A Brief Introduction to HPC I/O

Richard J Zamora

Data Science Group, ALCF
Argonne National Laboratory
rzamora@anl.gov

Scalable I/O Workshop — August 23rd 2018

Goal

- This is a high-level overview to get everyone on a similar page
- The goal of this talk is to review current I/O trends and practices in the world of HPC
- I will not talk about HEP applications or HEP-specific software technologies

Acknowledgement

Much of the content from this talk is borrowed from similar talks by other people. A valuable source is public content from The *Argonne Training Program on Extreme Scale Computing* (ATPESC), and other ALCF workshops. Others responsible for the content shared here:

- Francois Tessier
- Venkatram Vishwanath
- Paul Coffman
- Rob Lantham
- Rob Ross
- Phil Carnes
- Kevin Harms (*giving more ALCF-specific talk later*)
- …

Preview

1. **HPC I/O Basics**
2. **I/O Optimization Basics**
3. **Scalable I/O Libraries**
4. **Summary**

HPC I/O Basics

In the world of HPC, I/O usually corresponds to the storage and retrieval of persistent data to and from the a file system (by a software application)

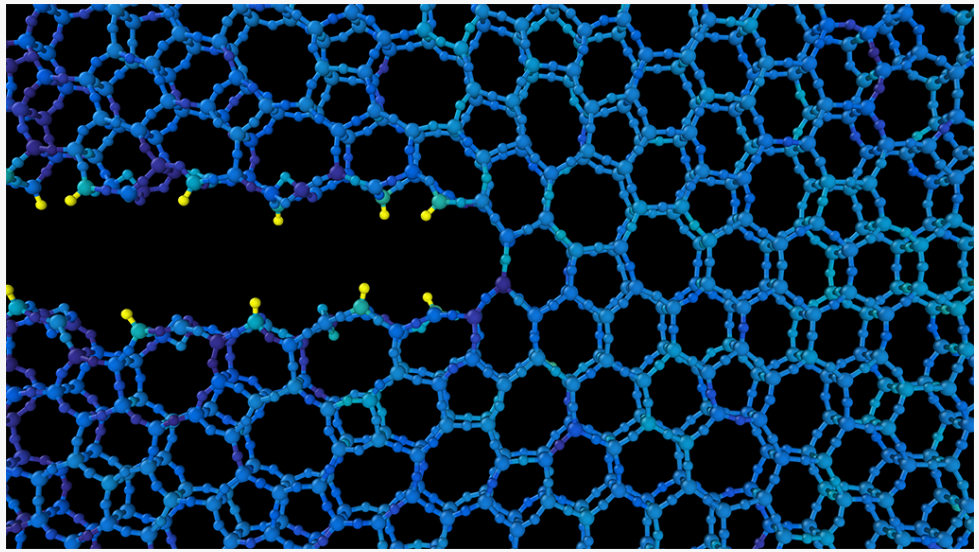
Key Points:

- Data is stored in POSIX files/directories (for now)
- HPC systems use parallel file systems (PFS)
- The PFS provides **aggregate** bandwidth
- User applications and/or I/O libraries must play to the PFS' **strengths**

Basic HPC I/O Flavors and Components

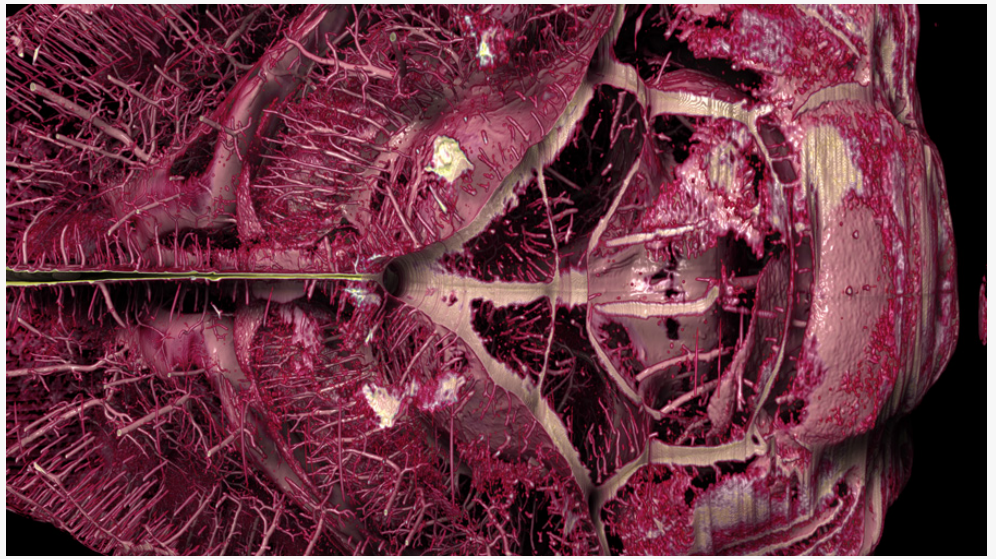
Basic flavors of HPC I/O:

- **Defensive**
  - Writing data to protect results from software and/or system failures (a.k.a checkpointing)
  - Priority: Speed



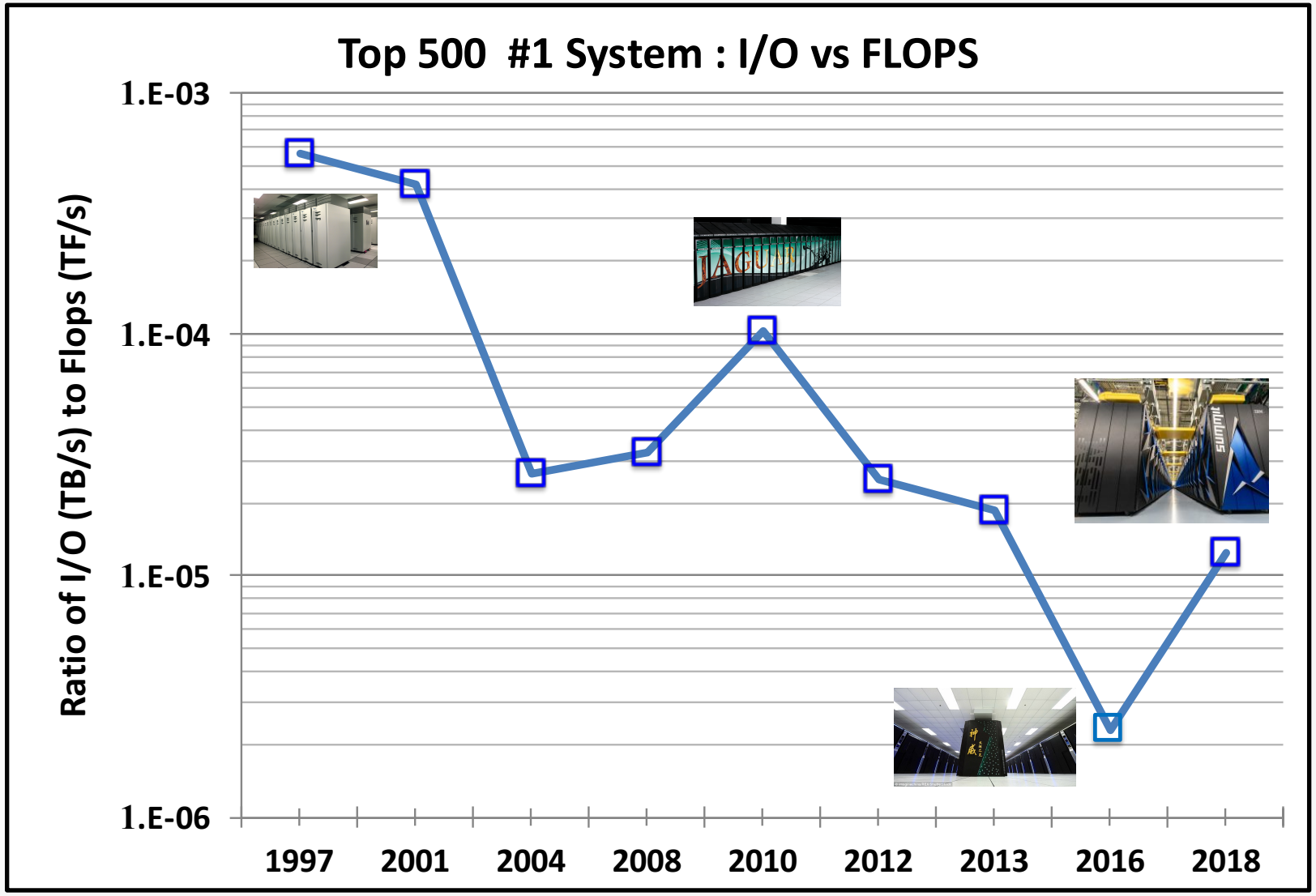
INCITE, 2016, James Kermode, University of Warwick

- **Productive**
  - Reading and/or writing data as a necessary component of the application workflow
  - Priority: Speed, Organization, Provenance



ADSP, 2017, Doga Gursoy, Argonne NAtional Laboratory

Historically: Compute has Outpaced I/O



How Data is Stored on HPC Systems

- **HPC systems use the *file system* (FS) model for storing data**
  - Use a **file** (POSIX) model for data access
  - The FS is **shared** across the system
  - There may be >1 FS on each system, and the FS might be shared between systems.
- **POSIX I/O API**
  - Lowest-level I/O API
  - **Well supported**: Fortran, C and C++ I/O calls are converted to POSIX
  - **Simple**: File is a *stream of bytes*

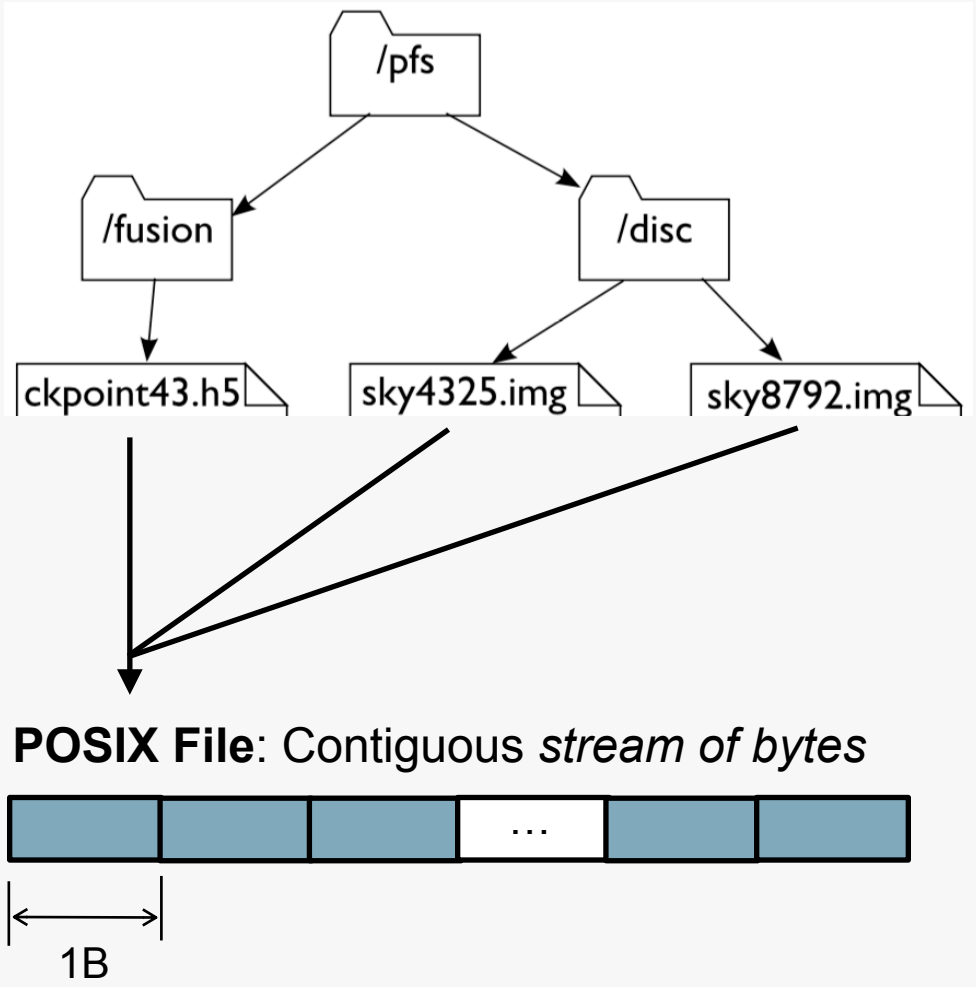
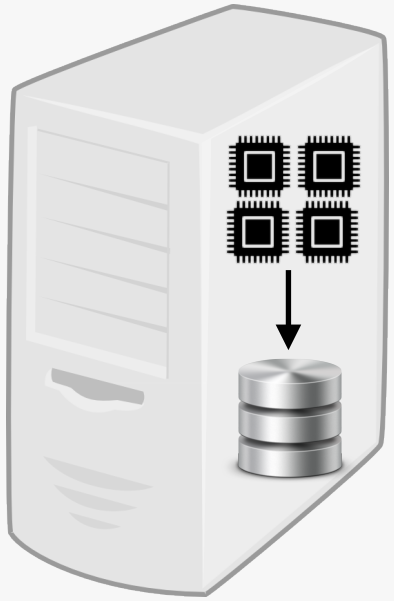


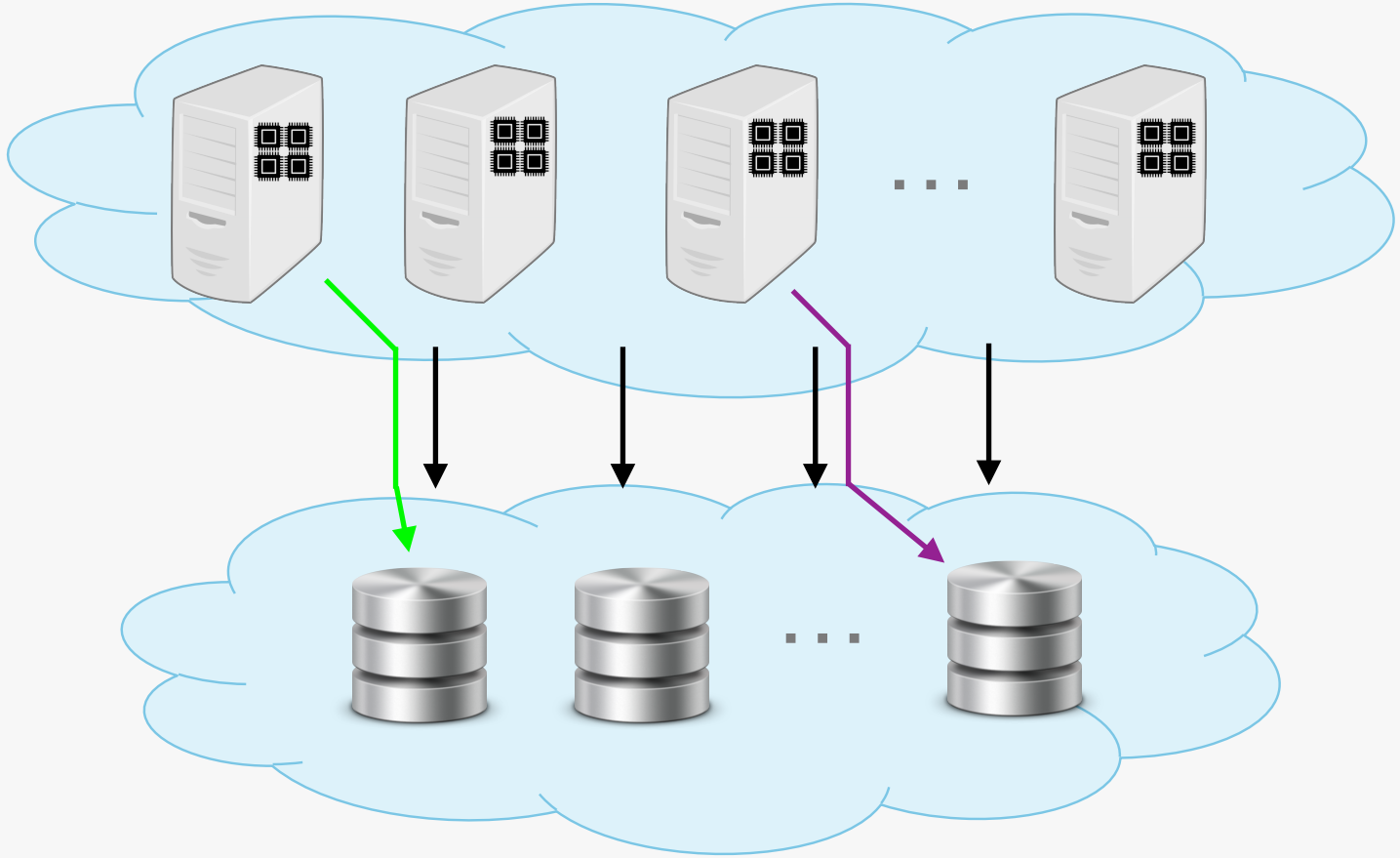
Image Source: Lantham et al. ATPESC 2014

Parallel I/O on a *Parallel* File System

Traditional: Serial I/O
(to *local* file system)

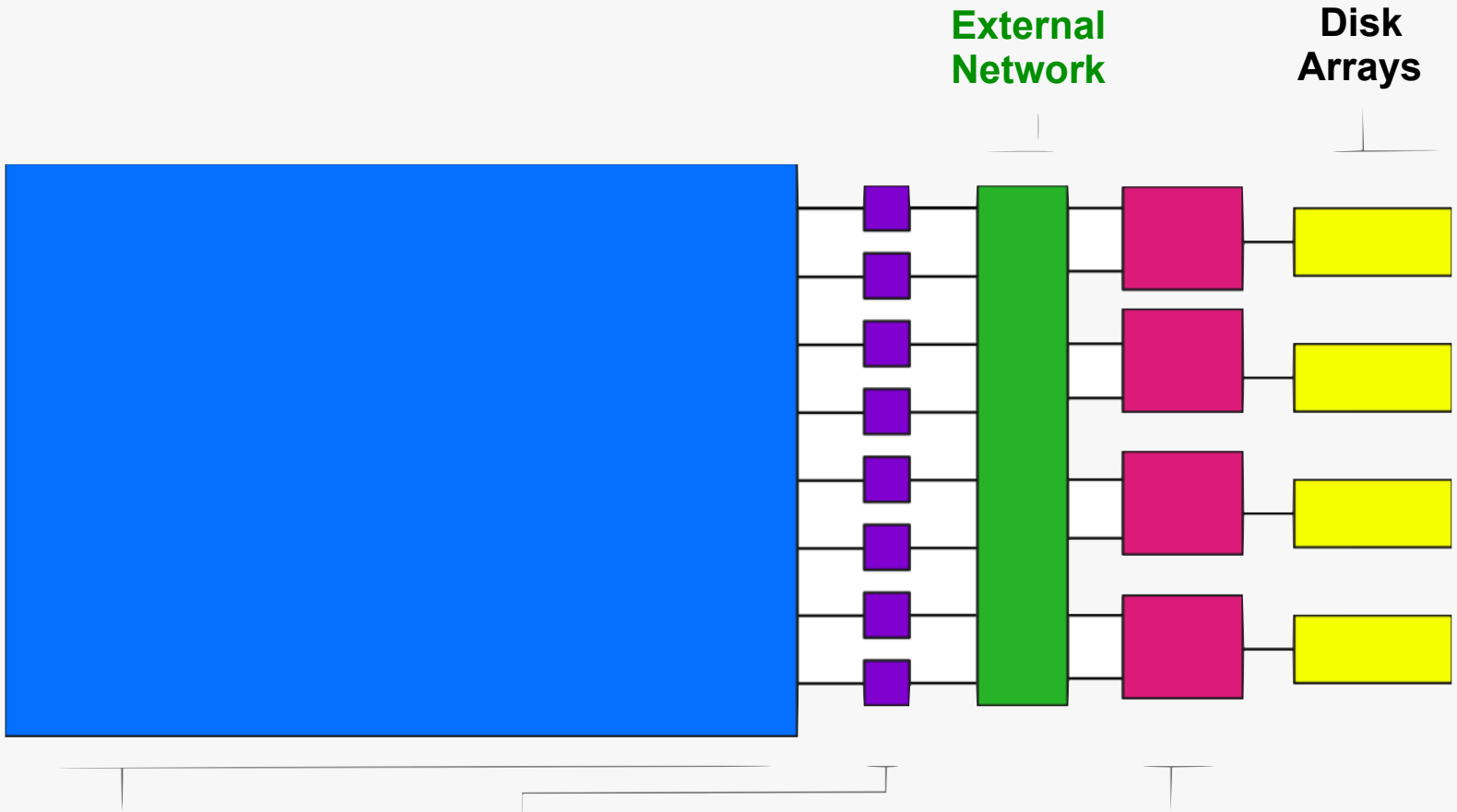


HPC: Parallel I/O (to *shared* file system)



Several Popular *Parallel* File Systems in HPC

Typical HPC I/O Network

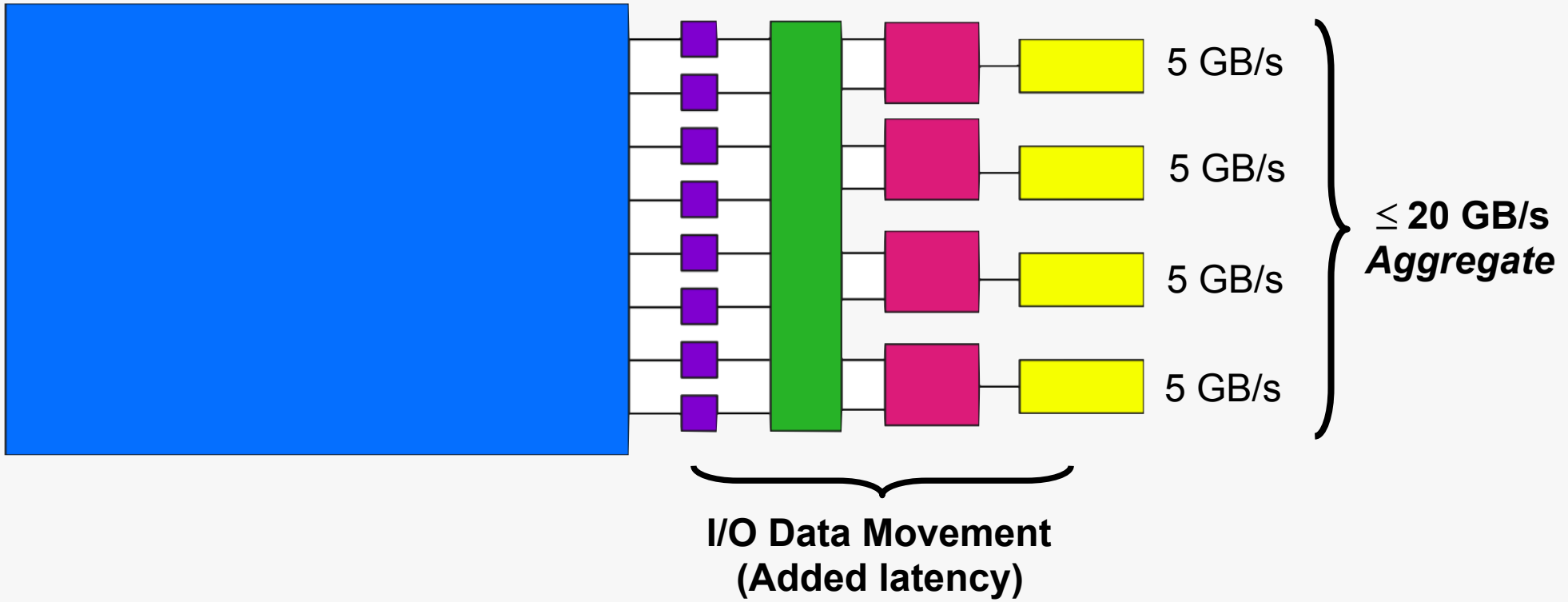


Compute nodes run user application. Data model software and some I/O transformations are also performed here.

I/O forwarding nodes (gateway nodes, LNET nodes,…) shuffle data between compute nodes and storage.

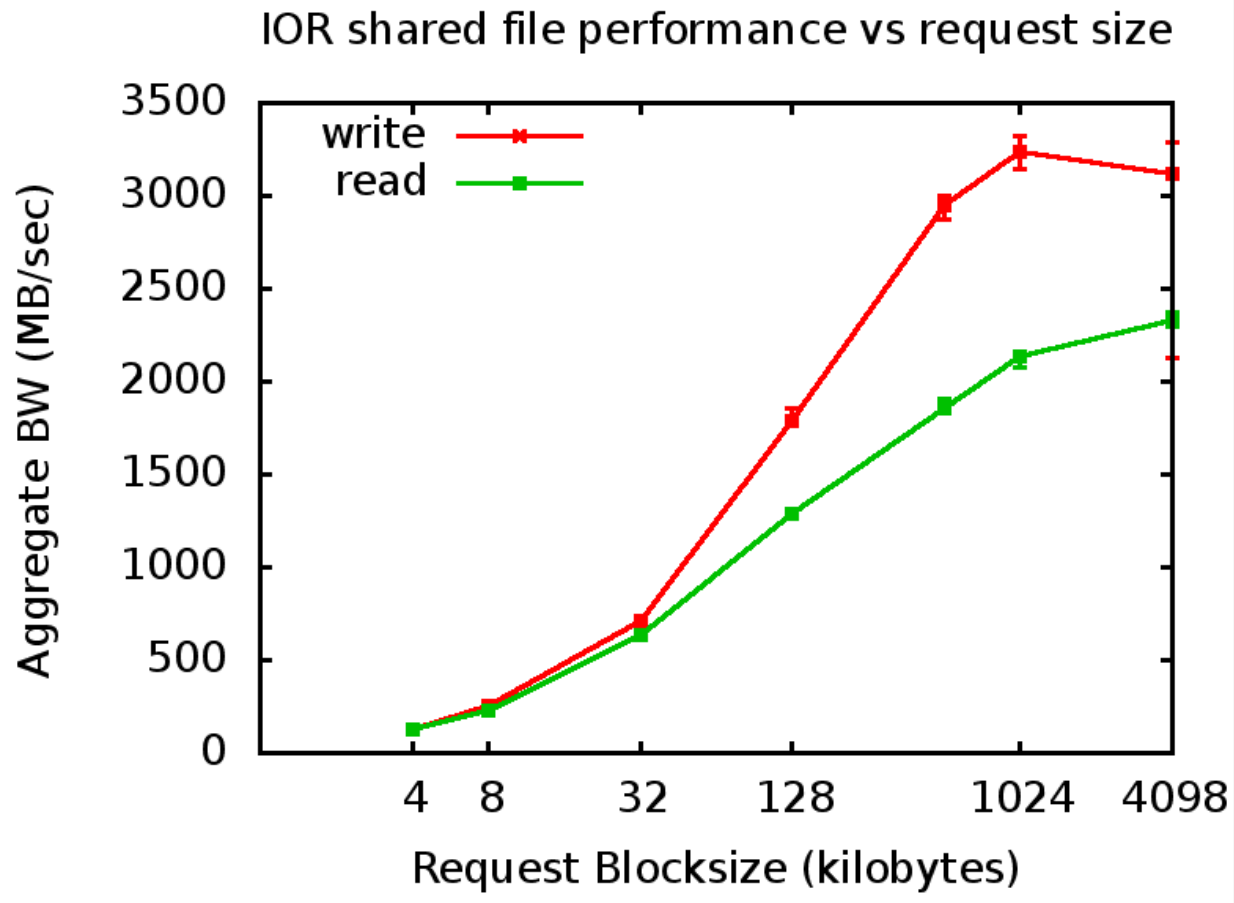
Storage nodes run the parallel file system.

HPC I/O provides *Aggregate* Bandwidth



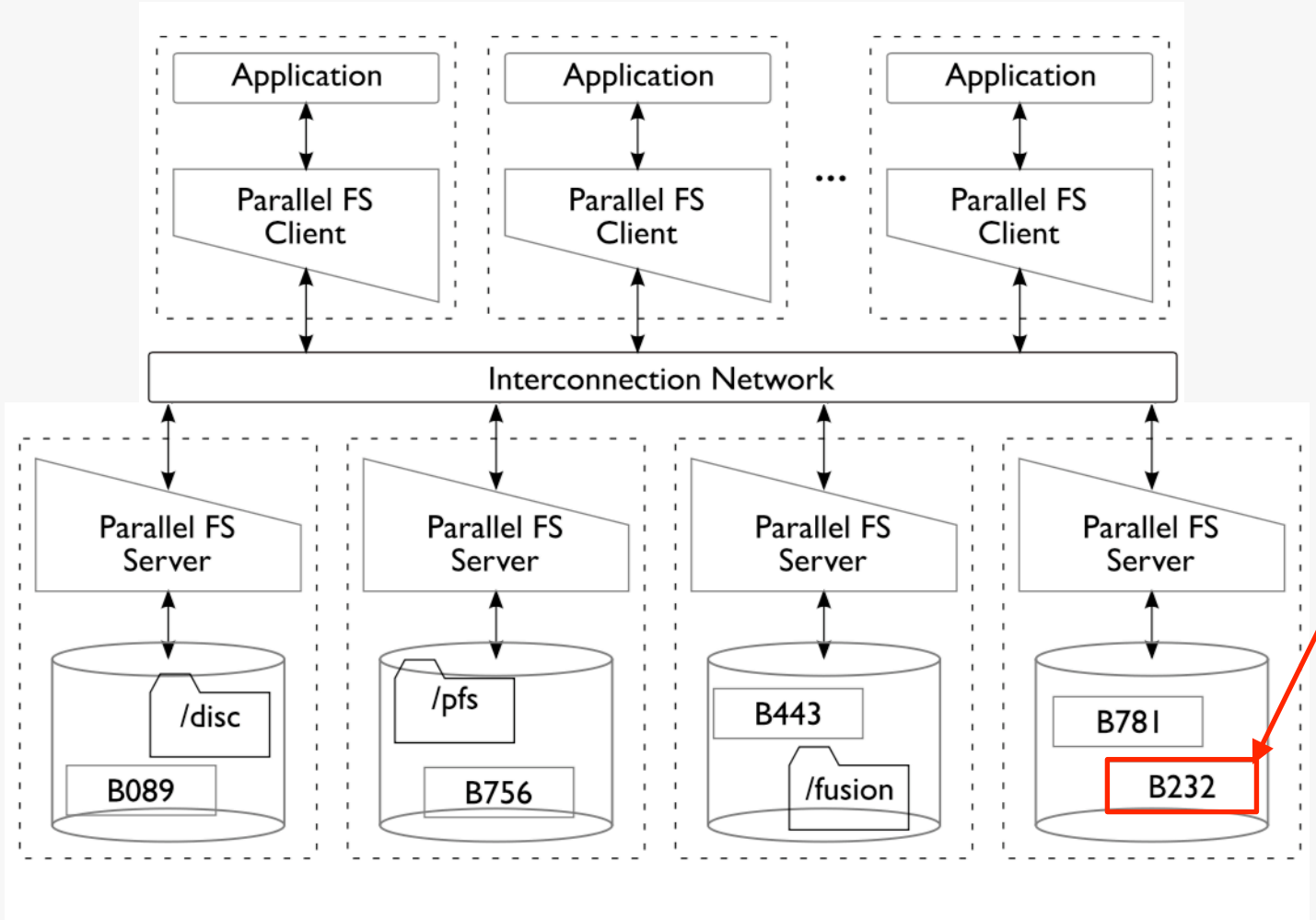
Data must take multiple hops to get to/from disk -> **high latency**
Multiple disks can be accessed concurrently -> **high aggregate bandwidth**

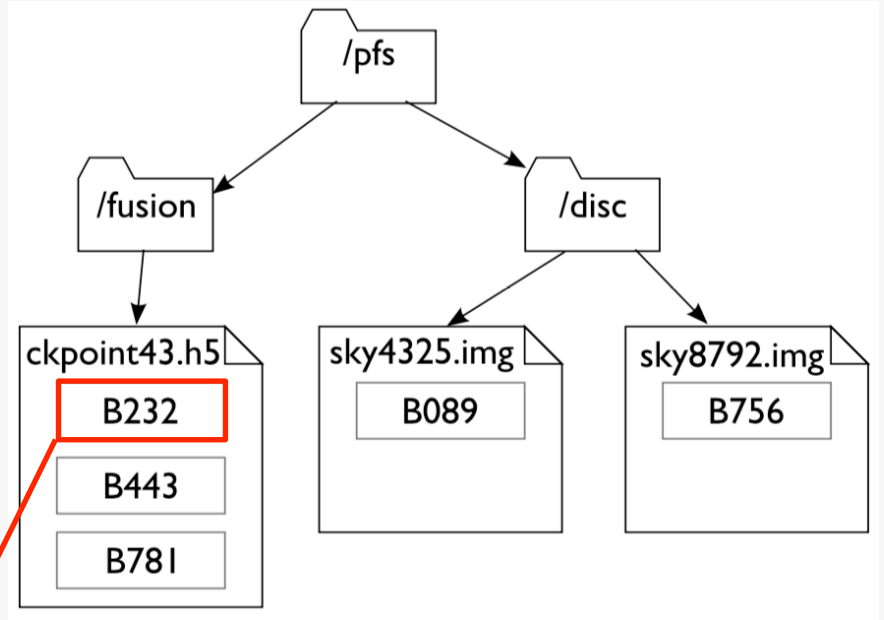
Example: Large Parallel Operations are *Faster*



2K processes of IBM Blue Gene/P at ANL

Mapping of Data Onto Parallel Storage





In a PFS, files are broken up into regions called ***extents*** (or **blocks**, or **stripes**, etc..)

Data Mapping Example: *Lustre*

File Stripes and Storage Targets (OSTs)

File **stripes** are distributed among **OST**s…

Clients = LNET Router Nodes
MDS = Metadata Server
MDT = Metadata Target
OSS = Object Storage Server
OST = Object Storage Target (Disk)

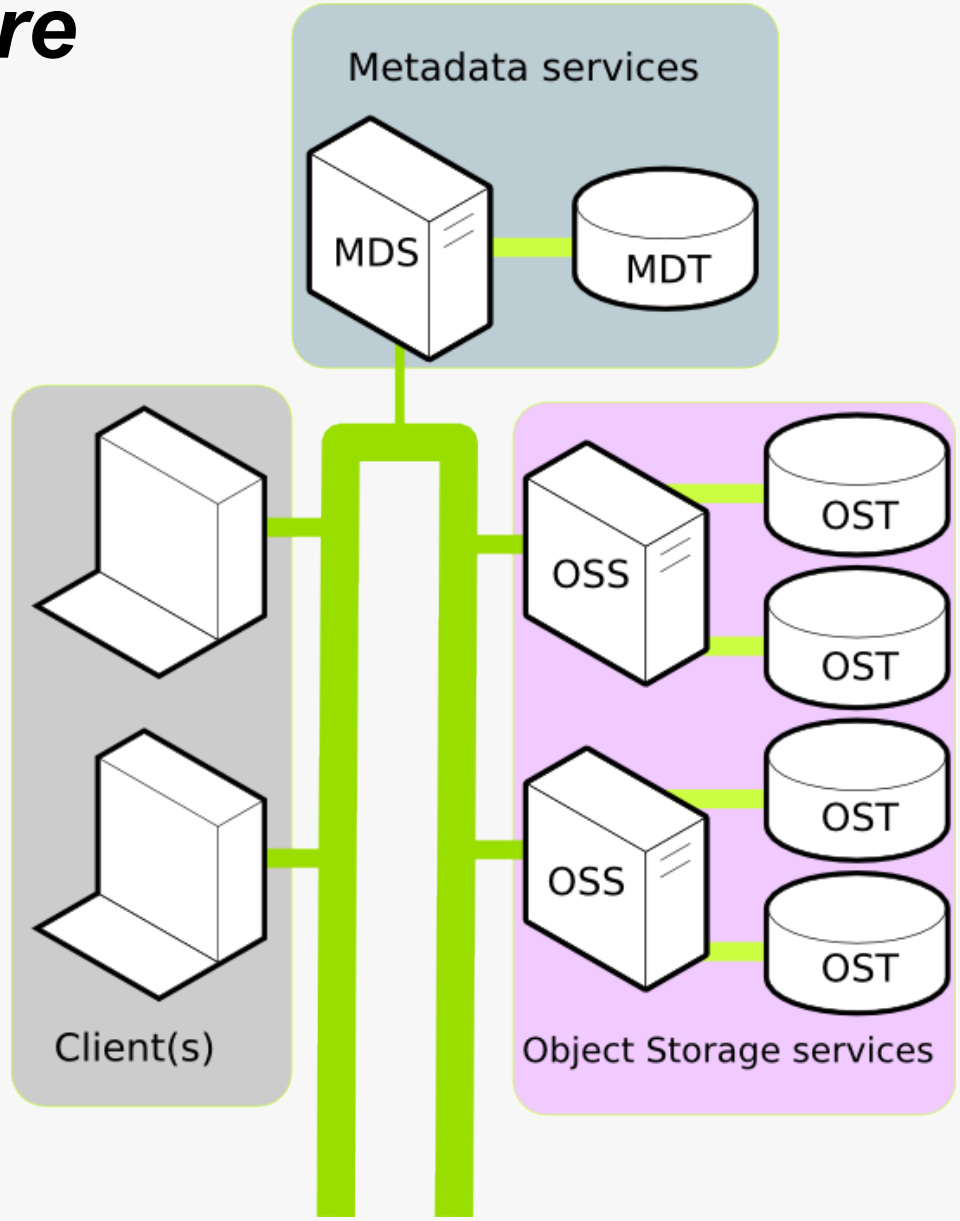


Image: https://wiki.hpdd.intel.com/display/PUB/Components+of+a+Lustre+filesystem

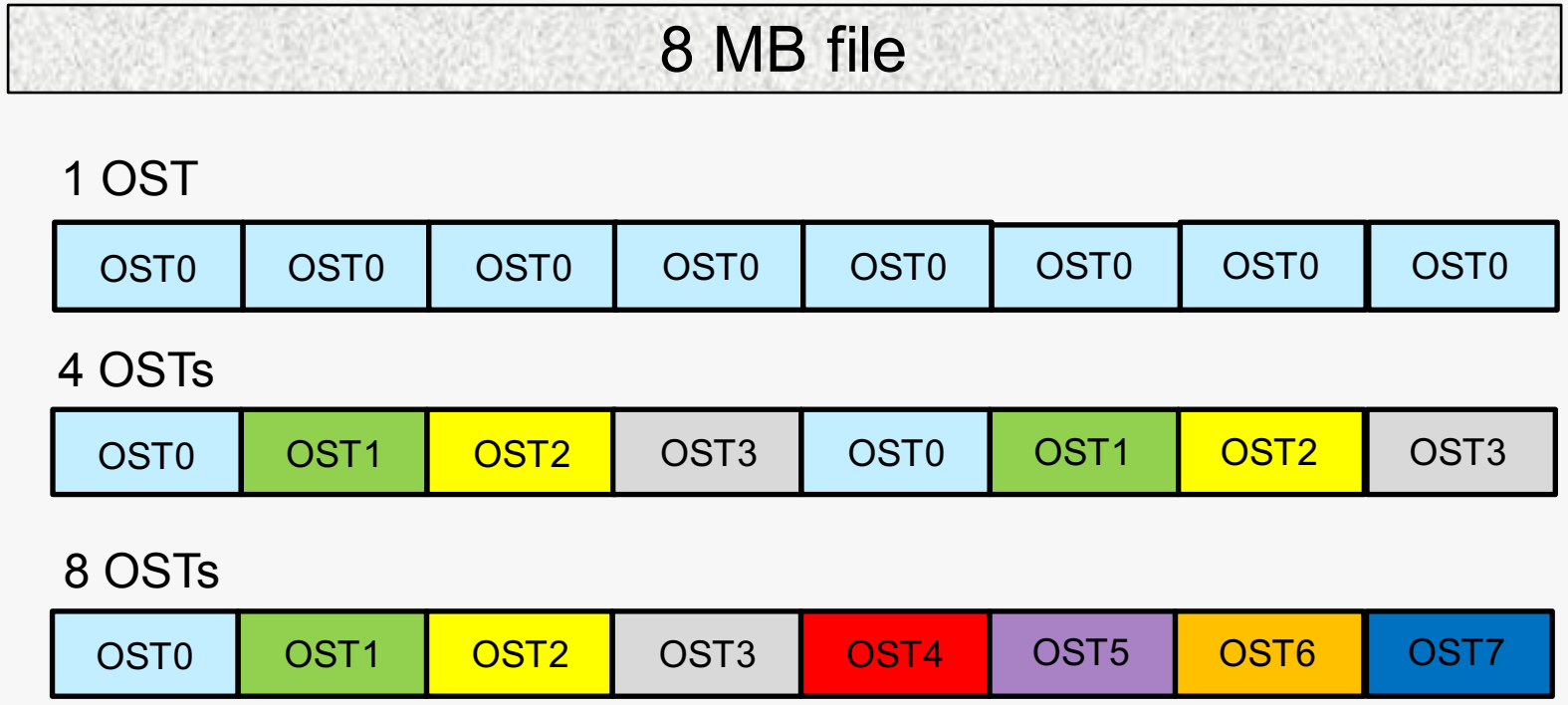
Data Mapping Example: *Lustre*

Round-robin Assignment of File Stripes to OST's

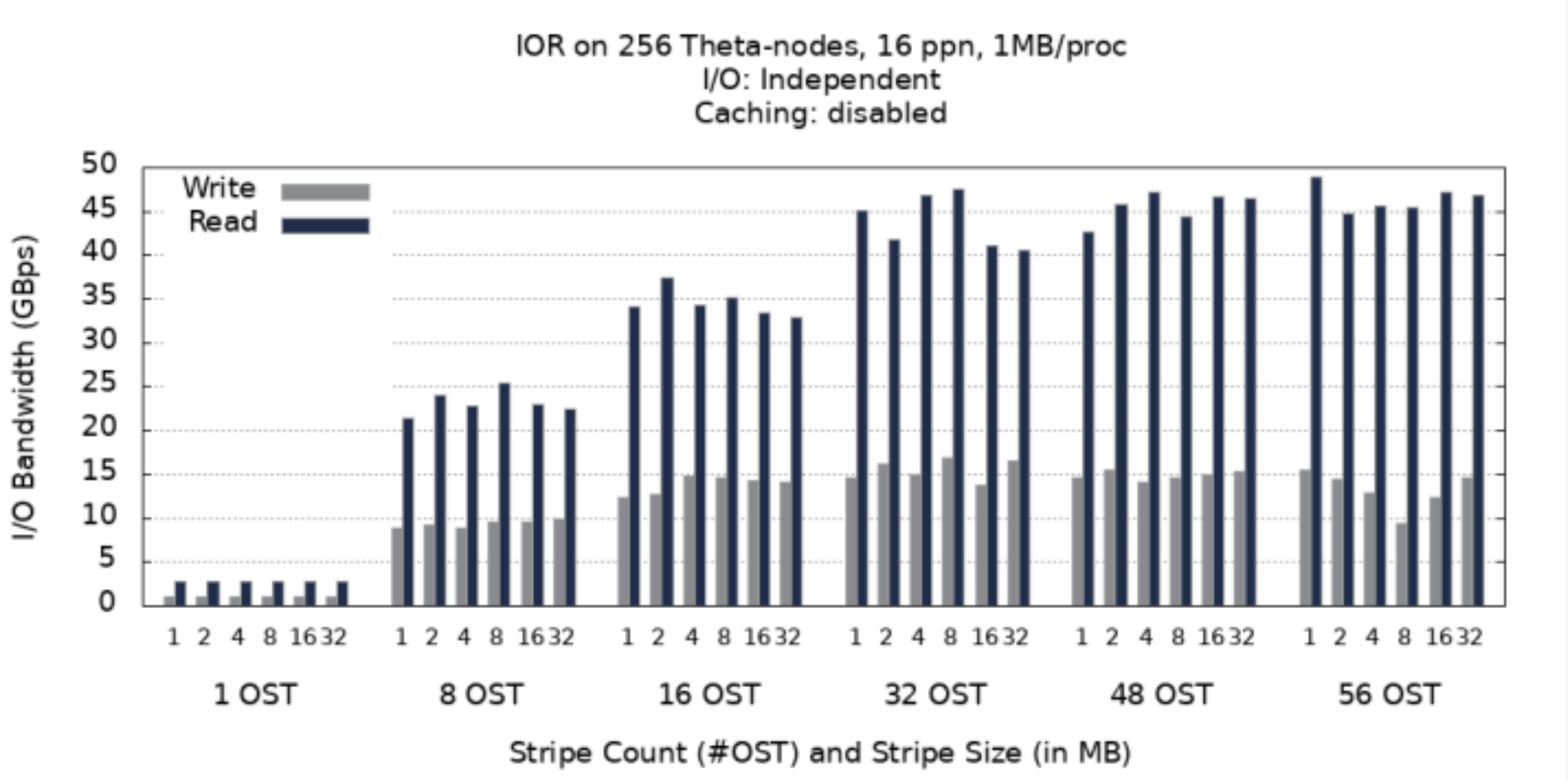
IMPORTANT: Unlike GPFS blocks, the user can specify the stripe **size** and **count** (number of OSTs) in Lustre

The **size** of stripes and **number** of OSTs can be user-specified for each file…

Example (stripe size = 1 MB):



As Expected: More OSTs, More Bandwidth

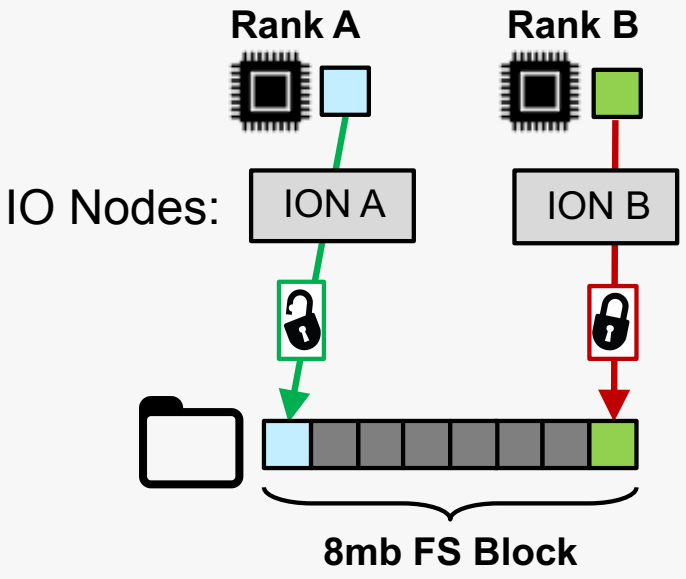


Locking of File Blocks/Stripes

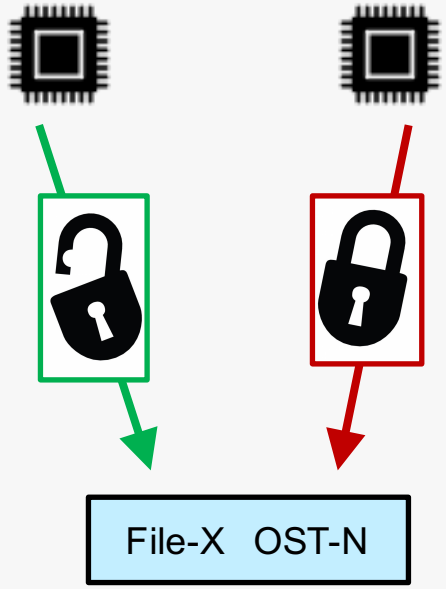
GPFS & Lustre Examples

- Block/Striped **aligned** I/O requests are important to avoid **lock contention** (false data sharing)
- Set **correct** file properties, and **environment variables** to avoid unnecessary locking

GPFS Block Locking

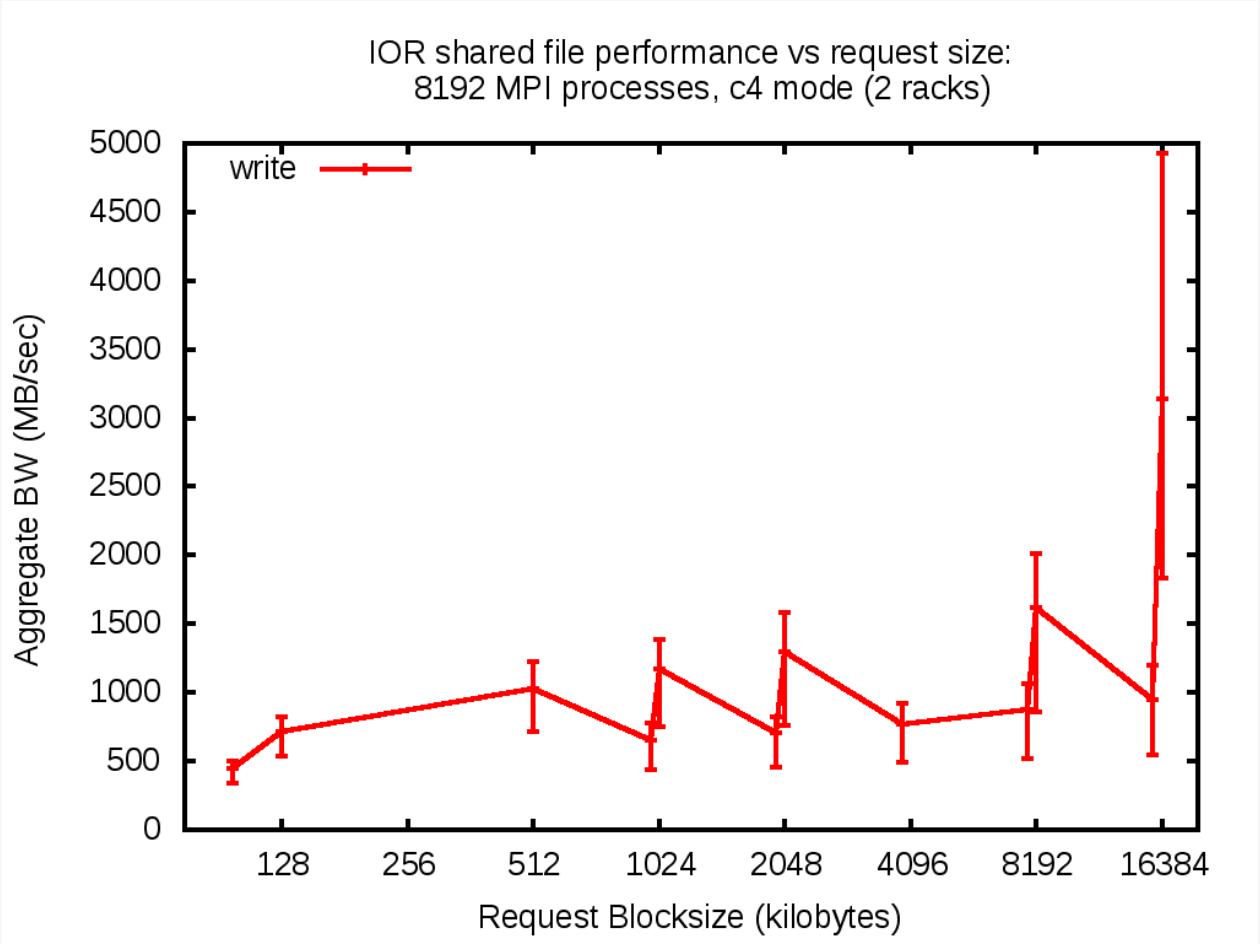


Lustre Stripe Locking



Ex. Cray+Lustre - Avoid whole-file locking:
`cray_cb_write_lock_mode=1`

Example: Block-Aligned I/O is *Faster*



8K processes of IBM Blue Gene/Q at ANL

Shared-Network *Traffic* and Performance *Jitter*

- Many HPC clusters rely on a shared network for I/O
- Many compute networks are also shared between jobs
- In Lustre, there are often a small number of **metadata** servers, making many basic FS operations (`ls`, `open`, `close`, etc..) highly dependent on network conditions

I/O Optimization Basics

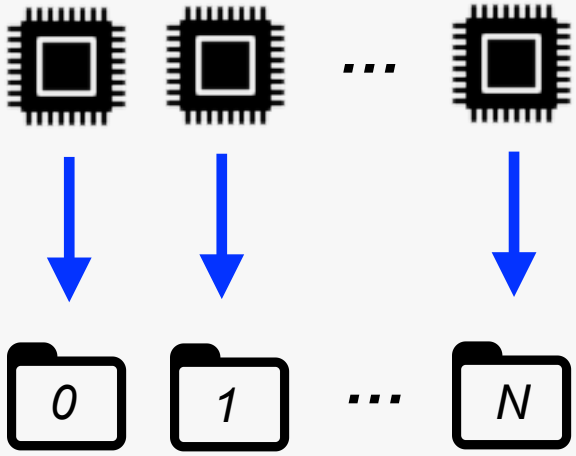
HPC machines usually rely on the PFS for I/O performance. Parallel algorithms and optimizations are needed to efficiently move data between compute and storage hardware

Key Points:

- **Sub-filing** will outperform both **shared** files file-**per-process** (at scale)
- **Collective** I/O will often outperform **independent** I/O
- Collective I/O algorithms are highly **tunable**
- **Compression** and **NVM**-utilization is also important

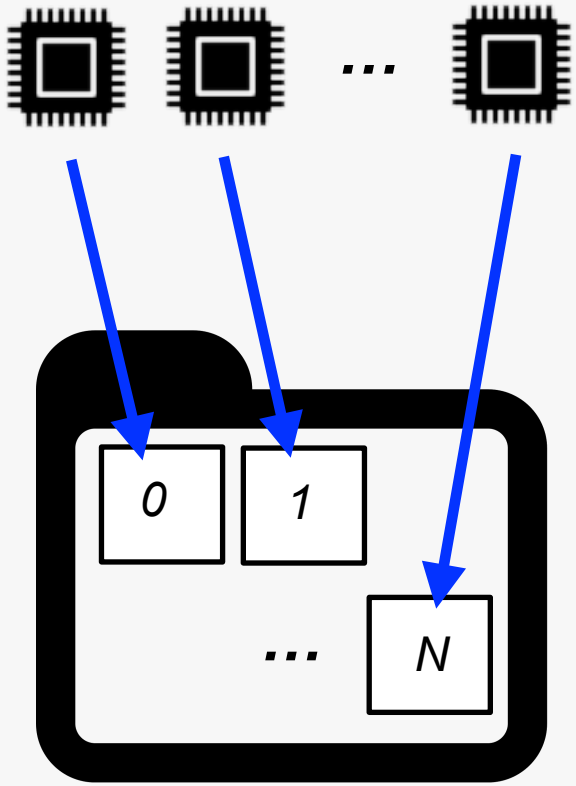
Types of Parallel I/O

File-per-process (FPP) Parallel

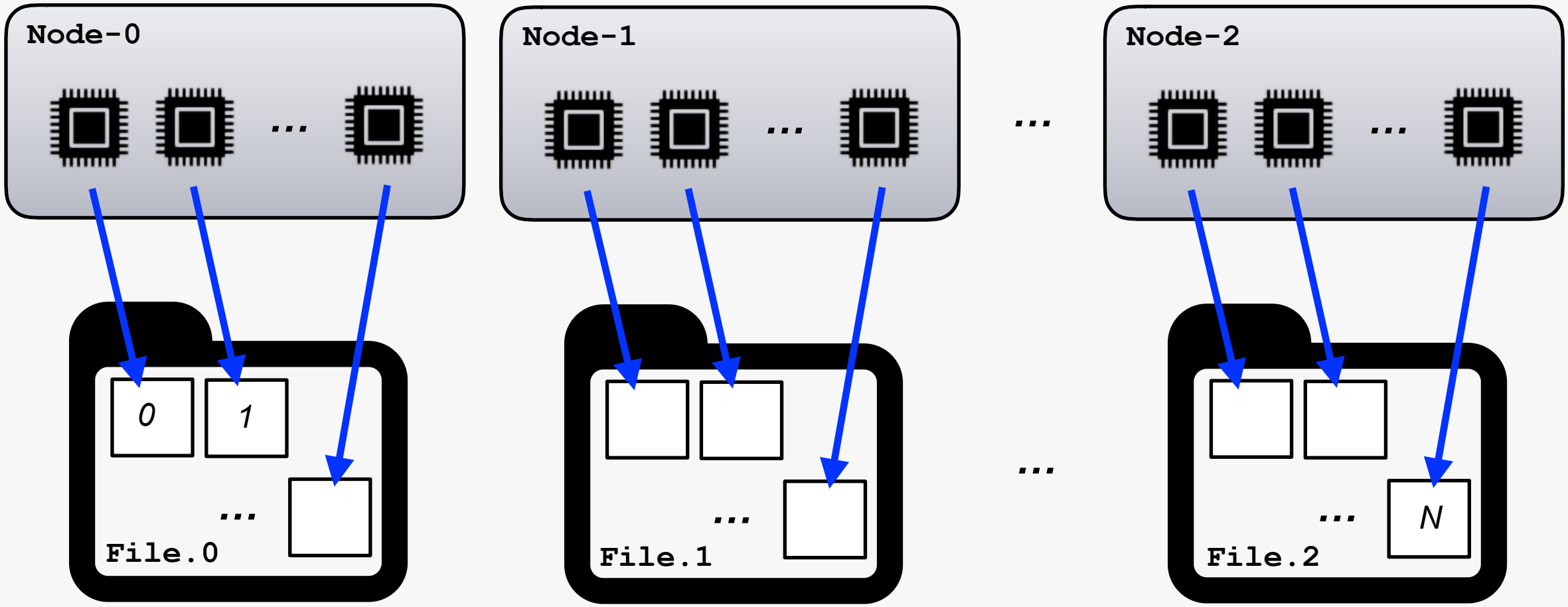


FPP can be fast for 10^1-10^3 ranks, but cannot scale to extreme scales (management and consumption issues)

Shared File Parallel



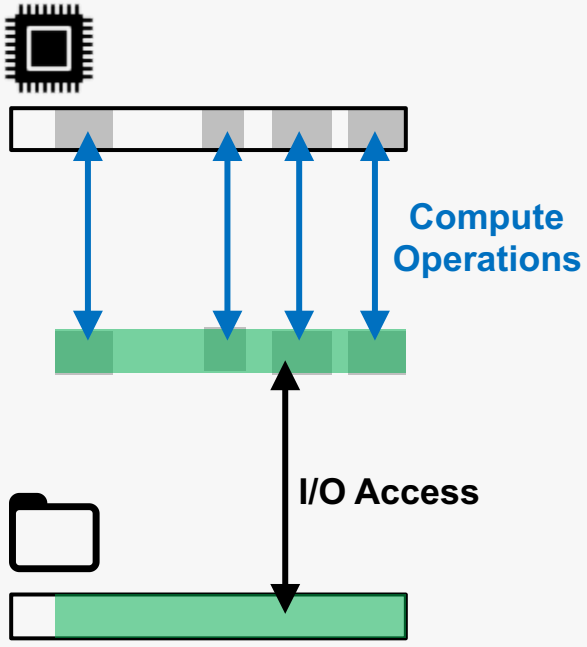
Mixing FPP with Shared Files: Sub-filing



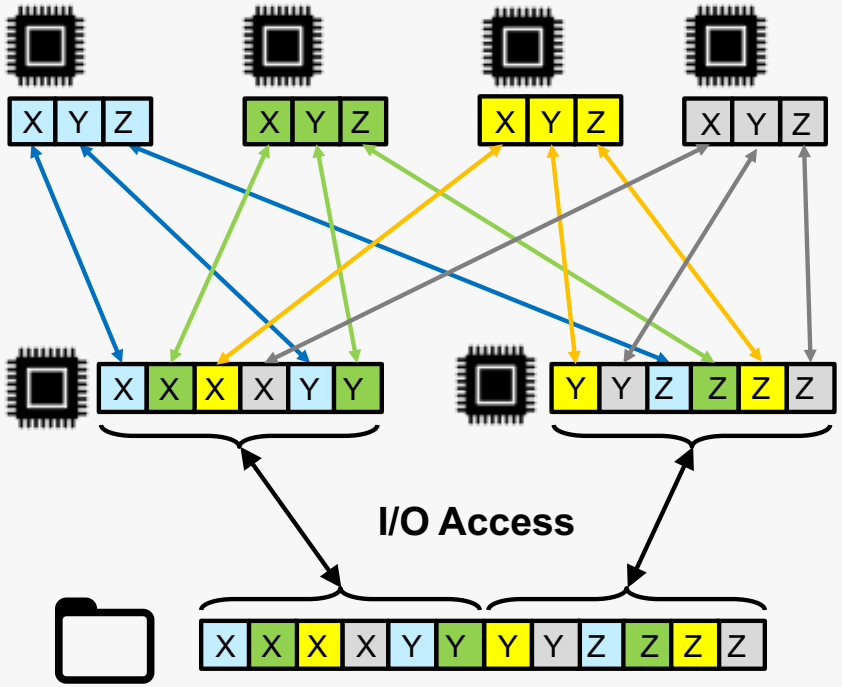
At large scale, it can be optimal to use a shared file for each subset of processes (Ex. Per-node)

Common I/O Optimizations

Data Sieving

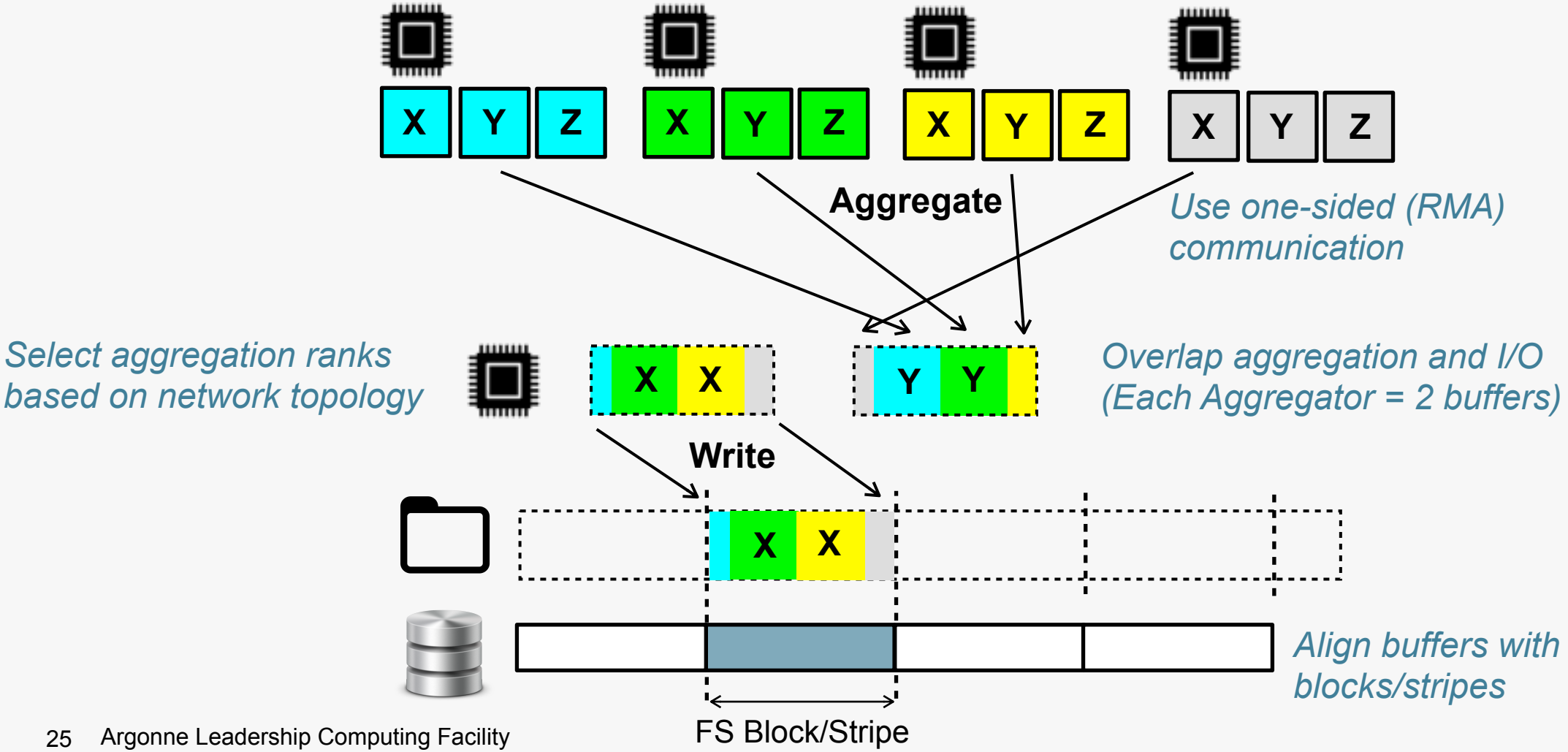


Two-Phase I/O
(Collective Aggregation)



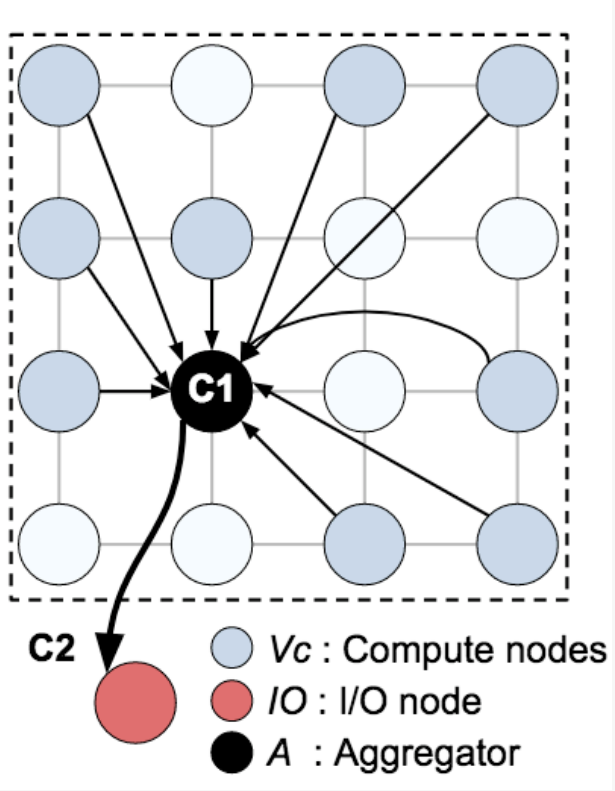
Optimizing Two-Phase I/O

Better collective I/O performance: Can optimize communication (One-sided MPI), can optimize aggregator selection, and can align aggregation buffers with file system blocks/stripes



Optimized Two-Phase I/O Example: TAPIOCA

- **Topology-Aware Parallel I/O: Collective Algorithm (TAPIOCA)**
  - An MPI-IO optimization library for two-phase I/O
- **Optimizations**:
  - Topology aware aggregator selection
  - One-sided (RMA)-based aggregation of data
  - Asynchronous I/O (overlapping aggregation with I/O)
  - Sub-filing

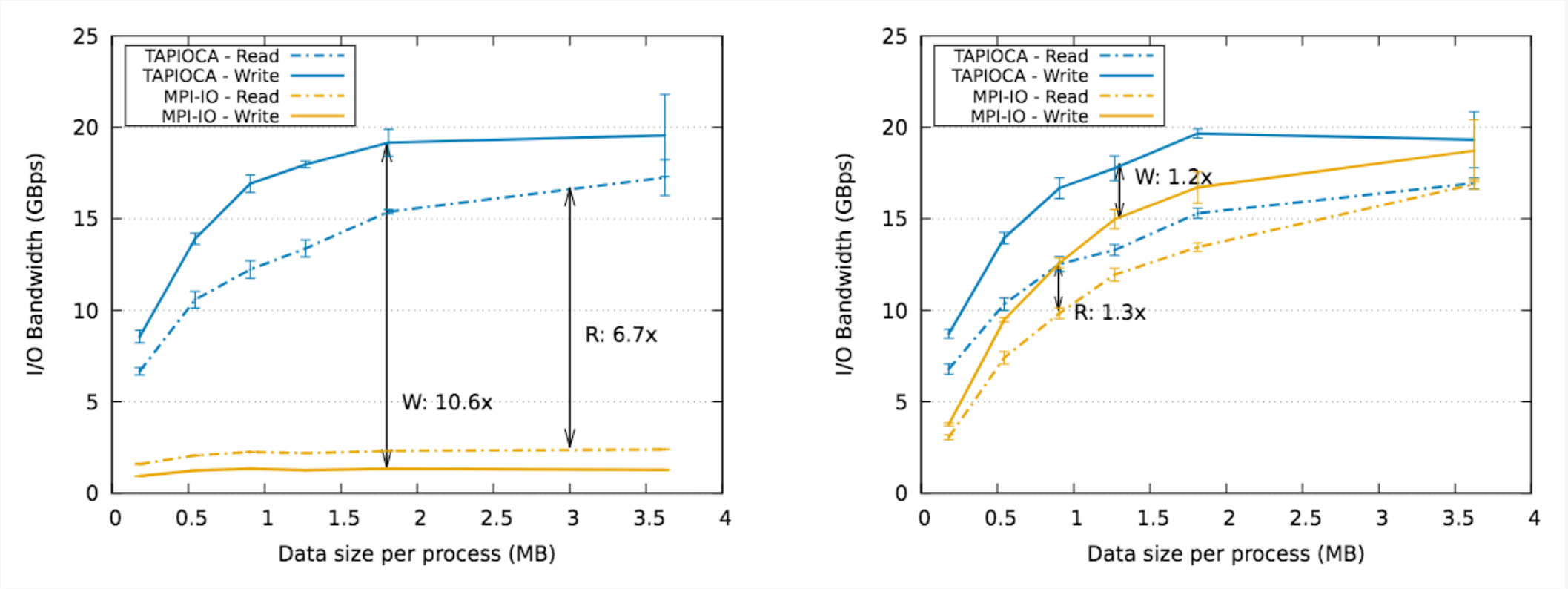


Topology aware aggregator selection.

Source: *Francois Tessier* et al. TAPIOCA, IEEE Cluster 2017

TAPIOCA Performance Example

TAPIOCA improvement over default MPI-IO for **HACC-IO** benchmark: *1024 BG/Q nodes (Mira), 16 ranks/node, **sub-filing** — 16 aggregators/Pset, 16MB aggregator buffer size*



(a) **Interleaved** Data Mapping to File

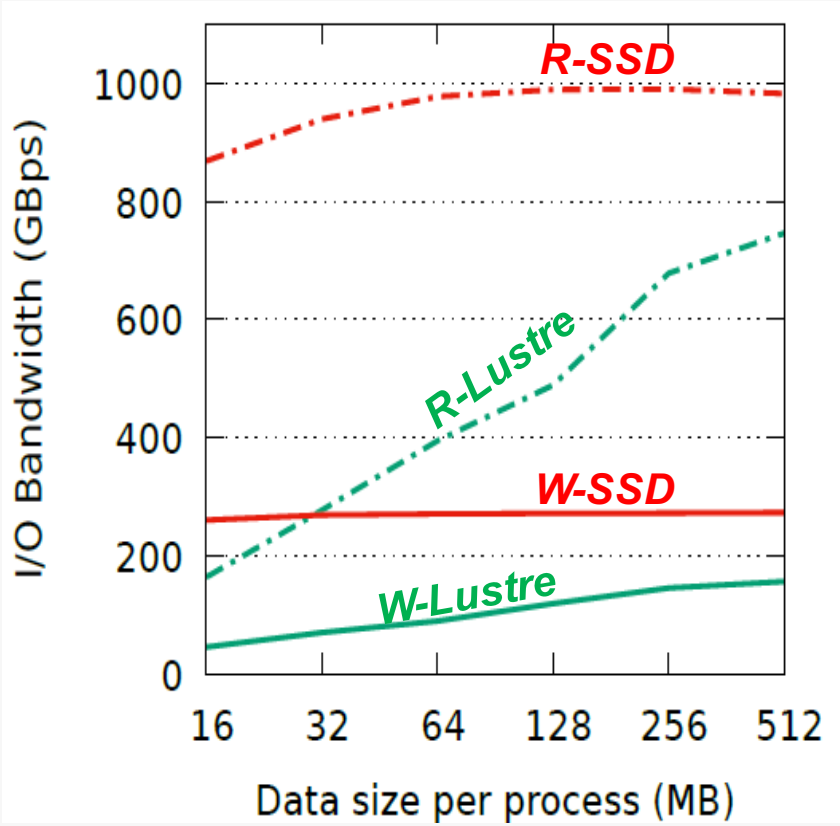
(b) **Contiguous** Data Mapping to File

Source: Tessier et al. TAPIOCA, IEEE Cluster 2017

Other Important Optimizations

- When reading/writing large data sets: Overlap data **compression/decompression** with I/O
- Use bonus I/O hardware whenever available: **Node-local SSDs** and Non-Volatile Memory (NVM)
  - Stage data
  - Cache data
  - Aggregate data
- Use burst buffers when they are available
  - *Not* covered in this talk

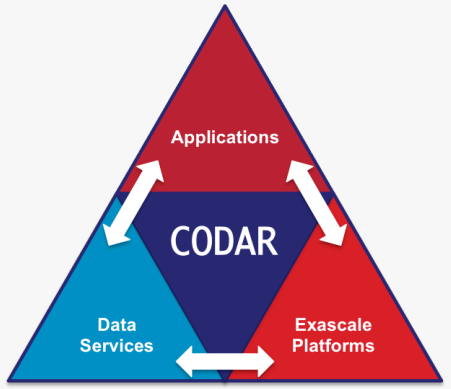
Node-local SSD vs Lustre
IOR, 1024 nodes, 2 ppn, 1 fpp



Example: ALCF Theta

Compression in HPC

- Compression/decompression is common for data-intensive HPC applications
- A range of popular compression libraries are used
  - File-per-process (and *single-writer*) can easily leverage compression
  - Manual implementation of MPI-IO + compression is also used
  - Support for compression in high-level *parallel* I/O libraries is improving (*more in next section*)



SZ: *Lossy Compression*

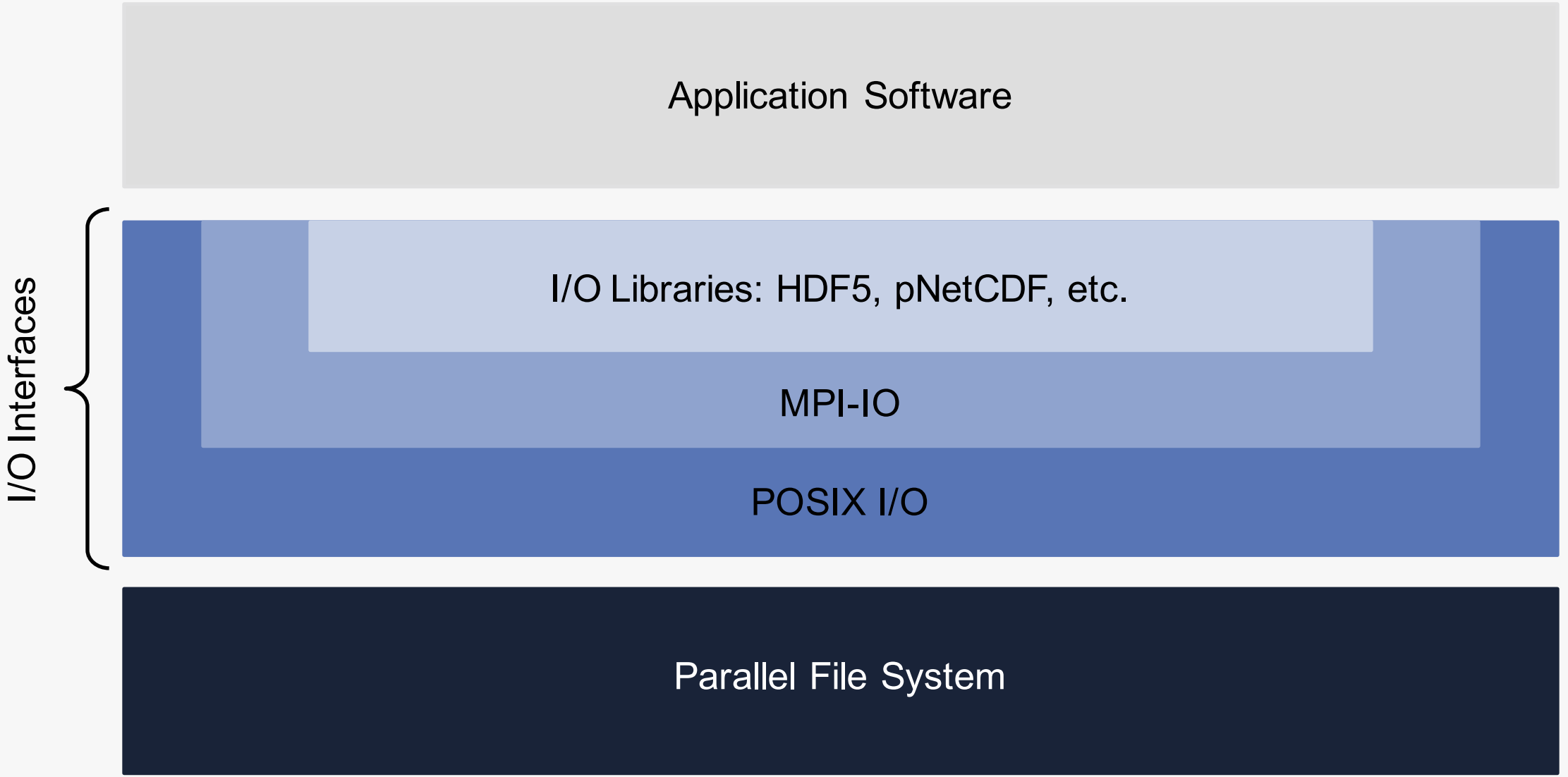
Scalable I/O Libraries

The common POSIX API can be used easily for FPP parallel I/O, but metadata and storage consumption is not scalable. MPI-IO and higher-level libraries are most popular for implementing scalable I/O workflows

Key Points:

- **MPI-IO** (an MPI replacement for POSIX) is popular in HPC
- **High-level** I/O libraries are key to productivity
- **Compression** support is improving in high-level I/O libraries

Common HPC I/O Software Stack



MPI-IO: MPI-Based POSIX-IO Replacement

- **POSIX has limitations**
  - Shared-file parallel I/O is possible, but complicated (parallel access, buffing, flushing, etc. must be explicitly managed)
- **Independent MPI-IO**
  - Each MPI task is handles the I/O independently using *non-collective* calls
    - Ex. `MPI_File_write()` and `MPI_File_read()`
  - Similar to POSIX I/O, but supports derived datatypes (useful for non-contiguous access)
- **Collective MPI-IO**
  - All MPI ranks (in a communicator) participate in I/O, and must call the same routines
    - Ex. `MPI_File_write_all()` and `MPI_File_read_all()`
  - Allows MPI library to perform collective I/O optimizations (often boosting performance)
- **Many HPC codes use MPI-IO or a high-level library (using MPI-IO)**
  - Python codes can use the `mpi4py` implementation of MPI-IO

*More code is needed for functionality

A Simple MPI-IO Example*

Simple MPI-IO code to concatenate local a 1-D arrays (on each rank) into a single global array in a file.

***Note:** Grey* **= Optional**

```
// Create array to write (localbuf)
localbuf = (int *) malloc((N / size) * sizeof(int));
for(i=0; i<(N / size); i++) localbuf[i] = i;

// Determine file offset
offset = (N / size) * rank * sizeof(int);

// Let rank 0 Create the file
if(rank == 0){
  MPI_File_open( MPI_COMM_SELF, filename, MPI_MODE_CREATE|MPI_MODE_WRONLY, info, &fh );
  MPI_File_set_size( fh, filesize );
  MPI_File_close( &fh );
}

// Open the file for writing
MPI_File_open( MPI_COMM_WORLD, filename, MPI_MODE_WRONLY, info, &fh );
MPI_File_set_atomicity( fh, 0 );

// Write the file
MPI_File_write_at_all( fh, offset, localbuf, (N/size), MPI_INT, &status );

// Close the file
MPI_File_close( &fh );
```

Creating and pre-allocating the file on a single rank can avoid metadata overhead

Specifying that "atomic" ordering is unnecessary

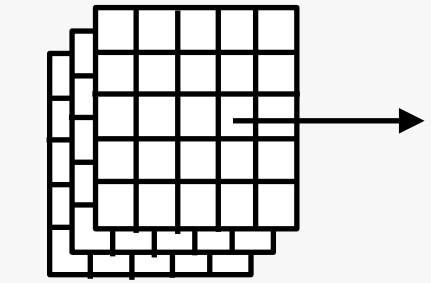
Collective write, starting at "offset"

High-level I/O Libraries

- High level I/O libraries provide an abstraction layer above MPI-IO/POSIX (but can sometimes talk directly to the low-level storage API)
  - Parallel HDF5, NetCDF, and ADIOS leverage MPI-IO (although ADIOS doesn't need to)
- Here we will focus mostly on HDF5
  - NetCDF-4 uses HDF5 as a backend
  - ADIOS is also a performant/popular option

Hierarchical Data Format 5 (HDF5)

- HDF5 is a high-performance I/O technology suite, designed for complex data sets
  - The HDF5 technology suite is composed of a data model, a file format, an API, a library, and a set of software tools
  - An HDF5 file organizes data into **datasets** organized within a **group** structure (like a file-system within a file)
  - **Datasets** are effectively **N-dimensional arrays** of elements (elements can also be complex data types)
  - `h5py`: 3rd-party python interface



Ex: 3D Dataset

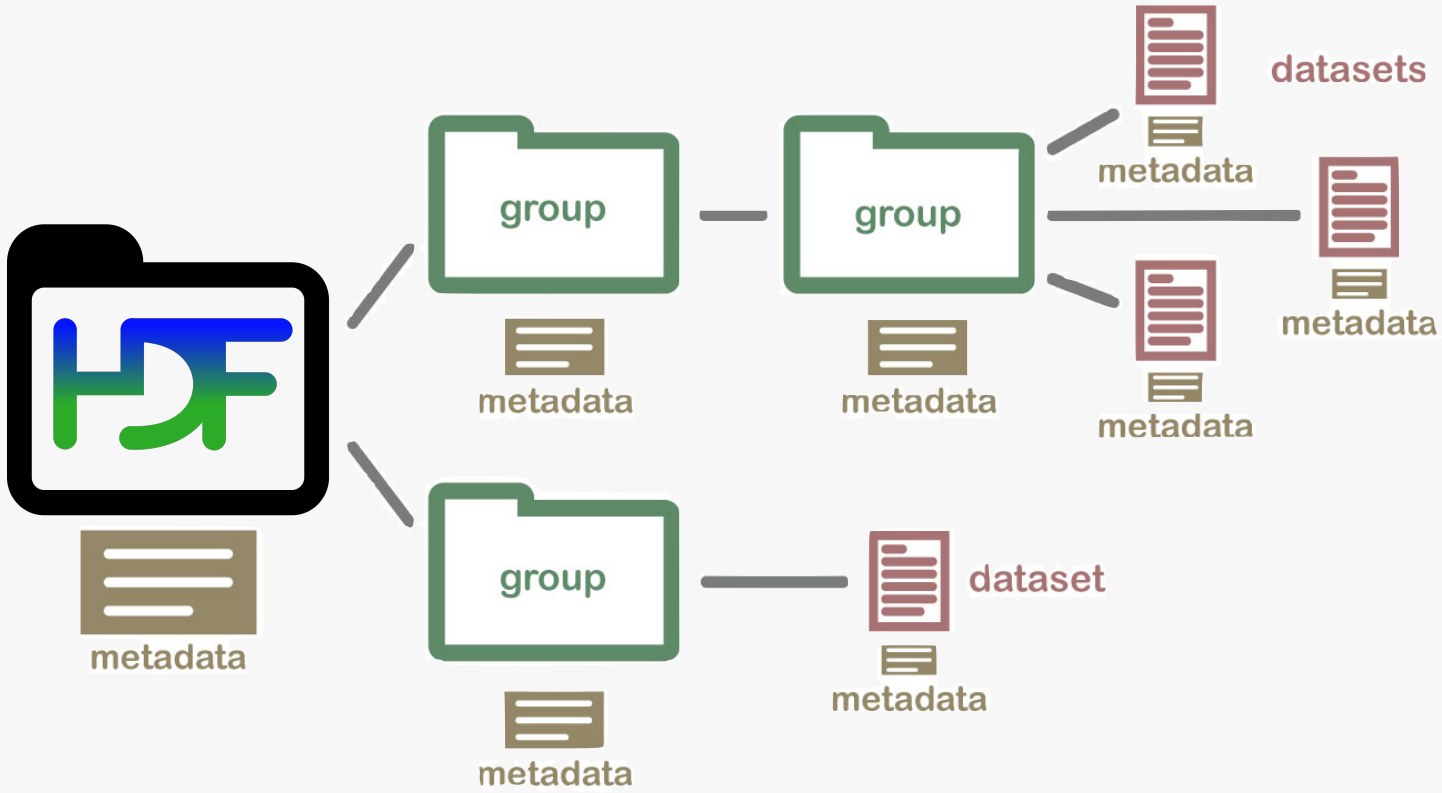


Image Source: https://www.neonscience.org/about-hdf5

HDF5: Partial and Parallel I/O

- HDF5 allows the application to read/write isolated datasets, as well as **points** and **slices** from datasets
  - Don’t need to load all data from the file, or even a single dataset.
  - N-dimensional slices are called **hyperslabs**
- **Parallel HDF5** is build on top of MPI-IO
  - MPI-IO performance determines HDF5 performance
  - Hyperslabs are internally represented as **MPI datatypes**
  - **Collective I/O**: User can tell HDF5 to use collective MPI-IO
  - **Independent I/O**: Independent MPI-IO routines are used by default
    - However, many HDF5 functions must be called by all processes (and are effectively collective)

Image Source: https://www.neonscience.org/about-hdf5

*More code is needed for functionality

A Simple Parallel HDF5 Example*

*Simple **HDF5** code to concatenate local a 1-D arrays (on each rank) into a single global array in a file.*

```
// Use Parallel I/O
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

// Create the file
file_id = H5Fcreate(filename, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);

// Create the file and memory data spaces
filespace = H5Screate_simple(1, dimsf, NULL);
memspace = H5Screate_simple(1, count, NULL);

// Create the dataset in the file
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace, … );

// Select a dataset hyperslab to write into the file
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL, count, NULL);

// Create property list for collective dataset write.
xplist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(xplist_id, H5FD_MPIO_COLLECTIVE);

// Write the data
status = H5Dwrite(dset_id, H5T_NATIVE_INT, memspace, filespace, xplist_id, data);
```

Note:** Red Boxes* = ***Parallel HDF5 Specific

Note: `h5py` code requires **fewer** lines :)

Chunking and Compression in HDF5

- HDF5 supports filters for **chunked** data
  - A filter is a data transformation (between memory and storage)
  - Chunks are contiguous in the file, but adjacent chunks in memory need not be adjacent in the file.
- **Compression** filters:
  - **gzip** (deflate), **Szip**, n-bit, scale-offset, and shuffling (with deflate)
- Parallel Compression (hdf5-1.10.2):
  - MPI ranks can compress/decompress and read/write in parallel

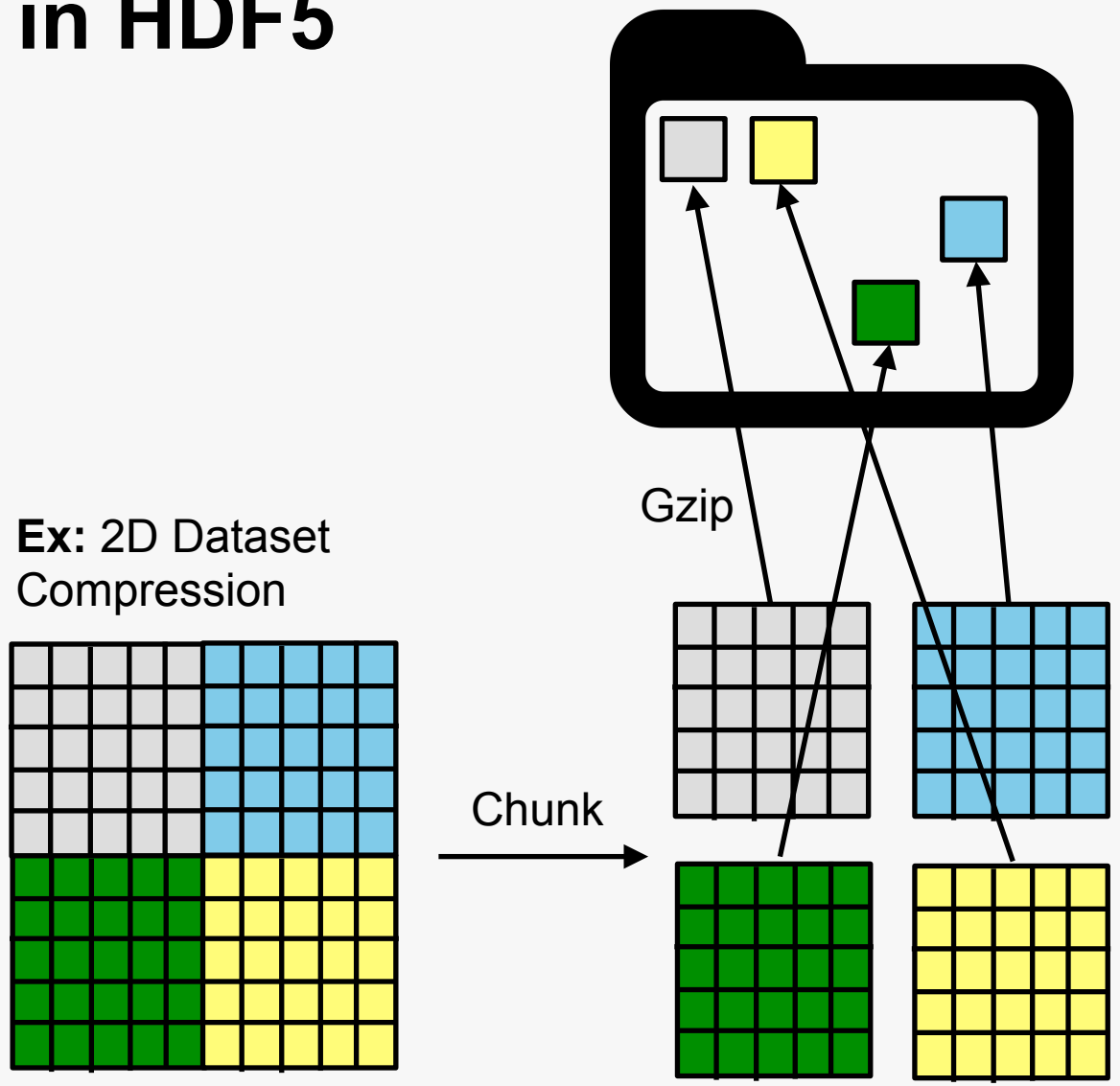
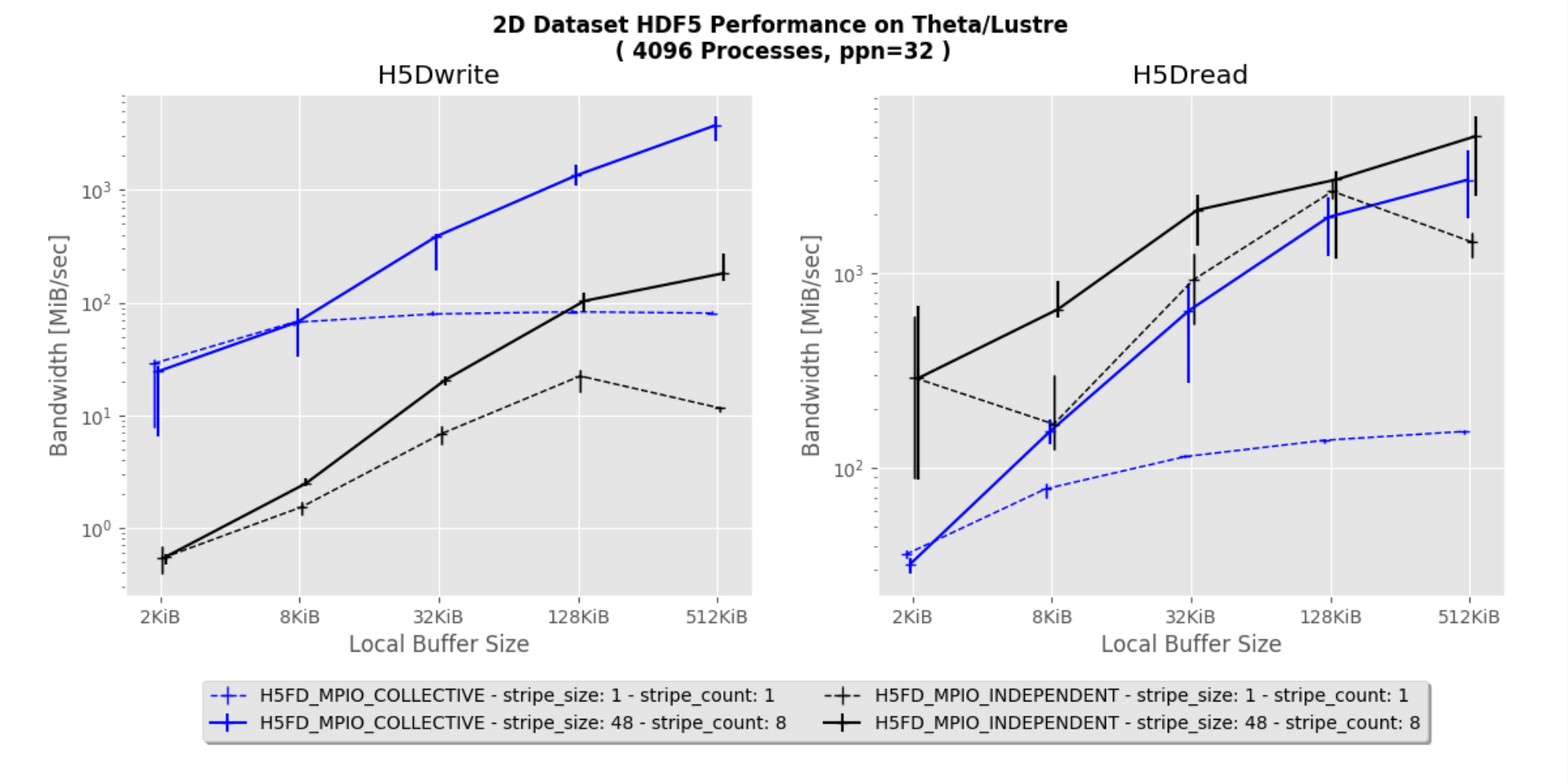
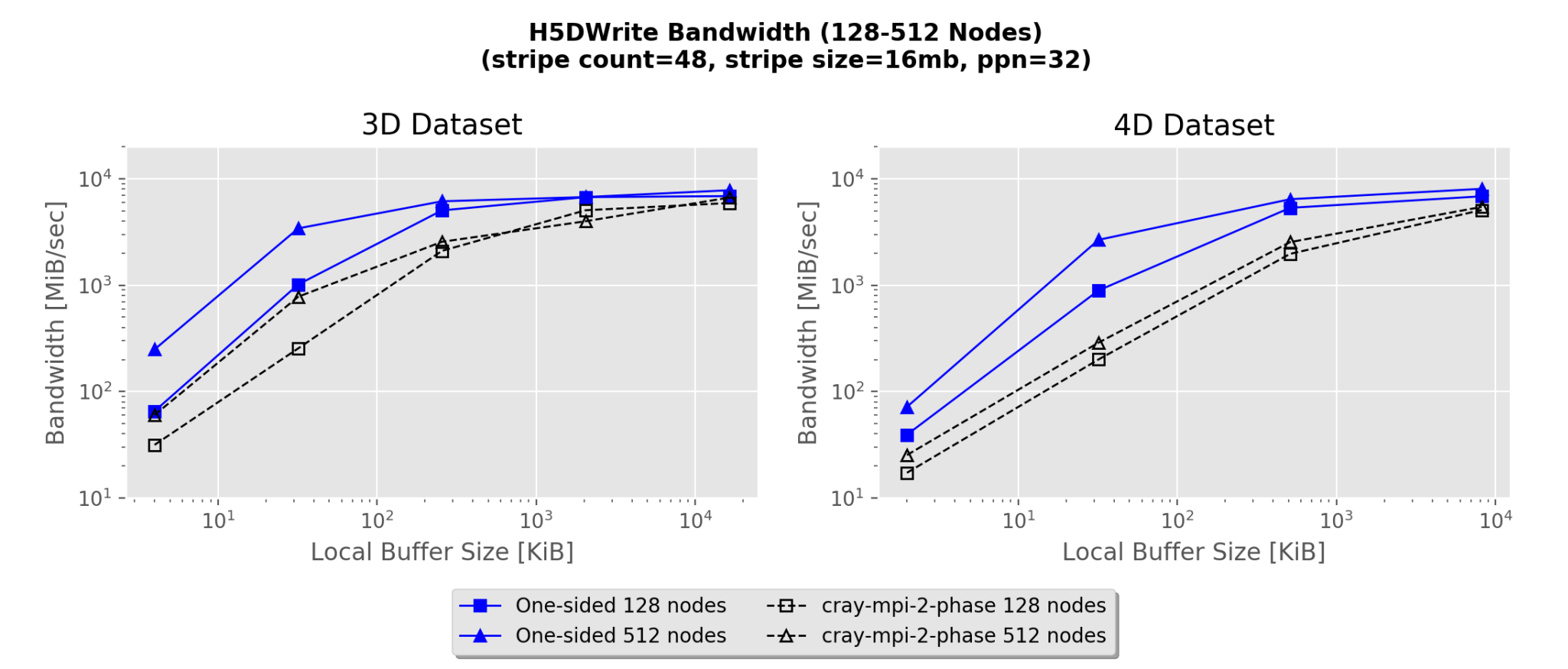


Image Source: https://www.neonscience.org/about-hdf5

Performance Depends on *Algorithm* and *PFS*



Collective I/O Details Matter in HDF5



ExaHDF5

EXASCALE COMPUTING PROJECT

Argonne NATIONAL LABORATORY

Summary

This talk has been a brief high-level overview of HPC I/O. Many topics (that I did and didn't mention) deserve far more detail (Ex: Key-value *object* storage, Burst-buffers, containerization, etc…)

Takeaways:

- Good HPC I/O Performance usually requires PFS-specific tuning of I/O
- Collective aggregation, Sub-filing, Compression and NVM utilization are all important
- MPI-IO and other (high-level) I/O libraries are valuable (but not perfect)

Thank You!