HPC I/O
I/O on Lustre

Richard Gerber
HPC Consultant
NERSC / Lawrence Berkeley National Laboratory

SC11 Tutorial

Scaling to Petascale and Beyond: Performance Analysis and Optimization of Applications

Nov. 13 2011

Thanks Katie Antypas of NERSC for some slides.

U.S. DEPARTMENT OF ENERGY | Office of Science

NERSC National Energy Research Scientific Computing Center

BERKELEY LAB Lawrence Berkeley National Laboratory

Tutorial Info & Presentations

http://www.nersc.gov/users/training/nersc-training-events/sc11/s10/

or

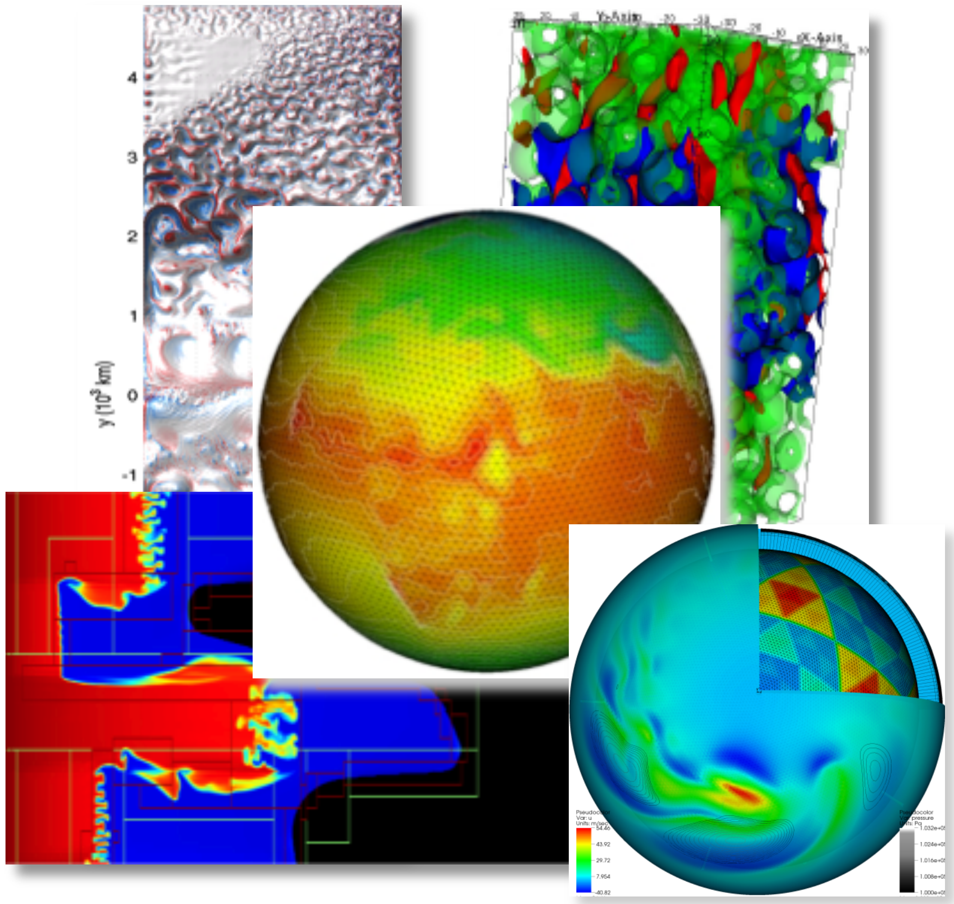
http://tinyurl.com/sc11-s10

Outline

- Storage Systems and Parallel File Systems
- High Level I/O Strategies
- Data Access Patterns
- Parallel I/O Interfaces
- I/O using Lustre File Systems
- 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
- Early 2011 – Hopper: 2 PB /scratch (we thought that was huge!)
- New systems at TACC and NCAR have ~ 18 PB / scratch!!!!



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

Should You Care About Architecture?

- Yes! It would be nice not to have to, but performance and perhaps functionality depend on it.
- You may be able to make simple changes to the code or runtime environment that make a big difference.
- Inconvenient Truth: *Scientists need to understand their I/O in order to get good* ^ *performance*

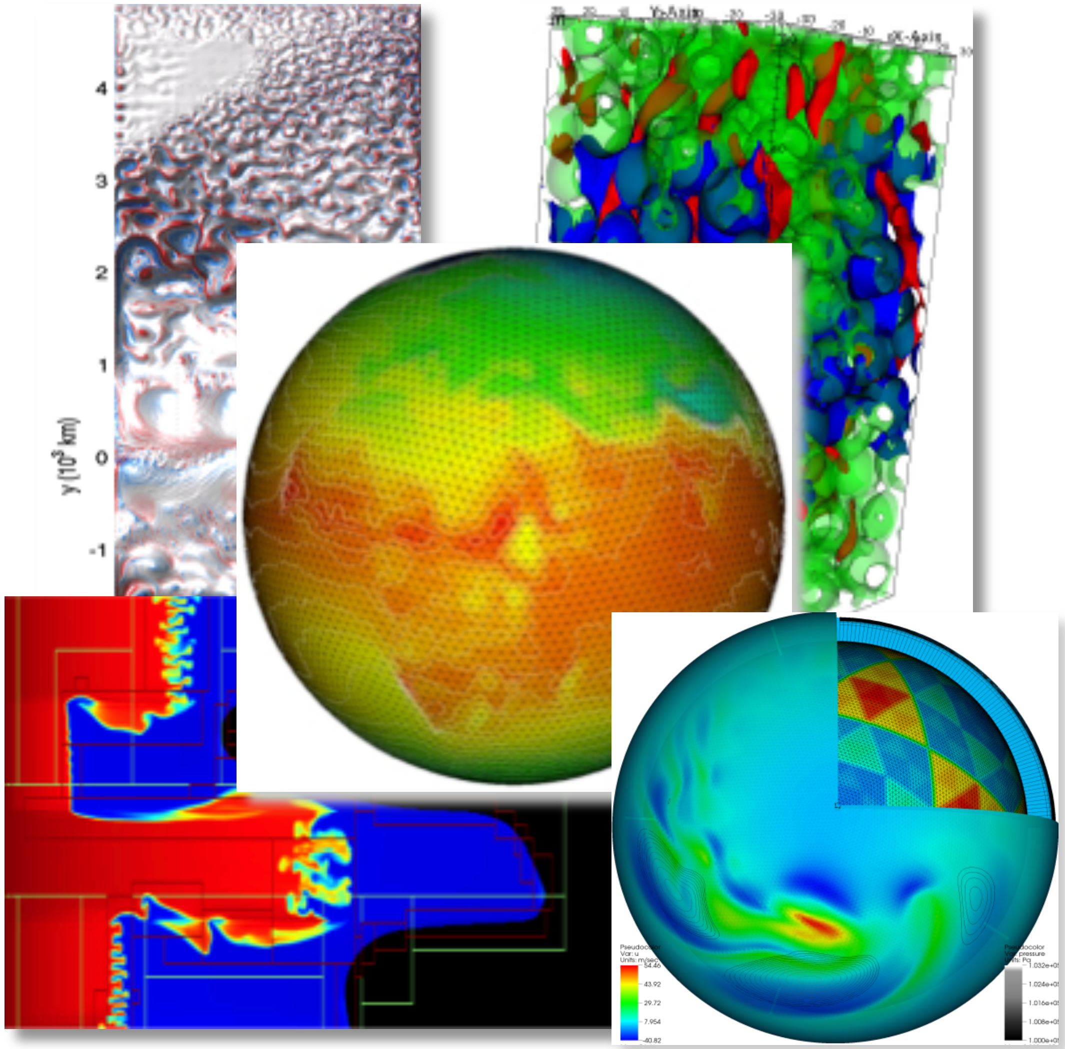
or acceptable

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB Lawrence Berkeley National Laboratory

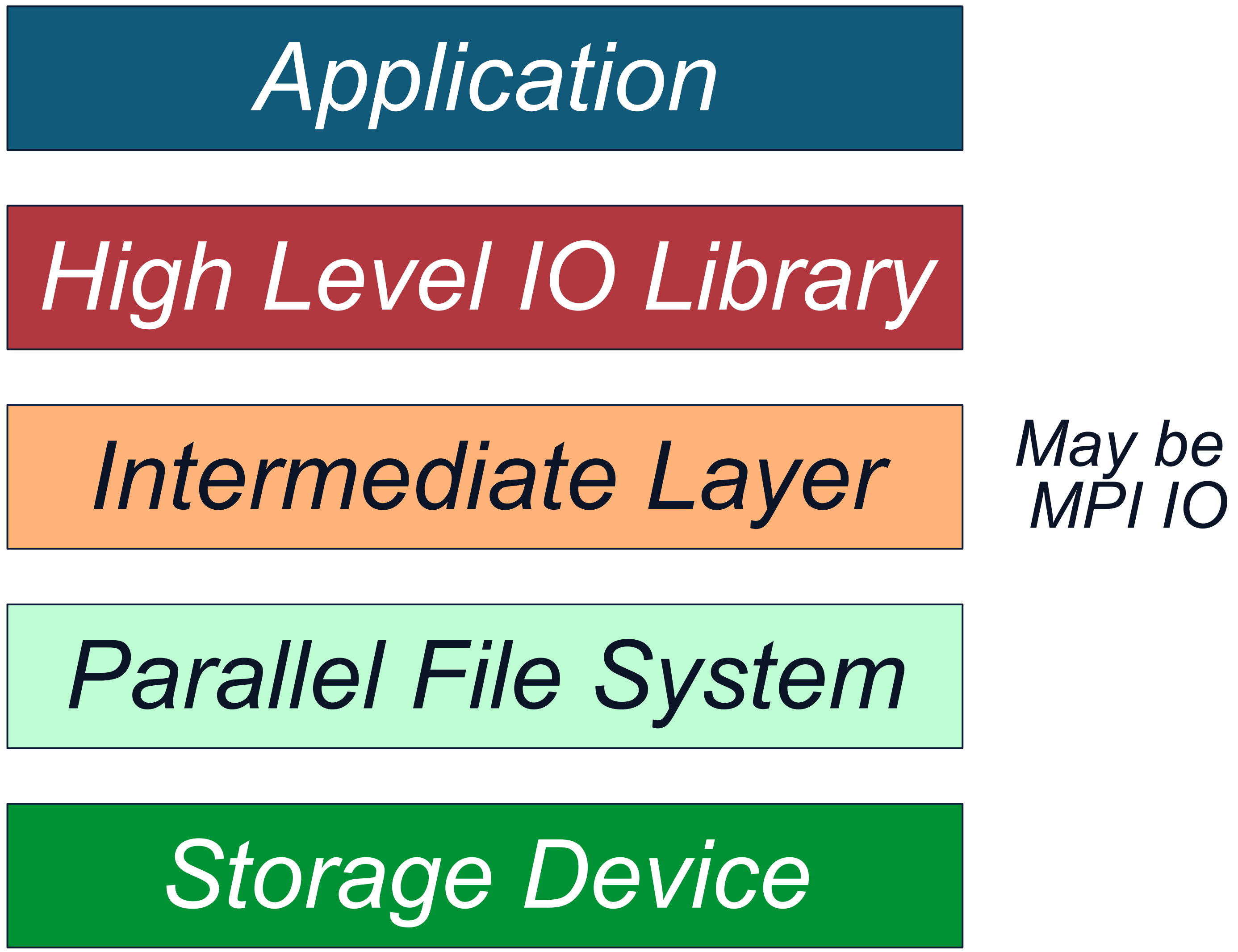
Why is Parallel I/O for science applications difficult?

- Scientists think about data in terms of how a system is represented in the code: as grid cells, particles, …
- Ultimately, data is stored on a physical device
- Layers in between the application and the device are complex and varied
- I/O interfaces and configurations are arcane and complicated



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

Simplified I/O Hierarchy



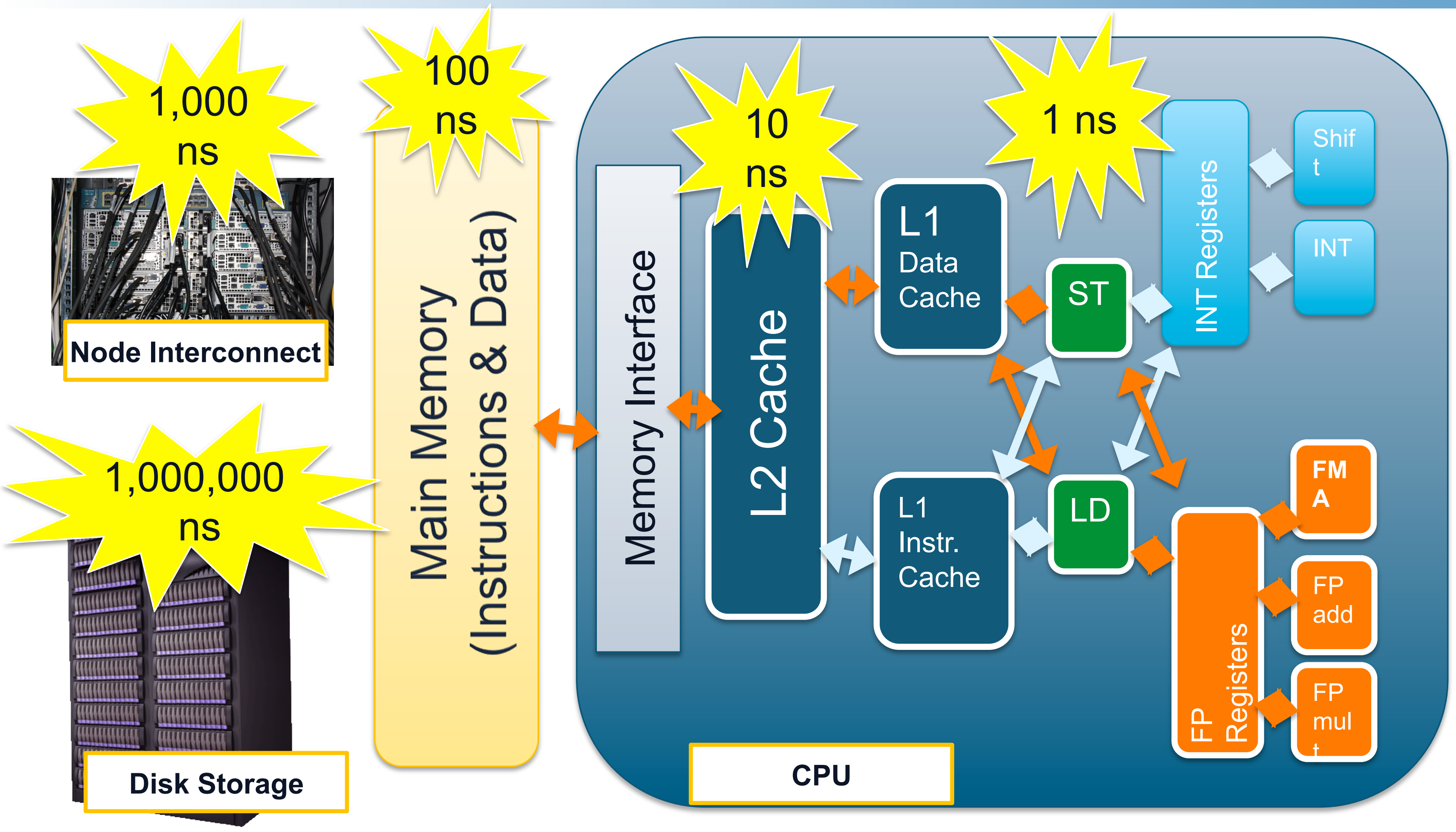
Storage Devices

- Usually we'll be talking about arrays of hard disks
- FLASH "drives" are being used as fast "disks," but are expensive
- Magnetic tapes are cheap, but slow and probably don't appear as standard file systems

Some Definitions

- **Capacity (in MB, GB, TB, PB)**
  - Depends on area available on storage device and the density data can be written
- **Transfer Rate (bandwidth) – MB/sec or GB/sec**
  - Rate at which a device reads or writes data
  - Depends on many factors: network interfaces, disk speed, etc.
  - Be careful with parallel BW numbers: aggregate? per what?
- **Access Time (latency)**
  - Delay before the first byte is read
- **Metadata**
  - A description of where and how a file or directory is stored on physical media
  - Is itself data that takes up space and has to be read/written with each file access
  - May be in a database

Latencies

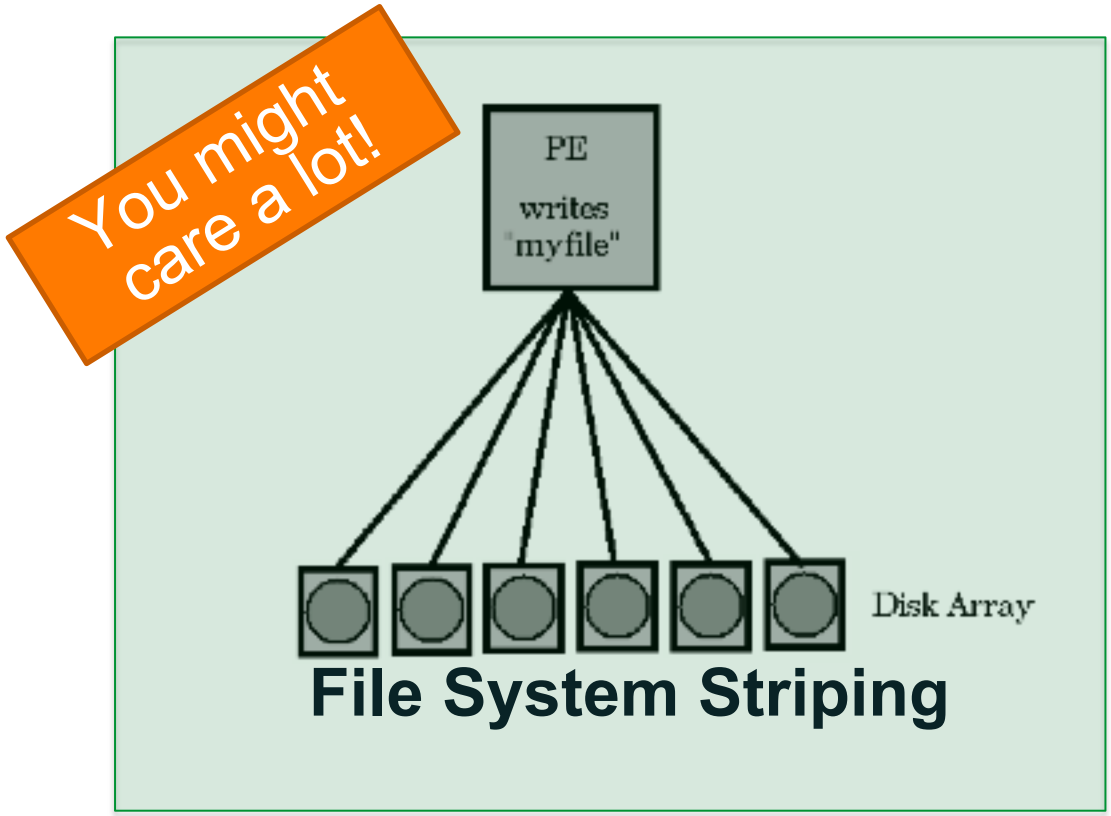

1,000 ns
Node Interconnect
1,000,000 ns
Disk Storage
100 ns
Main Memory (Instructions & Data)
10 ns
Memory Interface
L2 Cache
L1 Data Cache
L1 Instr. Cache
1 ns
ST
LD
INT Registers
Shift
INT
FP Registers
FMA
FP add
FP mult
CPU

Bandwidths

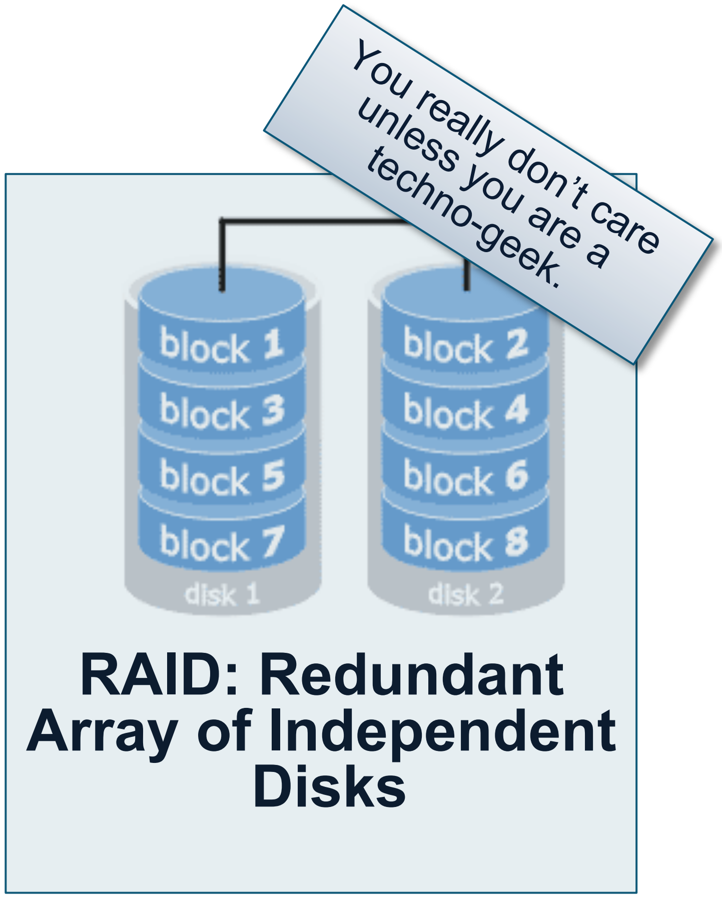
- How fast can you stream data from your application to/from disk?
- System aggregate bandwidths ~ 10s to now 100s GB/sec
- Serial bandwidths < 1 GB/sec
  - Limited by interfaces
  - and/or physical media
- The need for parallelism starts at the lowest level

Disk Parallelism

- Individual disk drives too slow for supercomputers
- Need parallelism



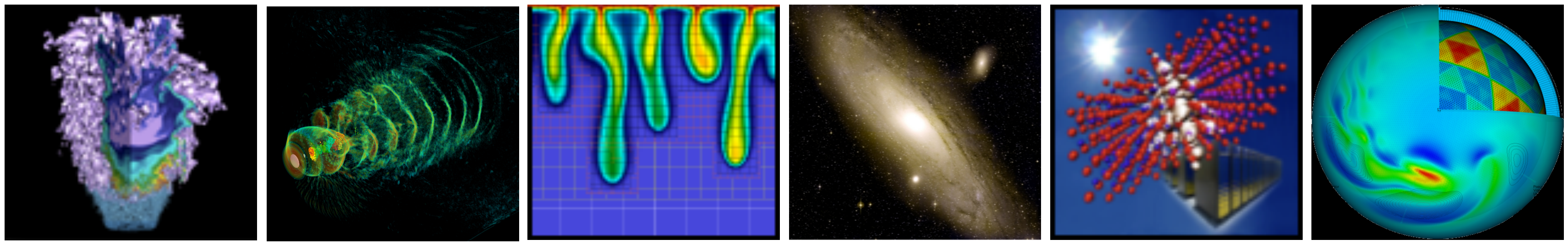
File System Striping



RAID: Redundant Array of Independent Disks

U.S. DEPARTMENT OF ENERGY | Office of Science

BERKELEY LAB Lawrence Berkeley National Laboratory



File Systems

What is a File System?

- Software layer between the Operating System and Storage Device which creates abstractions for
  - Files
  - Directories
  - Access permissions
  - File pointers
  - File descriptors
- Mediates moving data between memory and storage devices
- Coordinates concurrent access to files
- Manages the allocation and deletion of data blocks on the storage devices
- Has facilities for data recovery (not user accessible)

Local vs. Global File Systems

- **"On-board" (the old "local")**
  - Directly attached to motherboard via some interface
  - Few HPC systems have disks directly attached to a node
- **"Local" in HPC: Access from one system**
  - Network attached TB+ file systems
    - Via high-speed internal network (e.g. IB)
    - Direct from node via high-speed custom network (e.g. FibreChannel)
    - Ethernet
  - Contention among jobs on system
- **"Global": Access from multiple systems**
  - Networked file system
  - Activity on other systems can impact performance
  - Useful for avoiding data replication, movement among systems

What is a Networked File System

- **A file system that supports sharing of files as persistent storage over a network.**
- **Network File System (protocol) (NFS)**
  - Widely used and available, but not developed as a standard for high-performance parallel computing
  - Common for /home directories
  - Used for file systems that need high reliability, but not high performance
- **Other examples: AFS, NetWare Core Protocol, Server Message Block (SMB).**

Distributed Parallel Fault-Tolerant File Systems

- Networked
- Distributes data over multiple servers for high performance
- RAID for fault tolerance
- Efficiently manages up to 1,000s (?) of processors accessing the same file concurrently
- Coordinates locking, caching, buffering and file pointer challenges
- Scalable and high performing
- May have *Object Storage Device*
  - Storage “device” layer at higher level than physical media or even arrays of low-level media
- May have centralized metadata server (database)

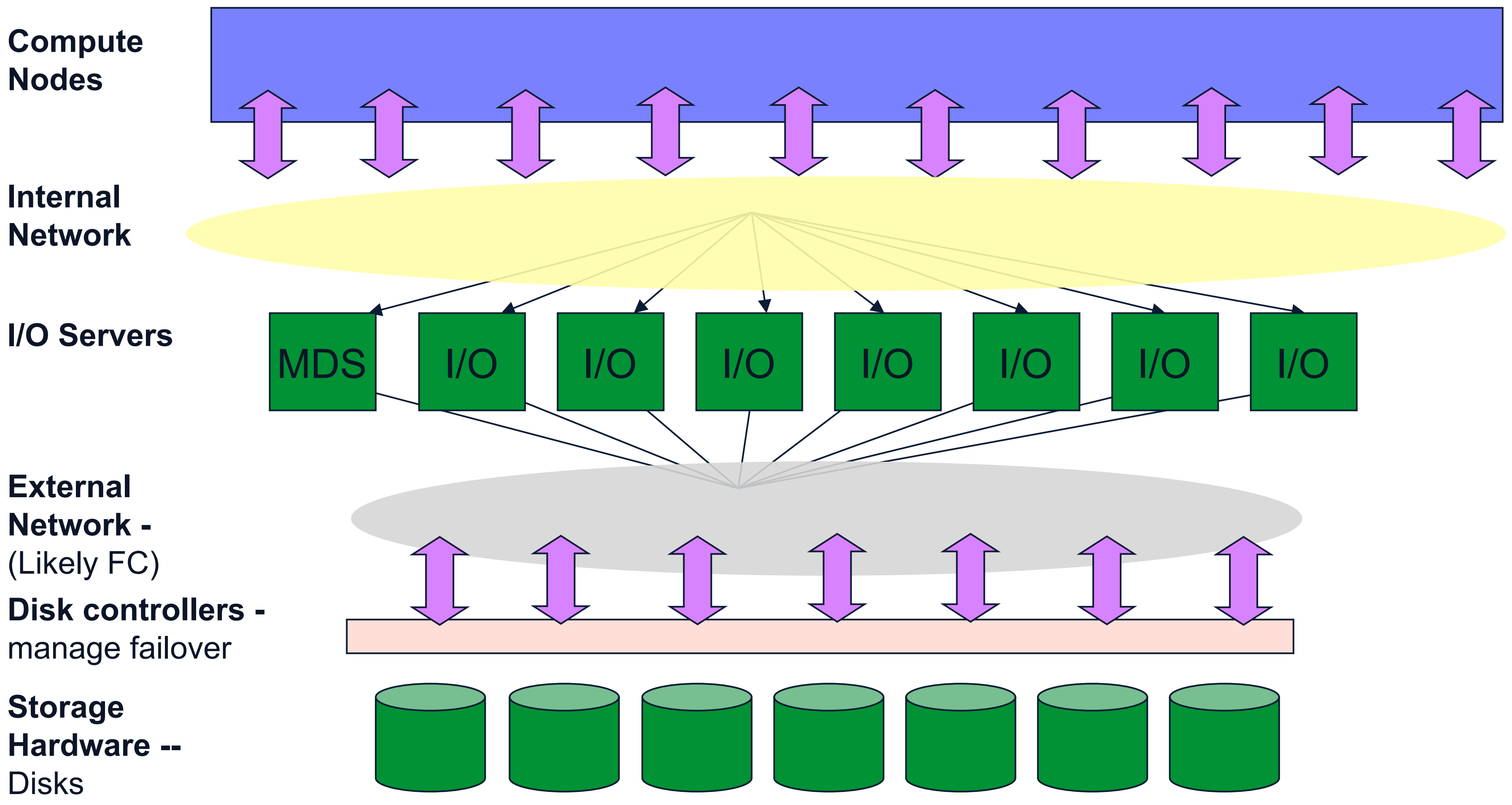
Top File Systems Used in HPC

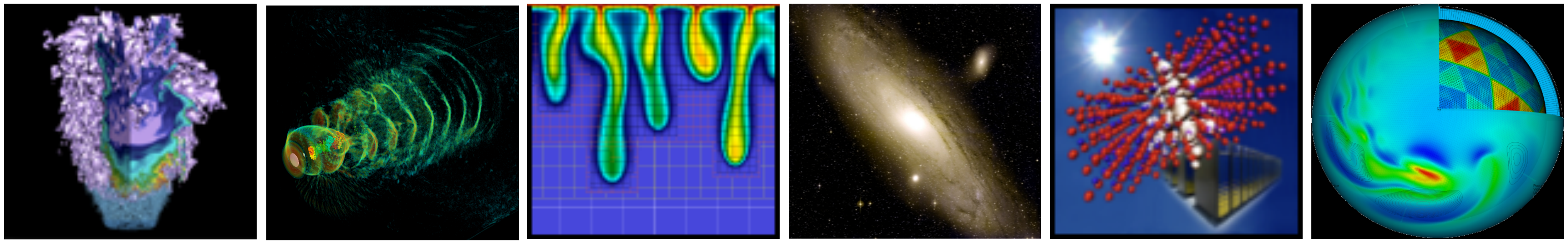
GPFS

More About Metadata

- 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, …
- Any time a file's attributes change or info is desired (e.g., ls –l) metadata has to be retrieved or written
  - Although there may be caching
- Metadata operations are IO operations (database queries) and inodes use 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



Now from the User's point of view

Some reasons you might need I/O

- **Checkpoint/Restart files**
  - System or node could fail; protect your application so you don't have to start from the beginning
  - Need to run longer than wall clock time allows
- **Write data for post run analysis and visualization**
- **You can use disk storage (large) as slow RAM memory (out-of-core algorithms)**
- **Reading in large datasets for analysis or visualization**

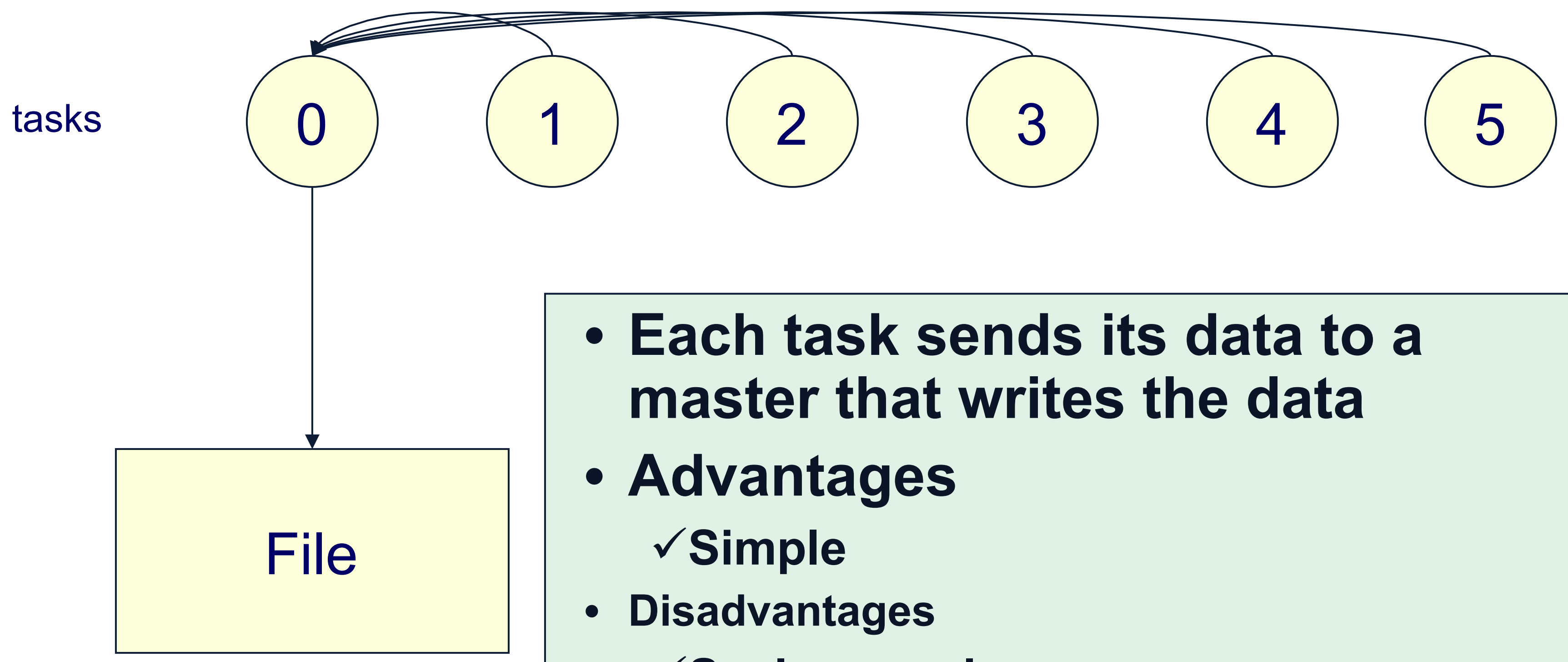
Application I/O

- All I/O performed by your job should use the file system designed for HPC applications.
- Home directories are often not optimized for large I/O performance
- Consult your center's documentation

High Level IO Strategies

- Single task does all IO
- Each task writes to its own file
- All tasks write to single shared file
- n<N tasks write to a single file
- n_1<N tasks write to n_2<N files

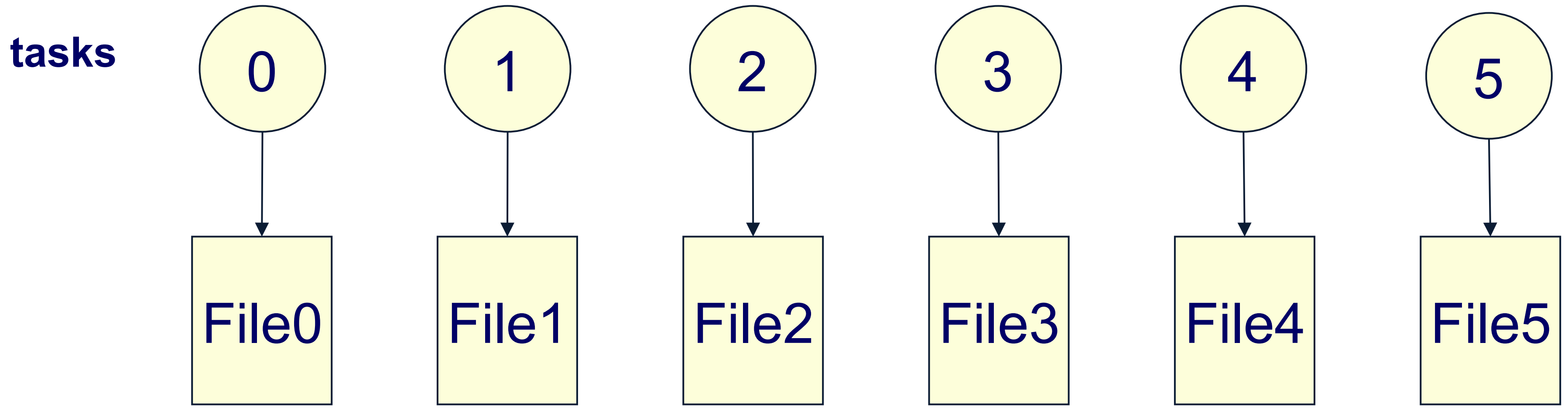
Serial I/O



- **Each task sends its data to a master that writes the data**
- **Advantages**
  - ✓**Simple**
- **Disadvantages**
  - ✓**Scales poorly**
  - ✓**May not fit into memory on task 0**
  - ✓**Bandwidth from 1 task is very limited**

Parallel I/O Multi-file

Each Processors Writes Its Data to Separate File



Advantages

Easy to program

Can be fast

Disadvantages

Many files can cause serious performance problems

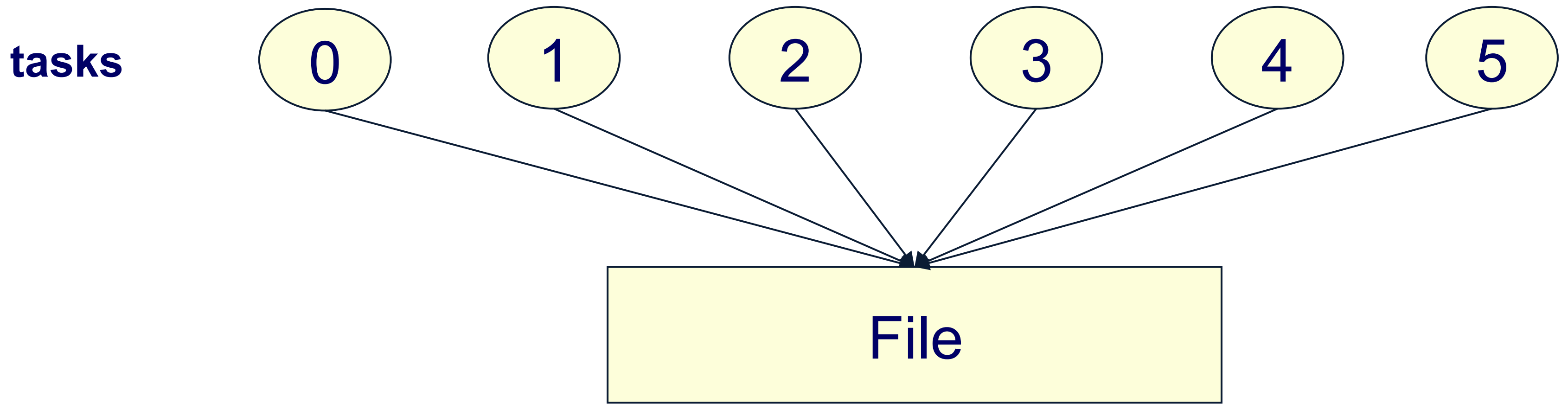
Hard for you to manage 10K, 100K or 1M files

Flash Center IO Nightmare…

- 32,000 processor run on LLNL BG/L
- Parallel IO libraries not yet available
- Every task wrote
  - Checkpoint files: .7 TB, every 4 hours, 200 total
  - Plot files: 20GB each, 700 files
  - Particle files: 470 MB each, 1,400 files
- Used 154 TB total
- Created 74 million files!
- UNIX utility problems (e.g., mv, ls, cp)
- It took 2 years to sift though data, sew files together

Parallel I/O Single-File

All Tasks to Single File



Advantages

Single file makes data manageable

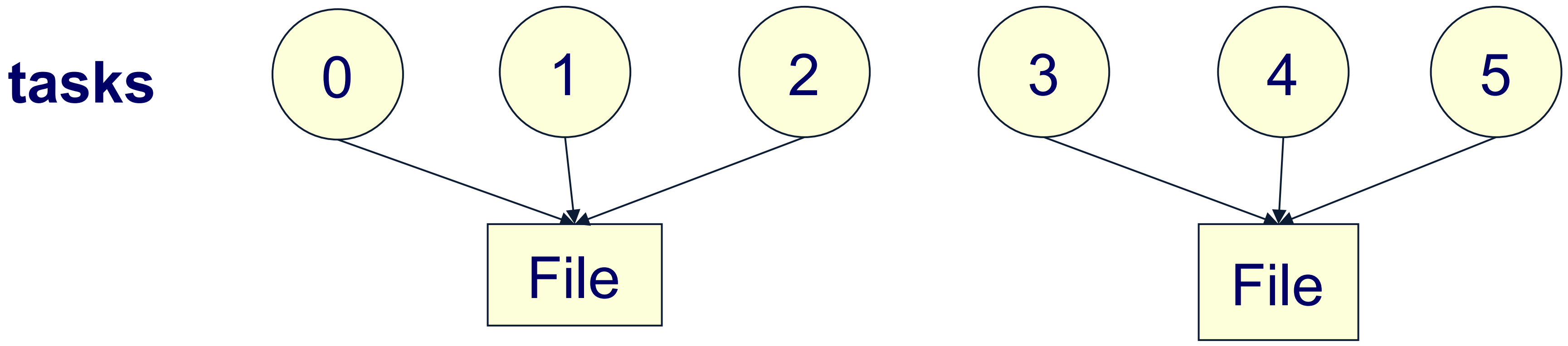
No system problems with excessive metadata

Disadvantages

Can be more difficult to program (use libs)

Performance may be less

Hybrid Model I
Groups of Tasks Access Different Files



Advantages

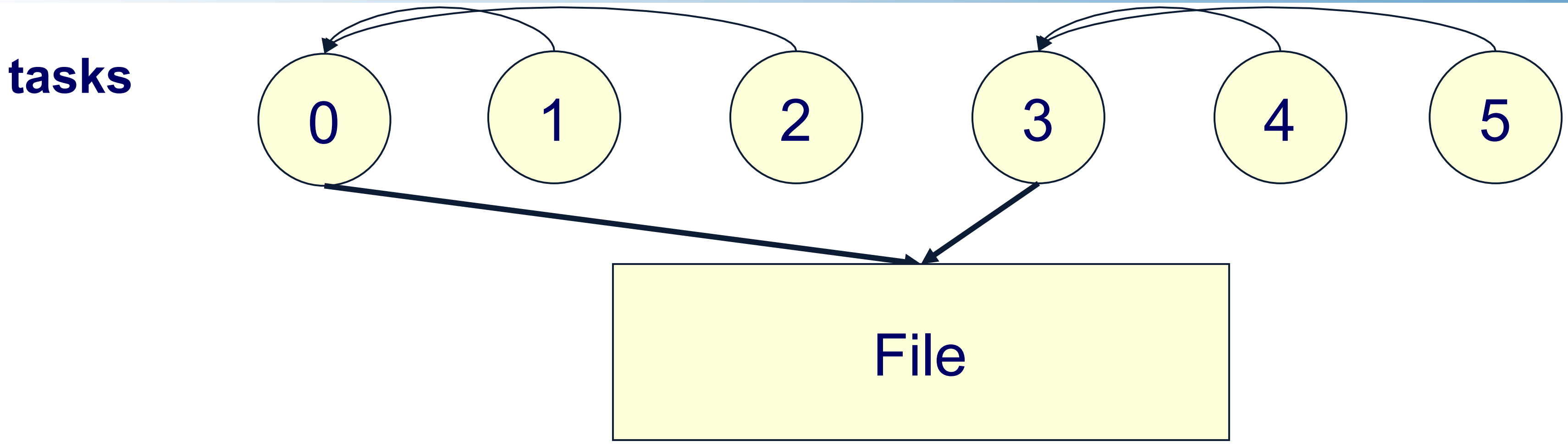
Fewer files than 1→1

Better performance than All→1

Disadvantages

Algorithmically complex

Hybrid II
Subset of Tasks Access Single File



Advantages

Single file makes data manageable

No system problems with excessive metadata

Disadvantages

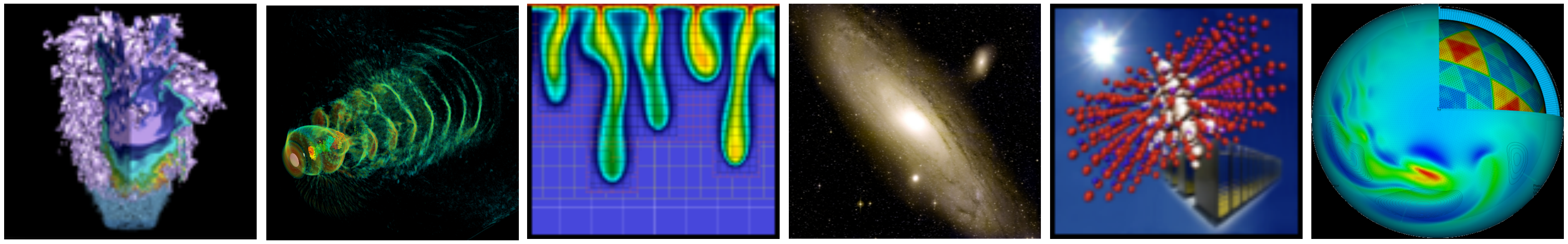
Can be more difficult to program (use libs)

Performance better than previous, but worse than 1->1

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



MPI-IO

U.S. DEPARTMENT OF ENERGY | Office of Science

National Energy Research Scientific Computing Center

BERKELEY LAB Lawrence Berkeley National Laboratory

What is MPI-IO?

- **Parallel I/O interface for MPI programs**
- **Part of the MPI standard (ubiquitous)**
- **Allows access to shared files using a standard API that is optimized and safe**
- **Key concepts:**
  - MPI communicators
    - open()s and close()s are collective within communicator
    - Only tasks in communicator can access file handle
  - Derived data types
    - All operations (e.g. read()) have an associated MPI data type
  - Collective I/O for optimization

Basic MPI IO Routines

- MPI_File_open() – associate a file with a file handle.
- MPI_File_seek() – move the current file position to a given location in the file.
- MPI_File_read() – read some fixed amount of data out of the file beginning at the current file position.
- MPI_File_write() – write some fixed amount of data into the file beginning at the current file position.
- MPI_File_sync() -- flush any caches associated with the file handle.
- MPI_File_close() – close the file handle.

MPI IO File Views

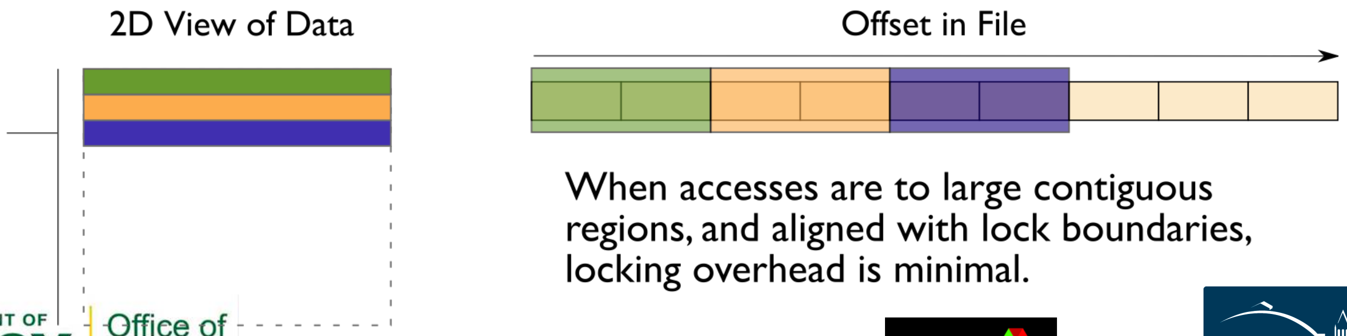
- **You can use MPI IO File Views to control how data is laid out on the file system**
  - Initial offset (default = 0)
  - Record type (size) (default = MPI_BYTE)
  - How records are laid out relative to each other (default=MPI_BYTE)
  - You can interleave data
  - Once defined, you may not need to seek() to explicit offsets

MPI-IO Collectives

- Allows the library to optimize the IO
- Must be called from all tasks in communicator
- Consolidates I/O requests from all tasks in communicator
- Only a subset of tasks (aggregators) access the file
- Also has a set of non-blocking routines
- Can give “hints” to optimize performance for your access patterns and/or the underlying file system structure

When To Use Collectives

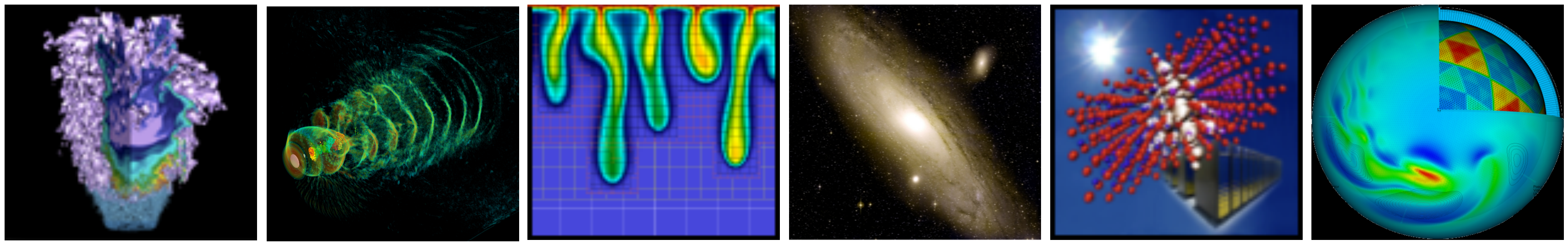
- **The smaller the write, the more likely it is to benefit from collective buffering**
- **Large contiguous I/O will not benefit from collective buffering.**
  - Non-contiguous writes of any size will not see a benefit from collective buffering



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

MPI-IO Summary

- Provides optimizations for 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
- MPI-IO works well in the middle of the I/O stack, letting high-level library authors write to the MPI-IO API



High Level Parallel I/O Libraries (HDF5)

U.S. DEPARTMENT OF ENERGY | Office of Science

National Energy Research Scientific Computing Center

Lawrence Berkeley National Laboratory

What is a High Level Parallel I/O Library?

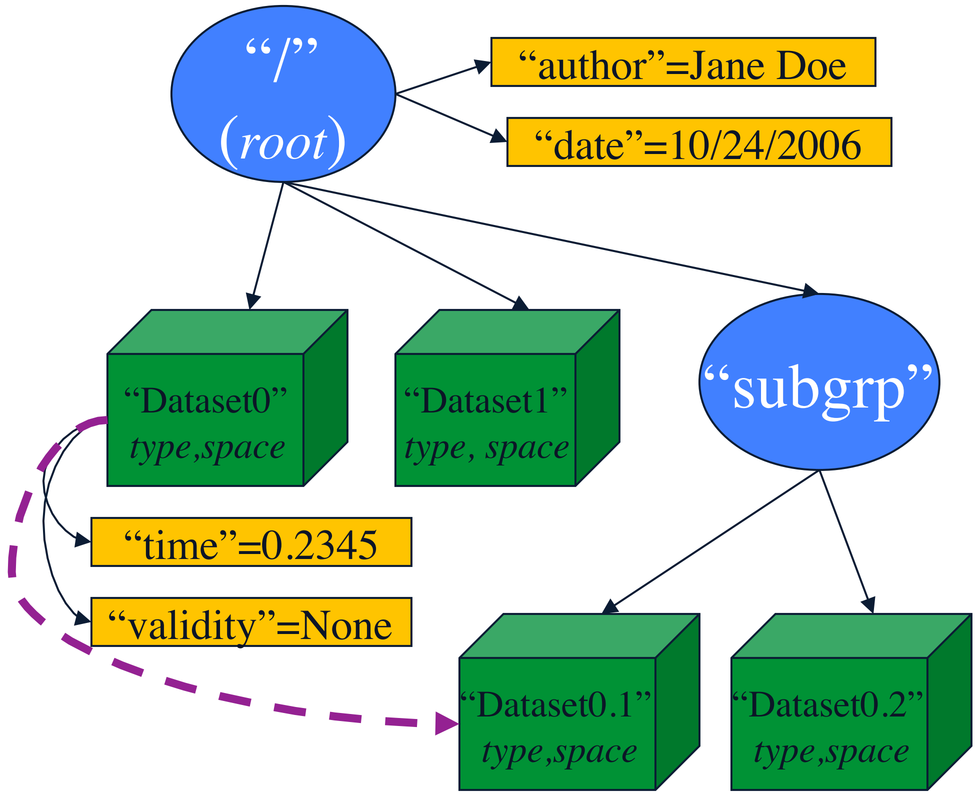
- **An API which helps to express scientific simulation data in a more natural way**
  - Multi-dimensional data, labels and tags, non-contiguous data, typed data
- **Typically sits on top of MPI-IO layer and can use MPI-IO optimizations**
- **Offer**
  - Simplicity for visualization and analysis
  - 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

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/
- http://www.hdfgroup.org/ftp/HDF5/examples/

HDF5 Data Model

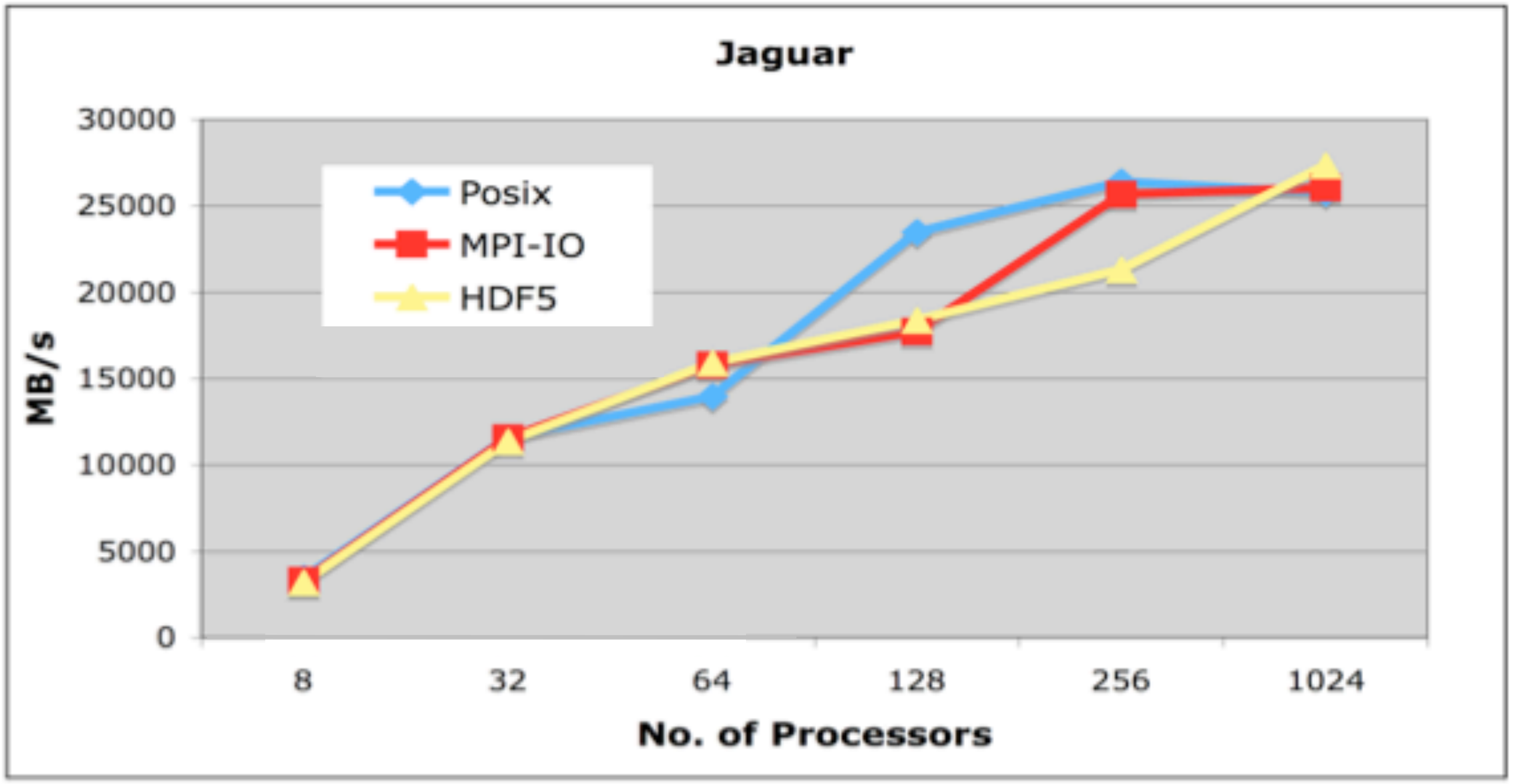
- **Groups** ●
  - Arranged in directory hierarchy
  - root group is always '/'
- **Datasets** 🟩
  - Dataspace
  - Datatype
- **Attributes** 🟨
  - Bind to Group & Dataset
- **References**
  - Similar to softlinks
  - Can also be subsets of data



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 can be optimized for given platforms and file systems by library developers

IO Library Overhead

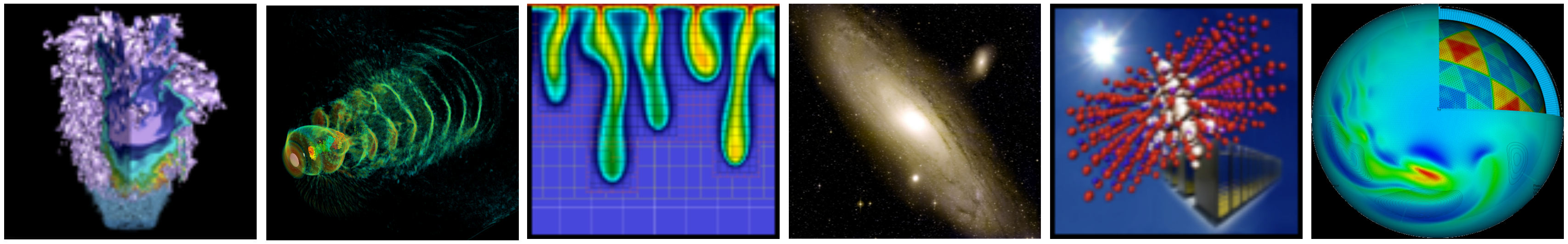


Very little, if any overhead from HDF5 for one file per processor IO compared to Posix and MPI-IO

Data from Hongzhang Shan

Performance

- IO performance is complicated to predict.
- Other users impact your job because IO uses a shared resource.
- Buffer caches exist throughout the system adding to the unpredictability.
- Data paths into single elements (e.g., a node) are limiting for large IO.
- Many small IO requests have a high overhead.
- You have to experiment!



I/O on Lustre File Systems

U.S. DEPARTMENT OF ENERGY | Office of Science

NERSC National Energy Research Scientific Computing Center

BERKELEY LAB Lawrence Berkeley National Laboratory

Terminology: Lustre

- Lustre (name derived from “Linux Cluster”)
- A clustered, shared file system
- Open software, available under GNU GPL
- Designed, developed, and maintained by Sun Microsystems, Inc., which acquired it from Cluster File Systems, Inc. in Oct. 2007
- Two types of Lustre servers (on IO service nodes)
  - Object Storage Servers (OSS)
  - Metadata Servers (MDS)

What is File Striping?

- Lustre file systems are 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

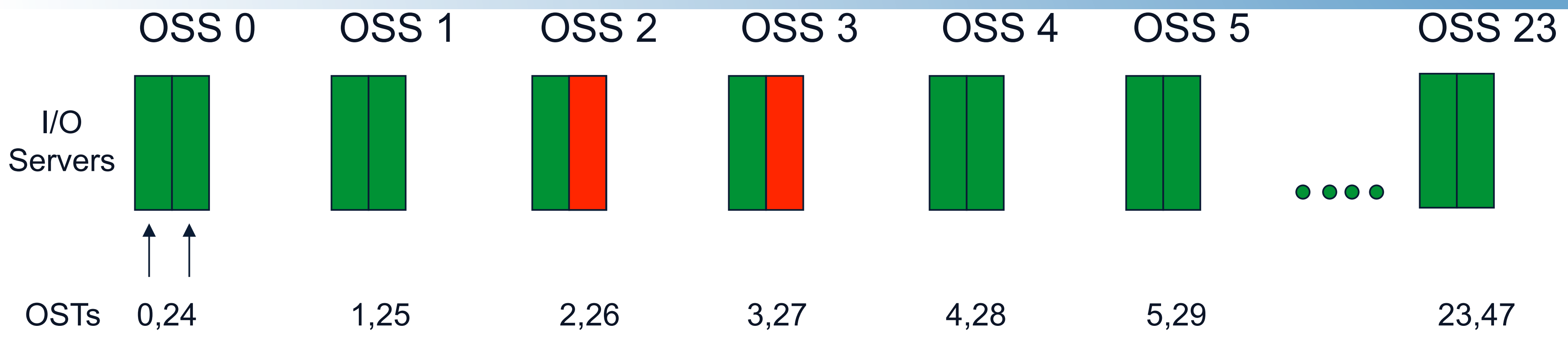
Lustre File Striping

- Files are broken into chunks that are stored on OSTs in a round-robin fashion.
- The size of the chunks and number of OSTs can be set by the user

```
lfs setstripe <name> -s <size> -I <start> -c <count>
```

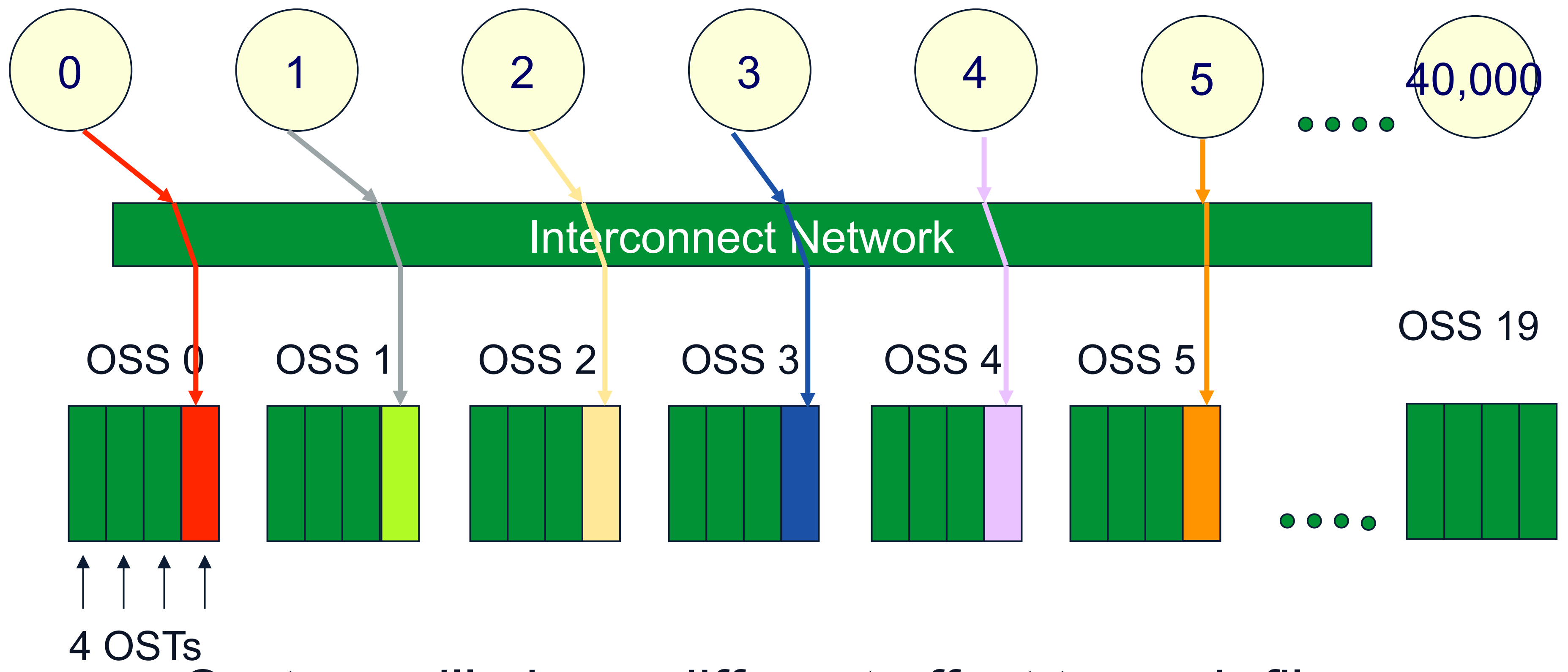
  – <name> can be a file or directory. If directory, new files in directory will inherit setting.
  – size = size of chunk, 0 signifies default of 1 MB
  – start = starting OST; you should use -1 to let the system decide
  – count = number of OSTs; 0 means use default, -1 means use all

A Stripe Count of 2



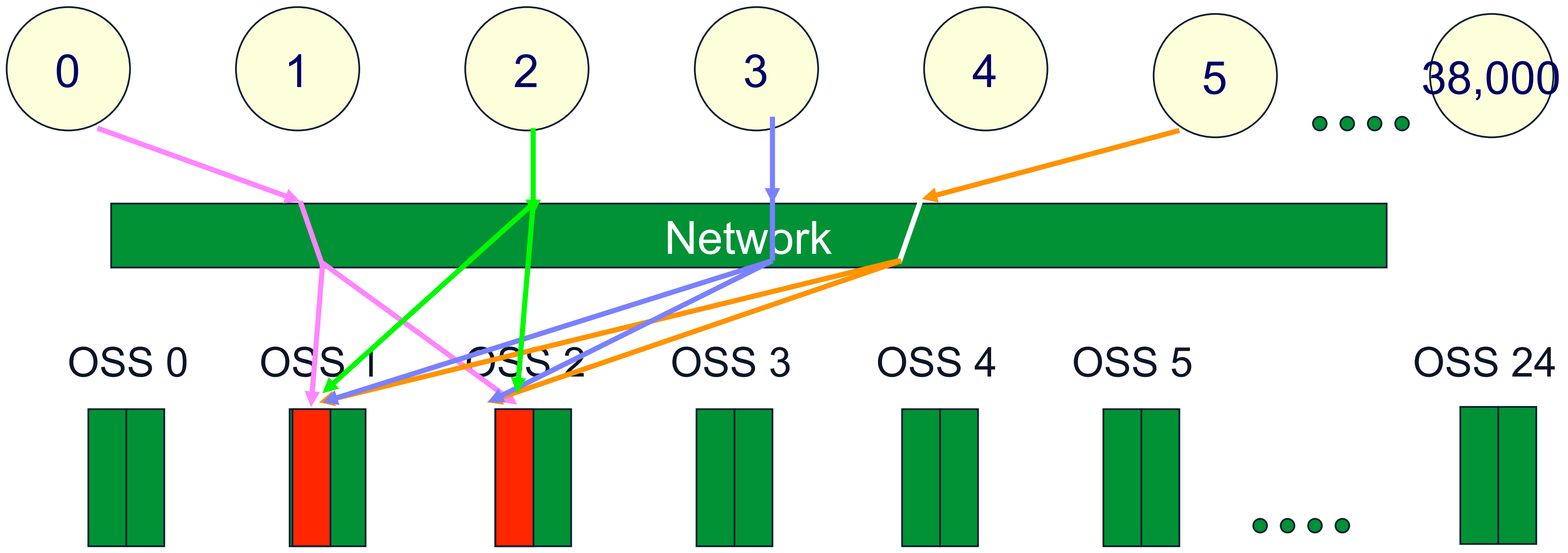
- **Pros**
  - Get 2 times the bandwidth you could from using 1 OST
  - Max bandwidth to 1 OST on NERSC's Hopper ~ 350 MB/Sec
  - Using 2 OSTs ~700 MB/Sec
- **Cons**
  - For better or worse your file now is in 2 different places
  - Metadata operations like 'ls -l' on the file could be slower
  - For small files (<100MB) no performance gain from striping

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



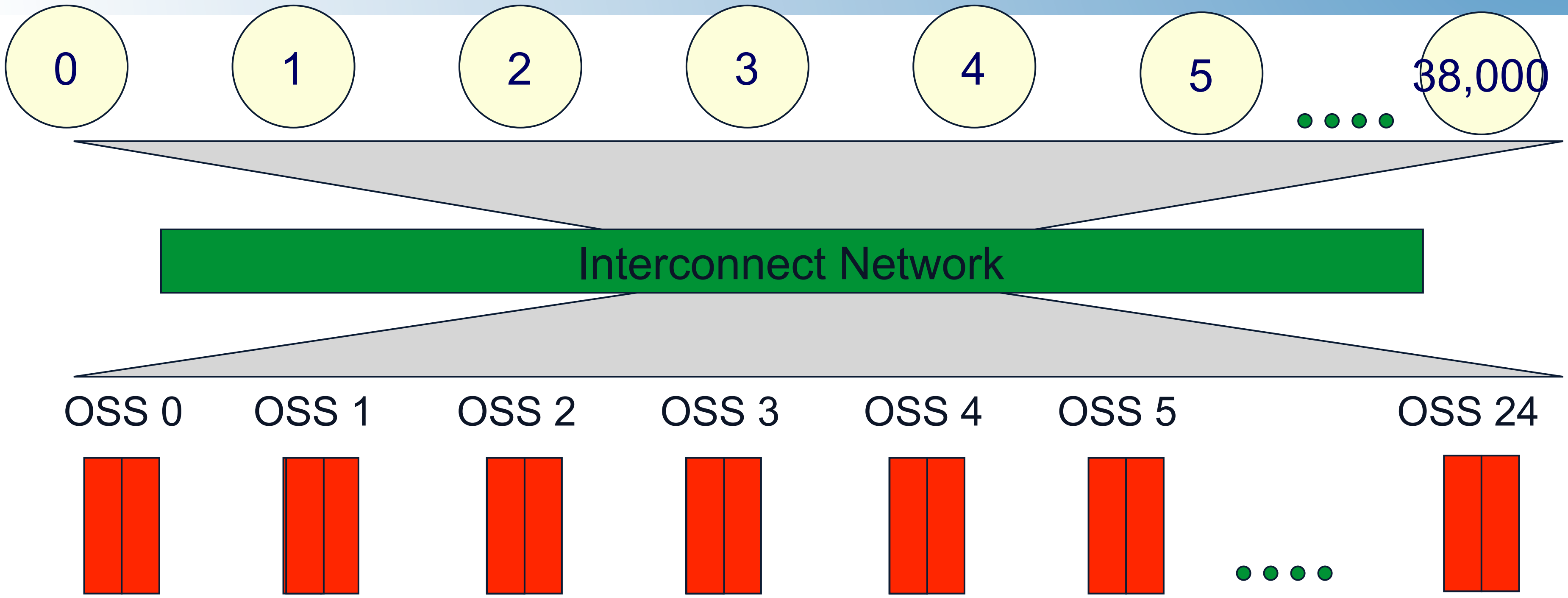
- System will give a different offset to each file (mod # of OSTs)
- If you have fewer writers than OSTs, and large files, you should stripe across >1 OST

Shared File I/O with Stripe Count 2



- 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)
- Need to use more OSTs for large shared files

Shared File I/O with Stripe Count = # OSTs



- Now Striping over all OSTs
- Increased available bandwidth to application

Striping Summary

- One File-Per-Processor I/O or shared files < 10 GB
  - Keep default, stripe count 1
- Medium shared files: 10GB – 100sGB
  - Set stripe count ~4-20
- Large shared files > 1TB
  - Set stripe count to 20 or higher, maybe all OSTs?
- You’ll have to experiment a little

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.

NERSC

National Energy Research Scientific Computing Center