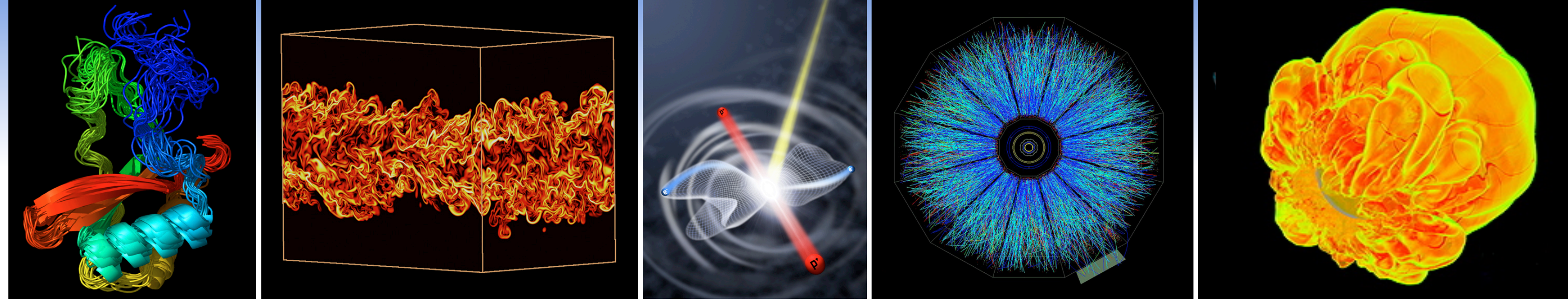


Parallel I/O

Katie Antypas

NUG Meeting

Boulder CO, October 8th 2009

Thanks to Rob Ross and Rob Latham at ANL for use of some slides

Outline

- Review of parallel file systems
- Application Parallel I/O strategies
  - MPI-IO
  - Parallel I/O libraries
- Lustre Optimizations
- Preview of Hopper I/O
- Best practices and recommendations

Getting bigger all the time

- User I/O needs growing each year in scientific community
- For our largest users I/O parallelism is mandatory
- I/O remains a bottleneck for many users

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

Disk Access Rates over Time

Average Internal Drive Access Rate (MBytes/sec)

1000
100
10
1
0.1
0.01
0.001

1950 1960 1970 1980 1990 2000 2010

RAMAC
2311
2314
3330
3350
3370
3380
3390
Spitfire
Ultrastar 18ES
Ultrastar 18ZX
Ultrastar 73LZX
Savvio 15K.1
Cheetah 15K.6
Ultrastar A7K1000

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

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

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

Slide from Rob Ross, Rob Latham at ANL

What is a File System?

- A file system is a method for storing, organizing, manipulating, navigating, accessing and retrieving data files
  - This is a layer that mediates transactions between the Operating System and the Storage Device.
- A file system deals with "data" and "metadata" (data about data)
- We often refer to a "file system name" as the root of a hierarchical directory tree, e.g. "the /home file system."
  - We can treat this as "one big disk," but it may actually be a complex collection of disk arrays, IO servers, and networks.

Terminology: Metadata

- Data about data
- File systems store information about files externally to those files.
- Linux uses an inode, which stores information about files and directories (size in bytes, device id, user id, group id, mode, timestamps, link info, pointers to disk blocks, file size…)
- Any time a file's attributes change or info is desired (e.g., `ls –l`) metadata has to be retrieved from the metadata server
- Metadata operations are IO operations which require time and disk space.

Generic Parallel File System Architecture

Compute Nodes

Internal Network

I/O Servers

MDS

I/O

I/O

I/O

I/O

I/O

I/O

I/O

External Network - (Likely FC)

Disk controllers - manage failover

Storage Hardware -- Disks

Franklin Luster Configuration in /scratch

Franklin Compute and Interactive Nodes

"The Torus"

24 OSS

48 OSTs

OST OST OST OST

FC Network

. . .

6 DDN

48 LUN

. . .

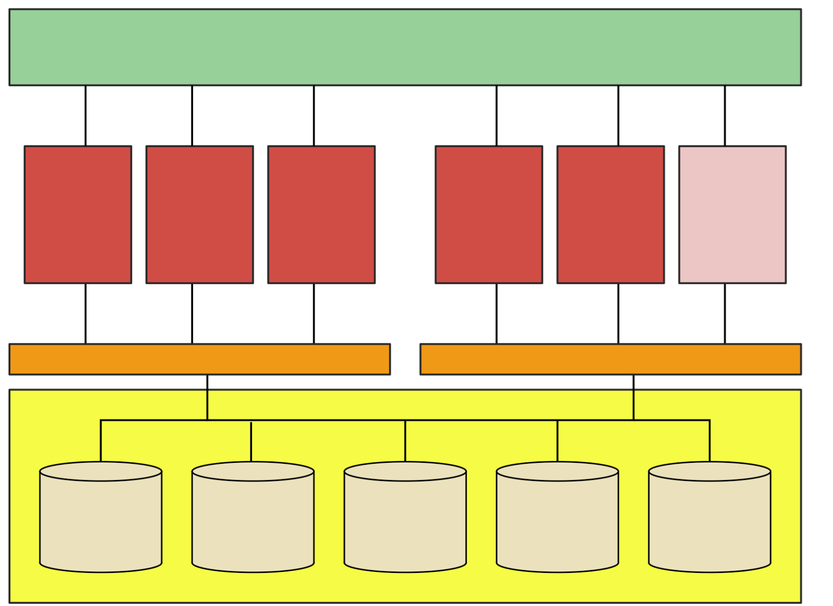
Peak I/O system bandwidth is ~17 Gbyte/sec.

Connectivity and configuration set in a "good" way for parallelism. Using 24 OSTs will spread evenly over the DDN appliances.

Fault Tolerance and Parallel File Systems

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

- RAID techniques hide disk failures
- Redundant controllers and shared access to storage
- Heartbeat software and quorum directs server failover



System network connects storage to compute resources.

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

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

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

Data Distribution in Parallel File Systems

Logically a file is an extendable sequence of bytes that can be referenced by offset into the sequence.

Metadata associated with the file specifies a mapping of this sequence of bytes into a set of objects on PFS servers.

Extents in the byte sequence are mapped into objects on PFS servers. This mapping is usually determined at file creation time and is often a round-robin distribution of a fixed extent size over the allocated objects.

checkpoint32.nc

H01 H02 H03 H04

Offset in File

E00 E01 E02 E03 E05 E06 E07 E08 E09 E10 E11

PFS Server H01 — E00 E08

PFS Server H02 — E01 E05 E09

PFS Server H03 — E02 E06 E10

PFS Server H04 — E03 E07 E11

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

A static mapping from logical file to objects allows clients to easily calculate server(s) to contact for specific regions, eliminating need to interact with a metadata server on each I/O operation.

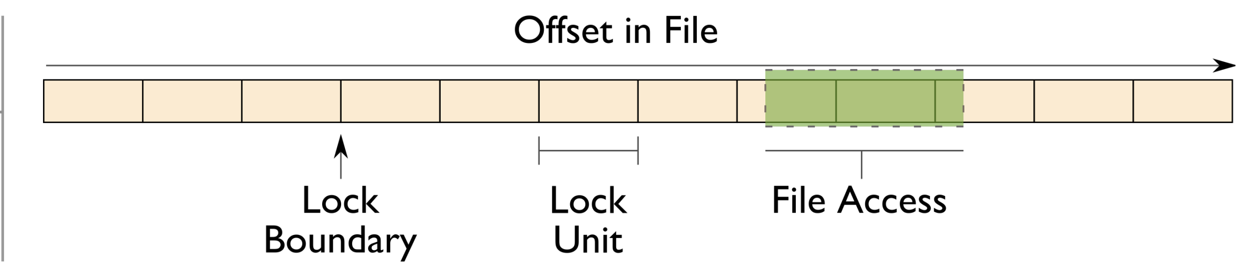
Slide Rob Ross, Rob Latham at ANL

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.

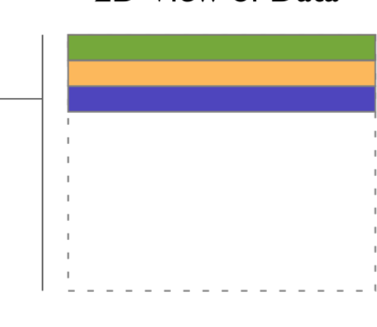


Slide from Rob Ross, Rob Latham at ANL

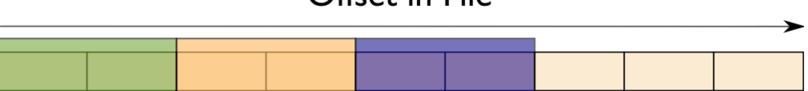
Locking and Concurrent Access

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.

2D View of Data

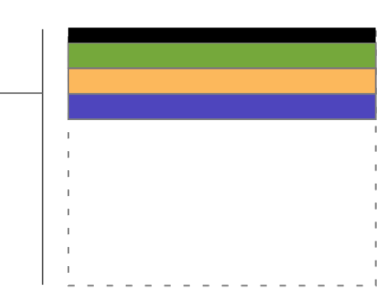


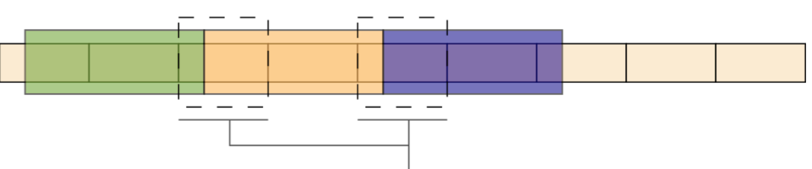
Offset in File



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

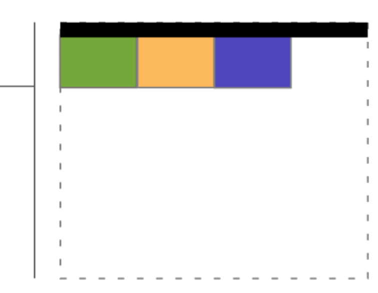
In this example a header (black) has been prepended to the data. If the header is not aligned with lock boundaries, false sharing will occur.

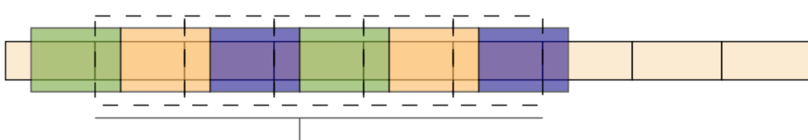




These two regions exhibit *false sharing*: no bytes are accessed by both processes, but because each block is accessed by more than one process, there is contention for locks.

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.

Slide from Rob Ross, Rob Latham at ANL

3D (reversing the decomp)

Logical

Physical

Slide from John Shalf

3D (block alignment issues)

Logical

Physical

720 bytes	720 bytes			

8192 bytes

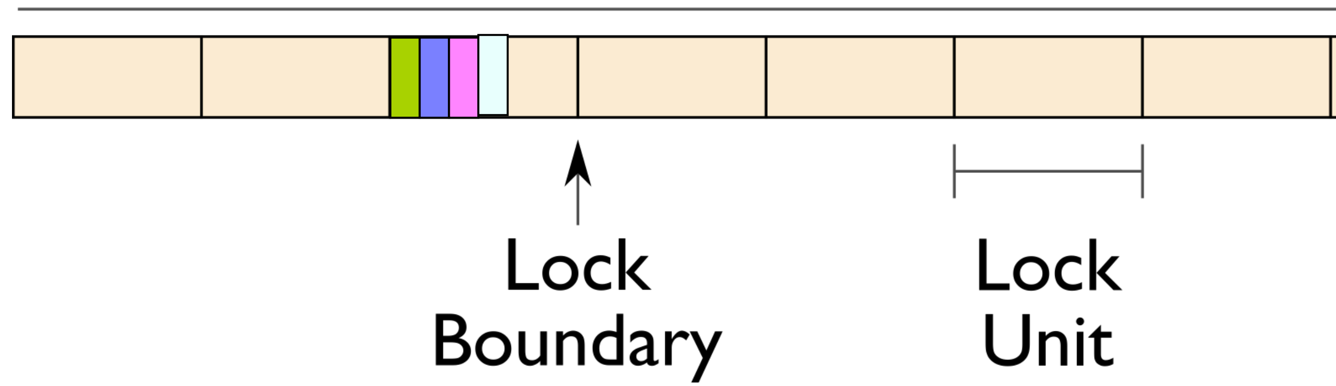
Writes not aligned to block boundaries

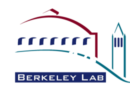
•Block updates require mutual exclusion
•Block thrashing on distributed FS
•I/O efficiency for sparse updates! (8k block required for 720 byte I/O operation
•Unaligned block accesses can kill performance! (but are necessary in practical I/O solutions)

Slide from John Shalf

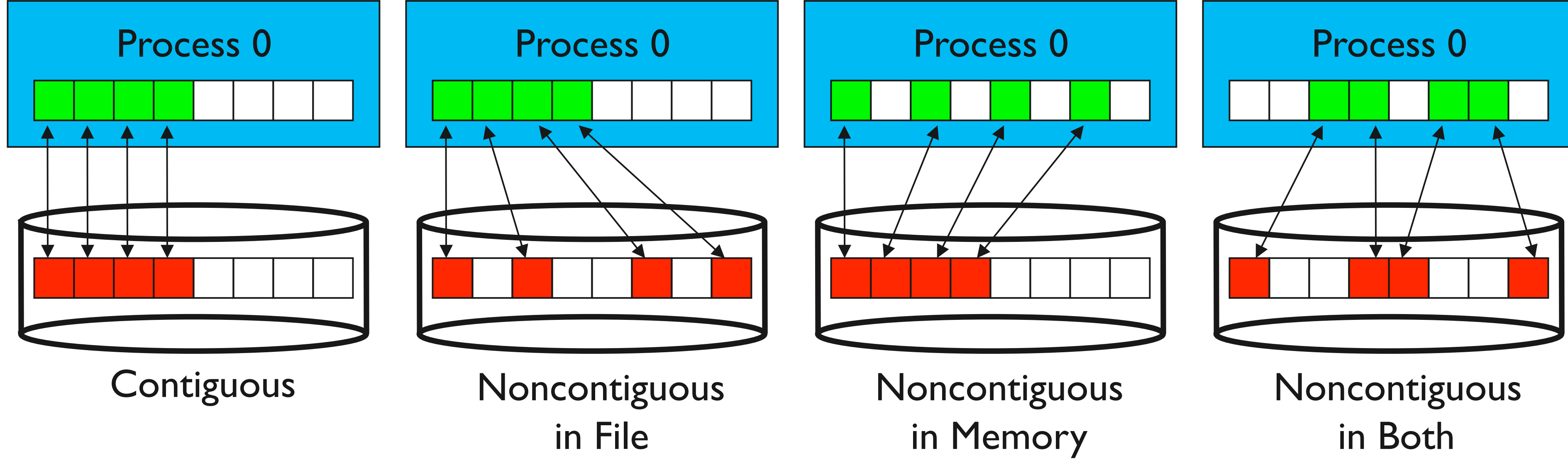
Small Writes

How will the parallel file system perform with small writes (less than the size of a lock unit)?





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**

Slide from Rob Ross, Rob Latham at ANL

Stressing the I/O System

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

Serial I/O

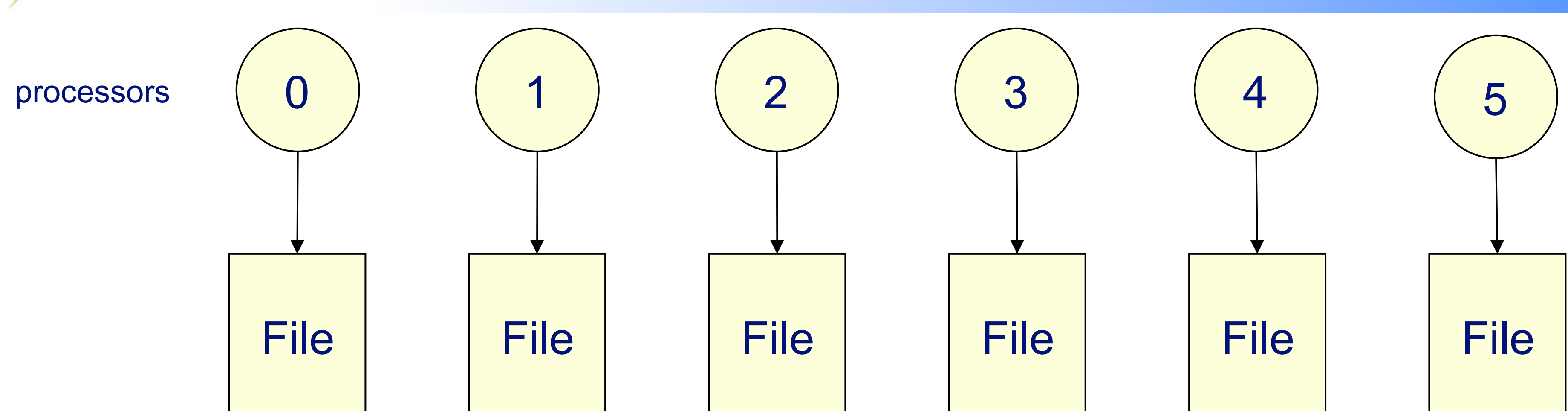
processors

0 1 2 3 4 5

File

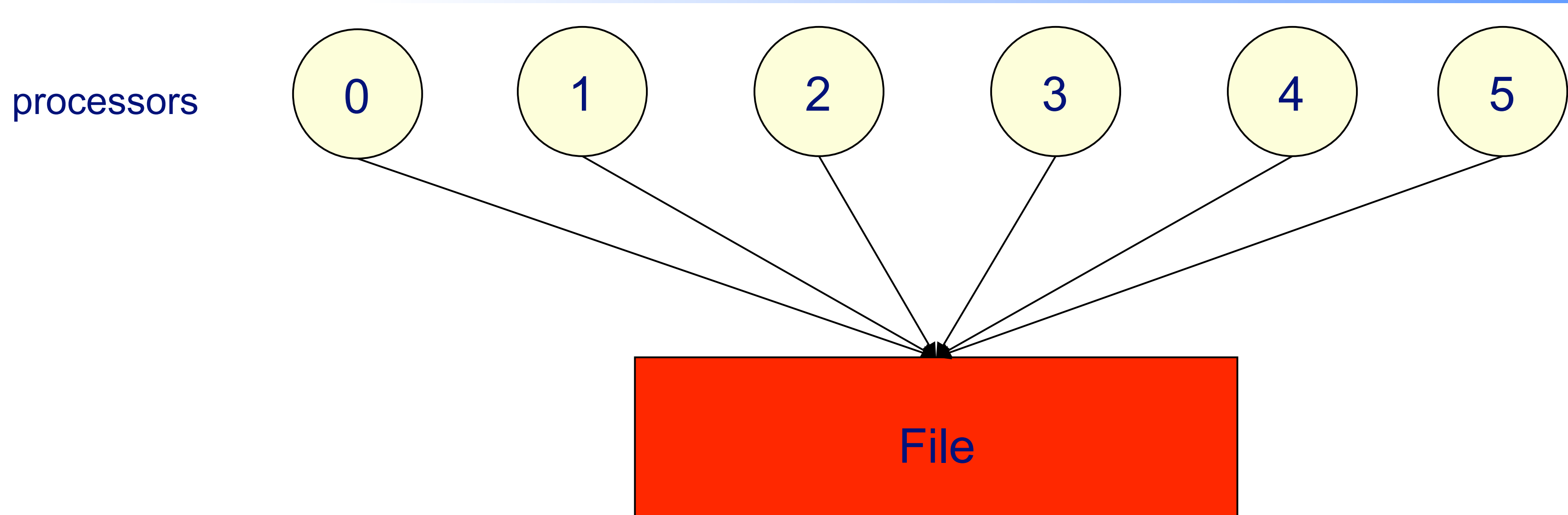
- **Each processor sends its data to the master who then writes the data to a file**
- **Advantages**
  - **Simple**
  - **May perform ok for very small IO sizes**
- **Disadvantages**
  - **Not scalable**
  - **Not efficient, slow for any large number of processors or data sizes**
  - **May not be possible if memory constrained**

Parallel I/O Multi-file



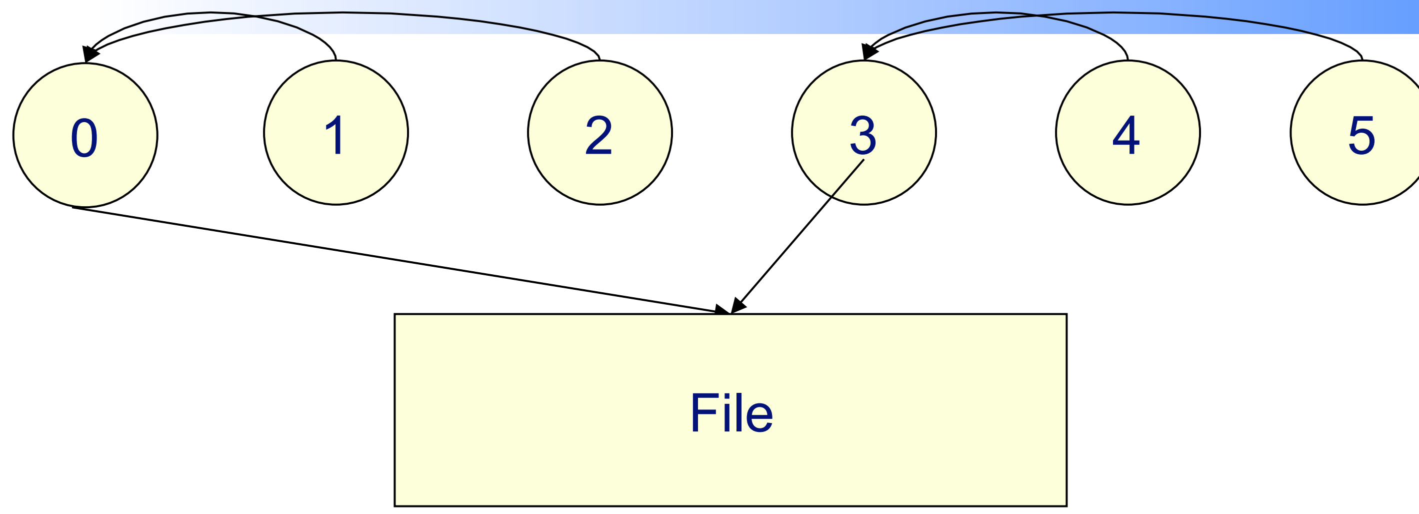
- Each processor writes its own data to a separate file
- Advantages
  - Simple to program
  - Can be fast -- (up to a point)
- Disadvantages
  - Can quickly accumulate many files
  - Hard to manage
  - Requires post processing
  - Difficult for storage systems, HPSS, to handle many small files
  - Can overwhelm the file system with many writers

Parallel I/O Single-file



- Each processor writes its own data to the same file using MPI-IO mapping
- Advantages
  - Single file
  - Manageable data
- Disadvantages
  - Shared files may not perform as well as one-file-per-processor models

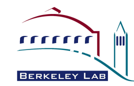
Reduced Writers to Single-file



- Best performance when # of writers is multiple of (1-4) # of IO nodes
- Subset of processors writes data to single file
- Advantages
  - Single file; manageable data
  - Better performance than all tasks writing for high concurrency jobs
- Disadvantages
  - This is a pain to program
  - User shouldn't have to do this!

Users don't need to do this at the application layer

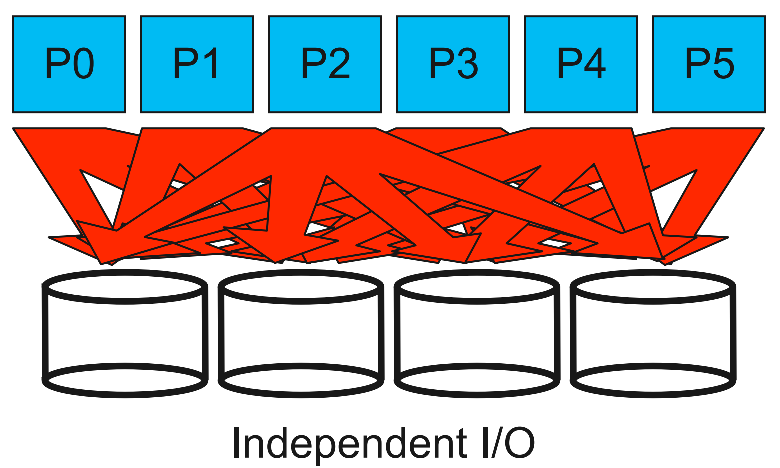
MPI-IO

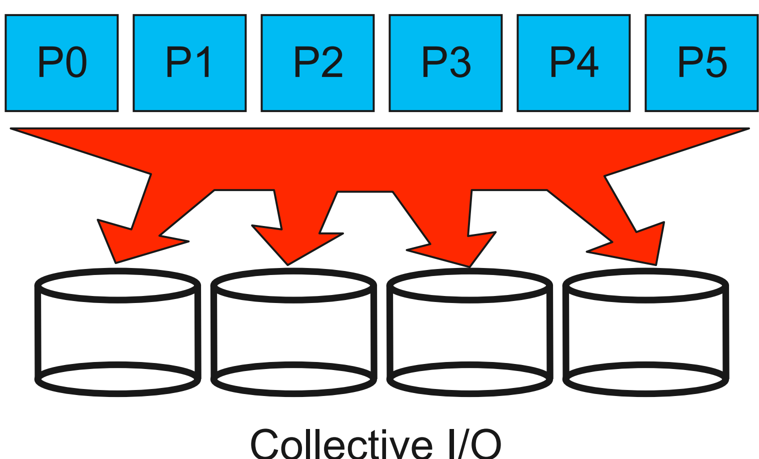


What is MPI-IO?

- Parallel I/O interface for MPI programs
- Allows users to write shared files with a simple interface
- Supports:
  - Derived data types
  - Collective I/O
  - Views

Independent and Collective I/O



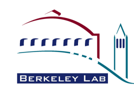


- Independent I/O operations specify only what a single process will do
  - Independent I/O calls obscure relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
- Collective I/O is coordinated access to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

Slide from Rob Ross, Rob Latham at ANL

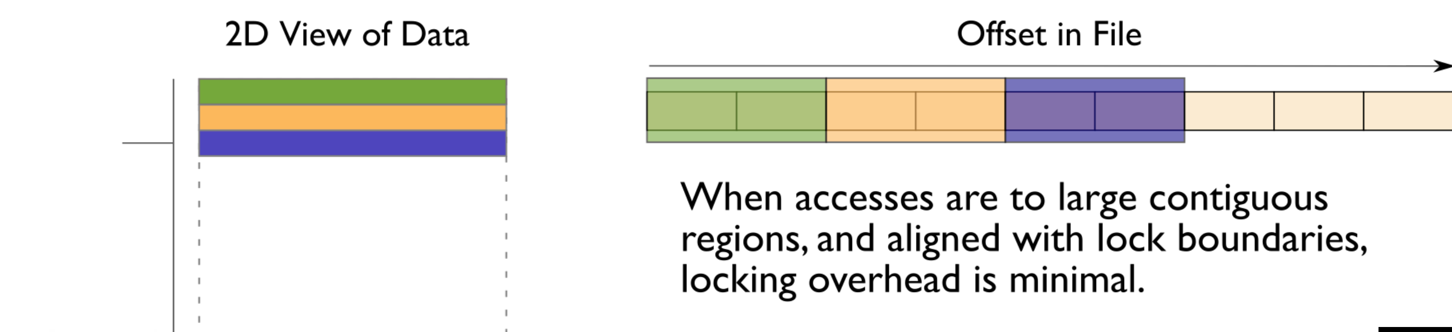
MPI-IO Optimizations

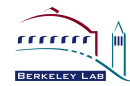
- Collective Buffering
  - Consolidates I/O requests from procs
  - Only a subset of procs (called aggregators) write to the file
  - Key point is to limit writers so that procs are not competing for same I/O block of data
  - Various algorithms exist for aligning data to block boundaries
  - Collective buffering is controlled by MPI-IO hints: romio_cb_read, romio_cb_write, cb_buffer_size, cb_nodes, cb_config_list



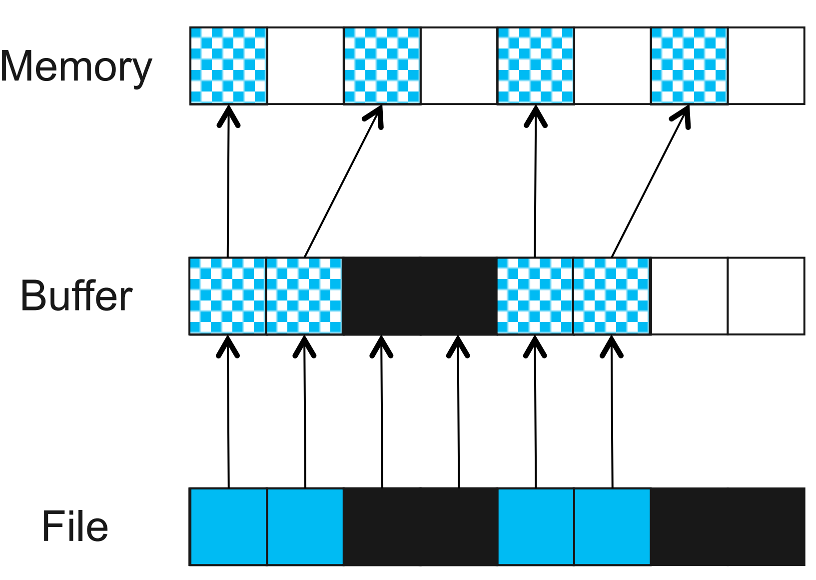
When to use collective buffering

- The smaller the write, the more likely it is to benefit from collective buffering
- Large contiguous I/O will not benefit from collective buffering. (If write size is larger than I/O block then there will not be contention from multiple procs for that block.)
- Non-contiguous writes of any size will not see a benefit from collective buffering
- Set number of collective buffering nodes to multiple of I/O nodes





Noncontiguous I/O Optimization: Data Sieving

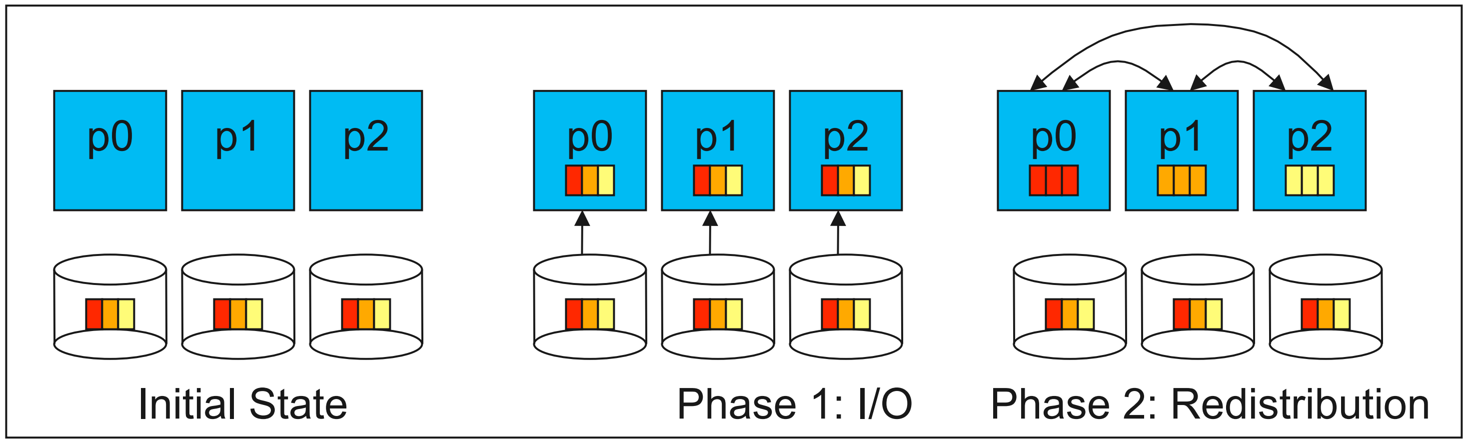


Data Sieving Read Transfers

- 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
- Trade off - read big data chunks, but need more memory

Slide material from Rob Ross, Rob Latham at ANL

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



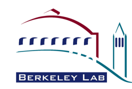
Two-Phase Read Algorithm

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

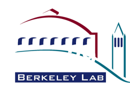
Slide from Rob Ross, Rob Latham at ANL

MPI-IO Summary

- MPI-IO is “middle ware” in the I/O stack
- Provides optimizations typically low performing I/O patterns (non-contiguous I/O and small block I/O)
- You could use MPI-IO directly, but better to use a high level I/O library



High Level Parallel I/O Libraries (HDF5 and Parallel-NetCDF)



What is a High Level Parallel I/O Library?

- An API which helps to express scientific simulation data in a more natural way
  - Multi-dimensional data, labels and tags, non-contiguous data, typed data
- Typically sits on top of MPI-IO layer and can use MPI-IO optimizations
- Offer
  - Simplicity for visualization and analysis
  - Portable formats - can run on one machine and take output to another
  - Longevity - output will last and be accessible with library tools and no need to remember version number of code

Common Storage Formats

- ASCII:
  - Slow
  - Takes more space!
  - Inaccurate
- Binary
  - Non-portable (eg. byte ordering and types sizes)
  - Not future proof
  - Parallel I/O using MPI-IO
- Self-Describing formats
  - NetCDF/HDF4, HDF5, Parallel NetCDF
  - Example in HDF5: API implements Object DB model in portable file
  - Parallel I/O using: pHDF5/pNetCDF (hides MPI-IO)
- Community File Formats
  - FITS, HDF-EOS, SAF, PDB, Plot3D
  - Modern Implementations built on top of HDF, NetCDF, or other self-describing object-model API

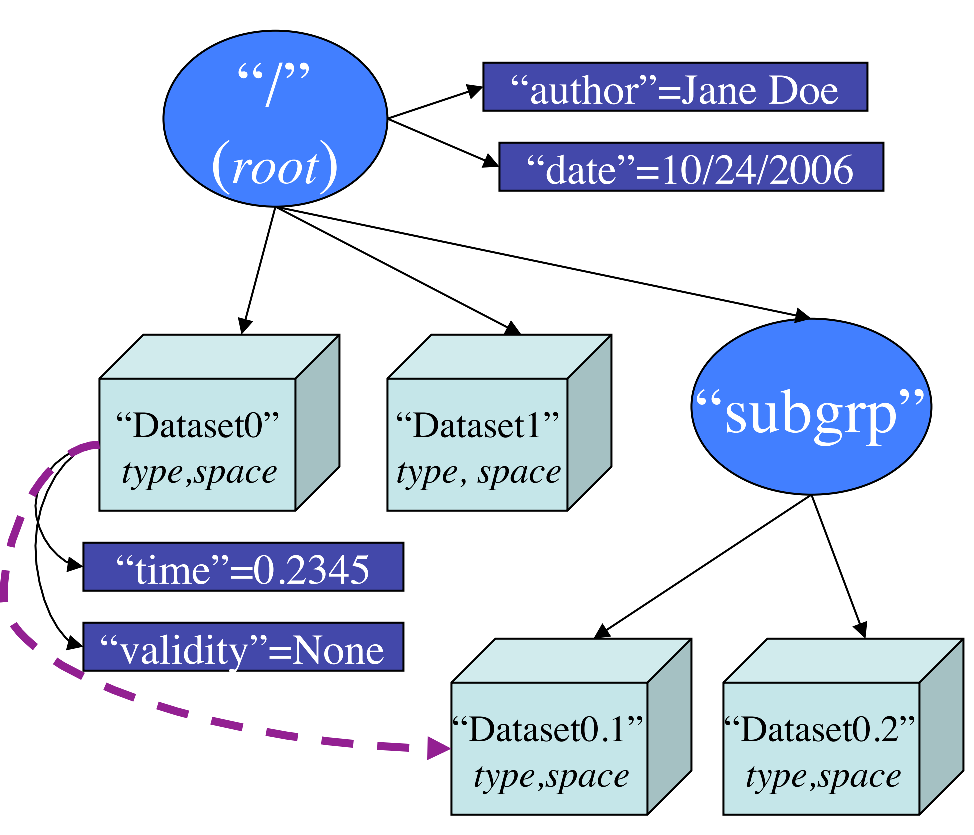
Many NERSC users at this level. We would like to encourage users to transition to a higher IO library

But what about performance?

- Hand tuned I/O for a particular application and architecture will likely perform better, but …
- Purpose of I/O libraries is not only portability, longevity, simplicity, but productivity
- Using own binary file format forces user to understand layers below the application to get optimal IO performance
- Every time code is ported to a new machine or underlying file system is changed or upgraded, user is required to make changes to improve IO performance
- Let other people do the work
  - HDF5/PnetCDF can be optimized for given platforms and file systems by library developers
- Goal is for shared file performance to be 'close enough'

HDF5 Data Model

- **Groups**
  - Arranged in directory hierarchy
  - root group is always '/'
- **Datasets**
  - Dataspace
  - Datatype
- **Attributes**
  - Bind to Group & Dataset
- **References**
  - Similar to softlinks
  - Can also be subsets of data



Example HDF5 file output

```
HDF5 "example_file.h5" {
GROUP "/" {
  DATASET "hamiltonian_000" {
    DATATYPE  H5T_IEEE_F64LE
    DATASPACE  SIMPLE { ( 10 ) / ( 10 ) }
    DATA {
    (0): 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
    }
  }
  DATASET "hamiltonian_001" {
    DATATYPE  H5T_IEEE_F64LE
    DATASPACE  SIMPLE { ( 10 ) / ( 10 ) }
    DATA {
    (0): 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
    }
  }
  DATASET "hamiltonian_002" {
    DATATYPE  H5T_IEEE_F64LE
    DATASPACE  SIMPLE { ( 10 ) / ( 10 ) }
    DATA {
    (0): 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
    }
  }
}
```

The HDF Group

- HDF5 is maintained by a non-profit company called the *HDF Group*
- Example code and documentation can be found here:
- http://www.hdfgroup.org/HDF5/

Parallel NetCDF Library

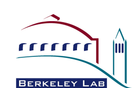
- Parallel implementation of netCDF storage format from Unidata
- Can read netCDF files
- Like HDF5 can store complex data structures, arrays, vectors, grids, text
- Built to take advantage of MPI-IO optimizations
- Portable Data Format

Parallel NetCDF Support

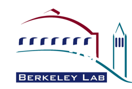
- Parallel NetCDF is maintained by a group at Argonne National Lab
- More information, code examples and documentation can be found here:
- http://trac.mcs.anl.gov/projects/parallel-netcdf

Recommendations

- Think about the big picture
  - Run time vs Post Processing trade off
  - Decide how much IO overhead you can afford
  - Data Analysis
  - Portability
  - Longevity
    - H5dump works on all platforms
    - Can view an old file with h5dump
    - If you use your own binary format you must keep track of not only your file format version but the version of your file reader as well
  - Storability



File Striping on Lustre File System



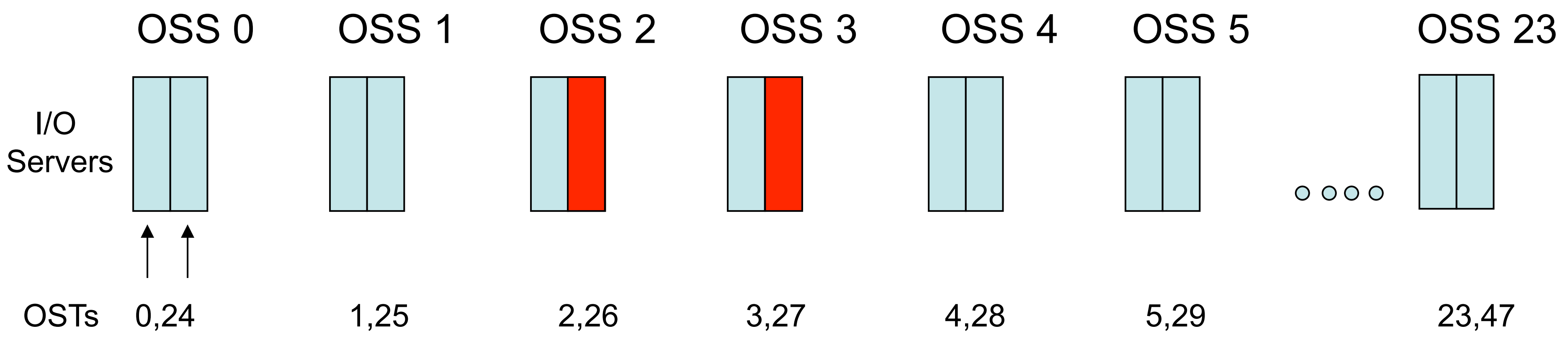
What is File Striping?

- Lustre file system on Franklin made up of an underlying set of parallel I/O servers
  - OSSs (Object Storage Servers) - nodes dedicated to I/O connected to high speed torus interconect
  - OSTs (Object Storage Targets) software abstraction of physical disk (1 OST maps to 1 LUN)
- File is said to be striped when read and write operations access multiple OSTs concurrently
- Striping can increase I/O performance since writing or reading from multiple OSTs simultaneously increases the available I/O bandwidth

Default Striping on Franklin /scratch

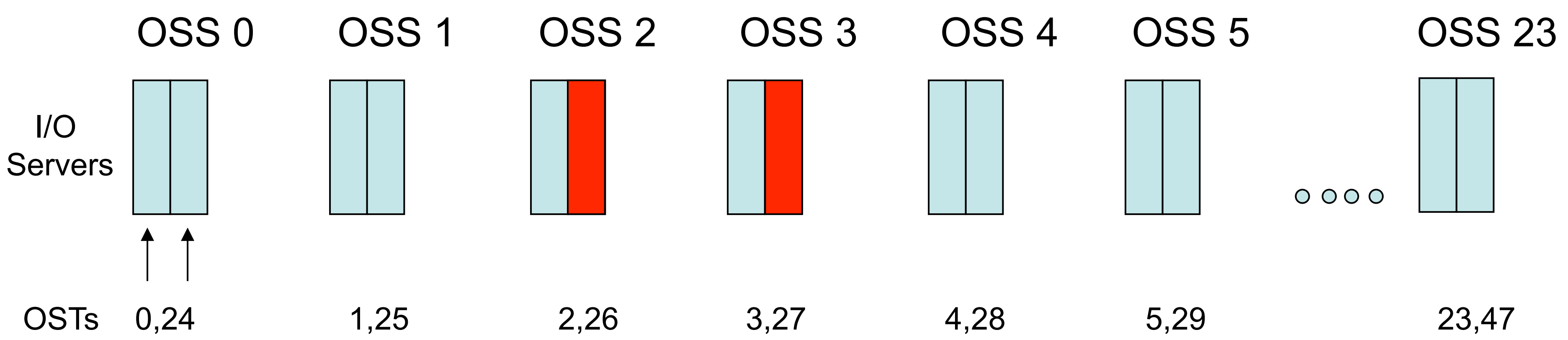
- 3 parameters characterize striping pattern of a file
  - Stripe count
    - Number of OSTs file is split across
    - Default is 2
  - Stripe size
    - Number of bytes to write on each OST before cycling to next OST
    - Default is 1MB
  - OST offset
    - Indicates starting OST
    - Default is round robin across all requests on system

Default Stripe Count of 2 on /scratch

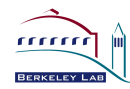


- **Pros**
  - **Get 2 times the bandwidth you could from using 1 OST**
  - **Max bandwidth to 1 OST ~ 350 MB/Sec**
  - **Using 2 OSTs ~700 MB/Sec**
- **Cons**
  - **For better or worse your file now is in 2 different places**
  - **Metadata operations like ‘ls -l’ on the file could be slower**

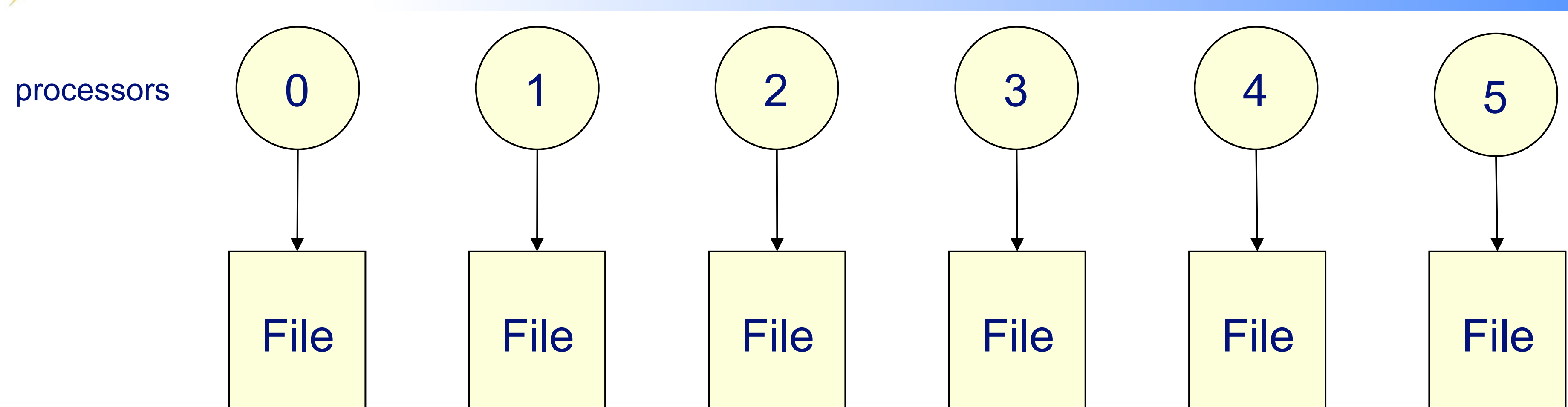
Why a stripe count of 2?



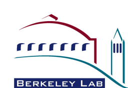
- **Balance**
  - **With a few important exceptions, should work decently for most users**
- **Protection**
  - **Each OST is backed up by a physical disk (LUN)**
  - **Stripe count of 1 leave us vulnerable to single user writing out huge amount of data filling the disk**
- **Striping of 2 is a reasonable compromise, although not good for large shared files**



Parallel I/O Multi-file



- Each processor writes its own data to a separate file
- Advantages
  - Simple to program
  - Can be fast -- (up to a point)
- Disadvantages
  - Can quickly accumulate many files
  - Hard to manage
  - Requires post processing
  - Difficult for storage systems, HPSS, to handle many small files
  - Can overwhelm the file system with many writers



One File-Per-Processor IO with Stripe Count of 1

0 1 2 3 4 5 38,000

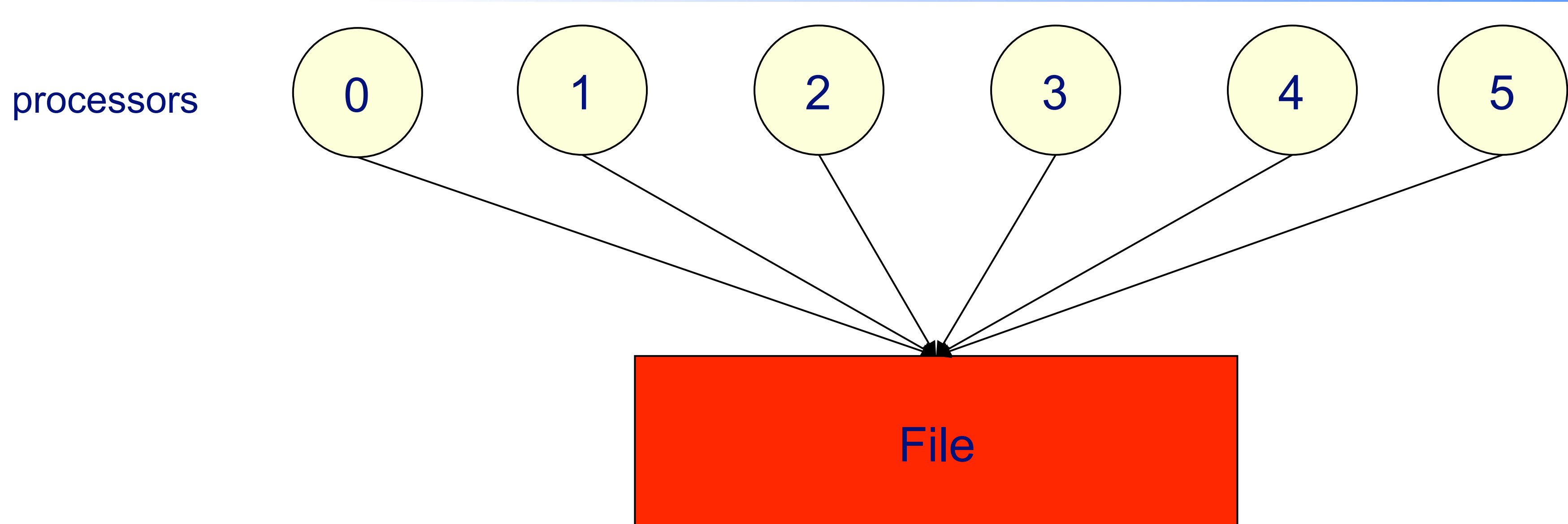
Torus Network

OSS 0 OSS 1 OSS 2 OSS 3 OSS 4 OSS 5 .... OSS 24

2 OSTs

- ***Use all OSTs but don’t add more contention than is necessary***

Parallel I/O Single-file



- Each processor writes its own data to the same file using MPI-IO mapping
- Advantages
  - Single file
  - Manageable data
- Disadvantages
  - Shared files may not perform as well as one-file-per-processor models

Shared File I/O with Default Stripe Count 2

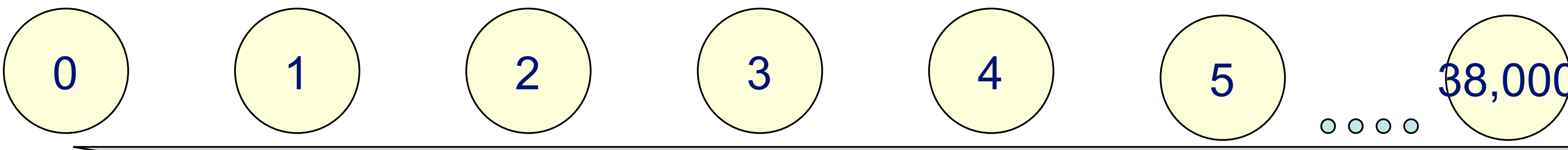
0 1 2 3 4 5 38,000

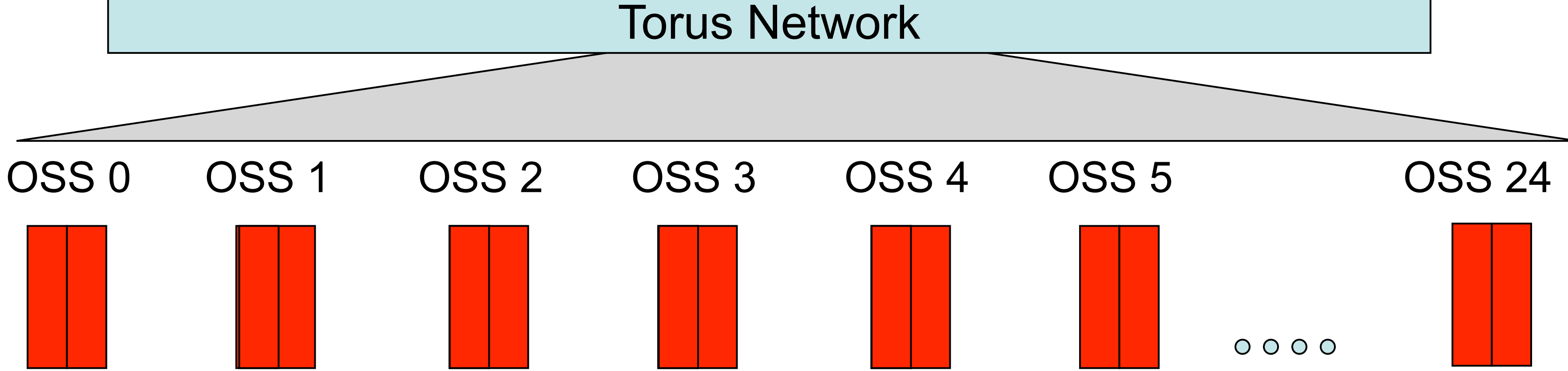
Torus Network

OSS 0 OSS 1 OSS 2 OSS 3 OSS 4 OSS 5 OSS 24

- *All processors writing shared file will write to 2 OSTs*
- *No matter how much data the application is writing, it won't get more than ~700 MB/sec (2 OSTs * 350 MB/Sec)*
- *Less sophisticated than you might think - no optimizations for matching processor writer to same OST*
- *Need to use more OSTs for large shared files*

Shared File I/O with Stripe Count 48





- *Now Striping over all 48 OSTs*
- *Increased available bandwidth to application*
  - *Theoretically (700 MB/Sec (OSS Max) * 20 OSSs)*
  - *In practice 11-12 GB/Sec*

Changing the Default Stripe Count

- A number of applications will see benefits from changing the default striping
- Striping can be set at a file or directory level
- When striping set on a directory: all files created in that directory with inherit striping set on the directory

lstripe <directory|file> <stripe size> <OST Offset> <stripe count>

lstripe mydirectory 0 -1 X

- Stripe size - # bytes written on each OST before cycling to next OST
- OST offset - indicates starting OST
- Stripe count - # of OSTs file is split across

Please contact consultants if you have low performing I/O. There may be something simple we can do to increase performance substantially

NERSC Striping Command Shortcuts

- Unfortunately users need to know about striping in order to get decent I/O performance for I/O intensive applications
- NERSC has tried to encapsulate messy details with 3 commands
- Usage >> stripe_large mydirectory

Size of File	Single File I/O	File Per Processor I/O
<1 GB	Do Nothing Use default striping	"stripe_fpp" or use default striping
1GB - 10 GB	"stripe_small"	"stripe_fpp" or use default striping
10GB - 100 GB	"stripe_med"	"stripe_fpp" or use default striping
100GB - 1TB+	"stripe_large"	Ask consultants

I/O on Hopper Phase 1

- Like Franklin, Hopper will use the Lustre file system
- Using LSI disks rather than DDN on Franklin, could be some performance differences
- However, performance recommendations remain the same
- Likely 2 scratch file systems with peak I/O rates of ~25GB/sec
- 2PB of disk

Best Practices

- Do large I/O: write fewer big chunks of data (1MB+) rather than small bursty I/O
- Do parallel I/O.
  - Serial I/O (single writer) can not take advantage of the system's parallel capabilities.
- Stripe large files over many OSTs.
- If job uses many cores, reduce the number of tasks performing IO
- Use a single, shared file instead of 1 file per writer, esp. at high parallel concurrency.
- Use an IO library API and write flexible, portable programs.

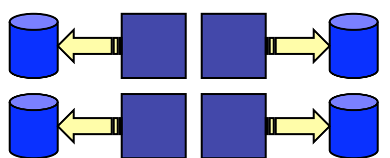
Questions?

Franklin Striping Summary

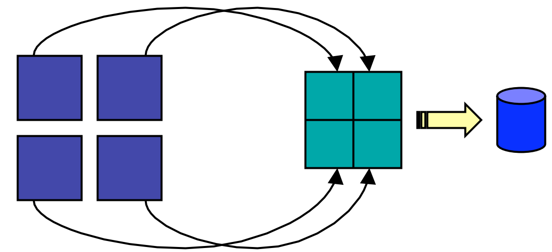
- Franklin Default Striping
  - Stripe size - 1MB (enter ‘0’ for default)
  - OST offset - round robin starting OST (enter ‘ -1’ for default)
  - Stripe over 2 OSTs (Stripe count 4)
- One File-Per-Processor
  - “lfs set stripe mydir 0 -1 1”
- Large shared files
  - “lfs setstripe mydir 0 -1 48”
- Medium shared files
  - Experiment a little 10-20 OSTs
  - “lfs setstripe mydir 0 -1 11”

Common Physical Layouts For Parallel I/O

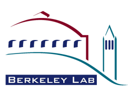
- One File Per Process
  - Terrible for HPSS!
  - Difficult to manage



- Parallel I/O into a single file
  - Raw MPI-IO
  - pHDF5 pNetCDF



- Chunking into a single file
  - Saves cost of reorganizing data
  - Depend on API to hide physical layout
  - (eg. expose user to logically contiguous array even though it is stored physically as domain-decomposed chunks)



Tutorial Outline

Walk through the I/O software stack in reverse order

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

HDF5, Parallel netCDF, ADIOS

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

IBM ciod

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

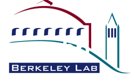
I/O Middleware
organizes accesses from many processes, especially those using collective I/O.

MPI-IO

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

PVFS, PanFS, GPFS, Lustre

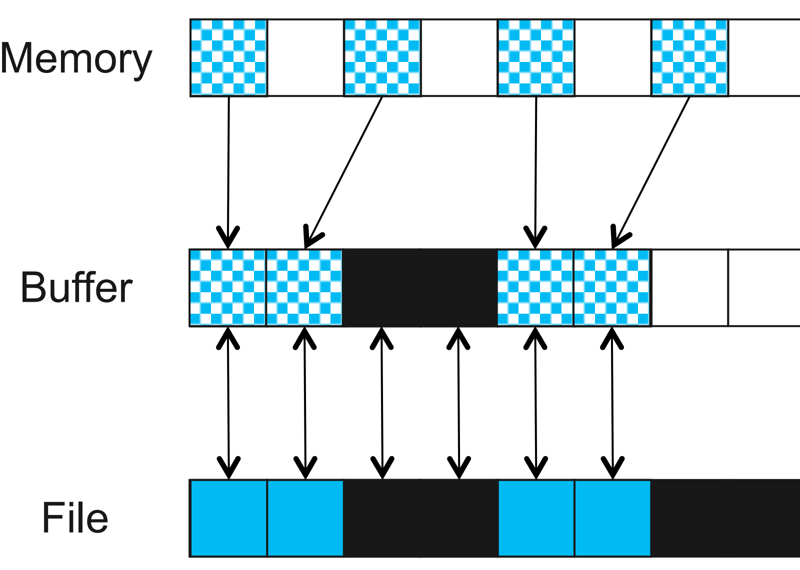
Slide from Rob Ross, Rob Latham at ANL



Parallel I/O:
A User Perspective

- **Wish List**
  - Write data from multiple processors into a single file
  - File can be read in the same manner regardless of the number of CPUs that read from or write to the file. (eg. want to see the logical data layout… not the physical layout)
  - Do so with the same performance as writing one-file-per-processor (only writing one-file-per-processor because of performance problems)
  - And make all of the above portable from one machine to the next

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

Slide from Rob Ross, Rob Latham at ANL