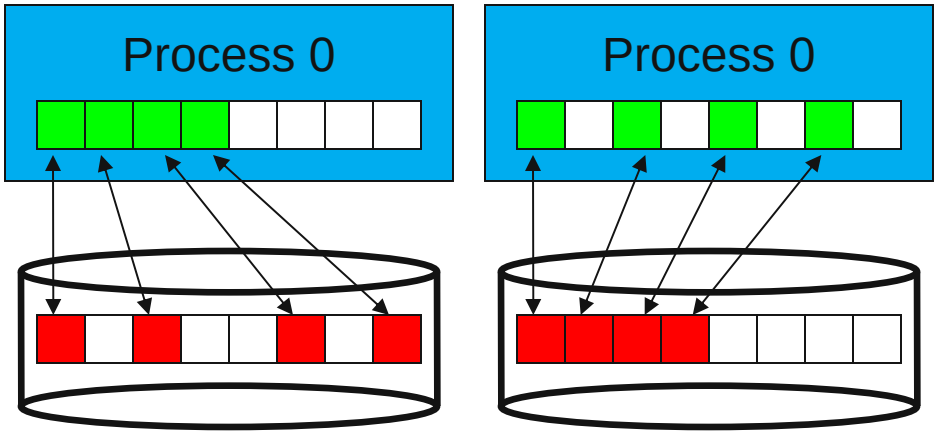
When we think about I/O transformations, we consider the mapping of data between application processes and locations in file.

Independent and Collective I/O



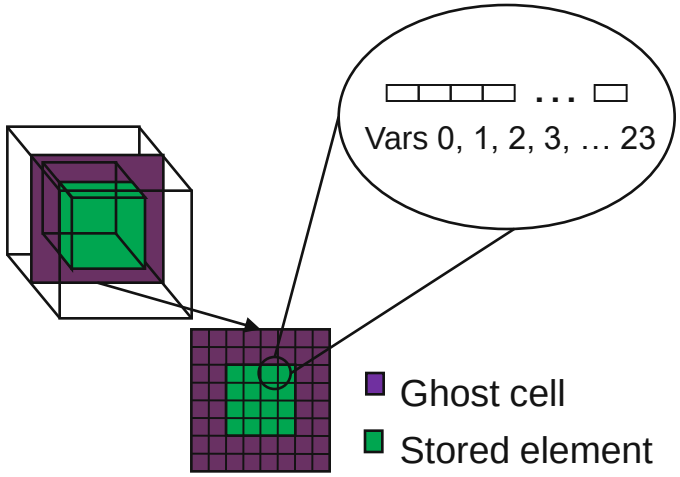
- Independent I/O operations specify only what a single process will do
  - Independent I/O calls do not pass on relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively 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

Contiguous and Noncontiguous I/O



Noncontiguous in File

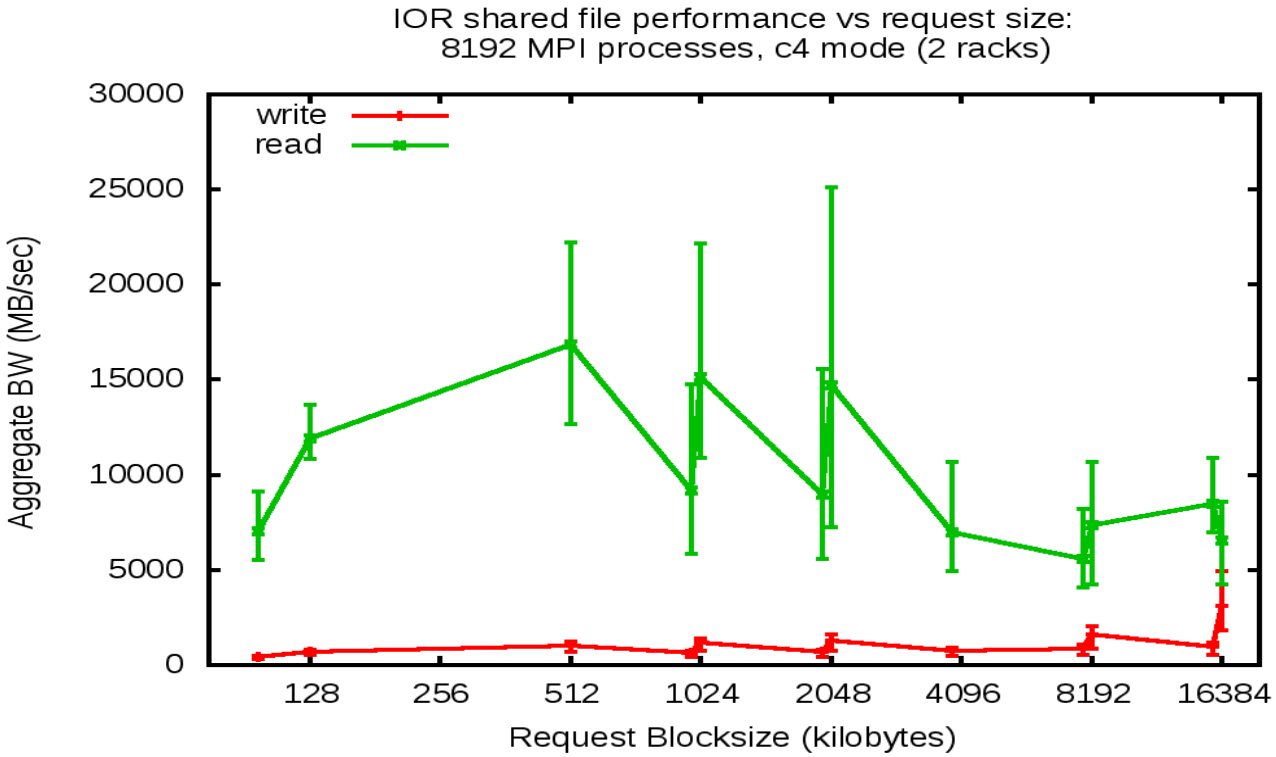
Noncontiguous in Memory



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

Request Size and I/O Rate



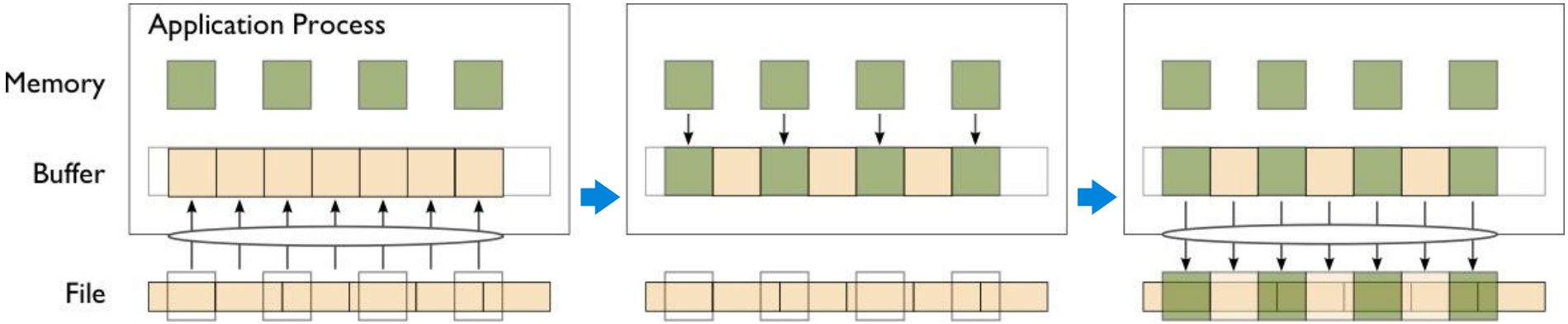
Interconnect latency has a significant impact on effective rate of I/O. Typically I/O should be in the O(Mbytes) range (at least 16 MiB on this GPFS config).

Tests run on 8K processes of IBM Blue Gene/Q at ANL.

Reducing Number of Operations

Since most operations go over the network, I/O to a PFS incurs more latency than with a local FS. *Data sieving* is a technique to address I/O latency by combining operations:

- When reading, application process reads a large region holding all needed data and pulls out what is needed
- When writing, three steps required (below)



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

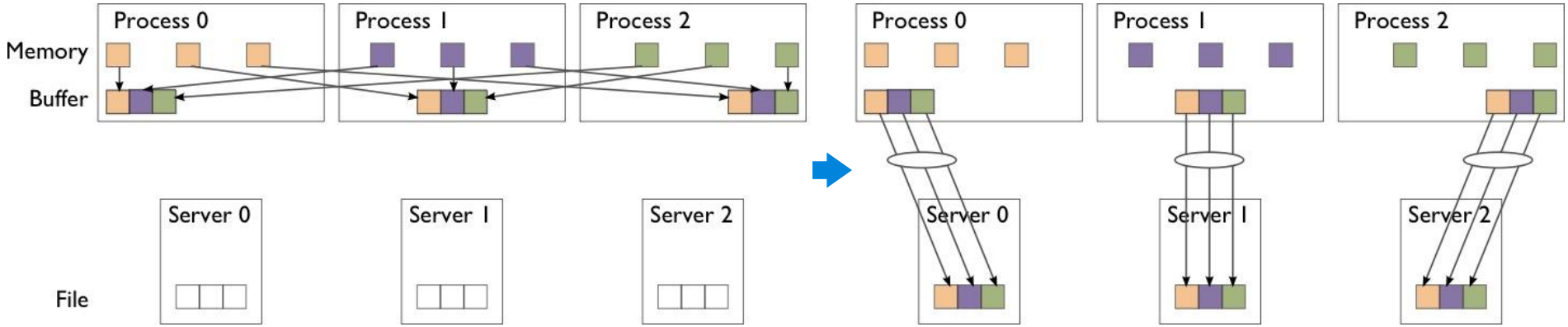
Step 2: Elements to be written to file are replaced in intermediate buffer.

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

Avoiding Lock Contention

To avoid lock contention when writing to a shared file, we can reorganize data between processes. *Two-phase I/O* splits I/O into a data reorganization phase and an interaction with the storage system (two-phase write depicted):

- Data exchanged between processes to match file layout

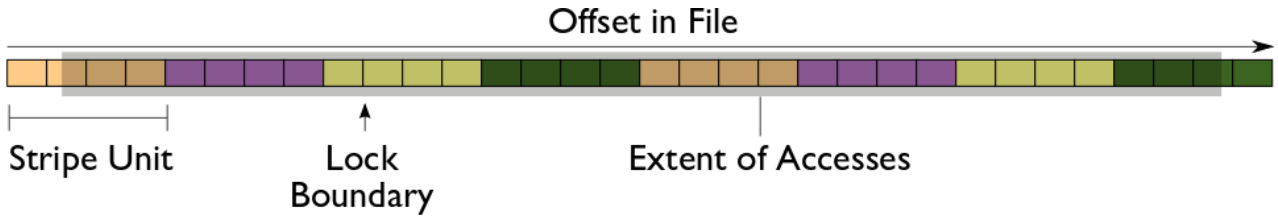


Phase 1: Data are exchanged between processes based on organization of data in file.

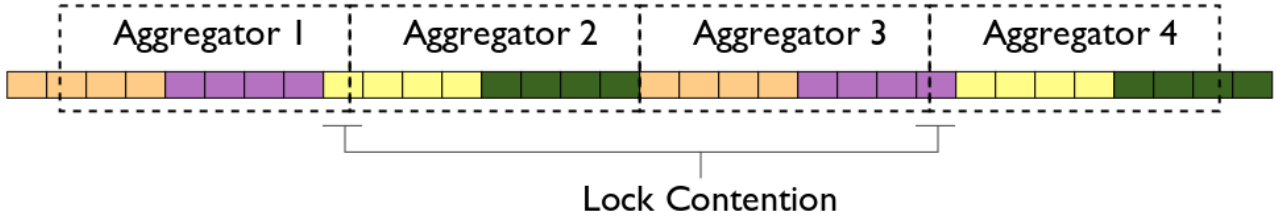
Phase 2: Data are written to file (storage servers) with large writes, no contention.

Two-Phase I/O Algorithms

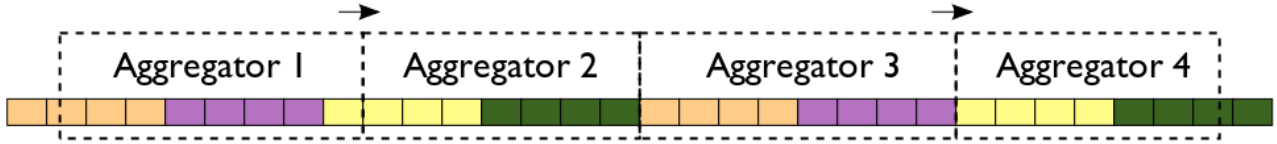
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



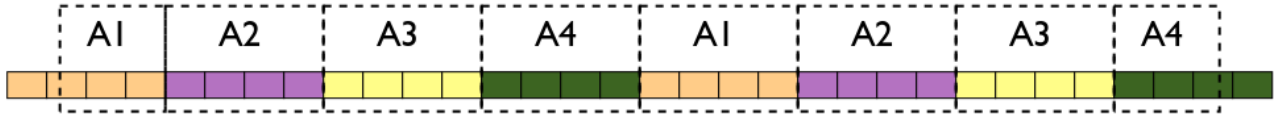
One approach is to evenly divide the region accessed across aggregators.



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).



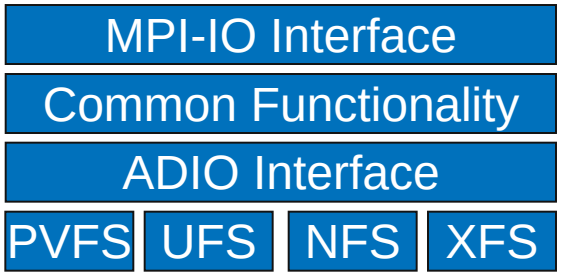
For more information, see W.K. Liao and A. Choudhary, “Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols,” SC2008, November, 2008.

Impact of 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	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 Implementations

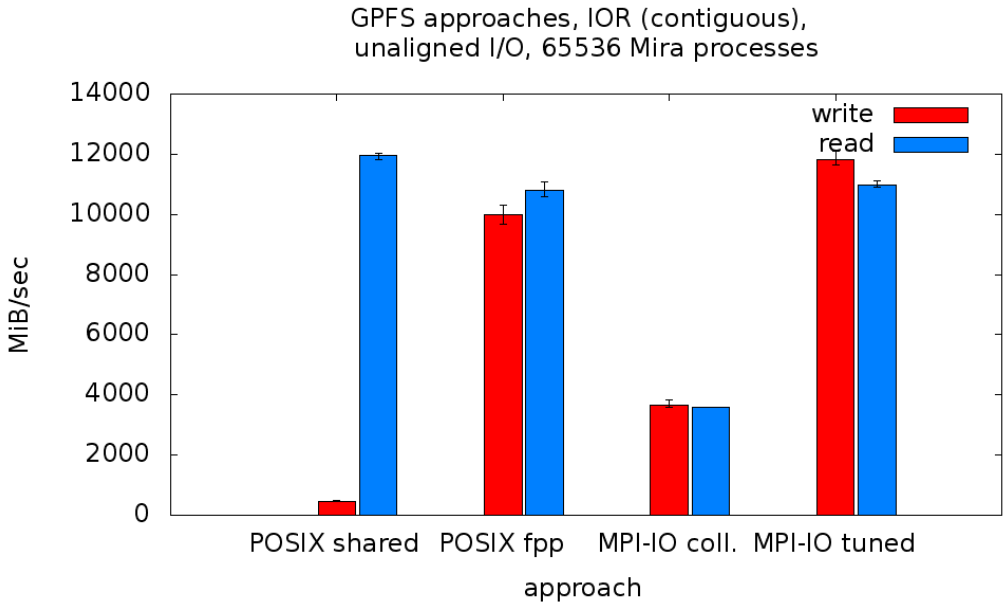


ROMIO's layered architecture.

- Different MPI-IO implementations exist
- Four better-known ones are:
  - ROMIO from Argonne National Laboratory
    - Leverages MPI-1 communication
    - Supports local file systems, network file systems, parallel file systems
      - UFS module works GPFS, Lustre, and others
    - Includes data sieving and two-phase optimizations
  - MPI-IO/GPFS from IBM (for AIX only)
    - Includes two special optimizations
      - Data shipping -- mechanism for coordinating access to a file to alleviate lock contention (type of aggregation)
      - Controlled prefetching -- using MPI file views and access patterns to predict regions to be accessed in future
  - MPI from NEC
    - For NEC SX platform and PC clusters with Myrinet, Quadrics, IB, or TCP/IP
    - Includes listless I/O optimization -- fast handling of noncontiguous I/O accesses in MPI layer
  - OMPIO from OpenMPI
    - Emphasis on modularity and tighter integration into MPI implementation

GPFS Access three ways

- POSIX shared vs MPI-IO collective
  - Locking overhead for unaligned writes hits POSIX hard
- Default MPI-IO parameters not ideal
  - Reported to IBM; simple tuning brings MPI-IO back to parity
  - “Vendor Defaults” might give you bad first impression
- File per process (fpp) extremely seductive, but entirely untenable on current generation.



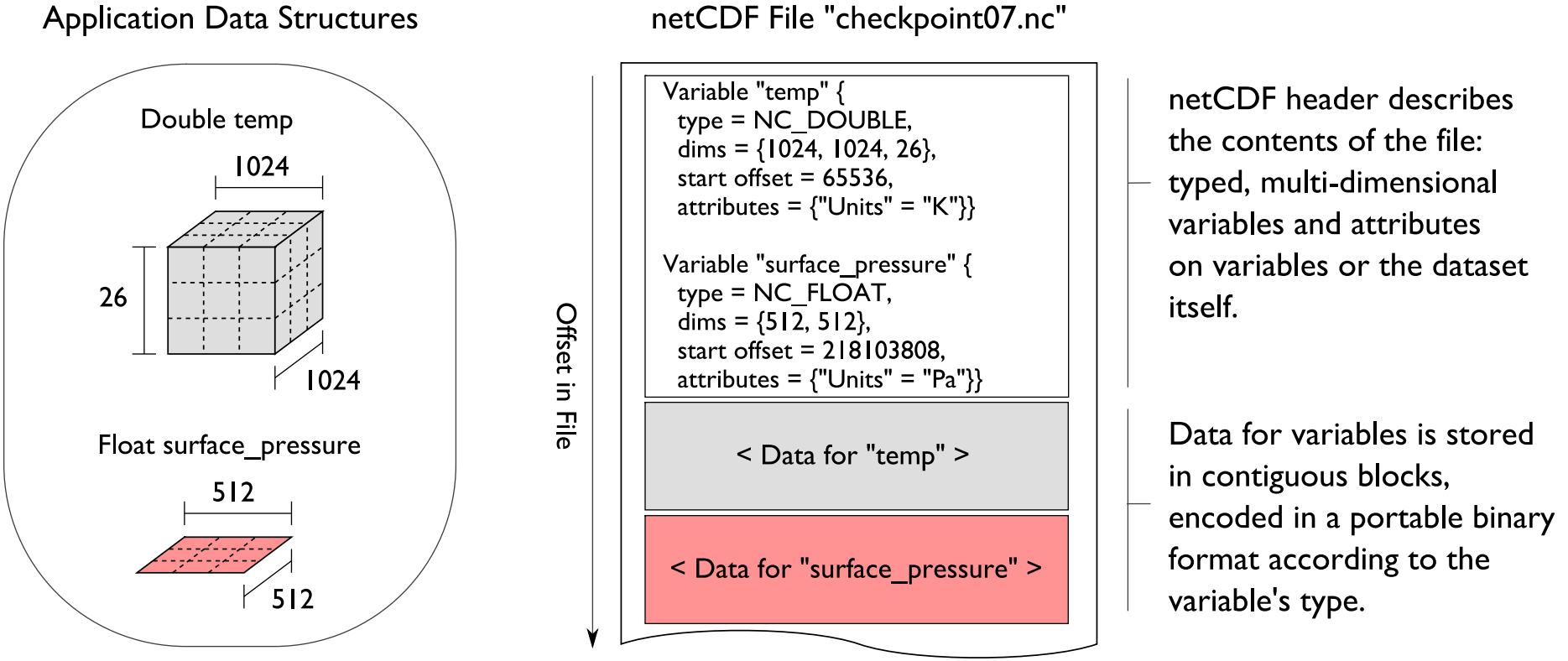
MPI-IO Wrap-Up

- 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
- Ideal location in software stack for file system specific quirks or optimizations
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

Parallel NetCDF (PnetCDF)

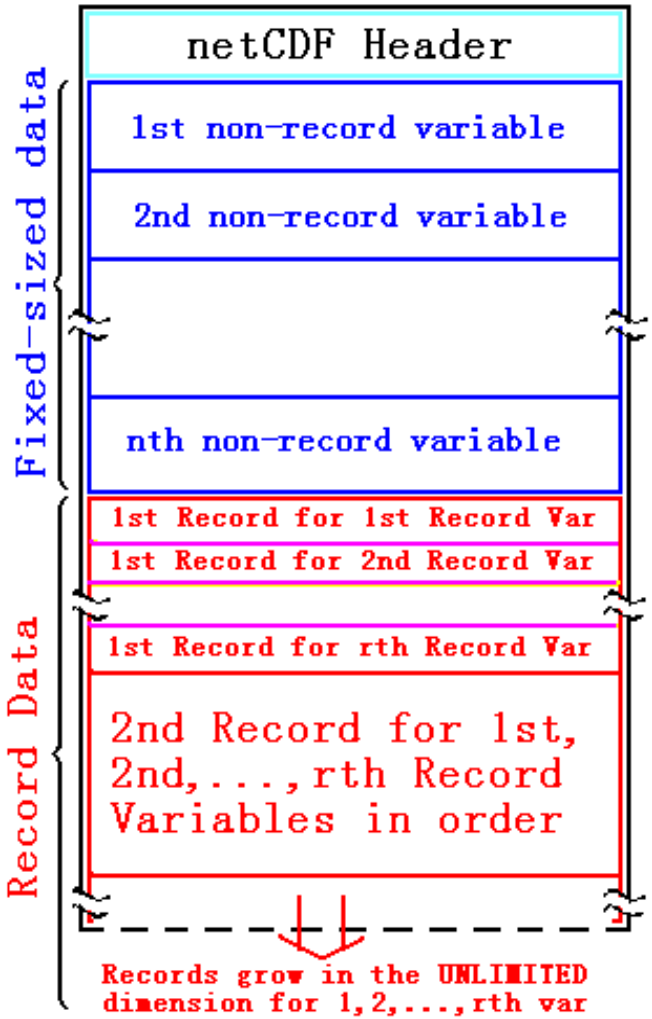
- Based on original “Network Common Data Format” (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C, Fortran, and F90 interfaces
  - Portable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

Data Layout in netCDF Files



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



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 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];
intvarids[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

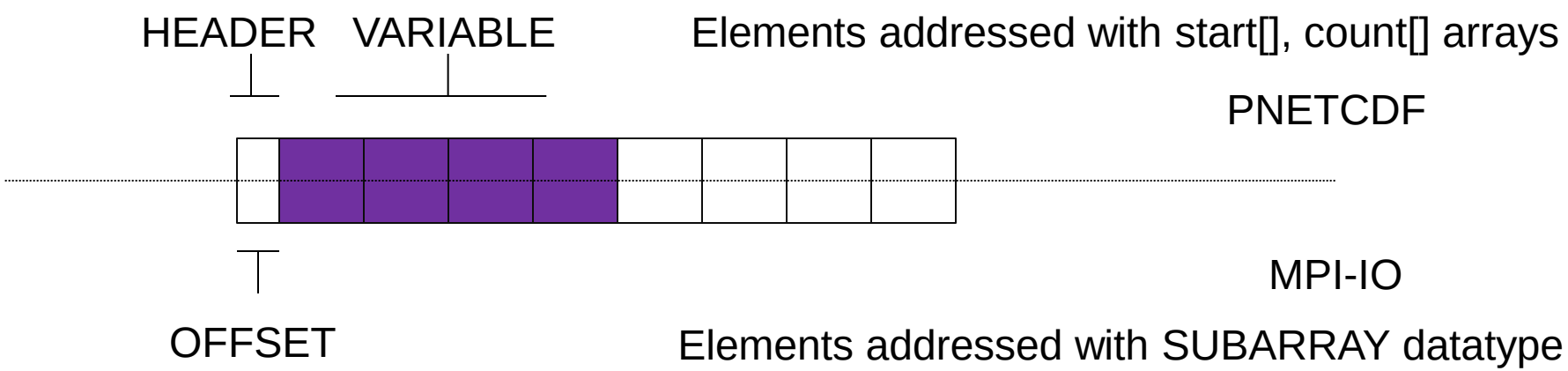
- In define mode (collective)
  - Use `MPI_File_open` to create file at create time
  - Set hints as appropriate (more later)
  - Locally cache header information in memory
    - All changes are made to local copies at each process
- At ncmpi_enddef
  - Process 0 writes header with `MPI_File_write_at`
  - `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

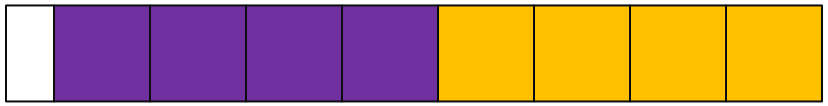
Parallel-NetCDF and MPI-IO



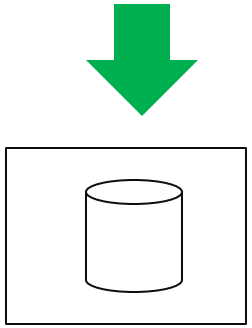
- `ncmpi_put_vara_all` describes access in terms of arrays, elements of arrays
  - E.g. Give me a 3x3 subcube of this larger 1024x1024 array
- Library translates into MPI-IO calls
  - `MPI_Type_create_subarray`
  - `MPI_File_set_view`
  - `MPI_File_write_all`

Parallel-NetCDF write-combining optimization

```
ncmpi_iput_vara(ncfile, varid1,
    &start, &count, &buffer1,
    count, MPI_INT, &requests[0]);
ncmpi_iput_vara(ncfile, varid2,
    &start,&count, &buffer2,
    count, MPI_INT, &requests[1]);
ncmpi_wait_all(ncfile, 2, requests, statuses);
```



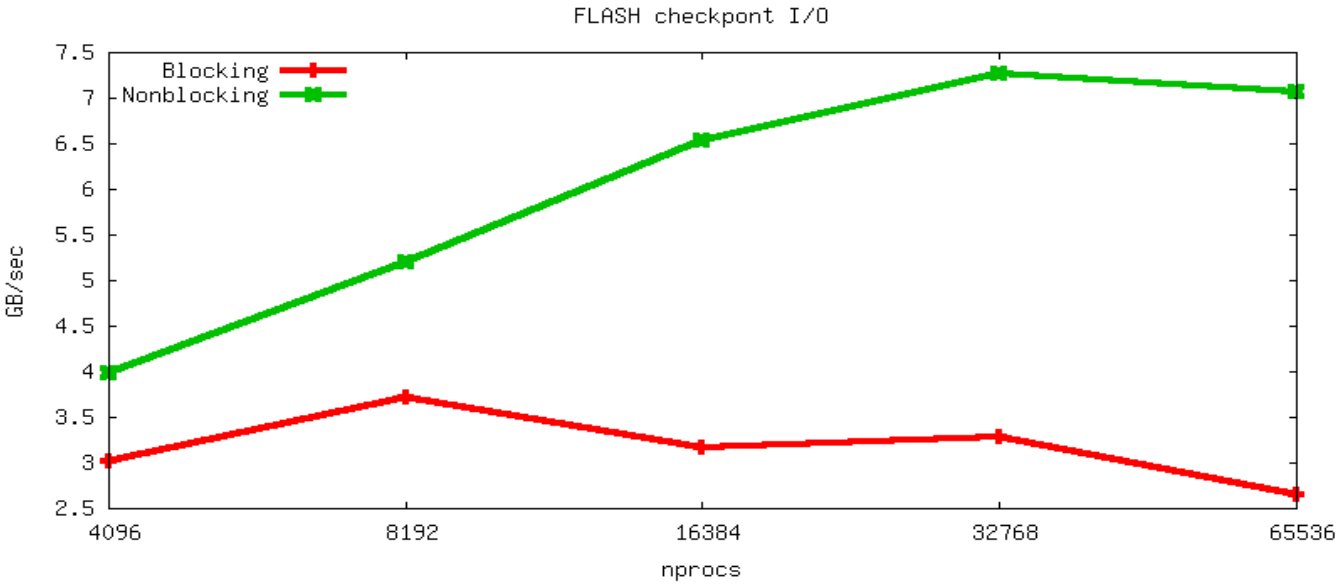
HEADER VAR1 VAR2



- netCDF variables laid out contiguously
- Applications typically store data in separate variables
  - temperature(lat, long, elevation)
  - Velocity_x(x, y, z, timestep)
- Operations posted independently, completed collectively
  - Defer, coalesce synchronization
  - Increase average request size

FLASH Astrophysics and the write-combining optimization

- FLASH writes one variable at a time
- Could combine all 4D variables (temperature, pressure, etc) into one 5D variable
  - Altered file format (conventions) requires updating entire analysis toolchain
- Write-combining provides improved performance with same file conventions
  - Larger requests, less synchronization.
  - Convinced HDF to develop similar interface

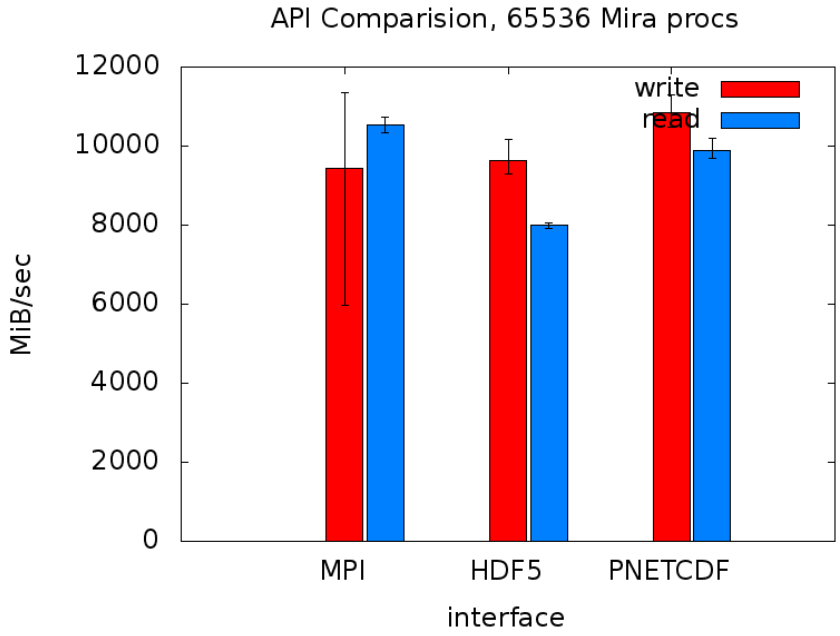


PnetCDF Wrap-Up

- PnetCDF gives us
  - Simple, portable, self-describing container for data
  - Collective I/O
  - Data structures closely mapping to the variables described
- If PnetCDF meets application needs, it is likely to give good performance
  - Type conversion to portable format does add overhead
- Some limits on (old, common CDF-2) file format:
  - Fixed-size variable: < 4 GiB
  - Per-record size of record variable: < 4 GiB
  - 2^{32} -1 records
  - New extended file format to relax these limits (CDF-5, released in pnetcdf-1.1.0; Integrated in Unidata NetCDF-4.4)

Comparing I/O libraries

- IOR to evaluate HDF5, pnetcdf somewhat artificial
  - HLL typically hold structured data
- HDF5, pnetcdf demonstrate performance parity for these access sizes (6 MiB on Mira)
- I/O libraries deliver benefits with slight (if any) cost to performance



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: likely one already exists for your domain

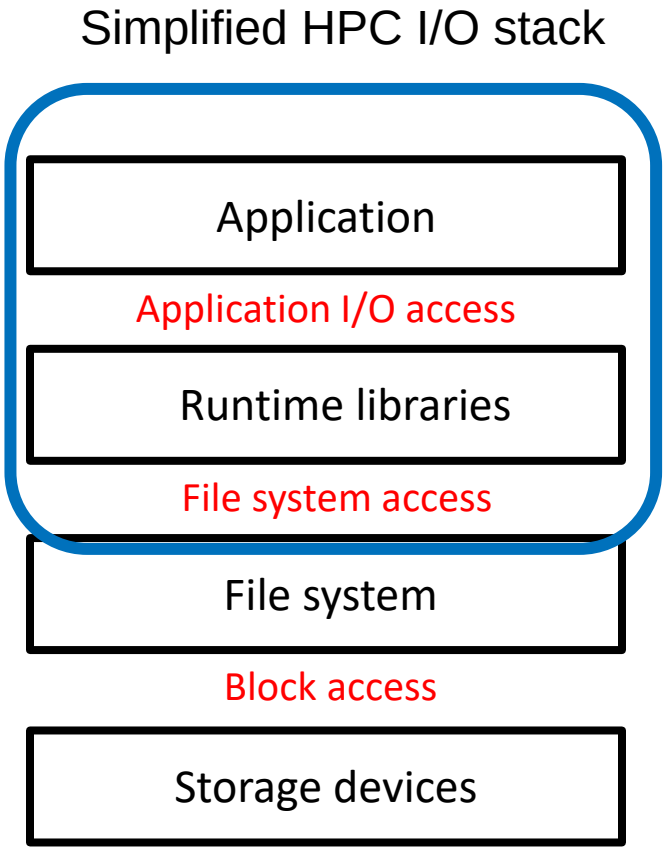
Parallel I/O Wrap-up

- Assess the cost benefit of using shared file parallel-IO for the lifetime of your project
  - How much overhead can you afford?
  - Slower runtime, could save years of post-processing, visualization and analysis time later
- Use high level parallel I/O libraries over MPI-IO.
  - They don't cost you performance (sometimes improve it)
  - Gain: portability, longevity, programmability
- MPI-IO is the layer where most optimizations are implemented – tune these parameters carefully
- Watch out for the key parallel-I/O pitfalls – unaligned block sizes and small writes
  - MPI-IO layer can often solve these pitfalls on your behalf.

Characterizing Application I/O

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

- The best way to answer these questions is by observing behavior at the application and library level
- What did the application intend to do, and how much time did it take to do it?
- In this portion of the training course we will focus on ***Darshan***, a scalable tool for characterizing application I/O activity.



I/O Performance Tuning "Rules of thumb"

- Use collectives when possible
- Use high-level libraries (e.g. HDF5 or PnetCDF) when possible
- A few large I/O operations are better than many small I/O operations
- Avoid unnecessary metadata operations, especially *stat()*
- Avoid writing to shared files with POSIX
- Avoid leaving gaps/holes in files to be written later
- Use tools like Darshan to check assumptions about behavior

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?
- Quincey Koziol, Prabhat, editors, High Performance Parallel I/O, Chapman and Hall/ CRC, October 2014
  - Survey of nearly every tool, library, file system available
- 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://www.hdfgroup.org/HDF5/
  - http://www.hdfgroup.org/HDF5/Tutor
- 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 (now branded “Spectrum Scale”)

  http://www-03.ibm.com/systems/storage/spectrum/scale/index.html

On-Line References (3 of 4)

- IOR benchmark
  - https://github.com/chaos/ior
- 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
  - https://fs.hlrs.de/projects/par/mpi/b_eff_io/

On Line References (4 of 4)

- NFS Version 4.1
  - 5661: NFSv4.1 protocol
  - 5662: NFSv4.1 XDR Representation
  - 5663: pNFS Block/Volume Layout
  - 5664: pNFS Objects Operation
- pNFSProblem 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

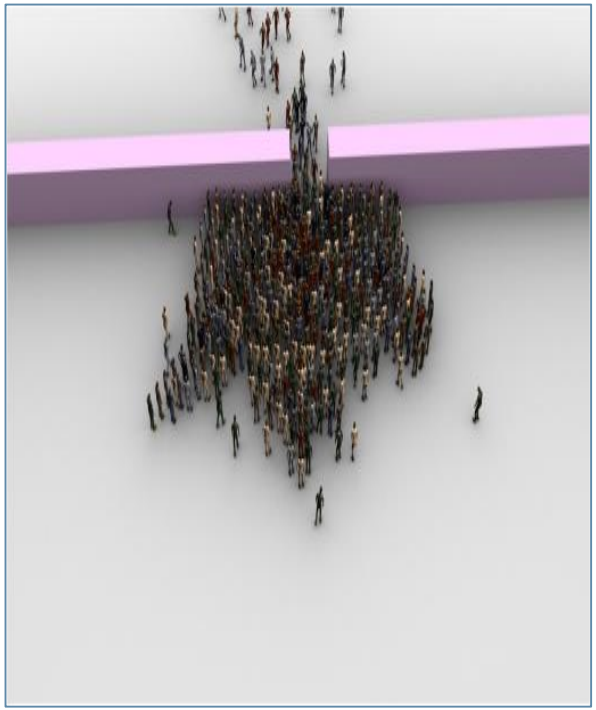
What are the problems?

Not enough I/O capacity on current HPC systems, and the trend is getting worse.

If there's not enough I/O, you can't write data to storage, so you can't analyze it: lost science.

Energy consumption: it costs a lot of power to write data to disk.

Opportunity for doing better science (analysis) when have access to full spatiotemporal resolution data.



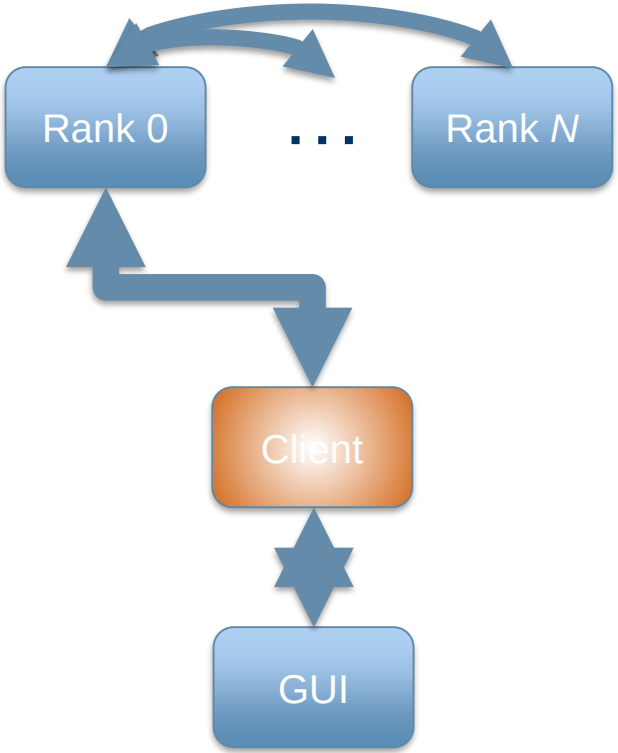
The problem is not going away

How does Summit compare to Titan

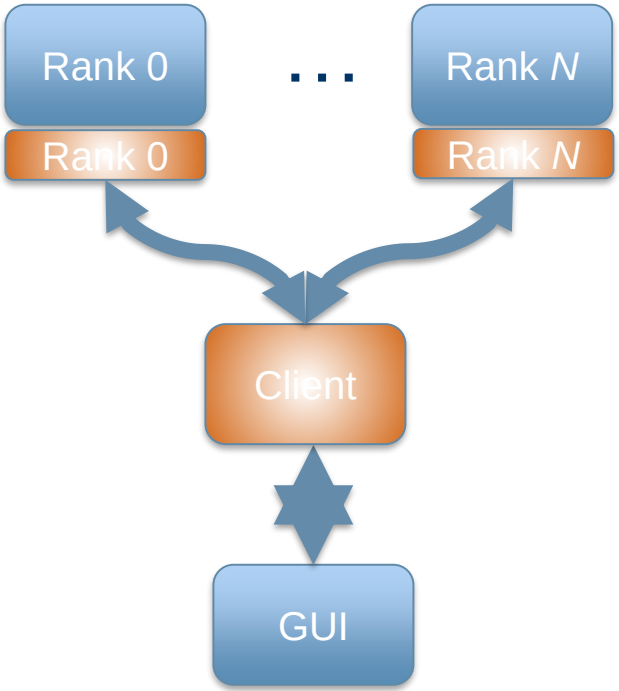
Feature	Summit	Titan
Application Performance	5-10x Titan	Baseline
Number of Nodes	~3,400	18,688
Node performance	> 40 TF	1.4 TF
Memory per Node	>512 GB (HBM + DDR4)	38GB (GDDR5+DDR3)
NVRAM per Node	800 GB	0
Node Interconnect	NVLink (5-12x PCIe 3)	PCIe 2
System Interconnect (node injection bandwidth)	Dual Rail EDR-IB (23 GB/s)	Gemini (6.4 GB/s)
Interconnect Topology	Non-blocking Fat Tree	3D Torus
Processors	IBM POWER9™ NVIDIA Volta™	AMD Opteron™ NVIDIA Kepler™
File System	120 PB, 1 TB/s, GPFS™	32 PB, 1 TB/s, Lustre®
Peak power consumption	10 MW	9 MW

Data courtesy A. Geist (ORNL)

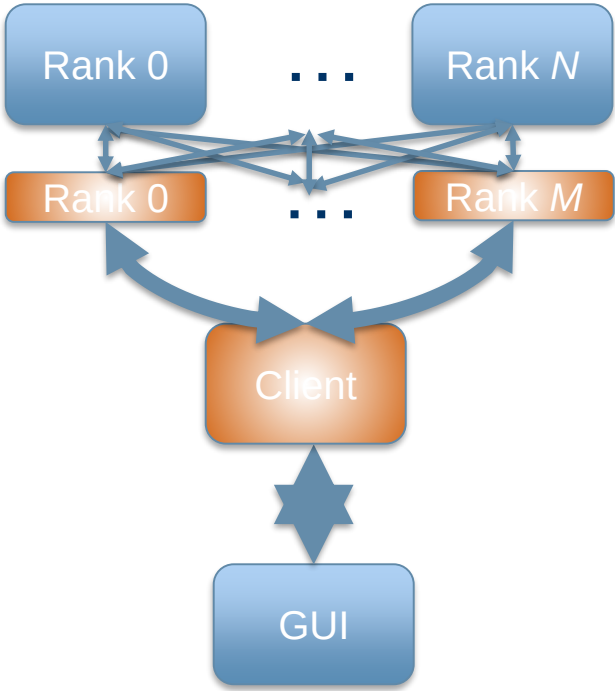
Common design patterns of 1990s



Many-to-one: AVS



“Tightly coupled”: pV3, custom projects



“Loosely coupled”, M-to-N: CUMULVS