TUTORIALS

Modern High Performance I/O: Leveraging Object Stores

Rob Latham, Argonne National Laboratory (robl@mcs.anl.gov)

Adrian Jackson, EPCC (a.jackson@epcc.ed.ac.uk)

Mohamad Chaarawi, HPE (mohamad.chaarawi@hpe.com)

Glenn Lockwood, VAST (glenn.lockwood@vastdata.com)

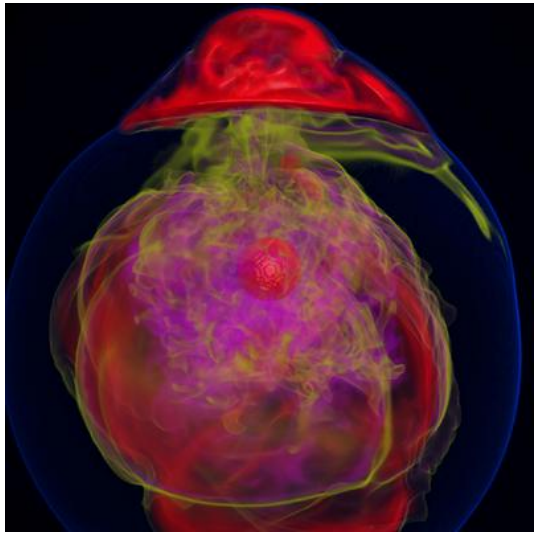
About us

- Rob Latham (robl@mcs.anl.gov)
  - Research Software Developer, MCS Division, Argonne National Laboratory
  - ROMIO MPI-IO implementation
  - Parallel netCDF high-level I/O library
  - Application outreach
- Adrian Jackson (a.jackson@epcc.ed.ac.uk)
  - Professor of High Performance Computing Technologies, EPCC, University of Edinburgh
  - I/O and application optimisation research
  - Object storage curious
- Mohamad Chaarawi
  - Distinguished Technologist, HPE
  - All things DAOS
  - Parallel I/O
- Glenn K. Lockwood (glenn.lockwood@vastdata.com)
  - Technical Strategist, VAST Data
  - Former system architect at Microsoft, NERSC/Berkeley Lab
  - Scalable infrastructure for AI
  - Former IOR and Darshan contributor

Computational Science

- Computer simulation as a tool promotes greater understanding of the real world
  - Complements experimentation and theory
- Problems are increasingly computationally expensive
  - Large parallel machines are needed to perform calculations
  - Leveraging parallelism in all phases is critical
- Data access is a huge challenge and includes
  - Using parallelism to obtain performance
  - Finding usable, efficient, and portable interfaces
  - Understanding and tuning I/O

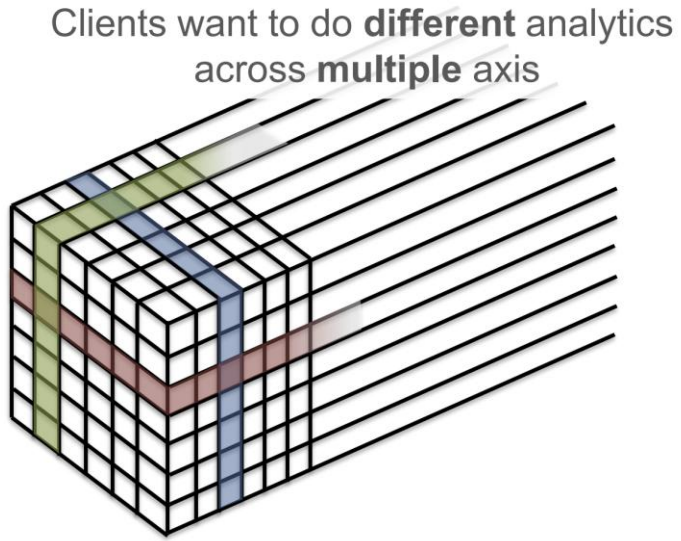
HPE/Inel Aurora system at Argonne National Laboratory.

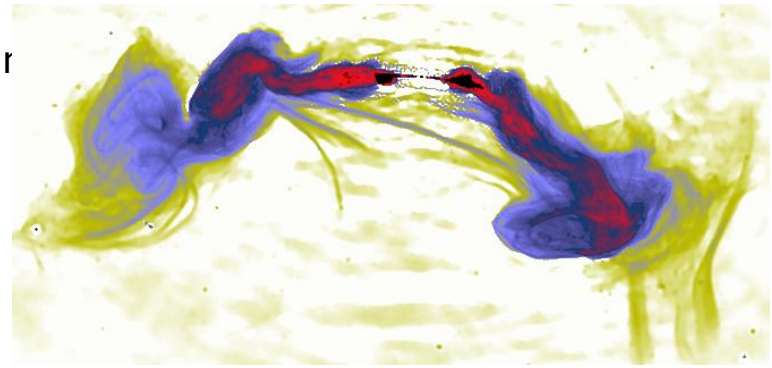


Visualization of entropy in Terascale Supernova Initiative application. Image from Kwan-Liu Ma's visualization team at UC Davis.

Object Storage

- Can enable efficient/fast access to data in different structures
  - Supports different creation, querying, analysis, and use patterns
- Granular storage with rich metadata
  - Data retrieval leverages metadata
  - **Build data structure on the fly**
- Examples:
  - Weather/climate
    - Pursuing optimal I/O for applications
      - Weather forecasting workflows
    - End-to-end workflow performance important
    - Simulation (data generation) only one part
      - Consumption workloads different layout/pattern from productior
  - Radio astronomy
    - Data collected and stored by antenna (frequency and location) and capture time
    - Reconstruction of images done in time order
    - Evaluation of transients or other phenomenon undertaken across frequency and location





Aims

- Start thinking about how application data and storage systems interact
- Understand object storage hardware and software
- Learn about DAOS, Ceph, and other object stores
- Programming interfaces ranging from low level byte movers to more sophisticated descriptions
- How to see through the abstraction layers
- How to evaluate these novel systems

Today's Plan

- Object Storage and hardware
- Benchmarking
- DAOS
- AM Break
- DAOS interfaces
- Programming DAOS
- Other Object Stores
- Lunch
- MPI-IO
- Darshan
- I/O libraries
- PM Break
- I/O Libraries (ctd)
- System evaluation
- AI Workloads

Summary

- Object Stores represent an interesting transformation from traditional file systems
- Tools and libraries exist to make the transition easy – maybe even invisible
- Please don’t hesitate to ask questions!
- Source code, examples, scripts all available
  - https://github.com/adrianjhpc/ObjectStoreTutorial

Storage Approaches and Object Storage

Adrian Jackson, EPCC

a.jackson@epcc.ed.ac.uk

NOVEMBER 2025

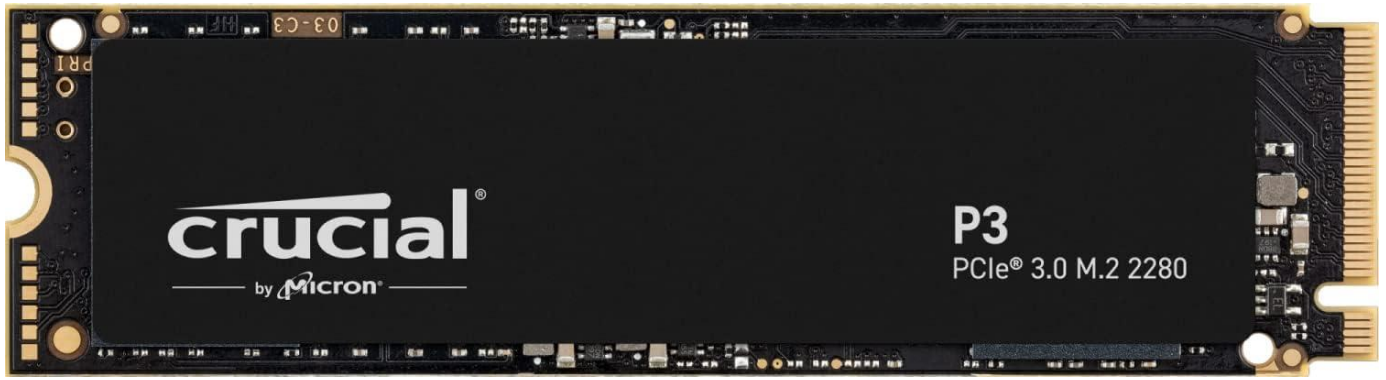
Object Storage

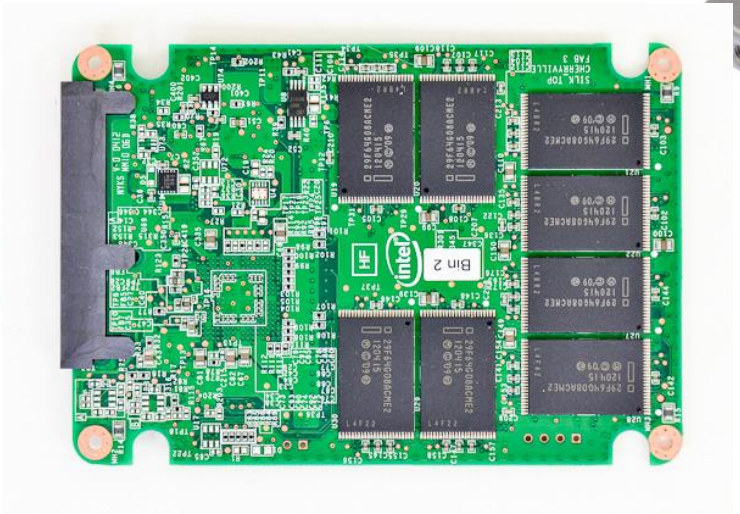
- Design and performance considerations are the challenge
  - Programming against the interfaces is (relatively) easy(ish)
  - Direct use often straight forward (i.e. filesystem interfaces)
  - More intelligent functionality takes more work/more specialised

- Design for functionality
  - When to store, when transactions should happen, what granularity I/O operations should be, what failures can you tolerate, etc..

- Design for performance
  - Memory size, I/O, data access costs, etc…

Storage


crucial
by Micron
P3
PCIe® 3.0 M.2 2280




crucial
2.5-inch SOLID STATE DRIVE

Filesystems

- Lots of ways to store data on storage devices
- Filesystems have two components:
  - Data storage
  - Indexing
- Data stored in blocks
  - Chunks of data physically stored on hardware somewhere
- Indexing is used to associate names with blocks

  - File names are the index
  - Files may consist of many blocks
- Variable sized nature of files makes this a hard problem to solve

Filesystems

- Data storage managed by the filesystem
  - Lots of different options (xfs, ext3, ext4, zfs, etc…)
- Programming interface provided by operating system
  - POSIX (Portable Operating System Interface) I/O interface
  - Gives program portability and data access abstraction
    - Defines standard operations: mount, umount, open, close, write, read, stat, chown, etc…
    - Defines filename and directory name structures/requirements
    - Requires strong (sequential) consistency, i.e. all writes be immediately visible to all subsequent reads
- Does not support parallel or HPC I/O well
  - Designed for one active writer
  - Consistency requirements hamper performance
  - Has a bunch of functions that can impact performance, i.e. locking (flock, etc…)
- Some filesystems/approaches relax POSIX semantics to improve performance
  - Moving beyond filesystems allows other semantics to be targeted

Object storage

- Data stored in unstructured objects
  - Data has identifier
  - Size and shape can vary
  - Metadata can also vary
- Originally designed for unstructured data sets
  - Bunch of data with no specific hierarchy required
- Can also enable efficient/fast access to data in different structures
  - Supports different creation, querying, analysis, and use patterns
- Key-Value storage but with additional functionality
  - More scope for metadata and structure
  - Designed for larger values

Object storage

- Filesystems use Files
  - container for blocks of data
  - lowest level of metadata granularity (not quite true)
- Object stores use Objects
  - container for data elements
  - lowest level of metadata granularity
- Allows individual pieces of data to be:
  - Stored
  - Indexed
  - Accessed separately
- Allows independent read/write access to “blocks” of data

Object storage

- Generally restricted interface
  - Put: Create a new object
  - Get: Retrieve the object
- Removes the requirements for lots of functionality r.e. POSIX style I/O
- Traditionally objects are immutable
  - Once created cannot be changed
  - This removes the locking requirement seen for file writes
  - Makes updates similar to log-append filesystems, i.e. copy and update
  - Can cause capacity issues (although objects can be deleted)
- Object ID generated when created
  - Used for access
  - Can be used for location purposes in some systems

Object stores

- Often helper services and interfaces
  - Manage metadata
  - Permissions
  - Querying
  - Etc…
- Distribution and redundancy etc… part of the complexity
  - Often eventual consistency
- Lots of complexity in implementations
- Commonly use web interfaces as part of the Put/Get interface

S3 – Simple Storage Service

- AWS storage service/interface
  - Defacto storage interface for a range of object stores
- Uses a container model
  - Buckets contain objects
  - Buckets are the location point for data
  - Defined access control, accounting, logging, etc…
  - Bucket names have to be globally unique
- Buckets can be unlimited in size
  - Maximum object size is 5TB
  - Maximum single upload is 5GB
- A bucket has no object structure/hierarchy
  - User needs to define the logic of storage layout themselves (if there is any)
- Fundamental operations corresponding to HTTP actions:
  - http://bucket.s3.amazonaws.com/object
  - POST a new object or update an existing object.
  - GET an existing object from a bucket.
  - DELETE an object from the bucket
  - LIST keys present in a bucket, with a filter.

S3

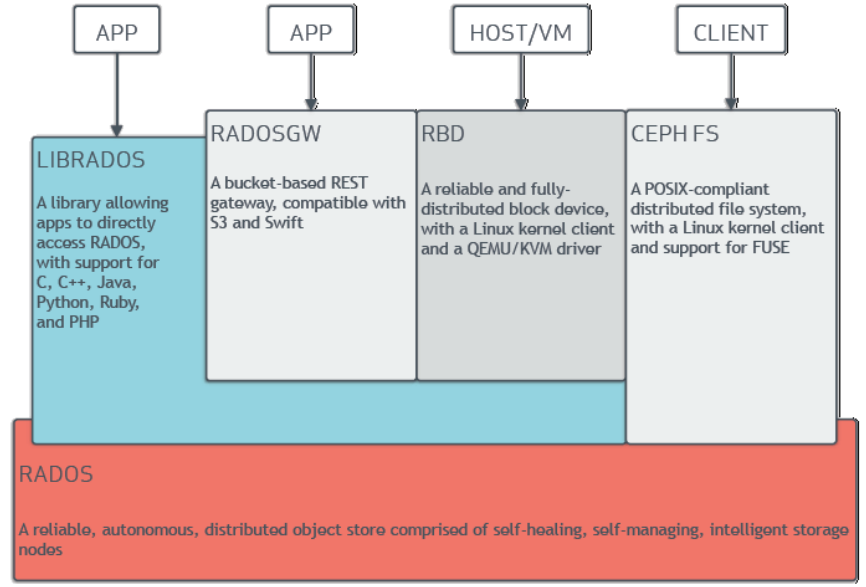
- Objects are combination of data and metadata
- Metadata is name – value pair (key) identifying the object
  - Default has some other information as well:
    - Date last modified
    - HTTP Content-Type
    - Version (if enabled)
    - Access Control List (if configured)
  - Can add custom metadata
- Data
  - An object value can be any sequence of bytes (up to 5TB)
  - Multi-part upload to create/update objects larger than 5GB (recommended over 100MB)

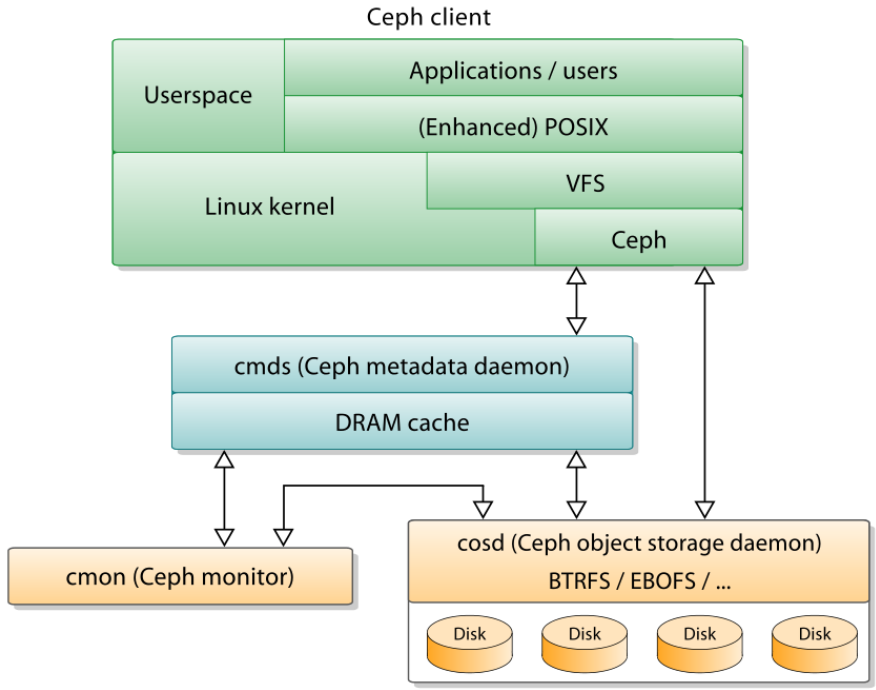
S3 Consistency Model

- Strong RAW (read after write) consistency
  - PUT (new and overwrite) and DELETE operations
  - READ on metadata also strong consistency
  - Across all AWS regions
- Single object updates are atomic
  - GET will either get fully old data or fully new data after update
  - Can’t link (at the S3 level) key updates to make them atomic
- Concurrent writers are *racy*
  - No automatic locking
- Bucket operations are eventually consistent
  - Deleted buckets may still appear after the delete has occurred
  - Versioned buckets may take some time to setup up initially (15 minutes)

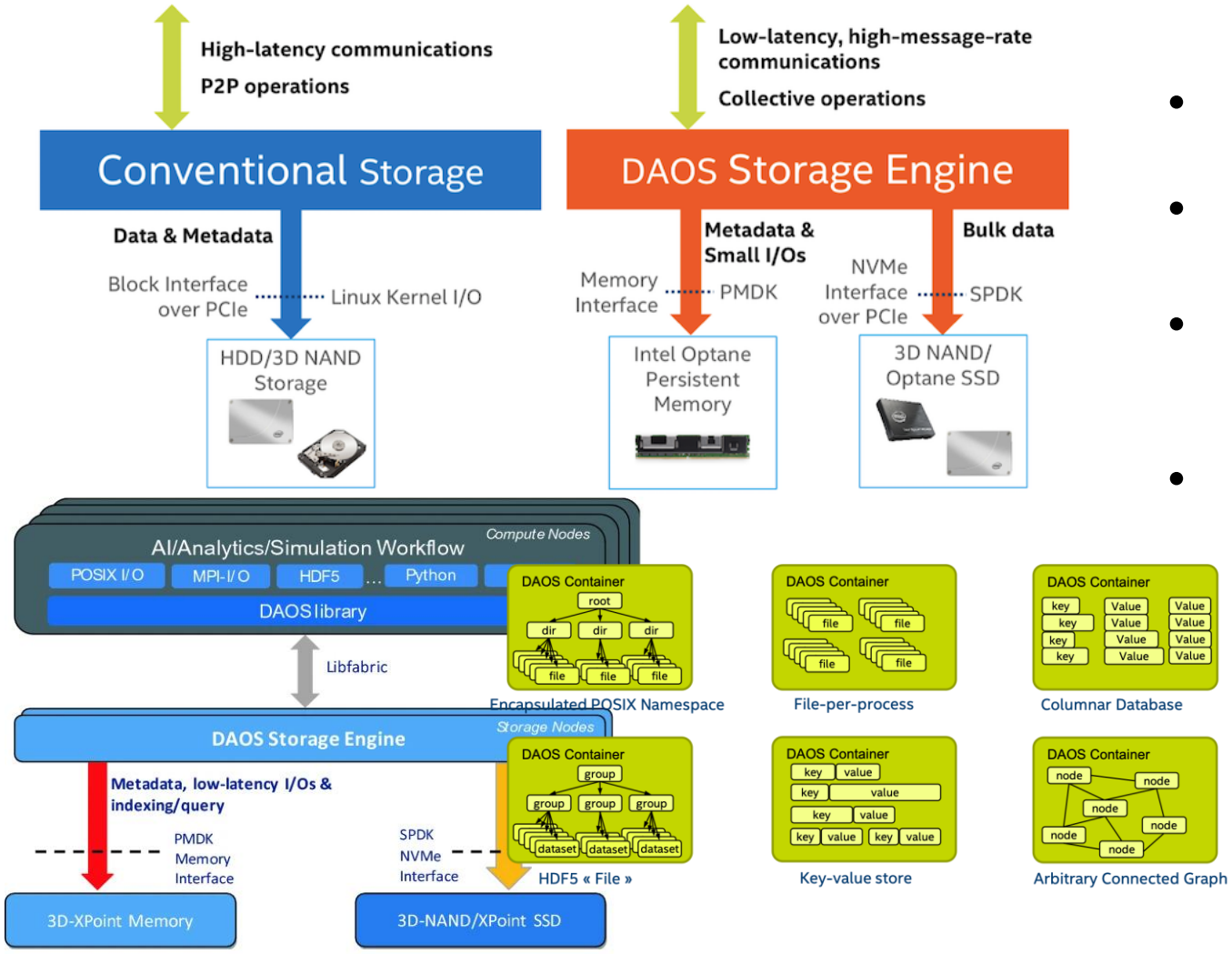
Ceph

- Widely used object store from academic storage project
- Designed to support multiple targets
  - Traditional object store: RadosGW → S3 or Swift
  - Block interface: RBD
  - Filesystem: Ceph FS
  - Lower-level object store: LibRados
- Distributed/replicated functionality
  - Scale out by adding more Ceph servers
  - Automatic replication/consistency
    - replication, erasure coding, snapshots and clones
- Supports striping
  - Has to be done manually if using librados
- Supports tiering
- Lacking production RDMA support





DAOS



- Native object store on non-volatile memory and NVMe devices and designed for HPC
- Pools
  - Define hardware range of data
- Containers
  - User space and data configuration definitions
- Objects
  - **Multi-level key-array** API is the native object interface with locality
  - **Key-value** API provides a simple key and variable-length value interface. It supports the traditional put, get, remove and list operations.
  - **Array API** implements a one-dimensional array of fixed-size elements addressed by a 64-bit offset. A DAOS array supports arbitrary extent read, write and punch operations.

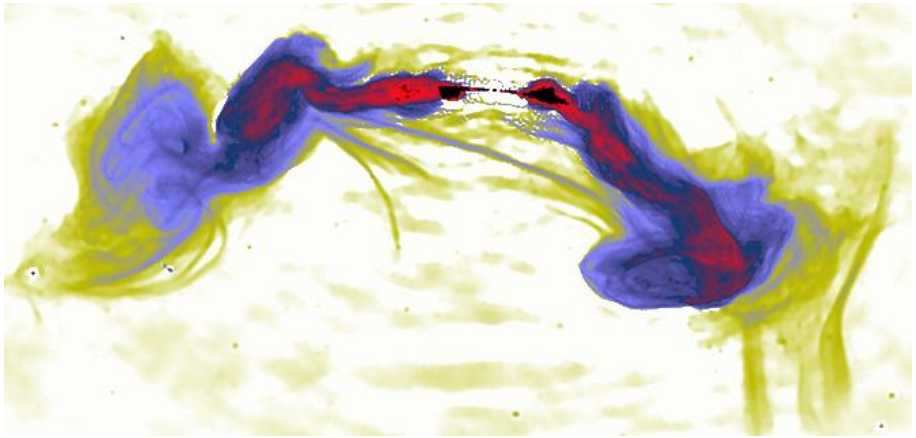
Object stores

- High performance object stores offer:
  - Server-side consistency by default
    - reducing round trip messaging for some operations
  - Distributed metadata functionality
    - no single performance bottleneck
  - Small object size performance
    - non-kernel space I/O operations so don’t have interrupt/context switch performance issues
    - Designed for faster hardware and for large scale operation
  - In-built redundancy control/configuration
  - Multi-versioning and transactions to reduce contention/provide consistency tools
  - Scaling across storage resources
  - Searching/discovery across varying data dimensions

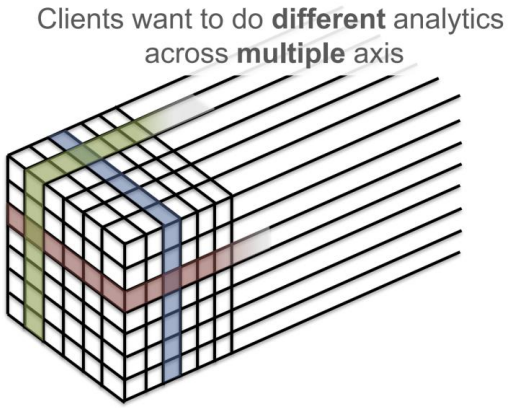
Object stores

- Object can have as much or as little complexity as you want
  - Single array
  - Single key-value
  - Nested object containing table of entries
  - etc..
- High performance object stores can’t:
  - Beat filesystems for bulk I/O with low metadata overheads
  - Support high performance alternative functionality without porting effort
  - Eliminate server-side contention
  - Fix poor storage design
  - Create your data layout and indexing for you
  - Fix configuration/resource issues

- Object Stores can unlock previously expensive I/O patterns



- Enable discovery as well as storage



Practical Setup

- https://github.com/adrianjhpc/ObjectStoreTutorial/Exercises/exercisesheet.pdf
- Take IOR source code
- Run on the GCP system
- SSH to provided IP
- You will get a username
  - tuXXX
  - And a ssh key

DAOS Tutorial

Mohamad Chaarawi, HPE

NOVEMBER 2025

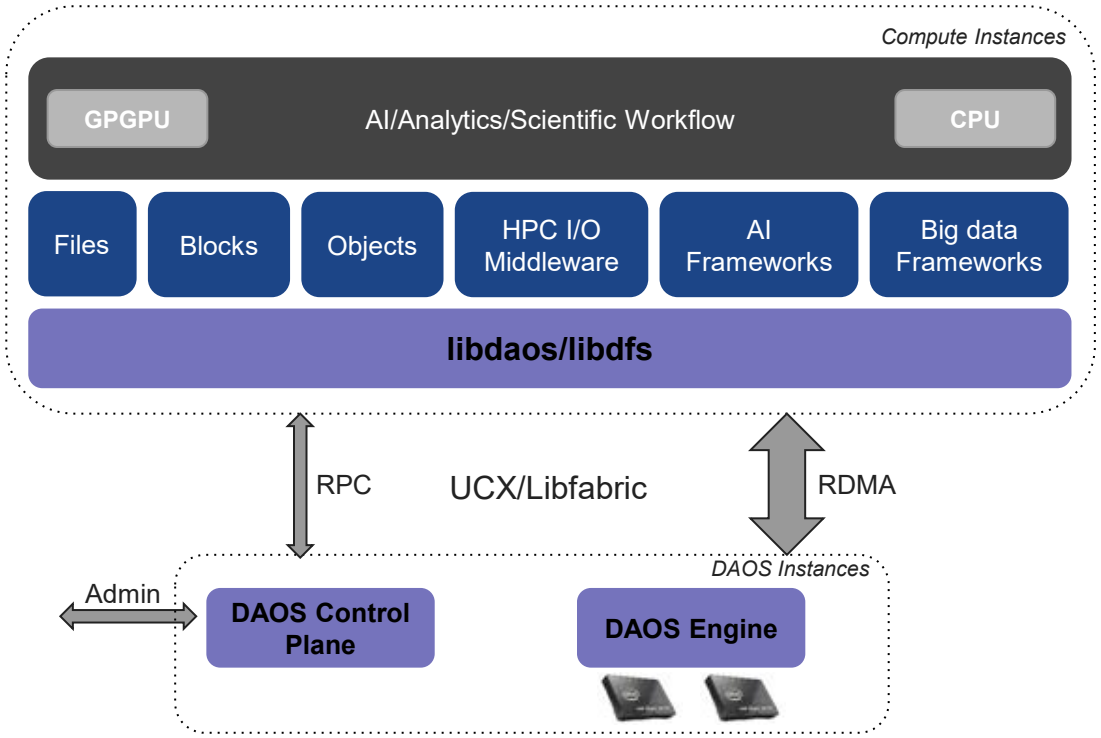
SC25 ST. LOUIS hpc ignites.

Presentation Outline

- DAOS Intro and Overview
- Pools:
  - Connect, disconnect
- Containers:
  - Create, destroy, open, close
- Objects: access APIs based on type
  - Flat KVS
  - Global array
  - Multi-level KVS
- POSIX Support:
  - DFS API (for modified applications to use daos)
  - dfuse mount, interception libraries (for unmodified applications to use daos)
  - Best practices

DAOS: Nextgen Open Storage Platform

- Open platform for innovation
- Files, blocks, objects and more
- Full end-to-end userspace
- Flexible built-in data protection
  - EC/replication with self-healing
  - No performance impact on reads
- Flexible network layer
- Strong distributed consistency
  - Version-based with MVCC
- Efficient single server
  - O(100)GB/s and O(1M) IOPS per server
- Highly scalable
  - TB/s and billions IOPS of aggregated performance
  - O(1M) client processes
- Time to first byte in O(10) µs

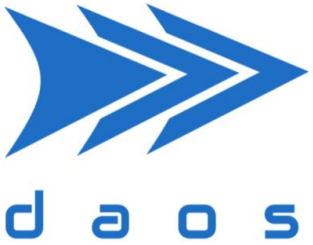


DAOS Use Cases

- Checkpoint/restart
  - High bandwidth and large capacity
  - Integration with HPC frameworks (HDF5, MPIIO, …)
- Out-of-core data / irregular accesses
  - Input/output data for simulation
  - High IOPS/bandwidth and low latency
  - Native POSIX and integration with HPC frameworks (HDF5, MPIIO, …)
- Analytics
  - High IOPS and low latency
  - Integration with Apache ecosystem (Spark)
- AI/ML
  - Optimizations for read-only workloads
  - High write bandwidth for checkpointing
  - Low latency **KVCache** for inference (VLLM or Dynamo)
  - Large capacity
  - Integration with AI framework (PyTorch)

What makes DAOS different from Lustre and other file systems?

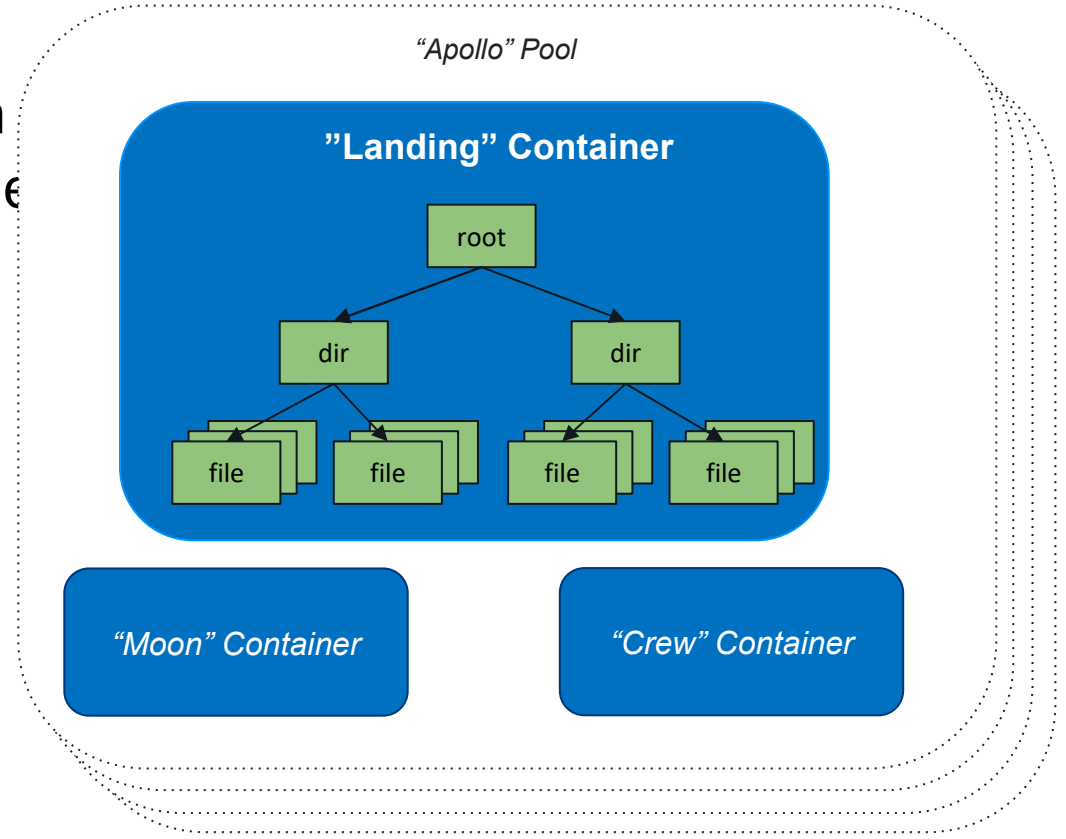
Enabling extreme scalability, performance and application integration for specific use cases



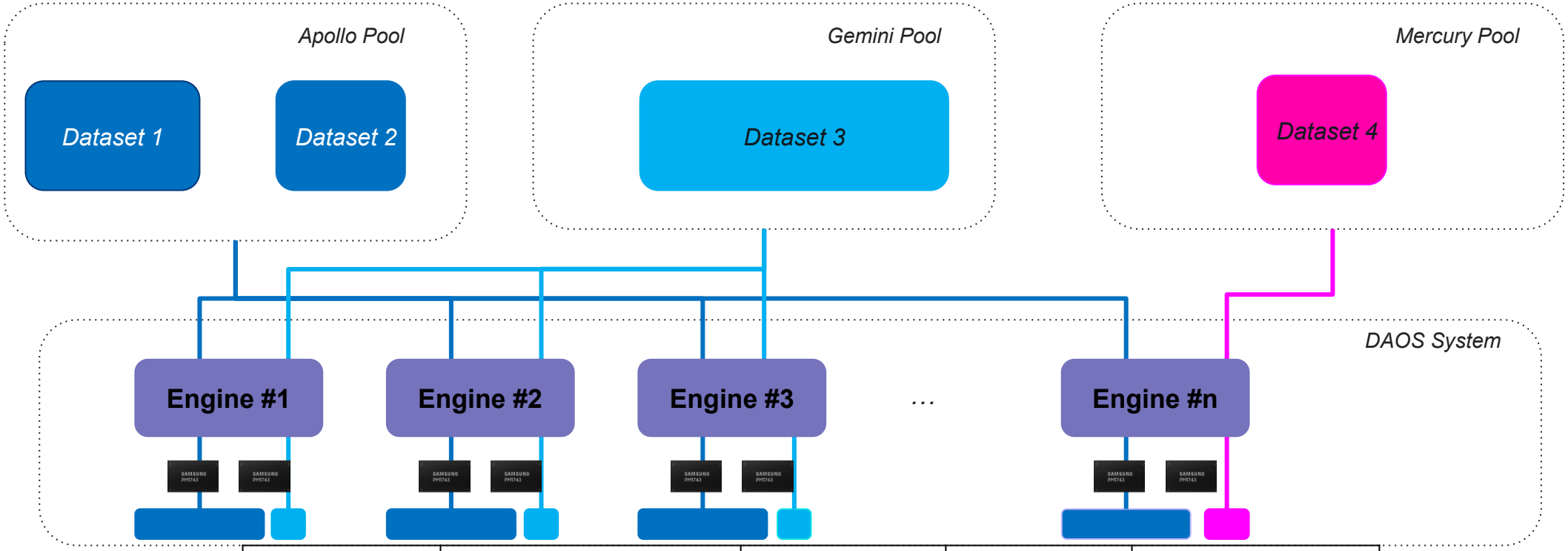
- Distributed key-value store / Versioned byte-granular I/Os
- No synchronous read-modify-write / No locking
- End-to-end in user space / No kernel code
- No centralized metadata servers / No global object table
- End-user managed snapshots / Self-healing data protection
- Multi-tenant storage pooling / Fine-grained access control
- Rich client software ecosystem (e.g. TensorFlow / PyTorch)

DAOS Concepts

- Pool
  - private partition/tenant allocated to a
  - distributed across all the storage node
    - Maximum BW/IOPS regardless of size
  - O(100) pools in a system
- Container
  - Dataset/bucket inside a pool
  - O(100) containers in a pool
  - e.g. checkpoints from May campaign
- Object
  - Array or key-value store
  - O(1T) objects in a container
  - e.g. files and directories in a POSIX container



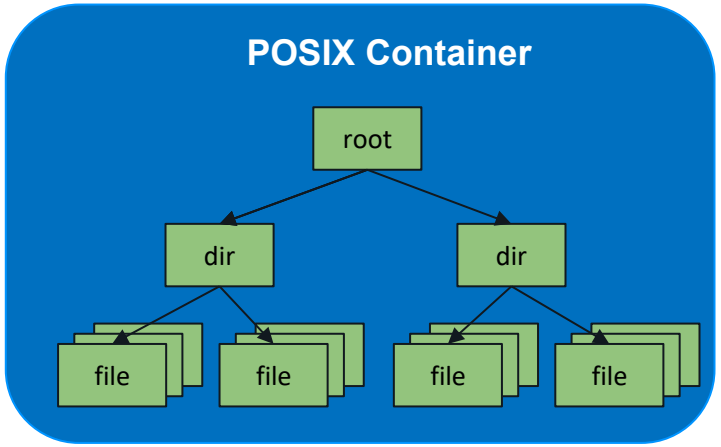
Storage Pooling And Multi-tenancy

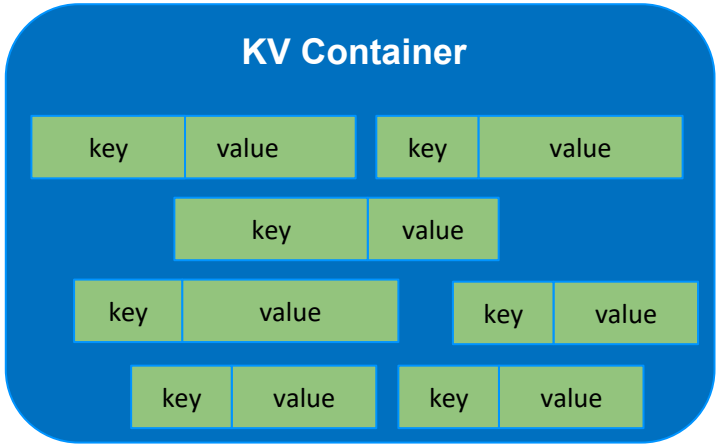


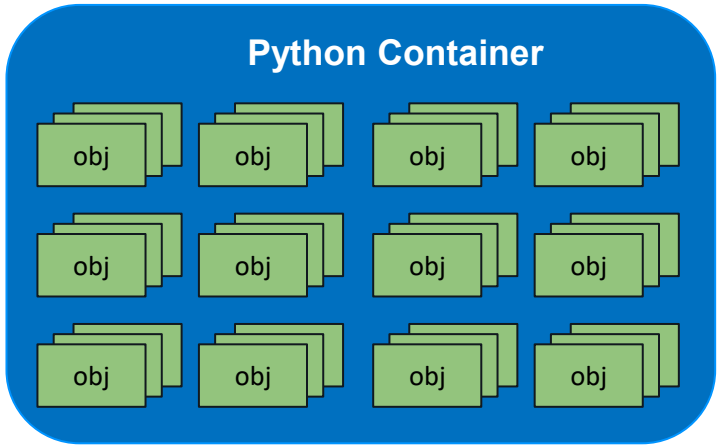
Pool 1	Apollo Pool	100PB	20TB/s	200M IOPS
Pool 2	Gemini Pool	10PB	2TB/s	20M IOPS
Pool 3	Mercury Pool	30TB	80GB/s	2M IOPS

DAOS Containers = Datasets/Buckets

- New data model to unwind 30+y of file-based management
- Introduce notion of dataset = container
- Basic unit of storage
- Containers have a type (e.g. POSIX, pyDAOS, …)
- POSIX containers can include trillions of files/directories
- Advanced dataset query capabilities
- Unit of snapshots
- ACLs
- Best practices:
  - 100's containers

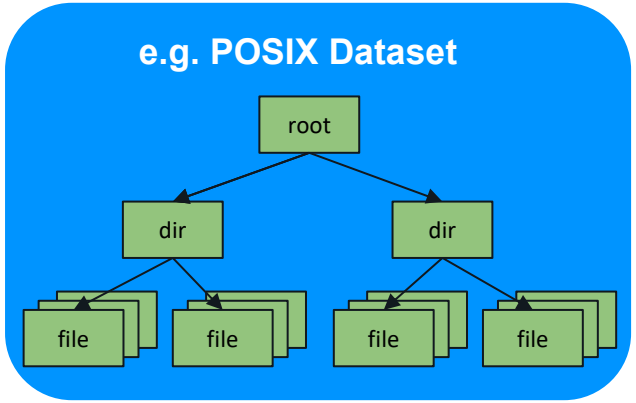






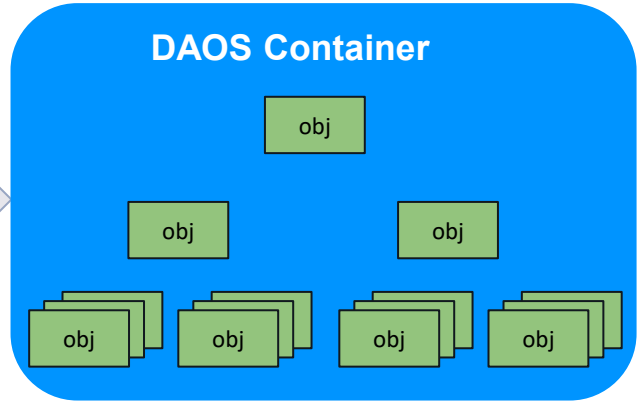
Object Interface

Middleware/Framework View



Mapping

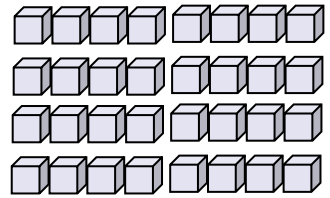
DAOS Layout View



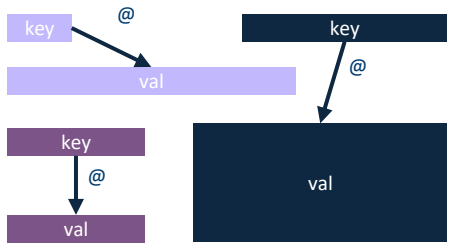
Object

128-bit
object Identifier

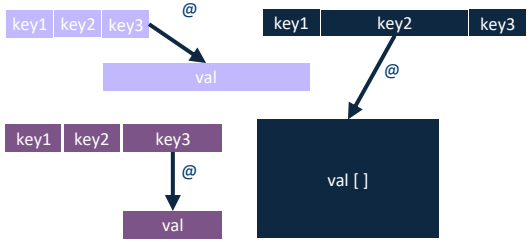
Array



Kev-value Store



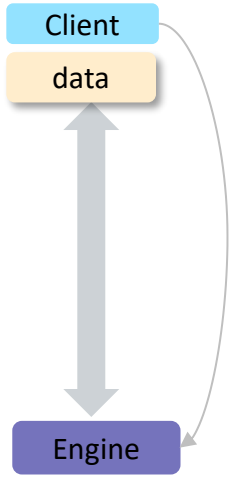
Multi-level
Key-value Store

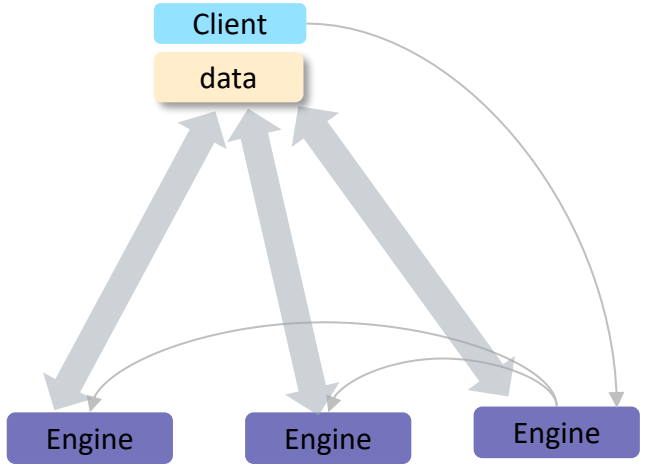


- No object create/destroy
- No size, permission/ACLs or attributes
- Sharded and erasure-coded/replicated
- Algorithmic object placement
- Very short Time To First Byte (TTFB)

Data Protection and Performance Trade-offs

No Data Protection
Full bandwidth/IOPS on
both read & write

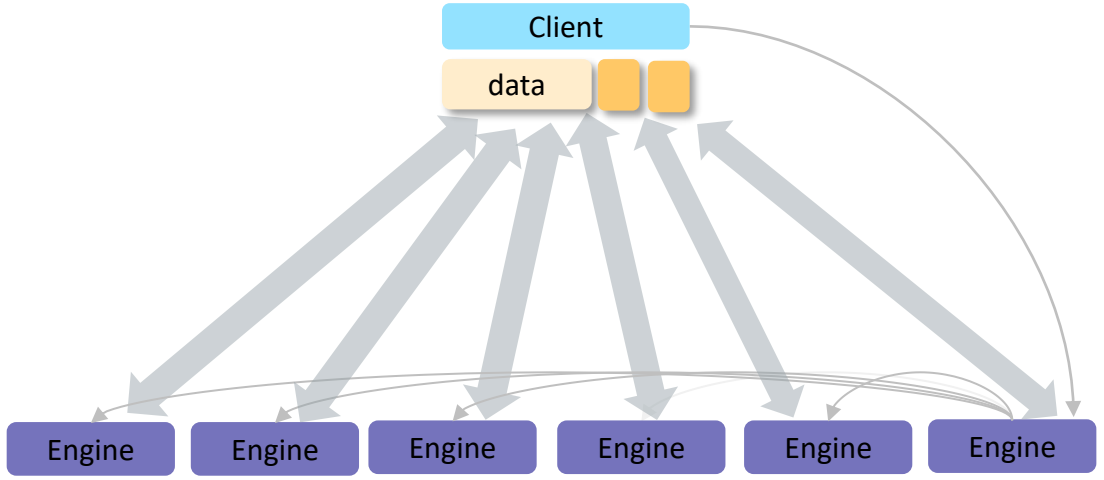




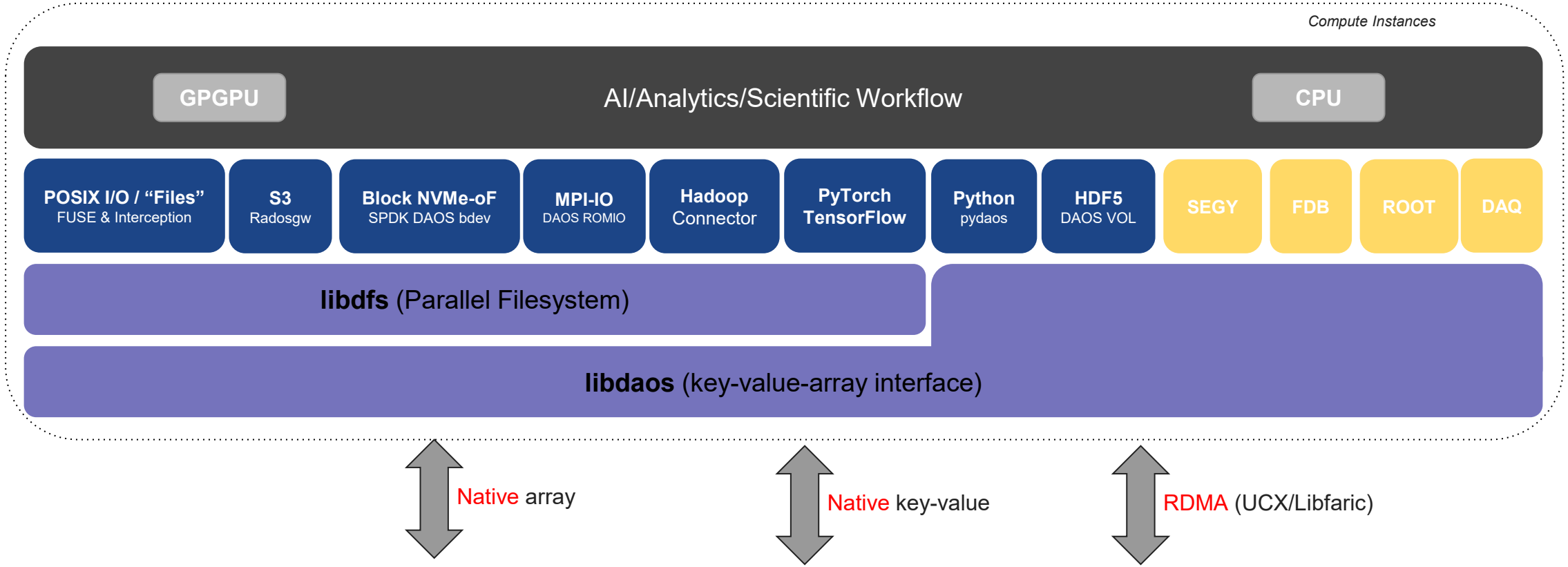
3-way Replication
Full bandwidth/IOPS on read
66% loss on write

16+2 Erasure Code
Full bandwidth/IOPS on read
12% loss on full stripe write
66% loss on partial write

4+2 Erasure Code
Full bandwidth/IOPS on read
33% loss on full stripe write
66% loss on partial write
(replicated turned into EC in background)



Software Ecosystem



DAOS API Usage, Program Flow

- Initialize DAOS stack
- Connect to a Pool
- Create / open a container
- Access an object in the container through the unique OID
  - Open object
  - update/fetch/list
  - Close object
- Close / disconnect from container & pool, finalize DAOS stack

Program Flow – Initialize DAOS, Connect to a Pool

- First (typically): initialize DAOS, connect to your pool:

```
int daos_init(void);
int daos_pool_connect(const char *pool, const char *sys,
        unsigned int flags, daos_handle_t *poh,
        daos_pool_info_t *info, daos_event_t *ev);
```

Note: pool already exists

Administrator used daos management utility "dmg" to create a pool – e.g.,

```
dmg pool create
--size=10TB mypool
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI

```
int daos_pool_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_pool_global2local(daos_handle_t poh,   d_iov_t *glob)
```

- Last, disconnect from your pool, finalize DAOS:

```
int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev);
int daos_fini(void);
```

Program Flow – Initialize DAOS, Connect to a Pool

- First (typically): initialize DAOS, connect to your pool:

```
int daos_init(void);
int daos_pool_connect(const char *pool, const char *sys,
        unsigned int flags, daos_handle_t *poh,
        daos_pool_info_t *info, daos_event_t *ev);
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI

```
int daos_pool_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_pool_global2local(daos_handle_t poh, d_iov_t *glob)
```

- Last, disconnect from your pool, finalize DAOS:

```
int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev);
int daos_fini(void);
```

const char *pool: label string

daos_handle_t: opaque handle type
- "poh" - "pool open handle"
- "coh" – "container open handle"
- "oh" – "object open handle"

daos_pool_info_t:
- capacity, free space, (im)balance
- Health, rebuild state
- Also output by `daos_pool_query()`

d_iov_t:
- Refers to a contiguous app buffer

daos_event_t: (not covered here)
- Asynchronous API invoke/test

Program Flow – Create a Container

- Using the daos tool:

```
daos cont create mypool mycont
```

```
Container UUID : 5d33d6e0-6c8b-4bf5-bb49-c8723bf30c91
Container Label: mycont
Container Type : unknown

Successfully created container 5d33d6e0-6c8b-4bf5-bb49-c8723bf30c91
```

User admin tool ‘daos’

API: input poh from pool_connect

Container also has a label id string

daos_prop_t: properties

- Label
- Type (POSIX, HDF5, untyped)
- Redundancy Factor (RF)

- Using the API:

```
int daos_cont_create_with_label(daos_handle_t poh, const char *label,
        daos_prop_t *cont_prop, uuid_t *uuid, daos_event_t *ev);


int daos_cont_destroy(daos_handle_t poh, const char *cont, int force, /* ev
*/);
```

Program Flow – Access a Container

- Need to open a container to access objects in it:

```
int daos_cont_open(daos_handle_t poh, const char *cont, unsigned int flags,
daos_handle_t *coh, daos_cont_info_t *info, /* ev */);
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI:

```
int daos_cont_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_cont_global2local(daos_handle_t poh, d_iov_t *glob)
```

daos_cont_info_t:

- Pool UUID
- Container UUID
- # container open handles
- Metadata open/close/modify times
- RF
- …
- Also output by `daos_cont_query()`

- Close container when done:

```
int daos_cont_close(daos_handle_t coh, daos_event_t *ev);
```

Program Flow – Recap

```c
#include <daos.h>
int main(int argc, char **argv)
{
        daos_handle_t   poh, coh;

        daos_init();
        daos_pool_connect("mypool", NULL, DAOS_PC_RW, &poh, NULL, NULL);
        daos_cont_create_with_label(poh, "mycont", NULL, NULL, NULL);
        daos_cont_open(poh, "mycont", DAOS_COO_RW, &coh, NULL, NULL);

        /** perform object I/O - presented next */

        daos_cont_close(coh, NULL);
        daos_pool_disconnect(poh, NULL);
        daos_fini();
        return 0;
}
```

DAOS Object IDs – Types and Classes

- DAOS Object Types:
  - DAOS Flat KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated on 1 target, and atomic update
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - Multi-level keys (distribution / attribute)
    - Different value types (single value, array w/ fine-grain update)
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

DAOS Object IDs – Types and Classes

- DAOS Object Types:
  - DAOS Flat KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated on 1 target, and atomic update
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - Multi-level keys (distribution / attribute)
    - Different value types (single value, array w/ fine-grain update)
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

DAOS Object IDs – Types and Classes

- DAOS Object Types:
  - DAOS Flat KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated on 1 target, and atomic update
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - Multi-level keys (distribution / attribute)
    - Different value types (single value, array w/ fine-grain update)
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

DAOS Object IDs – Types and Classes

- Object ID 128-bit space (Lower 96 user; upper 32 daos):
  - Object type (e.g., KV, Array, Multi-Level KV)
  - Object class (Replication, EC, None)

```
int daos_obj_generate_oid(
        daos_handle_t coh,
        daos_obj_id_t *oid,
        enum daos_otype_t type,
        daos_oclass_id_t cid,
        daos_oclass_hints_t hints, uint32_t args);

daos_obj_id_t oid;
oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid,
DAOS_OF_KV_HASHED, OC_RP_2GX, 0, 0);
```

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

DAOS KV Object – Management Operations

- Recall: KV store interface providing access operations: Put, Get, Remove, List

- Management API:

```
int daos_kv_open(daos_handle_t coh, daos_obj_id_t oid, unsigned int mode,
daos_handle_t *oh, daos_event_t *ev);

int daos_kv_close(daos_handle_t oh, daos_event_t *ev);

int daos_kv_destroy(daos_handle_t oh, daos_handle_t th, /* ev */);
```

KV: string key → opaque/atomic value

API:

- input coh from daos_cont_open()
- Input oid from daos_obj_generate_oid()
- output object handle (oh)

DAOS KV Object – Access Operations

- Access API:

```
int daos_kv_put(daos_handle_t oh, daos_handle_t th, uint64_t flags, const
char *key, daos_size_t size, const void *buf, daos_event_t *ev);

int daos_kv_get(daos_handle_t oh, daos_handle_t th,
        uint64_t flags, const char *key,
        daos_size_t *size, void *buf, daos_event_t *ev);

int daos_kv_remove(daos_handle_t oh, daos_handle_t th,
        uint64_t flags, const char *key, daos_event_t *ev);

int daos_kv_list(daos_handle_t oh, daos_handle_t th, uint32_t *nr,
        daos_key_desc_t *kds, d_sg_list_t *sgl, daos_anchor_t *anchor,);
```

API: input oh from kv_open()

Put/get/remove values (given string key)

List keys
- Key sizes in daos_key_desc_t *kds
- Key strings in sgl

DAOS KV Object – KV Conditional Operations

- By default, KV put/get operations do not check “existence” of key before operations:
  - Put(key): overwrites the value
  - Get(key): does not fail if key does not exist, just returns 0 size.
  - Remove(key): does not fail if key does not exist.

- One can use conditional flags for different behavior:

- `DAOS_COND_KEY_INSERT`: Insert a key if it doesn't exist (fail if it already exists)
- `DAOS_COND_KEY_UPDATE`: Update a key if it exists, (fail if it does not exist)
- `DAOS_COND_KEY_GET`: Get key value if it exists, (fail if it does not exist).
- `DAOS_COND_KEY_REMOVE`: Remove a key if it exists (fail if it does not exist).

DAOS KV Object – Put/Get Example

```
/** daos_init, daos_pool_connect, daos_cont_open */

oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid, DAOS_OF_KV_HASHED, OC_RP_2GX, 0, 0);
daos_kv_open(coh, oid, DAOS_OO_RW, &kv, NULL);

/** set val buffer and size */
daos_kv_put(kv, DAOS_TX_NONE, 0, "key1", val_len1, val_buf1, NULL);
daos_kv_put(kv, DAOS_TX_NONE, 0, "key2", val_len2, val_buf2, NULL);

/** to fetch, can query the size first if not known */
daos_kv_get(kv, DAOS_TX_NONE, 0, "key1", &size, NULL, NULL);
get_buf = malloc (size);
daos_kv_get(kv, DAOS_TX_NONE, 0, "key1", &size, get_buf, NULL);
daos_kv_close(kv, NULL);
/** free buffer, daos_cont_close, daos_pool_disconnect, daos_fini */
```

DAOS KV Object – List Keys Example

```
/** enumerate keys in the KV */
daos_anchor_t   anchor = {0};
d_sg_list_t     sgl;
d_iov_t         sg_iov;

/** size of buffer to hold as many keys in memory */
buf = malloc(ENUM_DESC_BUF_BYTES);
d_iov_set(&sg_iov, buf, ENUM_DESC_BUF_BYTES);
sgl.sg_nr               = 1;
sgl.sg_nr_out           = 0;
sgl.sg_iovs             = &sg_iov;
```

```
daos_key_desc_t kds[ENUM_DESC_NR];

while (!daos_anchor_is_eof(&anchor)) {
  /** how many keys to attempt to fetch in one call */
  uint32_t        nr = ENUM_DESC_NR;


  memset(buf, 0, ENUM_DESC_BUF_BYTES);
  daos_kv_list(kv, DAOS_TX_NONE, &nr, kds, &sgl,
               &anchor, NULL);

  if (nr == 0)
    continue;
  /** buf now contains nr keys */
  /** kds[] has nr key descriptors (length keys) */
}
```

DAOS Array Object – Management Operations

- 1-Dimensional Array object to manage records
  - cell_size: single array value size (bytes)
  - chunk_size: number of cells placed together in a storage target –controls striping of array regions across storage cluster

- Management API:

```
int daos_array_create(daos_handle_t coh, daos_obj_id_t oid, daos_handle_t th,
        daos_size_t cell_size, daos_size_t chunk_size, daos_handle_t *oh, /* ev
*/);
int daos_array_open(daos_handle_t coh, daos_obj_id_t oid, daos_handle_t th,
        unsigned int mode, daos_size_t *cell_size,
        daos_size_t *chunk_size,daos_handle_t *oh, daos_event_t *ev);

int daos_array_close(daos_handle_t oh, daos_event_t *ev);

int daos_array_destroy(daos_handle_t oh, daos_handle_t th, daos_event_t *ev);
```

DAOS Array Object – Access Operations

- Reading & writing record to an Array:

```
int daos_array_read(daos_handle_t oh, daos_handle_t th, daos_array_iod_t
*iod,
      d_sg_list_t *sgl, daos_event_t *ev);
int daos_array_write(daos_handle_t oh, daos_handle_t th, daos_array_iod_t
*iod,
      d_sg_list_t *sgl, daos_event_t *ev);
```

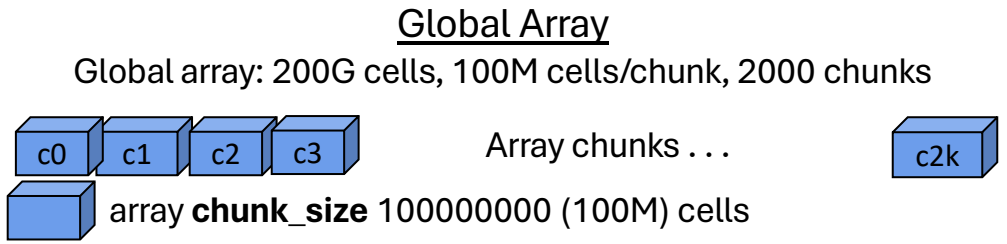
- Misc

```
int daos_array_get_size(daos_handle_t oh, daos_handle_t th, daos_size_t
*size, …);
int daos_array_set_size(daos_handle_t oh, daos_handle_t th, daos_size_t
size, …);
int daos_array_get_attr(daos_handle_t oh, daos_size_t *chunk_size,
                        daos_size_t *cell_size);
```

DAOS Array Object – Example

```
/** create array – 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);
```

`



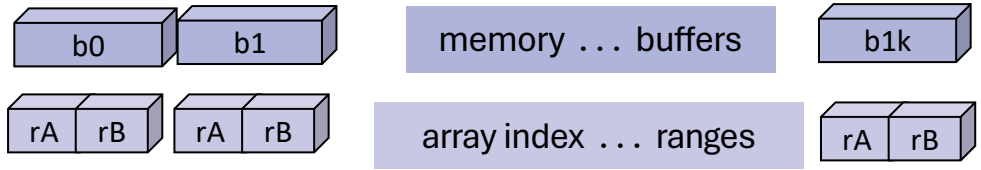
DAOS Array Object – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;              /** two array (sub)ranges */
iod.arr_rgs   = rgs;            /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;       /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;         /** length (and rgs[1] len=NCELLS) */
```

`

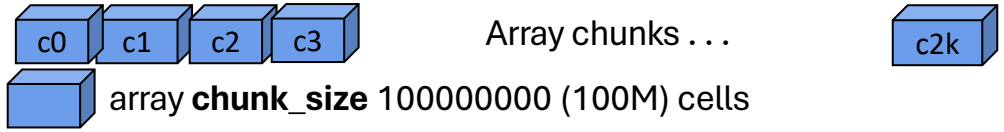
Scaled Application

1000 clients (ranks) each produce 200M cells of data



Global Array

Global array: 200G cells, 100M cells/chunk, 2000 chunks

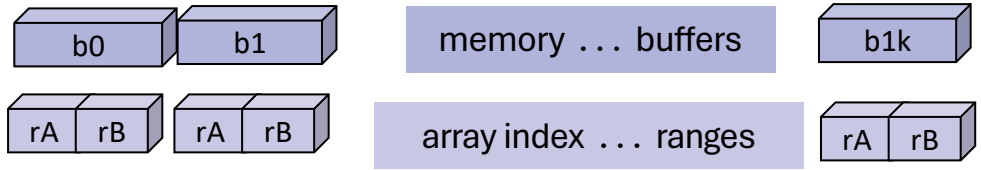


DAOS Array Object – Example

```
/** create array - 1 byte/cell, NCELLS=100,000,000 cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, NCELLS, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;              /** two array (sub)ranges */
iod.arr_rgs   = rgs;            /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;       /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;         /** length (and rgs[1] len=NCELLS) */

/** write array data to DAOS storage, and read back */
daos_array_write(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_read(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_close(array, NULL);
```

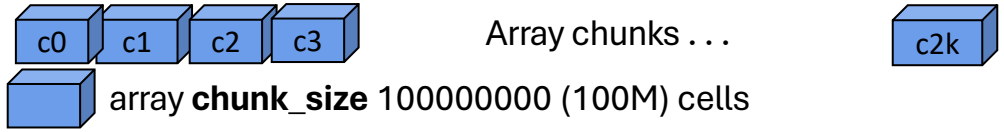
Scaled Application

1000 clients (ranks) each produce 200M cells of data



Global Array

Global array: 200G cells, 100M cells/chunk, 2000 chunks

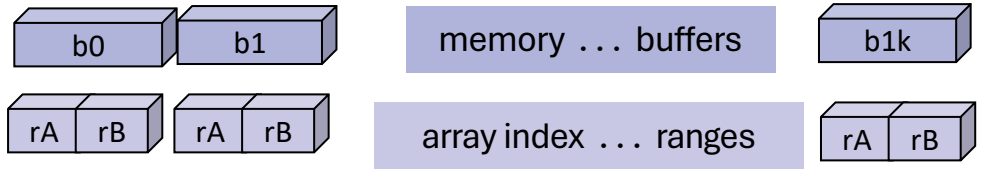


DAOS Array Object – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;                /** two array (sub)ranges */
iod.arr_rgs   = rgs;              /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;         /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;           /** length (and rgs[1] len=NCELLS) */

/** write array data to DAOS storage, and read back */
daos_array_write(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_read(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_close(array, NULL);
```

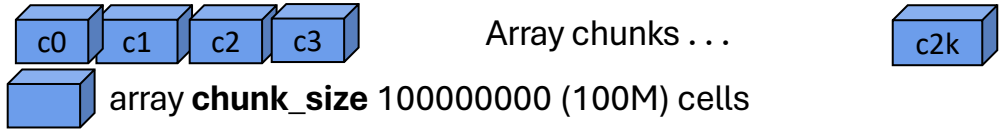
Scaled Application

1000 clients (ranks) each produce 200M cells of data



Global Array

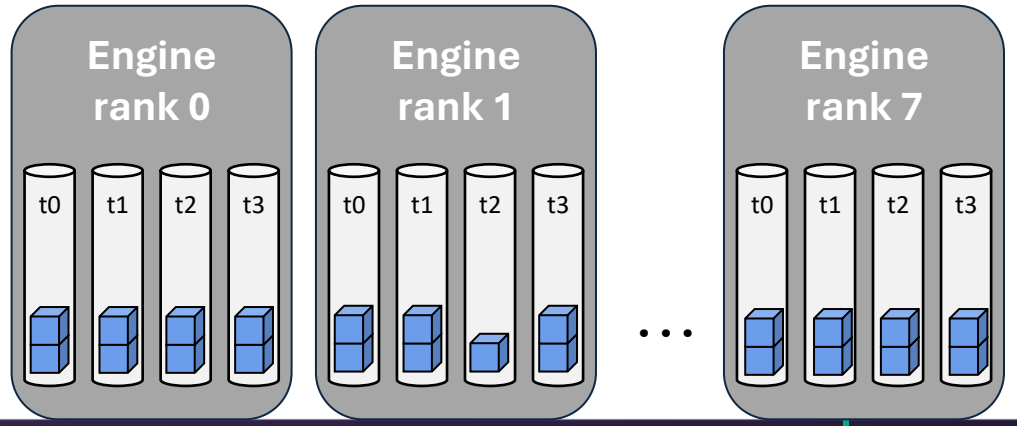
Global array: 200G cells, 100M cells/chunk, 2000 chunks



DAOS Storage

Stored in <num_daos_engines_in_pool> x tgts/daos_server

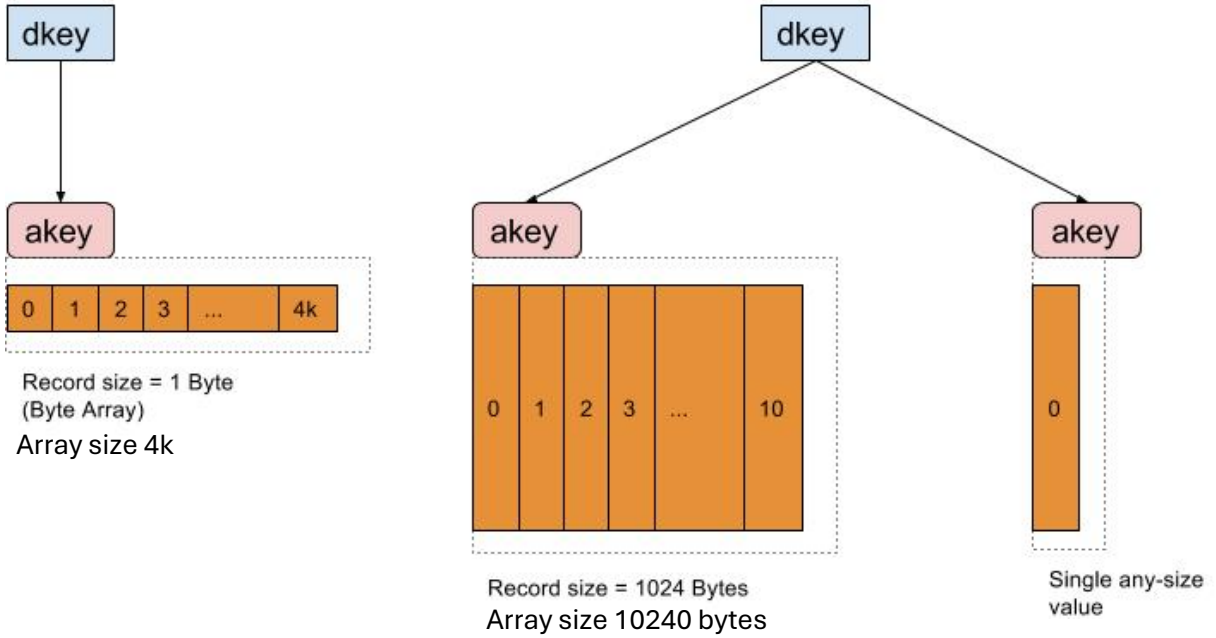
Ex: 8 servers x 4 tgts/server = 32 targets



Multi-Level KV Object

- Two-level key:
  - Distribution Key - Dkey (collocate all entries under it), holds multiple akeys
  - Attribute Key - Akey (lower level to address records)
  - Both are opaque (support any size / type)

- Value types (under a single akey):
  - Single value: one blob (traditional value in KV store)
  - Array value:
    - Array of fixed-size cells (records) that can be updated in a fine-grained manner via different range extents
    - This is different than a DAOS (global/distributed) array



- **Intentionally very flexible, rich API**
- **(at the expense of higher complexity for the typical user)**

Multi-Level KV Object – Management Operations

```
int daos_obj_open(daos_handle_t coh, daos_obj_id_t oid, unsigned int mode,
daos_handle_t *oh, daos_event_t *ev);

int daos_obj_close(daos_handle_t oh, daos_event_t *ev);

int daos_obj_punch(daos_handle_t oh, daos_handle_t th, uint64_t flags, /*
ev */);

int daos_obj_punch_dkeys(daos_handle_t oh, daos_handle_t th, uint64_t
flags, unsigned int nr, daos_key_t *dkeys, daos_event_t *ev);

int daos_obj_punch_akeys(daos_handle_t oh, daos_handle_t th, uint64_t
flags, daos_key_t *dkey, unsigned int nr, daos_key_t *akeys, …);
```

API:

- input coh from daos_cont_open()
- Input oid from daos_obj_generate_oid()
- output object handle (oh)

Multi-Level KV Object – Access Operations (Update, Fetch)

```
int daos_obj_update(daos_handle_t oh, daos_handle_t th,
        uint64_t flags, daos_key_t *dkey, unsigned int nr,
        daos_iod_t *iods, d_sg_list_t *sgls, daos_event_t *ev);
```

```
daos_key_t iod_name;       /* akey */
daos_iod_type_t iod_type; /* value type (single value or array value) */
daos_size_t iod_size;      /* SV: value size, array: record size */
uint32_t iod_nr;           /* SV: 1, array: number of record extents */
daos_recx_t *iod_recxs;    /* SV: NULL, array: (offset, length) pairs */
        uint64_t rx_idx, rx_nr;
```

```
uint32_t sg_nr;
uint32_t sg_nr_out;
d_iov_t *sg_iovs;
```

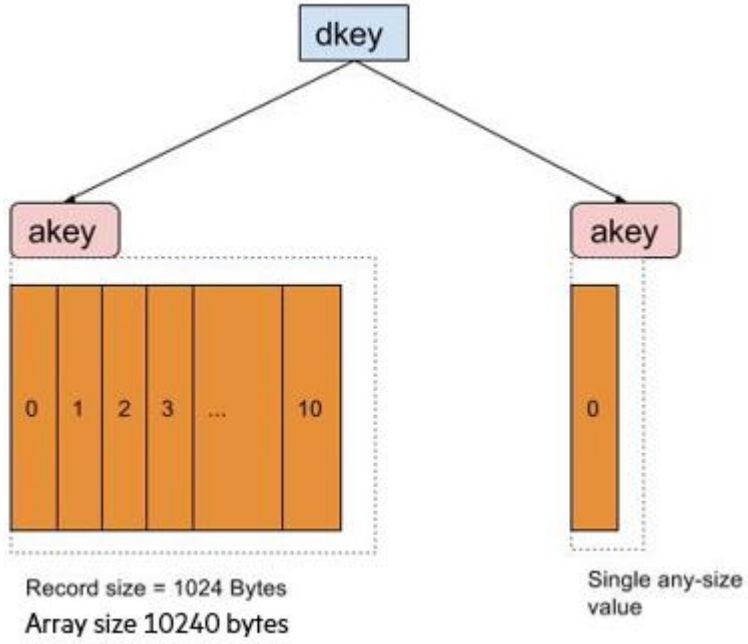
```
int daos_obj_fetch(daos_handle_t oh, daos_handle_t th, uint64_t flags,
        daos_key_t *dkey, unsigned int nr, daos_iod_t *iods,
        d_sg_list_t *sgls, daos_iom_t *ioms, daos_event_t *ev);
```

Multi-Level KV Object – Access Operations (List)

```
int daos_obj_list_dkey(daos_handle_t oh, daos_handle_t th, uint32_t
*nr,
      daos_key_desc_t *kds, d_sg_list_t *sgl, daos_anchor_t
*anchor,…);

int daos_obj_list_akey(daos_handle_t oh, daos_handle_t th,
      daos_key_t *dkey, uint32_t *nr, daos_key_desc_t *kds,
      d_sg_list_t *sgl, daos_anchor_t *anchor, daos_event_t *ev);
```

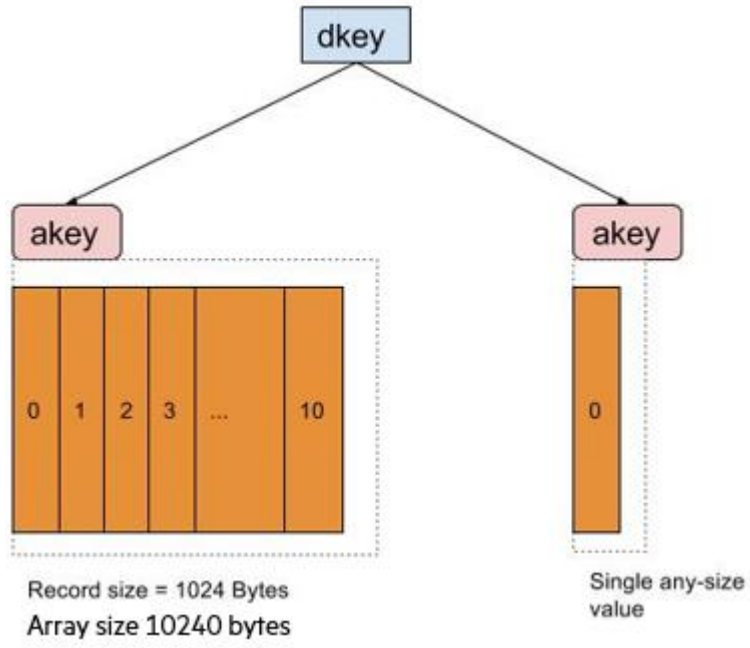
Multi-Level KV Object – Update Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);

/* application buffers */
const char *buf2 = "single_value, my string";
d_iov_set(&sg_iovs[0], buf1, BUF1LEN);      /* 10240 byte array val: (dkey1,akey1) */
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iovs[0];
d_iov_set(&sg_iovs[1], buf2, strlen(buf2)); /* string val: (dkey1,akey2) */
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iovs[1];

/* keys */
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&iods[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iods[1].iod_name, "akey2", strlen("akey2"));

/* IODs for each akey */
iods[0].iod_type = DAOS_IOD_ARRAY;
iods[0].iod_size = 1;                   /* 1 byte/array cell */
recx.rx_idx = 0;                        /* array index range (0, BUF1LEN) */
recx.rx_nr = BUF1LEN;
iods[0].iod_nr = 1;
iods[0].iod_recxs = &recx;
iods[1].iod_type = DAOS_IOD_SINGLE;
iods[1].iod_size = strlen(buf2);
iods[1].iod_nr = 1;                     /* iod_recxs=NULL for SV */

daos_obj_update(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], NULL);
```



Multi-Level KV Object – Fetch Example

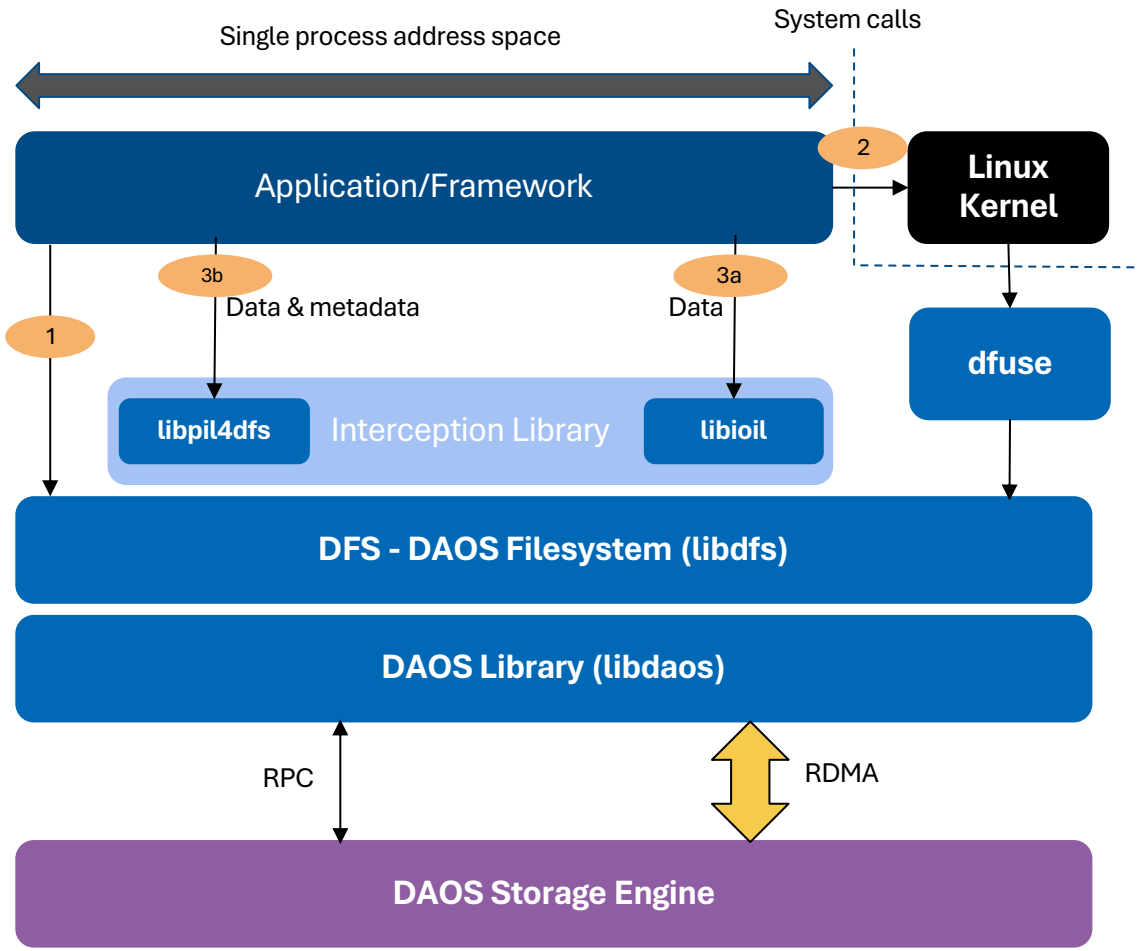
```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);

/* application buffers */
char rbuf2[128];
d_iov_set(&sg_iovs[0], rbuf1, BUF1LEN);     /* 10240 byte array val: (dkey1,akey1) */
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iovs[0];
d_iov_set(&sg_iovs[1], rbuf2, strlen(buf2)); /* string val: (dkey1,akey2) */
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iovs[1];

/* keys */
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&iods[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iods[1].iod_name, "akey2", strlen("akey2"));

/* IODs for each akey */
/** If iod_size is unknown: specify DAOS_REC_ANY, NULL sgl */
iods[0].iod_type = DAOS_IOD_ARRAY;
iods[0].iod_size = 1;                /* 1 byte/array cell */
recx.rx_idx = 0;                     /* array index range (0, BUF1LEN) */
recx.rx_nr = BUF1LEN;
iods[0].iod_nr = 1;
iods[0].iod_recxs = &recx;
iods[1].iod_type = DAOS_IOD_SINGLE;
iods[1].iod_size = strlen(buf2);
iods[1].iod_nr = 1;                  /* iod_recxs=NULL for SV */

daos_obj_fetch(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], NULL, NULL);
```



More Examples

- https://github.com/daos-stack/daos/blob/master/src/tests/simple_obj.c
- https://github.com/daos-stack/daos/blob/master/src/tests/simple_dfs.c

POSIX Support & Interception



1

2

3

3a

3b

Userspace DFS library with API like POSIX

- **Requires** application changes
- Low-latency and high-concurrency
- No caching

DFUSE daemon to support POSIX API

- **No changes** to application
- VFS mount point and high-latency
- Caching done by Linux kernel

DFUSE + Interception Library

- **No changes** to application
- 2 flavors, choose with LD_PRELOAD
- Libioil
  - Intercept data only - (f)read/write
  - Metadata handled by dfuse / kernel
- Libpil4dfs
  - Intercept data and metadata
  - Aim to deliver same performance as DFS (#1)
  - mmap() and binary execution via fuse

POSIX – How to Use DFS API (From a Modified Application)

- You should have access to a pool (identified by a string label).

- Create a POSIX container with the daos tool:
  - `daos cont create mypool mycont --type=POSIX`
  - Or: use API to create a container to use in your application (if using DFS and changing your app).

- Open the DFS mount:
  - `dfs_connect (mypool, mycont, O_RDWR, .. &dfs);`
  - `dfs_disconnect (dfs);`

POSIX – DFS API

POSIX	DFS
mkdir(), rmdir()	dfs_mkdir(), dfs_rmdir()
open(), close(), access()	dfs_open(), dfs_release(),dfs_lookup()
pwritev(), preadv()	dfs_read/write()
{set,get,list,remove}xattr()	dfs_{set,get,list,remove}xattr
stat(), fstat()	dfs_stat(),ostat()
readdir() …	dfs_readdir() …

- Mostly 1-1 mapping from POSIX API to DFS API.
- Instead of File & Directory descriptors, use DFS objects.
- All calls need the DFS mount which is usually done once (initialization time).

POSIX – DFUSE (With Unmodified Applications)

- To mount an existing POSIX container with dfuse, run the following command:
  - `dfuse /mnt/dfuse mypool mycont`
  - No one can access your container / mountpoint unless access is provided on the pool and container (via ACLs)

- Now you have a parallel file system under /mnt/dfuse on all nodes where that is mounted
  - “Easy path” for unmodified apps and daos – access files / directories as a namespace in the container

- dfuse + Interception Libraries:
  - Approach: intercept POSIX I/O calls, issue I/O directly from application through libdaos (kernel bypass)
  - To use: set LD_PRELOAD to point to the shared library in the DAOS install dir
    - (newer approach – metadata+data intercept) LD_PRELOAD=/path/to/daos/install/lib64/libpil4dfs.so
    - (original approach – read/write only intercept) LD_PRELOAD=/path/to/daos/install/lib64/libioil.so

POSIX – Best Practices: Redundancy Factor (rd_fac) Container Property

- The number of (not yet rebuilt) concurrent failures container objects are protected against (without loss)
  - A number in the range 0-5

- Production systems recommendation: rd_fac:3
  - `daos cont create –type=POSIX –properties=rd_fac:3 <pool> <container>`

- Note: all objects must use a class with at least this degree of protection. Some legal examples:

	rd_fac:0	rd_fac:1	rd_fac:2	rd_fac:3
No Protection Classes	OC_S<*>	None	None	None
Replication Classes	Any	OC_RP_2G<*> OCP_RP_3G<*> …	OC_RP_3G<*> OC_RP_4G<*> …	OC_RP_4G<*> OC_RP_5G<*> …
Erasure Code Classes	Any	OC_EC_8P1G<*> OC_EC_16P1G<*> >	OC_EC_8P2G<*> OC_EC_16P2G<*> >	OC_EC_8P3G<*> OC_EC_16P3G<*> >

POSIX – Best Practices: Object Class Data Protection

- Recall: data protection is part of an object's "object class" – None, Replication, or Erasure Code

- Erasure Code:
  - Best for large IO access patterns.
  - Full stripe write: 12%-33% lower performance (vs. no data protection).
  - Partial stripe write: 66% lower performance (vs. no data protection).
  - Read performance should be the same.
  - Not supported for directory objects

- Replication:
  - Best for metadata objects (directories) and small files (<= 16k).
  - Write IOPS: slower (than no data protection) by the number of replicas created.
  - Read IOPS: equal or better (than no data protection) – more shards to serve concurrent requests.

POSIX – Best Practices: Object Class Striping (Wide or Narrow)

```
daos cont create –type=POSIX –dir-oclass=<OC> --file-oclass=<OC>
```

	rd_fac:0	rd_fac:1	rd_fac:2
Defaults - Widely-striped (“X”) objs for: - Large files (GBs), Lean dirs. (<10k ent) - Single-shared access, high BW required	File: SX Dir : S1	File: EC_16P1GX Dir : RP_2G1	File: EC_16P2GX Dir : RP_3G1
Small-stripe (1/2/4/16/32) objs for: - Something in-between huge and tiny files - File per process to large files.	File: S32 (S1/2/.../32) Dir : S1	File: EC_16P1_G32 (G1/2/.../32) Dir : RP_2G1	File: EC_16P2_G32 (G1/2/.../32) Dir : RP_3G1
One-stripe objs for: - tiny files, more IOPS required	File/Dir: S1	File/Dir: RP_2G1	File/Dir: RP_3G1

Recall: Sample Object Classes (daos_oclass_id_t)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

POSIX – Best Practices: Object Class Striping – Tradeoffs

```
daos cont create –type=POSIX –dir-oclass=<OC> --file-oclass=<OC>
```

	rd_fac:0	rd_fac:1	rd_fac:2
Defaults - Widely-striped (“X”) objs for: - Large files (GBs), Lean dirs. (<10k ent) - Single-shared access, high BW required Tradeoffs: - Slow file stat(), remove, directory listing – RPC to all engines, query all targets.	File: SX Dir : S1	File: EC_16P1GX Dir : RP_2G1	File: EC_16P2GX Dir : RP_3G1
Small-stripe (1/2/4/16/32) objs for: - Something in-between huge / tiny files - File per process to large files. Tradeoffs: - Faster stat() and directory listing - Limited bandwidth to number of targets (if using single shared file) - Benchmarking file create/remove/stat could benefit from widely-striped dirs.	File: S32 (S1/2/.../32) Dir : S1	File: EC_16P1_G32 (G1/2/.../32) Dir : RP_2G1	File: EC_16P2_G32 (G1/2/.../32) Dir : RP_3G1
One-stripe objs for: - tiny files, more IOPS required	File/Dir: S1	File/Dir: RP_2G1	File/Dir: RP_3G1

Recall: Sample Object Classes (daos_oclass_id_t)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

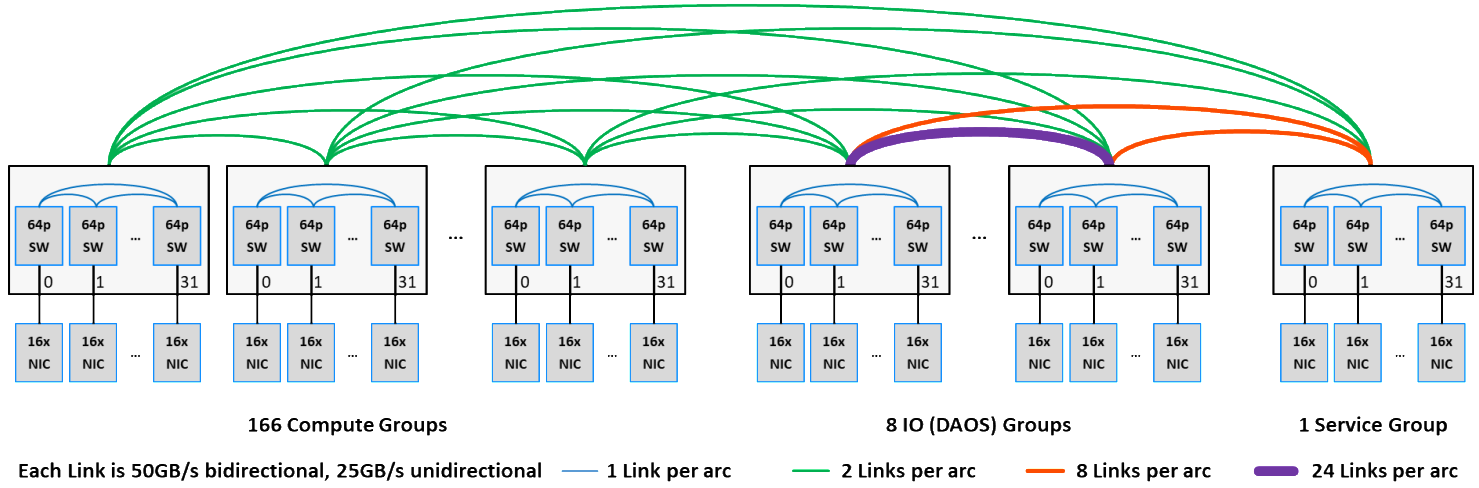
POSIX – Best Practices: – EC Properties for Performance

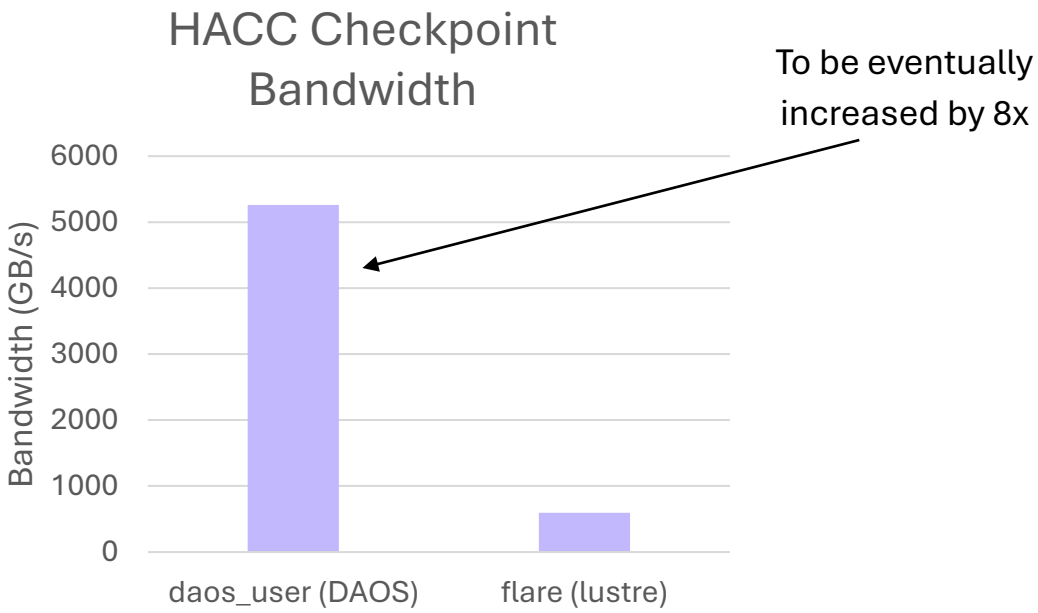
- `ec_cell_sz` container property
  - DAOS splits application buffer into `ec_cell_sz` byte parts (16 parts for EC_16P2, 8 parts for EC_8P2, etc.)
  - Default is 64k – not ideal. 128k is better for generic use (will be updated in future releases).
  - Can increase the ec_cell_sz if you think your IO is large most of the time (data shards x ec_cell_size)
- DFS Chunk Size (`daos container create –chunk_size=`)
  - Set automatically in newer releases. But before had to set it to ec_cell_size x EC data shards
- Full (vs. Partial) Stripe Write – application buffer chunk is an even multiple of `ec_cell_sz` (or not).
  - Full stripe write is more efficient

EC Object Class	EC Cell Size	DFS Chunk size	Full or Partial Write?
16P2	128k	1m	Partial: 1m gets divided into 8 128k data parts which < 16 data shards of the object class used
8P2	128k	1m	Full
16P2	128k	2m, 4m, 8m, etc	Full
16P2	256k	2m	Partial
16P2	256k	4m, 8m, etc.	Full

Aurora DAOS System

- 1k storage nodes
  - Namely “aurora-daos-xxxx”
  - Dual NICs/Sockets running 2 DAOS engines
  - Raw capacity of 244TB per node
  - Raw read/write@~40GB/s per node
  - 1k storage nodes spread over 64 racks
  - >220PB and >31TB/s when at full capacity (vs 600GB/s for flare/Lustre)
- Directly connected to the Slingshot fabric
  - 8 DragonFly I/O groups (8 racks per group)
  - Adaptive routing enabled
  - More links between I/O groups for failure handling/rebuild





DAOS Community Release (September'25)

Color coding schema:
- Committed (or released) release/features
- In-planning release/features
- Future possible release/features

DAOS 2.6
(Intel Release)

OS Packages:
- Leap 15.5
- RHEL/Rocky/Alma 8.8/9.2

Networking:
- Change provider w/o reformat
- MD duplicate RPC detection

Features:
- Non-PMem support phase 1
- libpil4dfs
- Intel VMD hotplug
- Delayed rebuild

Tech preview:
- Distributed consistency checker (CR)

UX Improvements:
- Improved version interoperability

DAOS 2.8
(DAOS Foundation Release)

OS Packages:
- Leap 15.6/7
- RHEL/Rocky/Alma 8.10/9.4
- New packaging

Networking:
- DOCA-OFED support

Features:
- Catastrophic recovery tool improvement
- Better defaults (e.g. RF3, cell size, …)
- Rebuild bug fixing and perf improvement
- Explicit rebuild control
- Improved allocator for non-PMem phase 1
- PyTorch data loader

Tech preview:
- Non-PMem support phase 2
- Rolling upgrade preparation

UX Improvements:
- Reintegration of all pools
- daos pool listing

DAOS 3.0
(DAOS Foundation Release)

OS Packages:
- Leap 15.7 (x86_64)
- RHEL/Rocky/Alma 8.10/9.x (x86_64)
- RHEL/Rocky/Alma 9.x client (ARM64)
- Ubuntu 22.04 client (x86_64/ARM64)

Features:
- Non-PMem support phase 2
- SSD hotplug & LED without VMD
- Mount POSIX snapshots RO
- Incremental rebuild/reintegration

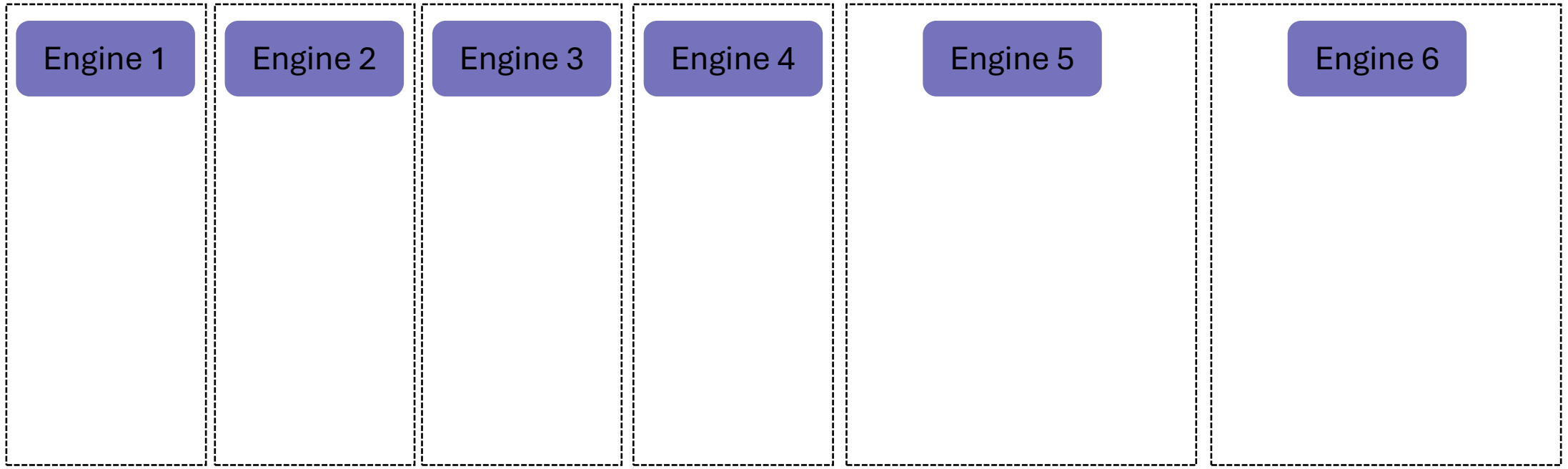
Tech preview:
- Rolling upgrade
- Multi-provider support
- SSD encryption support via SED
- Fault domains beyond servers

Future Releases
(DAOS Foundation Release)

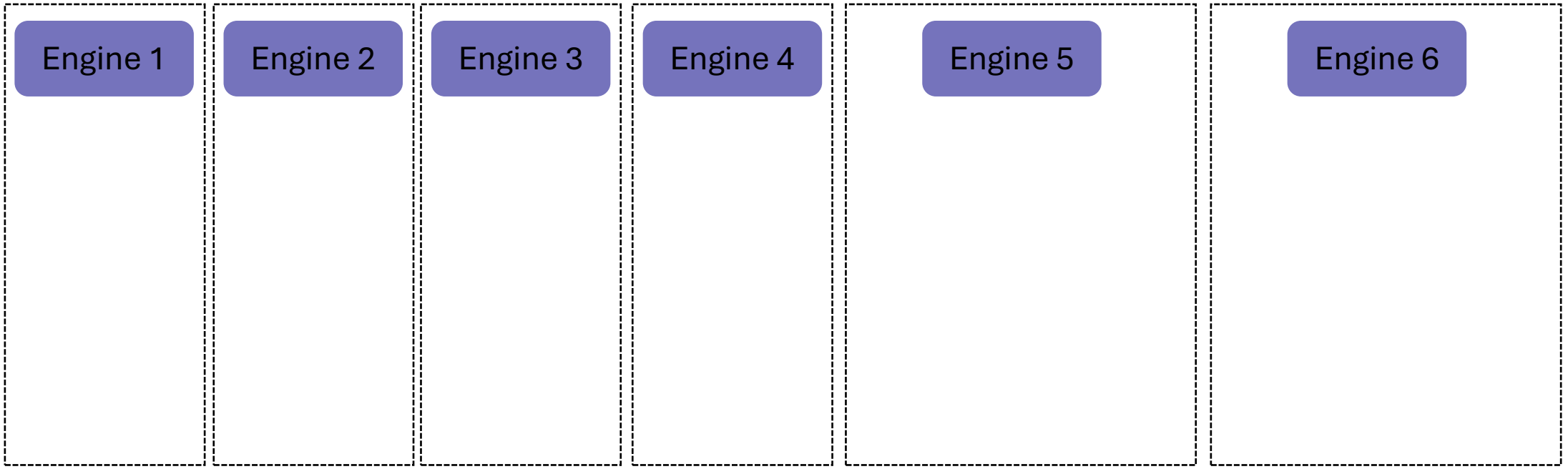
OS Packages:
- Leap 15.7 (x86_64)
- RHEL/Rocky/Alma 8.10/9.x (x86_64)
- RHEL/Rocky/Alma 9.x (ARM64/x64_64)
- Ubuntu 24.04 client (x86_64/ARM64)

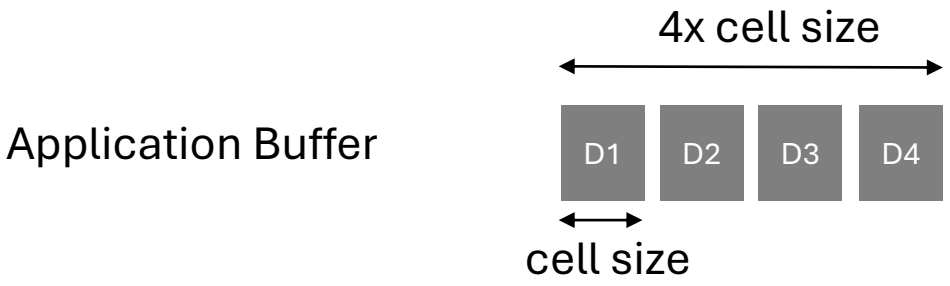
Features:
- Pool resizing
- Inline compression
- Inline encryption
- Inline deduplication
- Middleware consistency checker
- Progressive layout
- Pipeline API
- SQL support with predicate pushdown
- Distributed transactions
- Pool/container freeze
- CXL SSD support / QLC
- Tiered container phase 1
- Support for multiple DAOS systems
- multi-NIC support per engine/process
- hardlinks support in libdfs
- network multipath support
- Container parking/serialization
- WORM containers
- flock support

Erasure Code Full Stripe I/O (4+2)

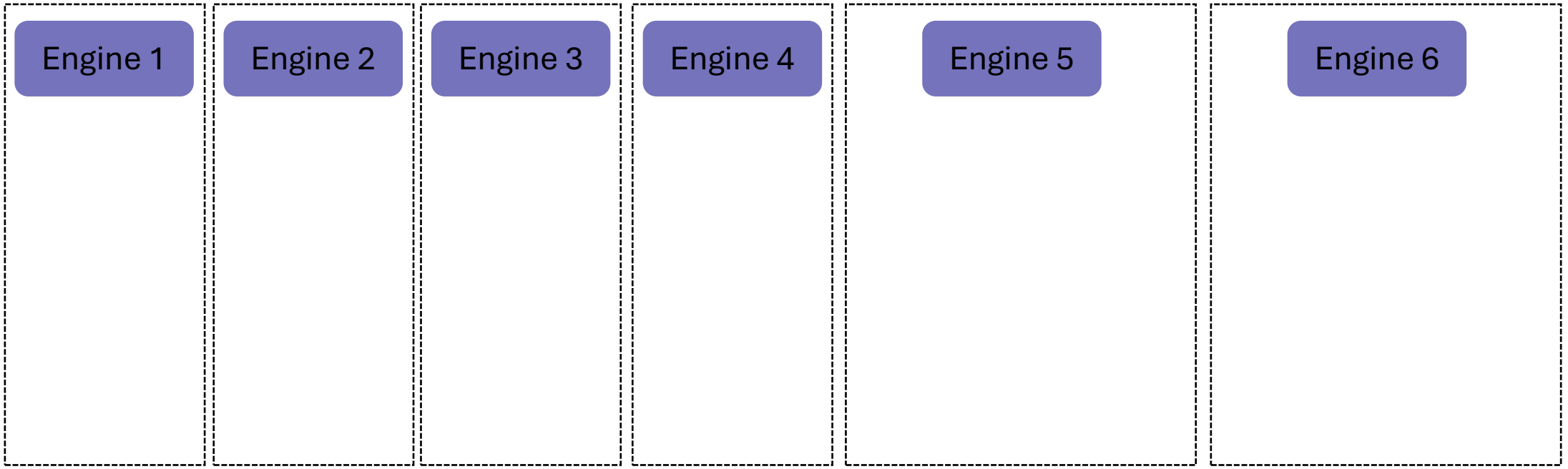


Erasure Code Full Stripe I/O (4+2)

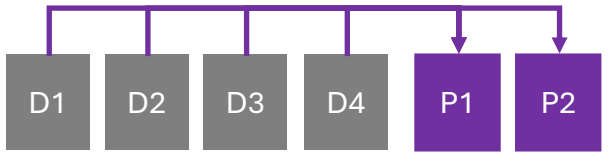




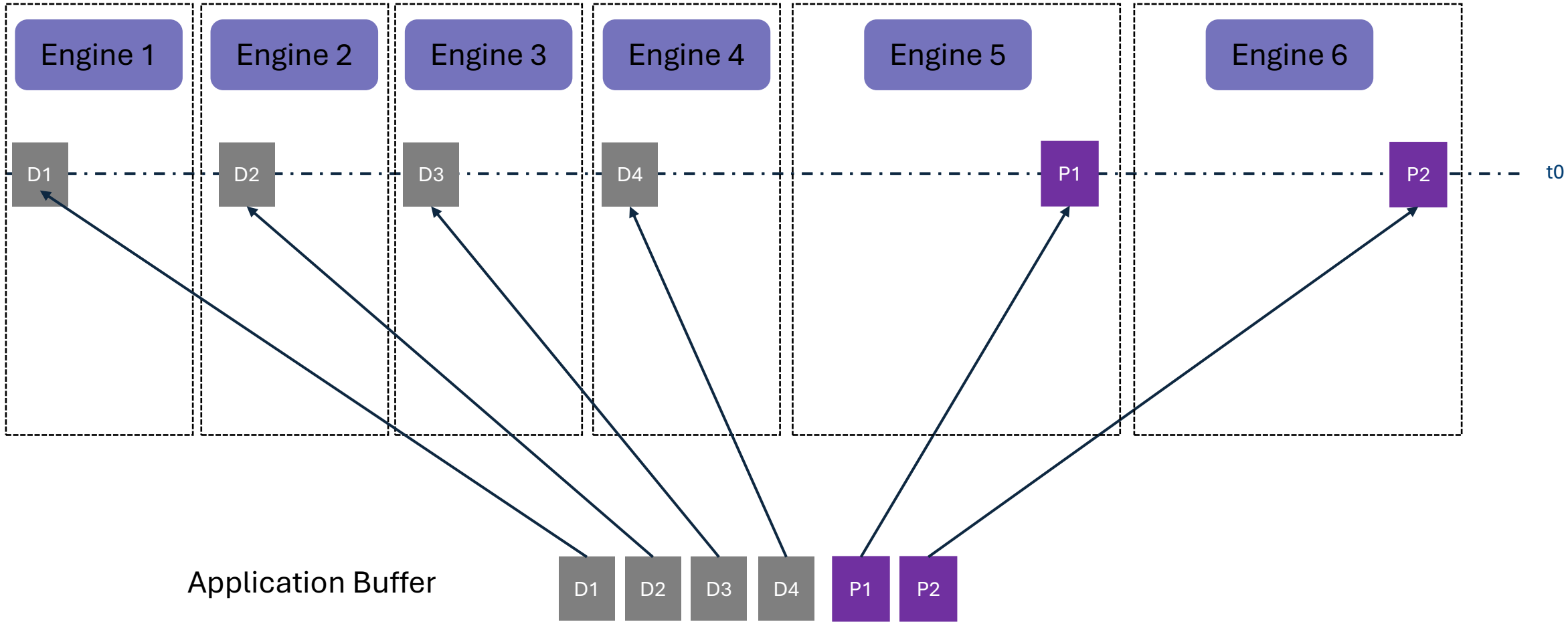
Erasure Code Full Stripe I/O (4+2)



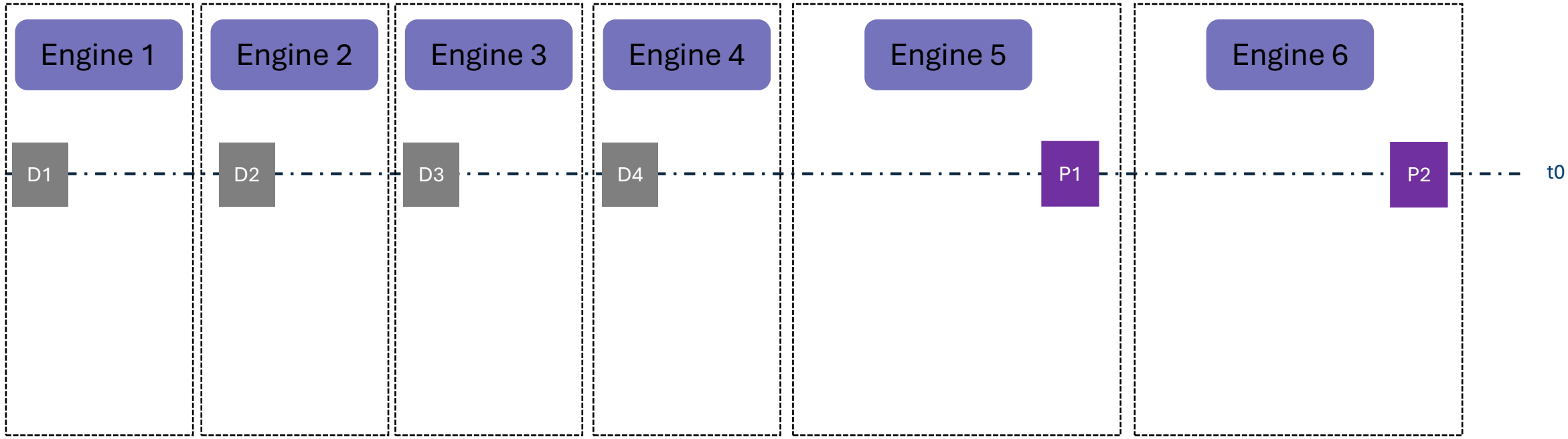
Application Buffer



Erasure Code Full Stripe I/O (4+2)

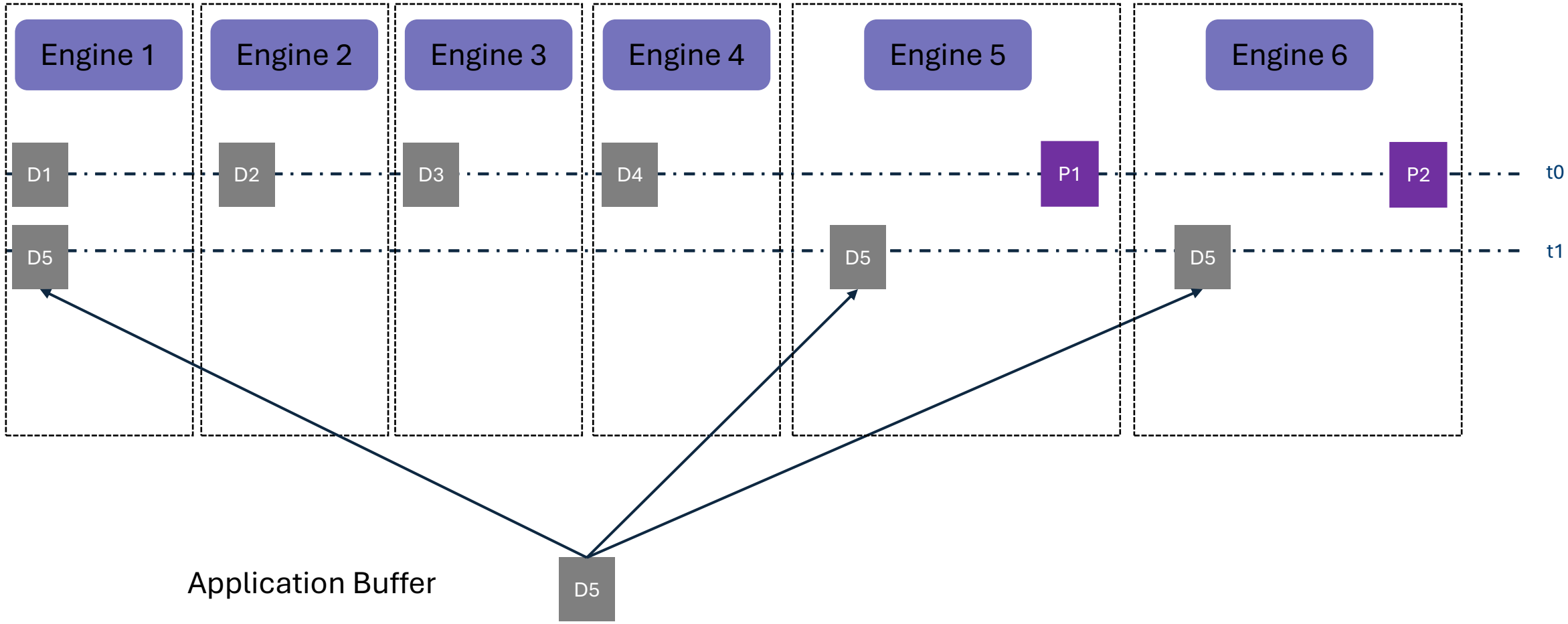


Erasure Code Partial I/O (4+2)

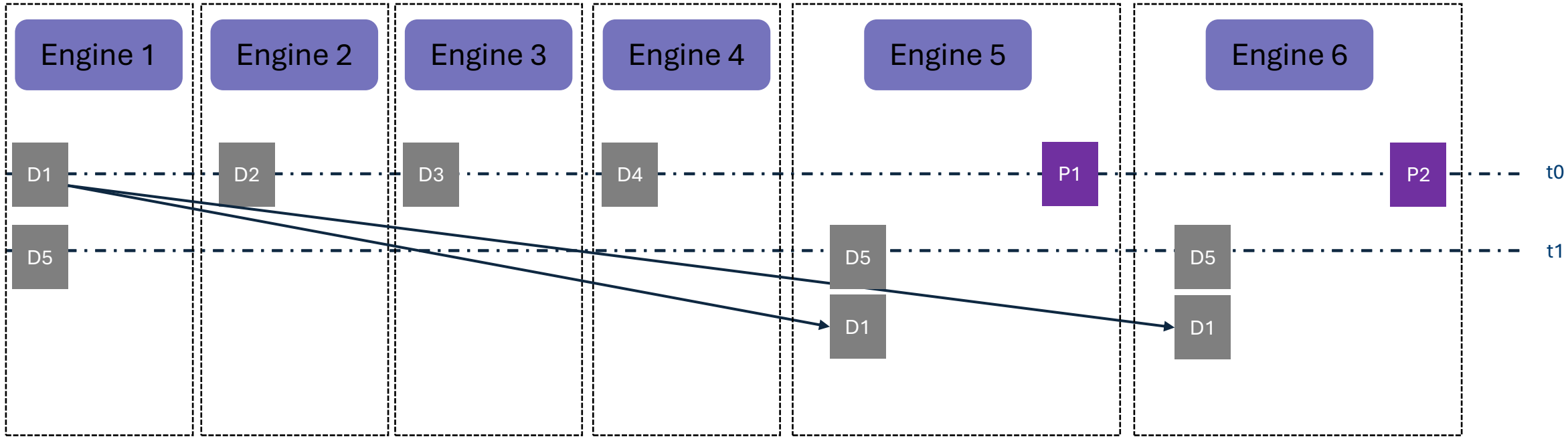


Application Buffer

Erasure Code Partial I/O (4+2)

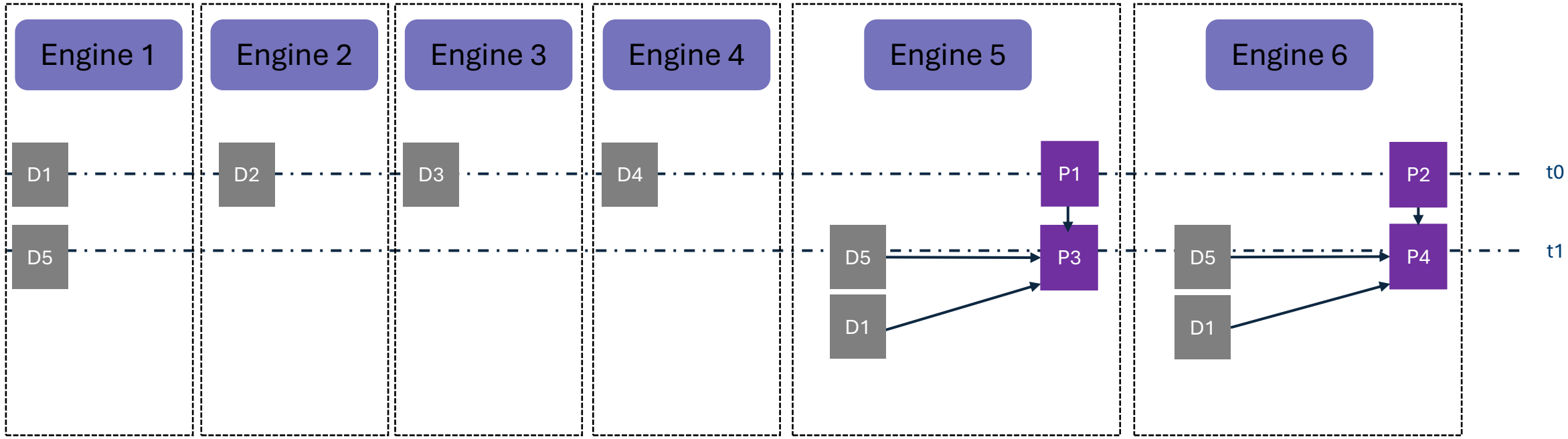


Erasure Code Aggregation (4+2)



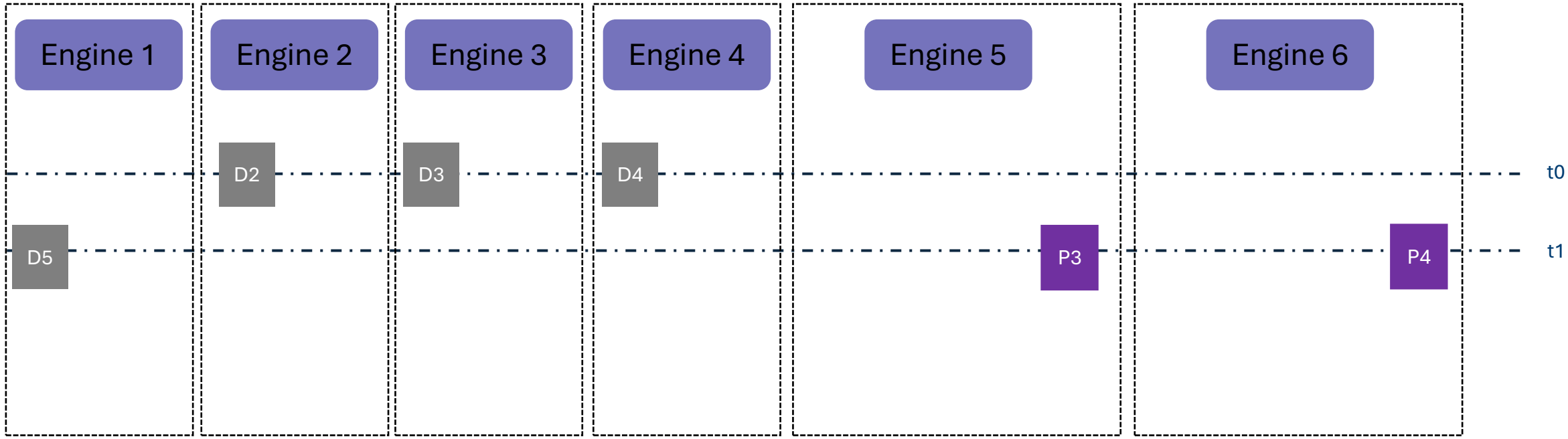
Application Buffer D5

Erasure Code Aggregation (4+2)



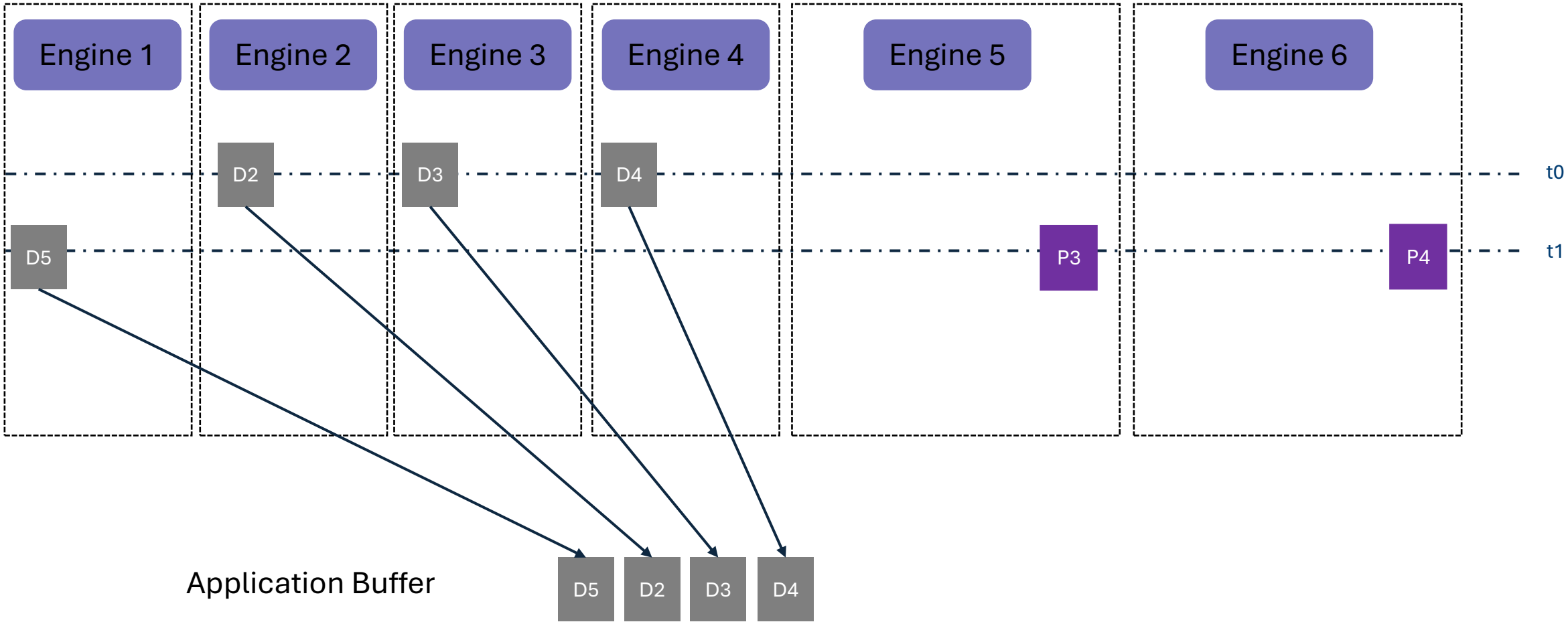
Application Buffer

Erasure Code Aggregation (4+2)



Application Buffer D5

Erasure Code Read (4+2)



DAOS Service Structure

daos_server process (Go)
Managed by systemd

Control plane

DAOS Server

gRPC

yaml

Network

start/stop
monitor

daos_engine process (C)
Managed by daos_server

Data plane

DAOS Engine

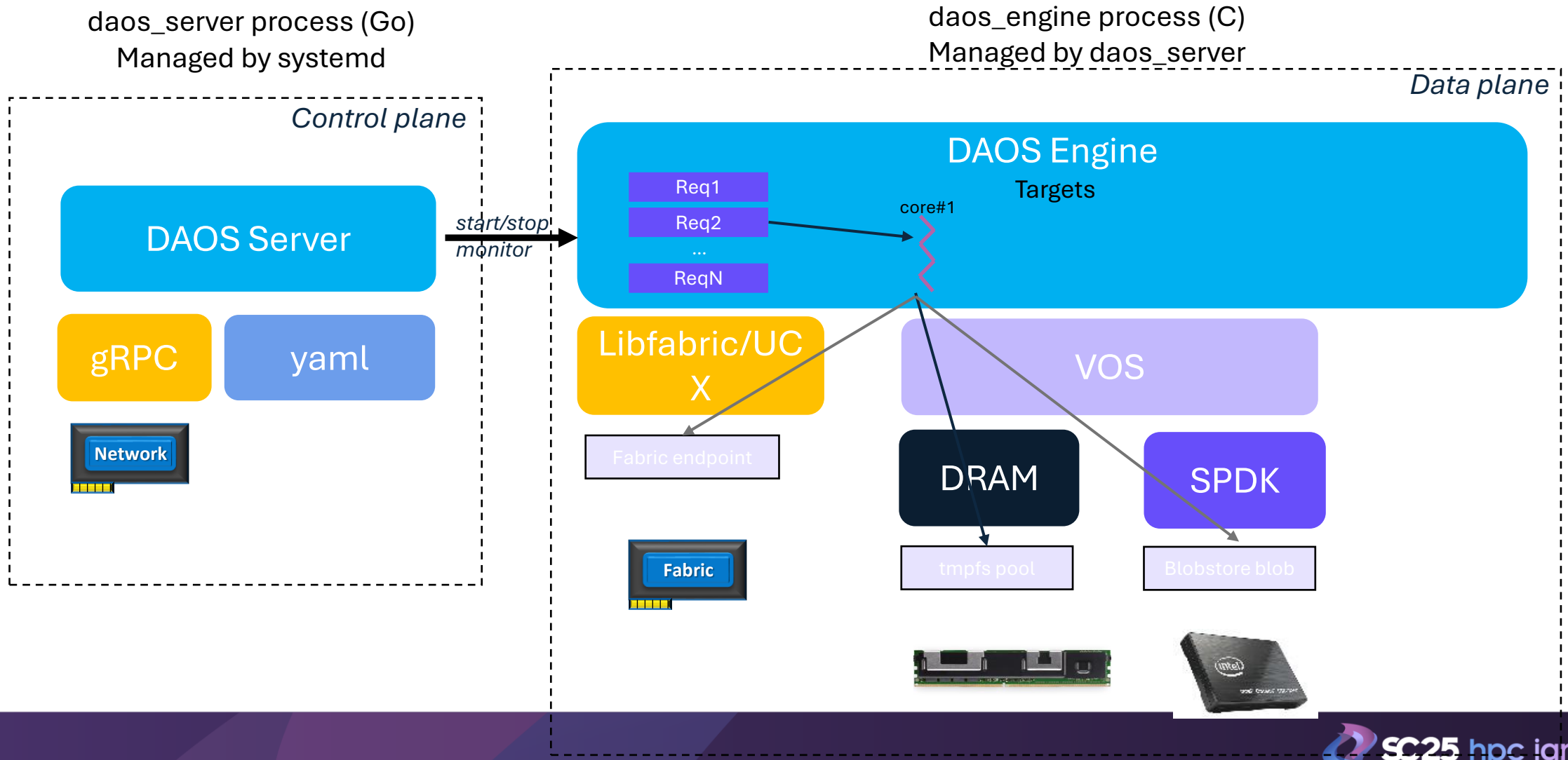
Libfabric/UCX

DRAM

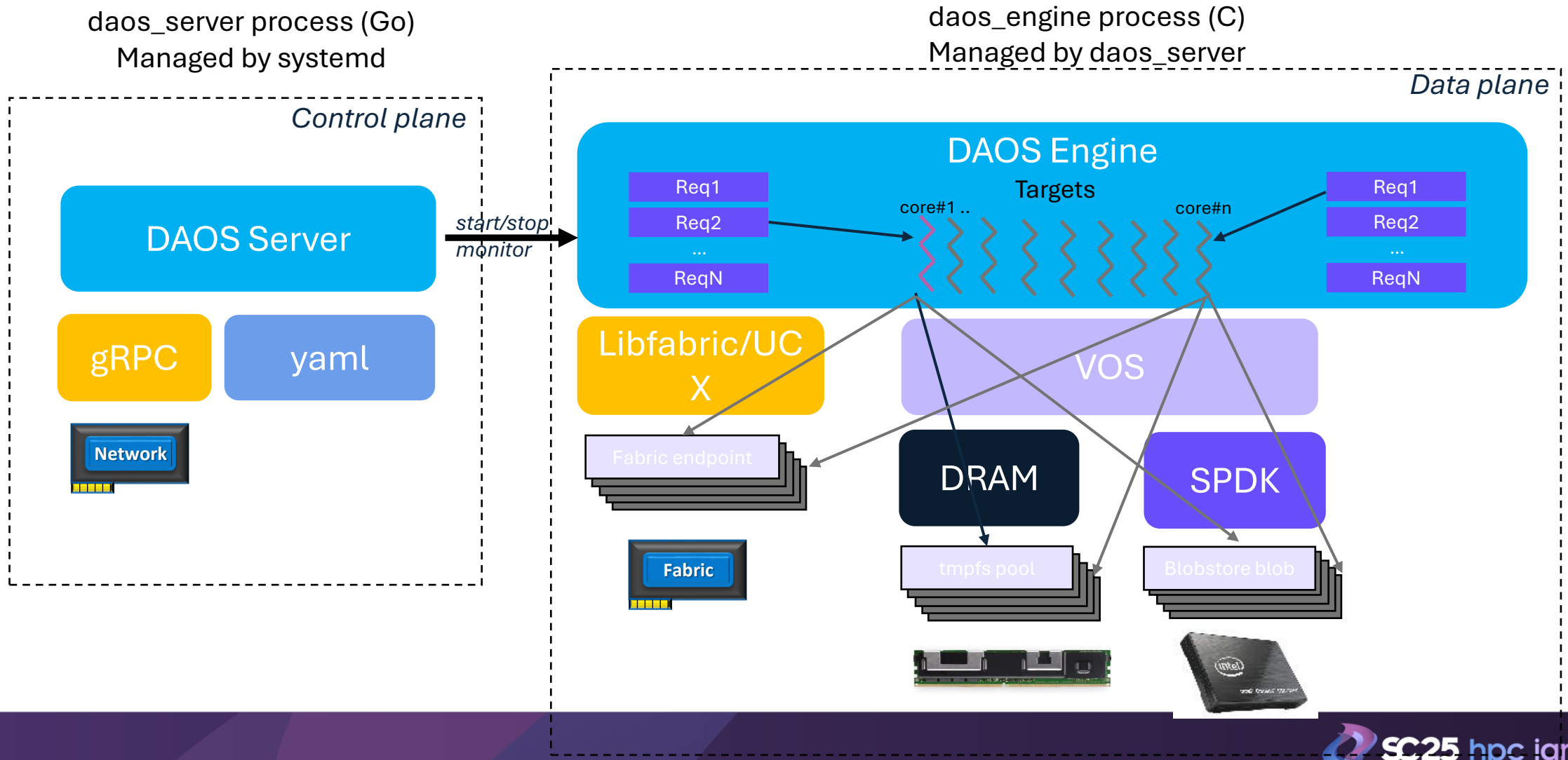
SPDK

Fabric

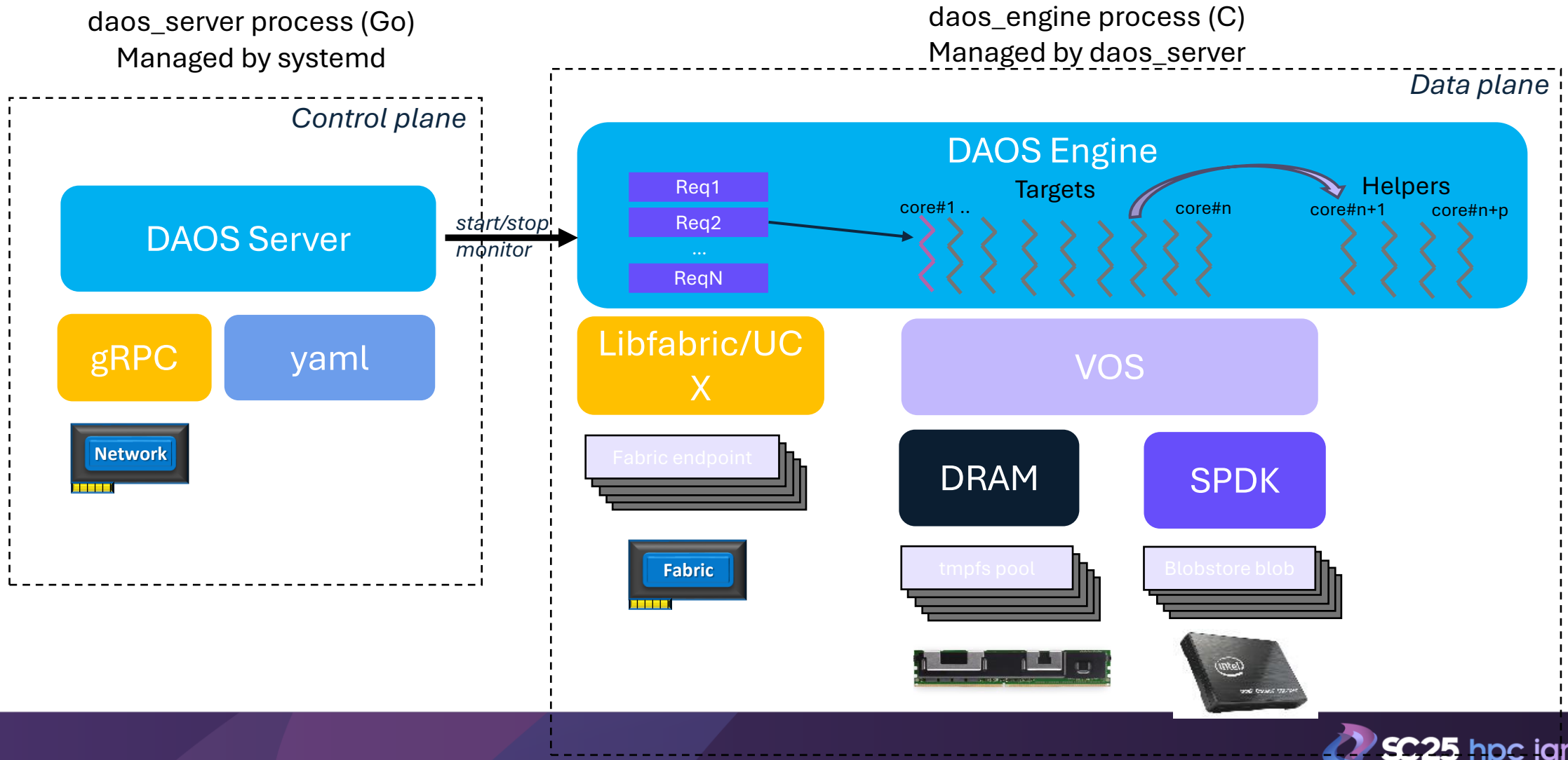
DAOS Service Structure



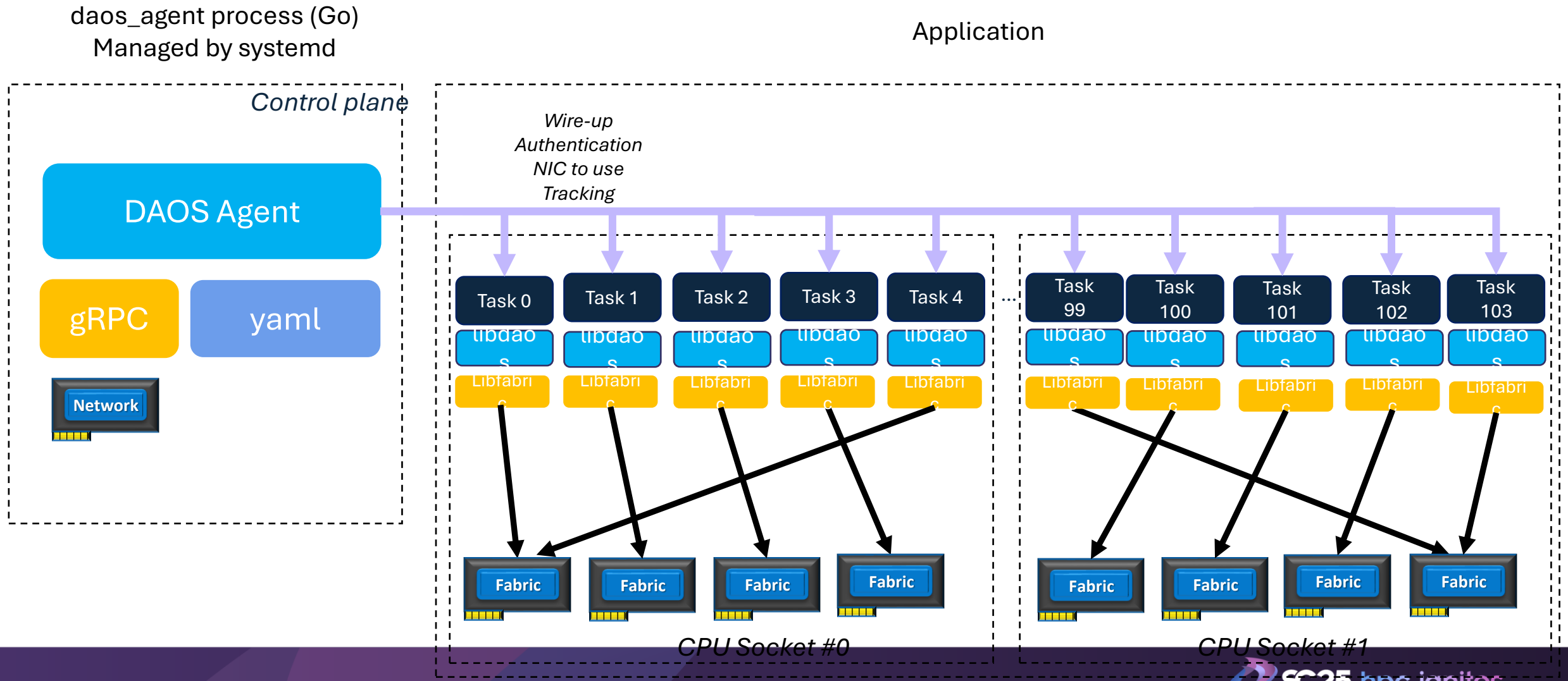
DAOS Service Structure



DAOS Service Structure

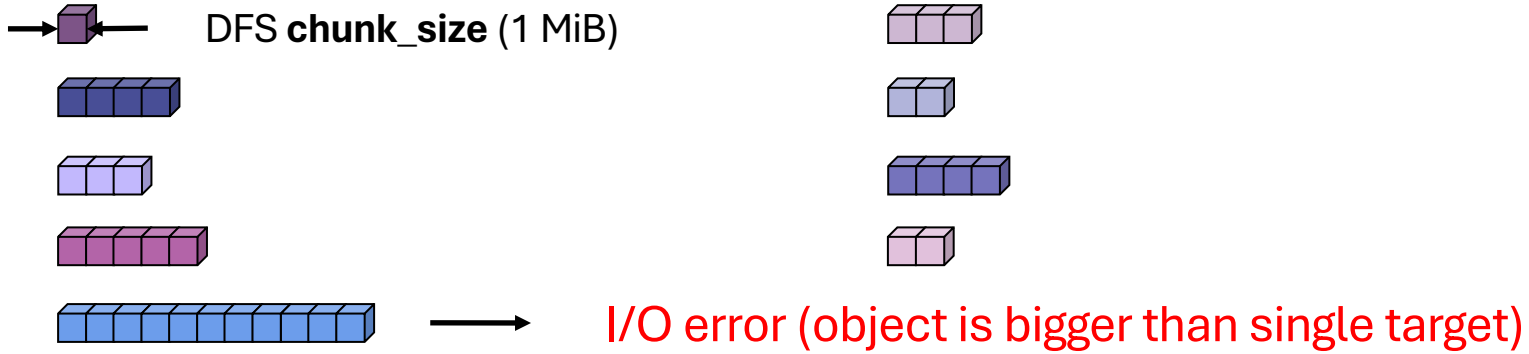


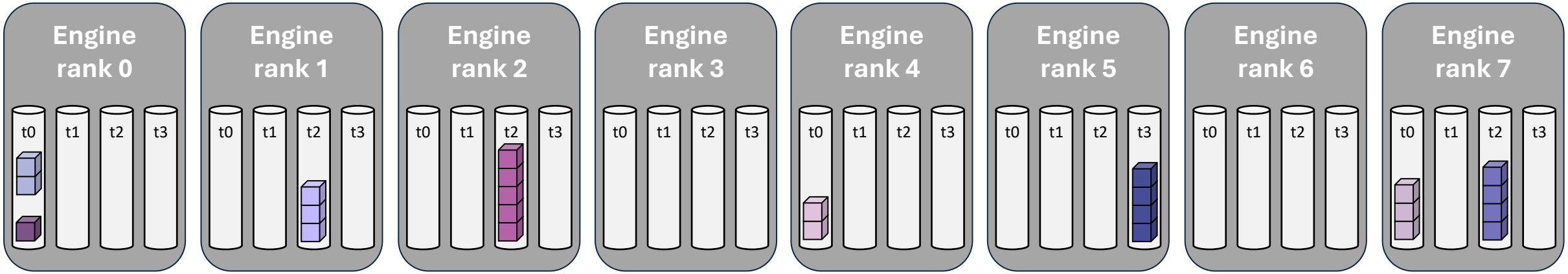
DAOS Client Structure



Data Distribution and Data Protection

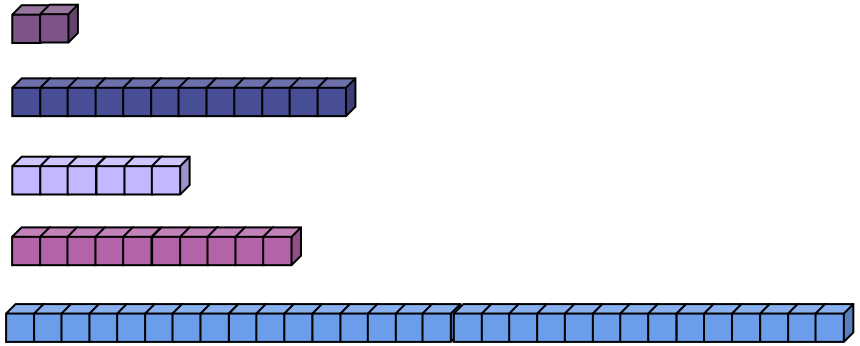
Sharding, object class S1

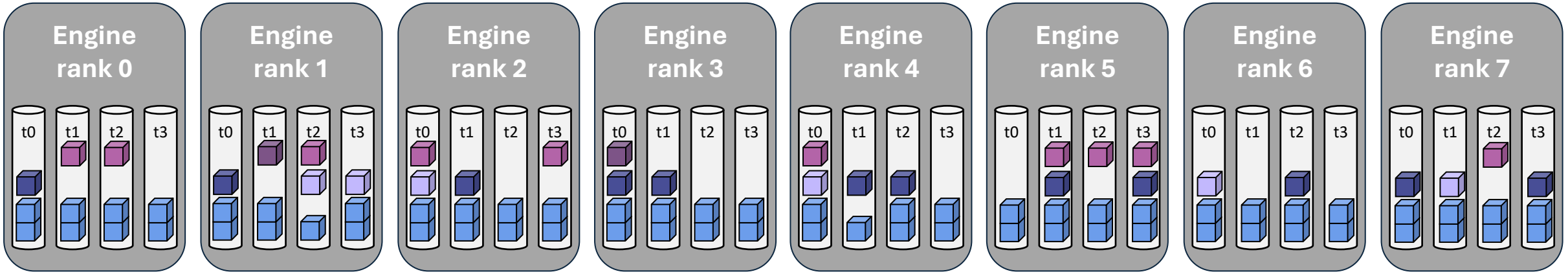




Data Distribution and Data Protection

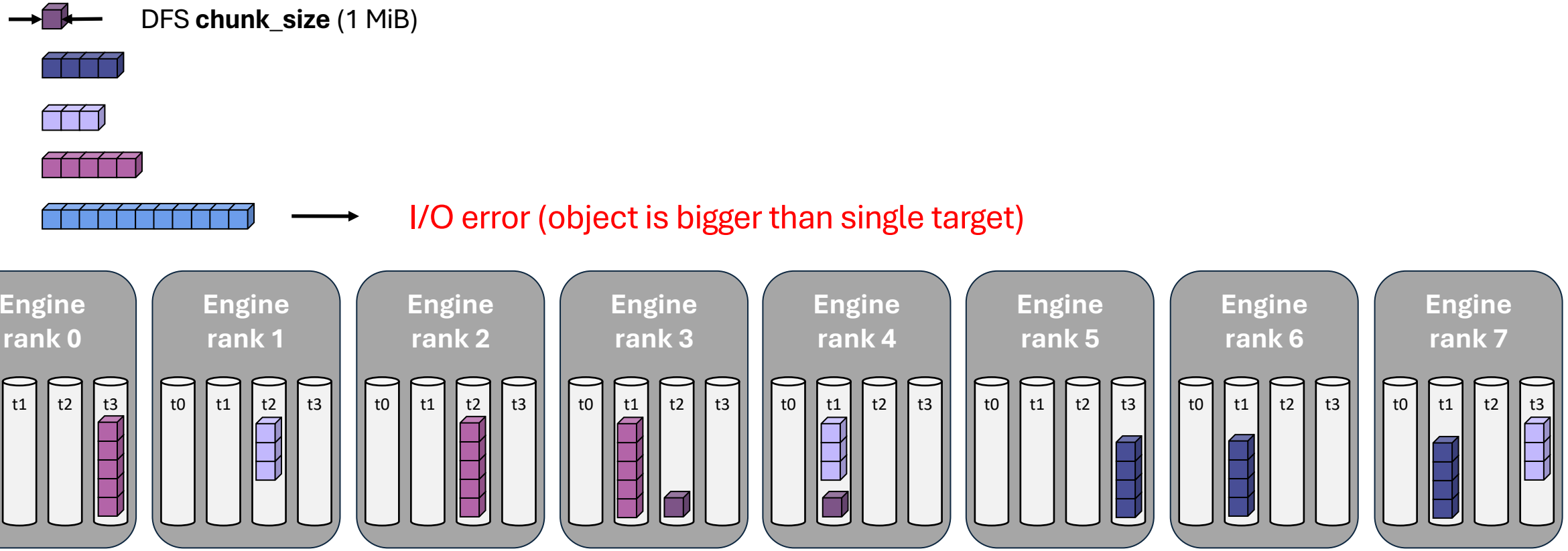
Sharding, object class SX





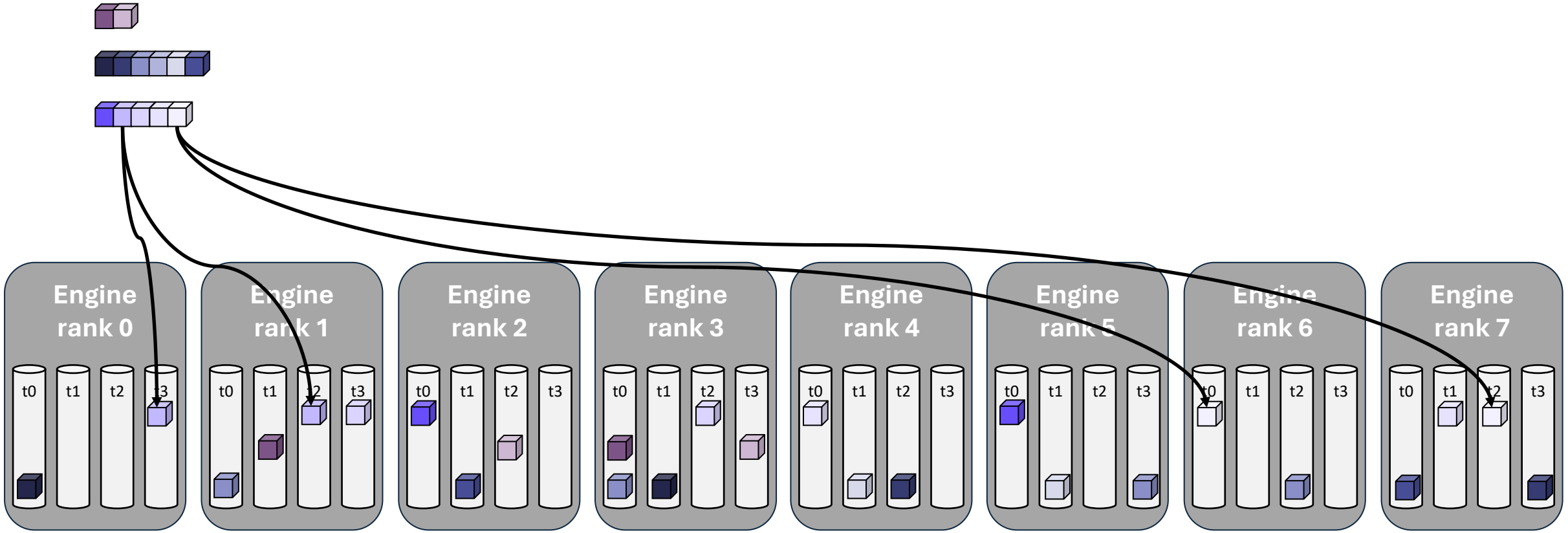
Data Distribution and Data Protection

Replication, object class RP_3G1 (rf_lvl=engine)



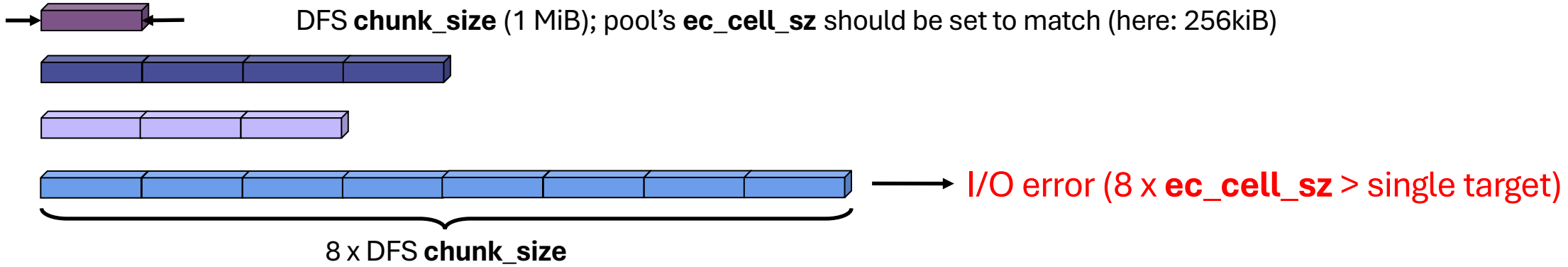
Data Distribution and Data Protection

Replication, object class RP_2GX (rf_lvl=engine)



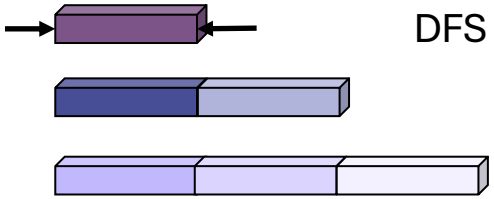
Data Distribution and Data Protection

Erasure Coding, object class EC_4P2G1 (rf_lvl=engine)

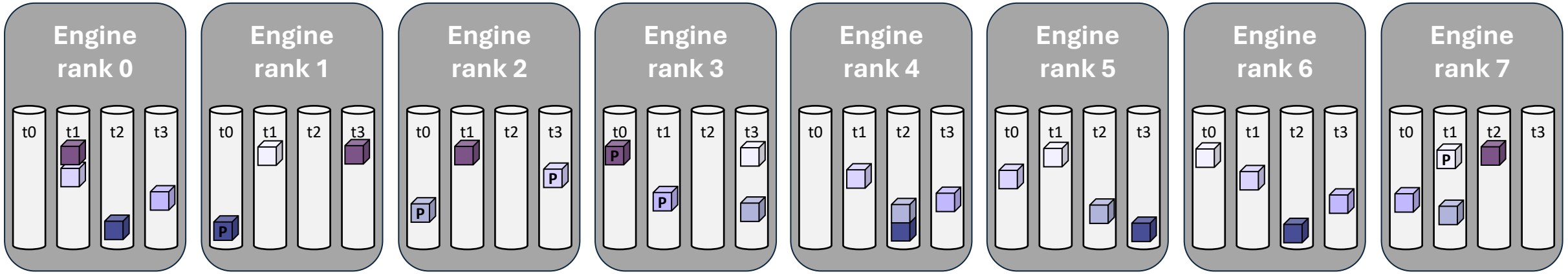


Data Distribution and Data Protection

Erasure Coding, object class EC_4P1GX (rf_lvl=engine)



DFS **chunk_size** (1 MiB); pool's **ec_cell_sz** should be set to match (here: 256kiB)



DAOS Object - Old Update Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN;
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1;
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_update(oh, DAOS_TX_NONE, 0, &dkey, 2, &iod, &sgl, NULL);
```

Multi-Level KV Object – Old Fetch Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iov;
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_fetch(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], NULL, NULL);
```

Ceph

Adrian Jackson, EPCC

a.jackson@epcc.ed.ac.uk

NOVEMBER 2025

The Ceph Distributed Storage System

- Open-source
- Designed for
  - Commodity hardware
  - Resilience and data safety
  - Scalability
- Popular in Cloud/Industry

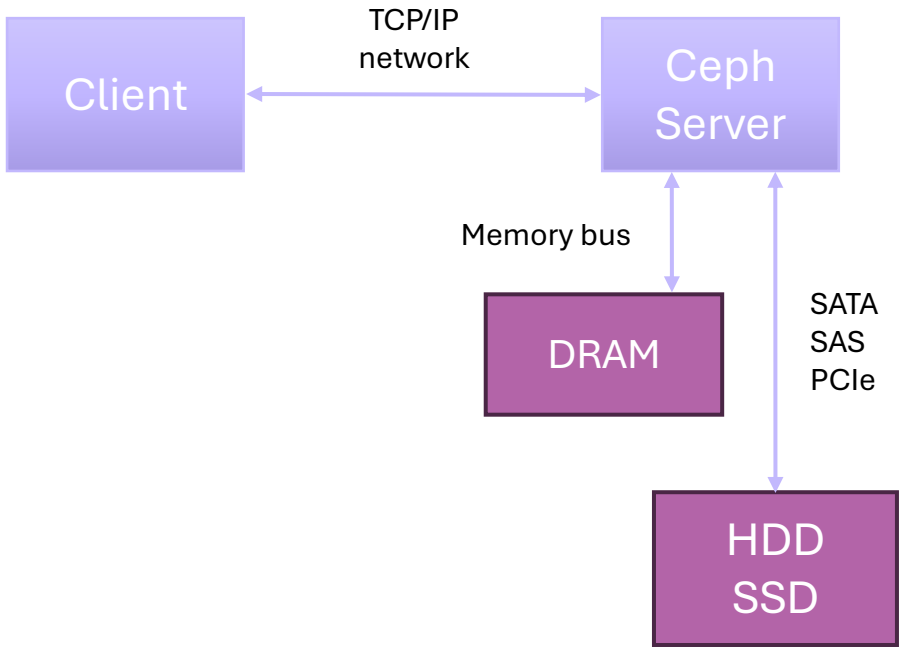
https://ceph.io/en/discover/

https://docs.ceph.com/en/latest/

https://www.youtube.com/watch?v=PmLPbrf-x9g

Hardware Support

- Ceph is software-defined
  - No specific hardware required
- Supports commodity and production hardware
  - HDDs
  - SSDs (SATA/SAS/NVMe)
  - TCP/IP networks
- No DPDK or RDMA support out-of-the-box yet



https://docs.ceph.com/en/latest/start/hardware-recommendations/#data-storage

https://docs.ceph.com/en/latest/rados/configuration/network-config-ref/#general-settings

Ceph Architecture

- Ceph Storage Cluster daemons (a.k.a. RADOS)
  - Object Storage Daemon (OSD)
  - Monitor Daemon
  - Manager Daemon
- Other daemons
  - Rados Block Device (RBD)
  - Rados GateWay (RGW) → S3
  - MetaData Server (MDS) → POSIX
- All daemons can be deployed and scaled independently

ceph-mon

ceph-mgr

ceph-osd

Object Storage Daemon (OSD) – ceph-osd

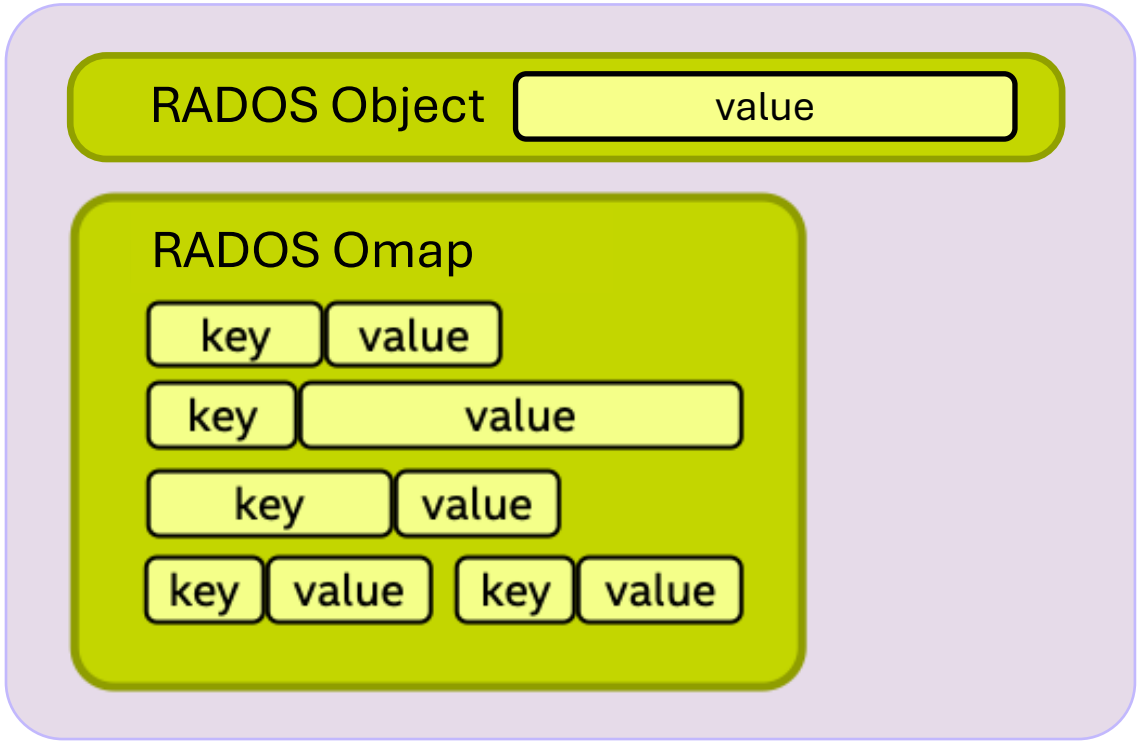
- Manages local storage in a node
  - Potentially multiple OSDs per node
- Exploits raw devices
- Stores object data
- Stores metadata index
  - Can exploit fast storage layer if present
- Clients perform I/O directly to OSDs

- At least 3 OSDs per system for data safety
- Require 2-4 GiB DRAM each
- Scales up to 10.000s of OSDs

ceph-osd

RADOS object storage API

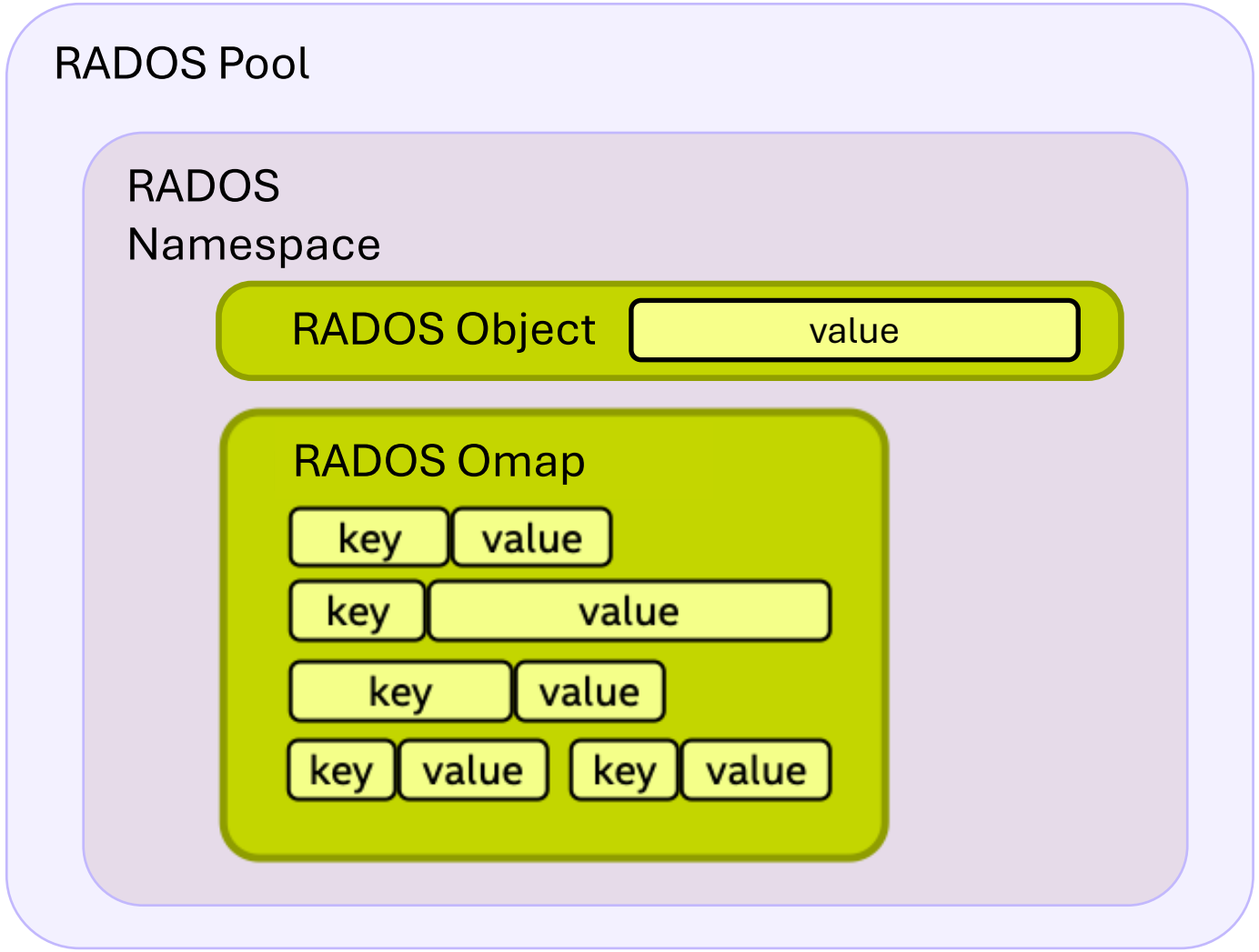
- Clients can interact with the API via librados
- Object
  - Identified by name
  - Can have attributes
  - Regular object
    - stores string of bytes
  - Omap object
    - provides key-value dictionary



https://docs.ceph.com/en/reef/rados/api/librados/

RADOS object storage API

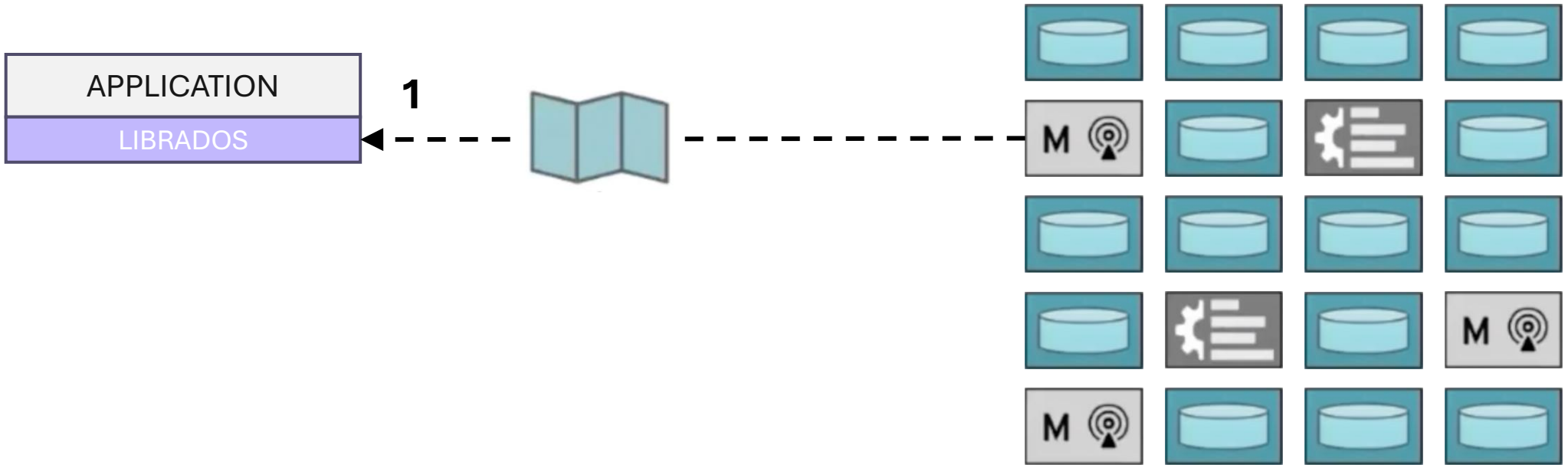
- Pool
  - Partitions object namespace
  - Usually, one pool is created for each type of application
  - Can be bound to specific OSDs
  - Can be configured for
    - Replication
    - Erasure-Coding
- Namespace
  - Partitions object namespace within a pool
  - Lightweight w.r.t. pool



Algorithmic placement

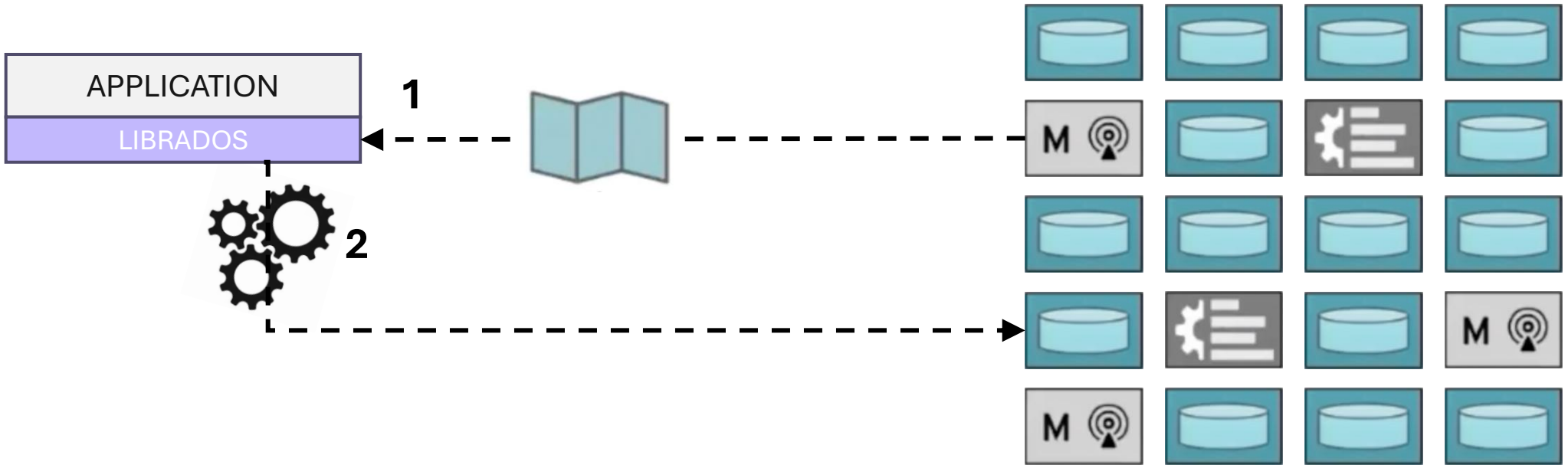
Algorithmic placement

- 1: retrieve up-to-date cluster map



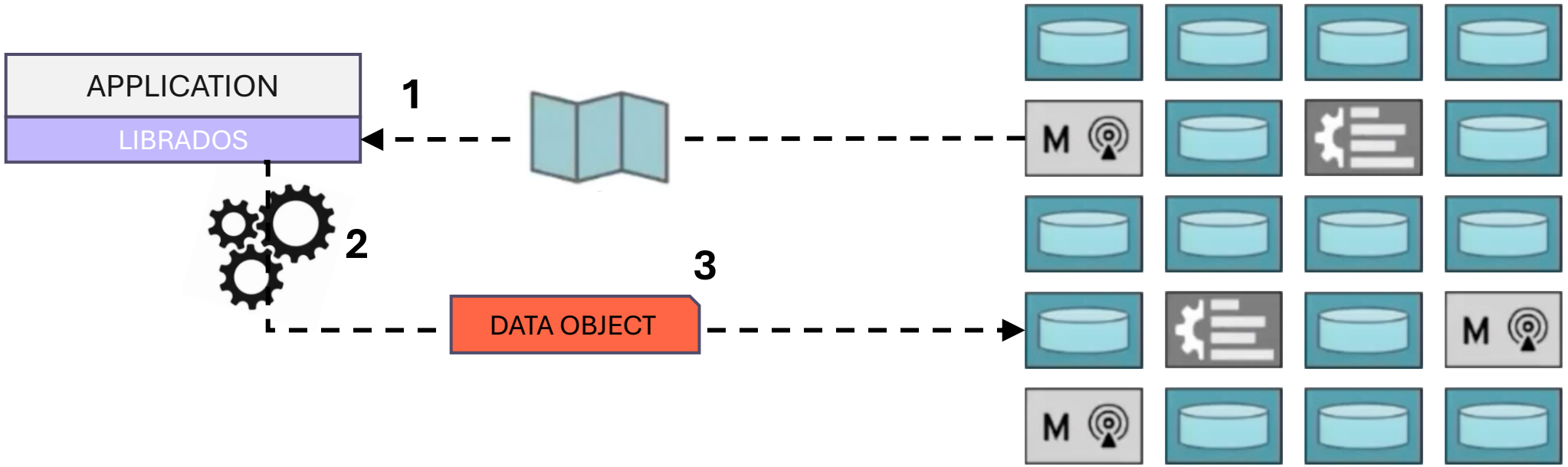
Algorithmic placement

- 1: retrieve up-to-date cluster map
- 2: calculate placement based on object name and map



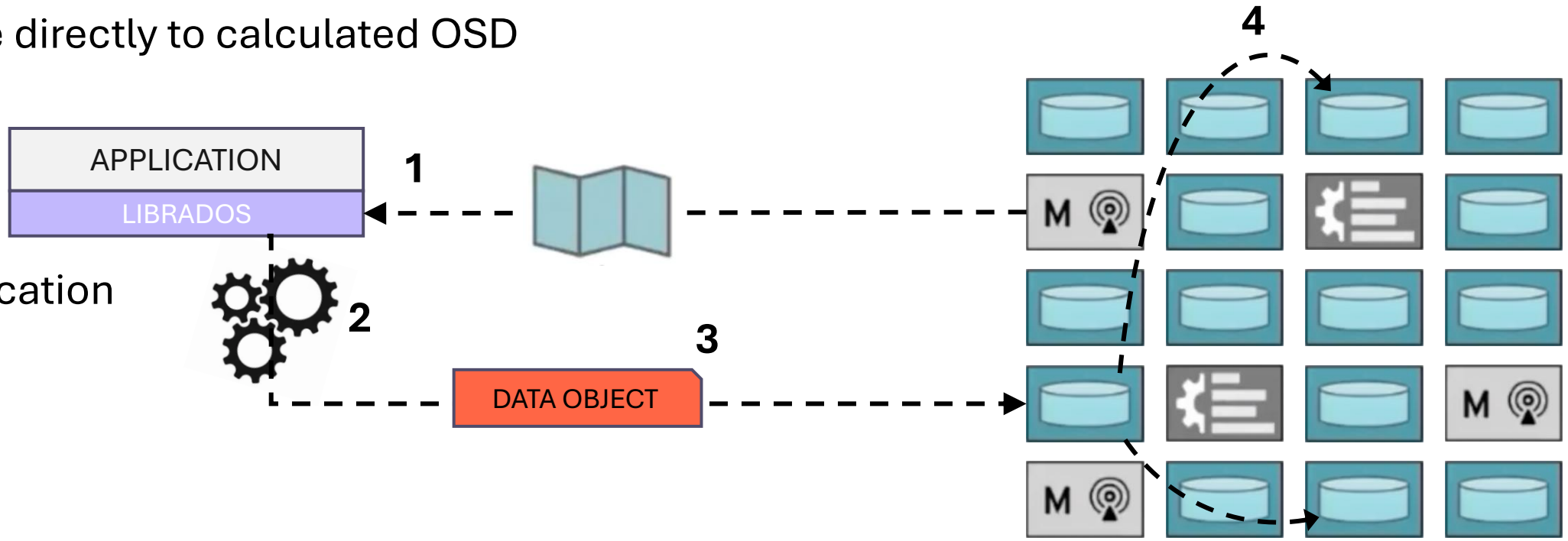
Algorithmic placement

- 1: retrieve up-to-date cluster map
- 2: calculate placement based on object name and map
- 3: write directly to calculated OSD



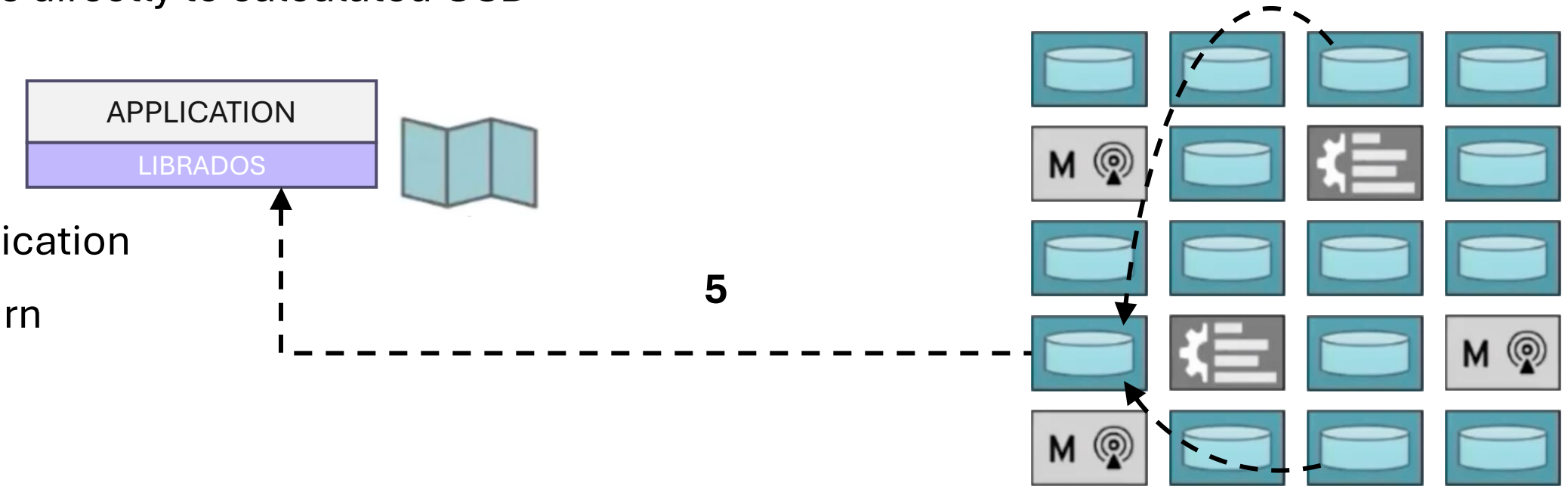
Algorithmic placement

- 1: retrieve up-to-date cluster map
- 2: calculate placement based on object name and map
- 3: write directly to calculated OSD
- 4: replication

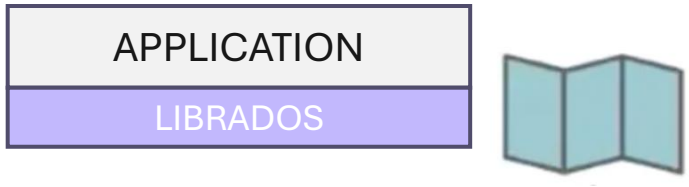


Algorithmic placement

- 1: retrieve up-to-date cluster map
- 2: calculate placement based on object name and map
- 3: write directly to calculated OSD
- 4: replication
- 5: return

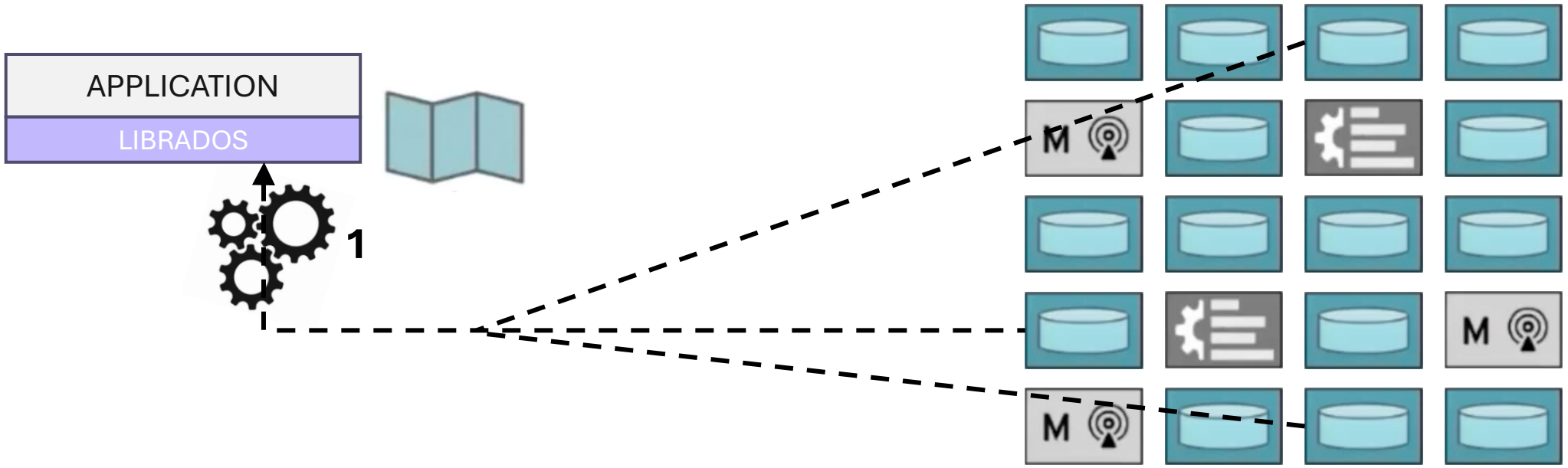


Algorithmic placement – read



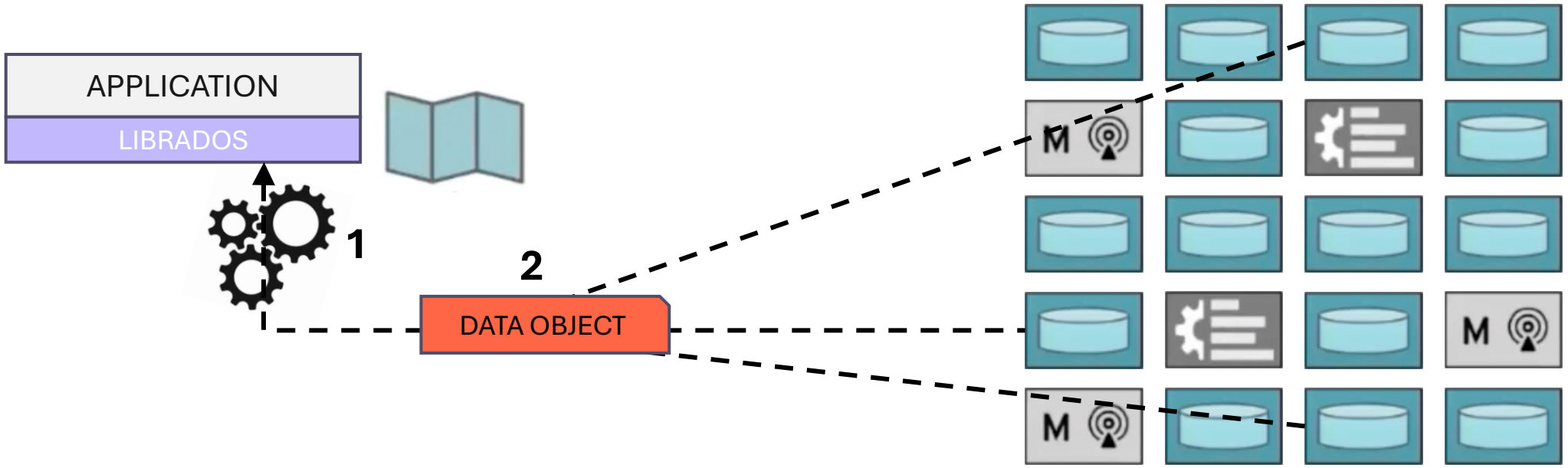
Algorithmic placement – read

- 1: calculate placement based on object name and map



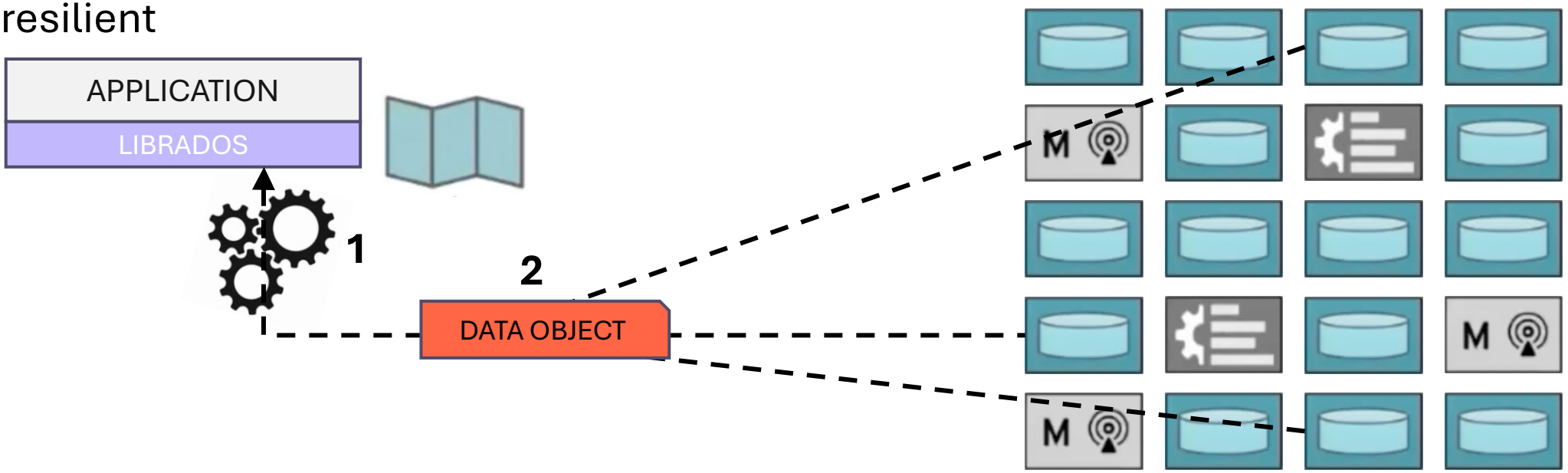
Algorithmic placement – read

- 1: calculate placement based on object name and map
- 2: read object directly from OSDs



Algorithmic placement

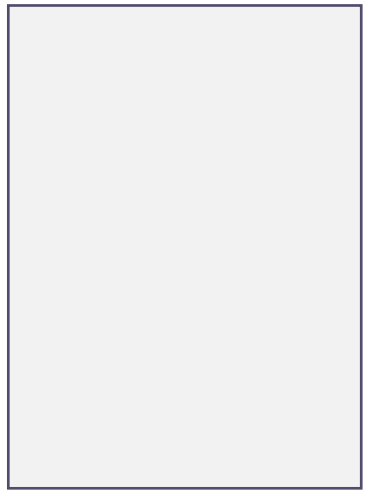
- Algorithmic placement enables scalability
- RADOS' placement algorithm is deterministic and repeatable – CRUSH
- Failure resilient



Placement Groups

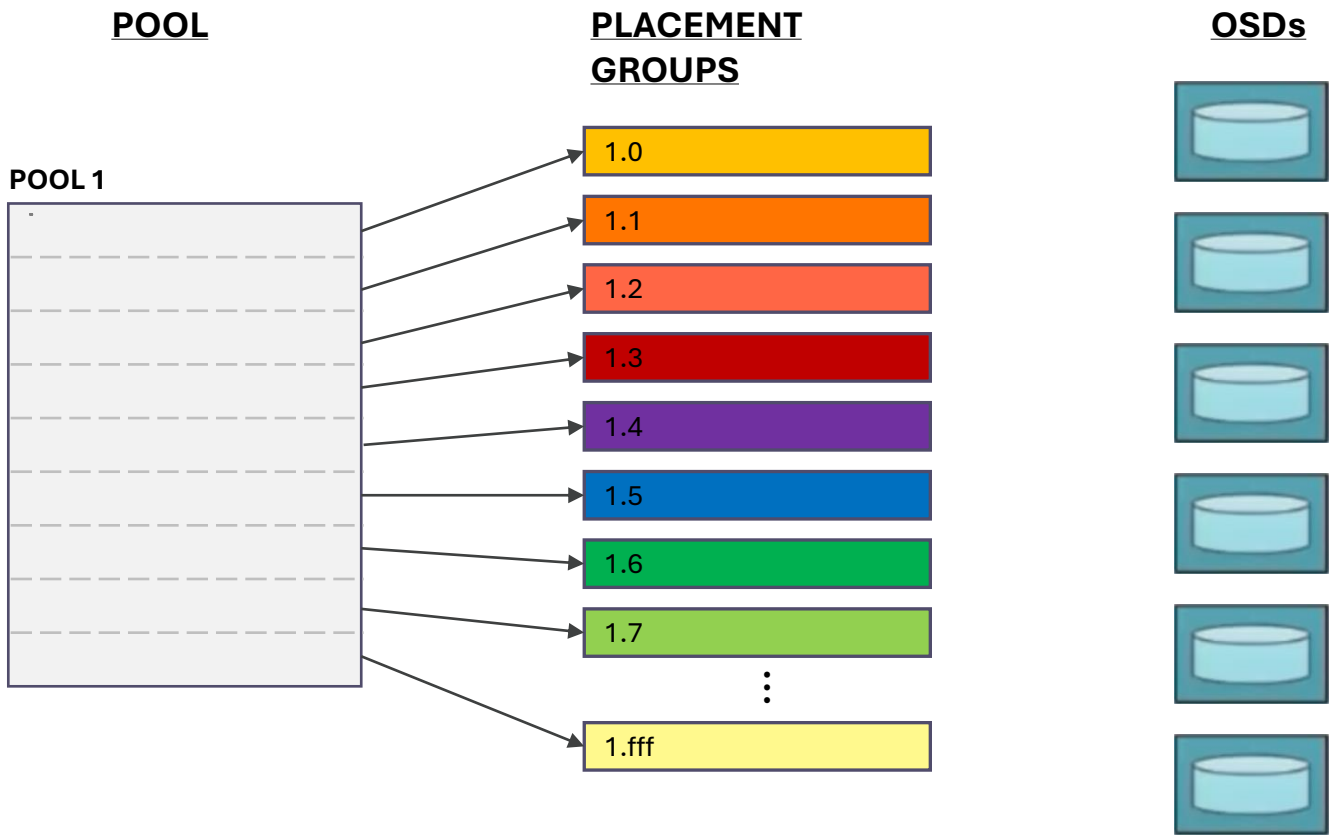
POOL

POOL 1

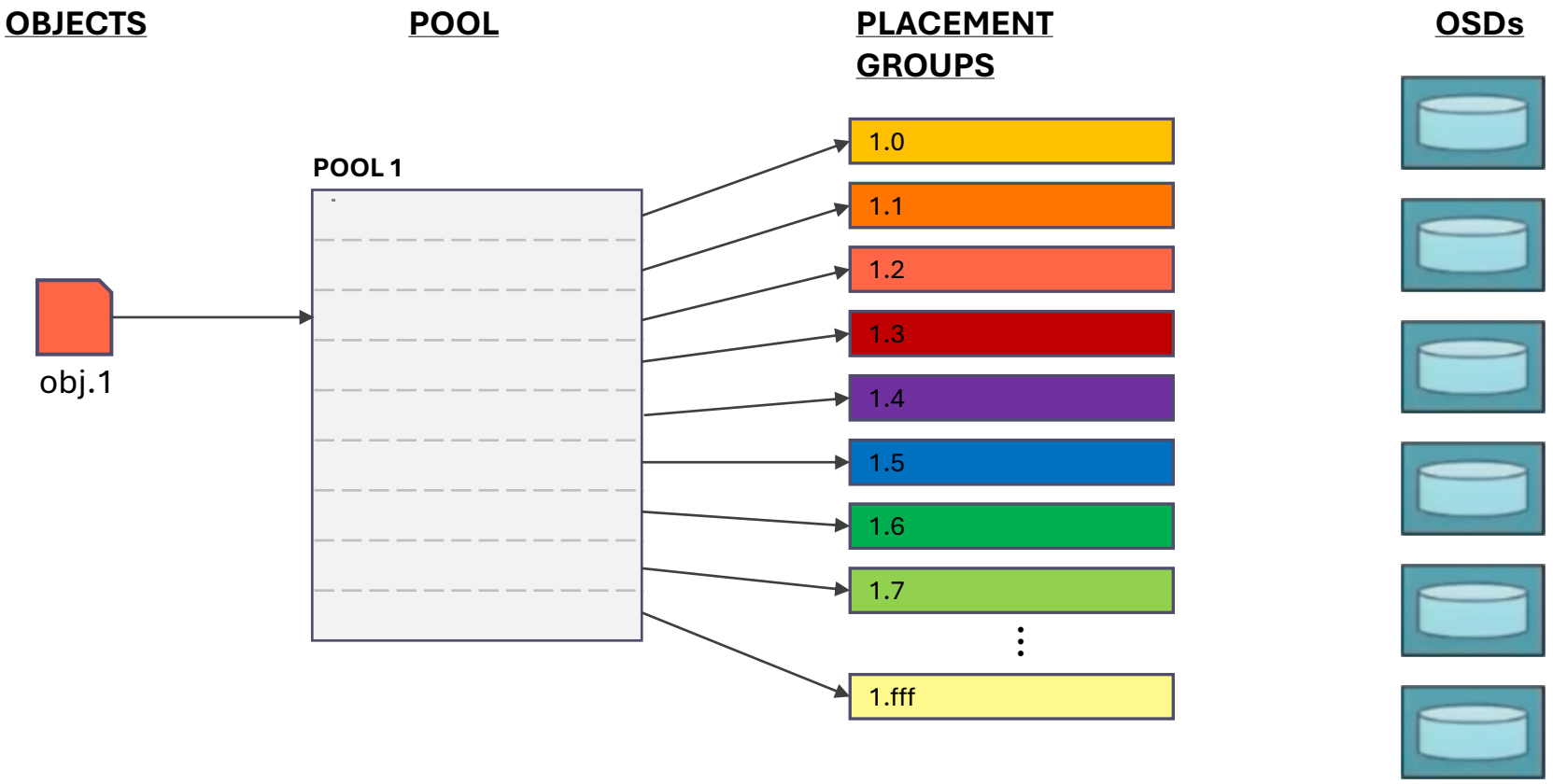


OSDs

Placement Groups

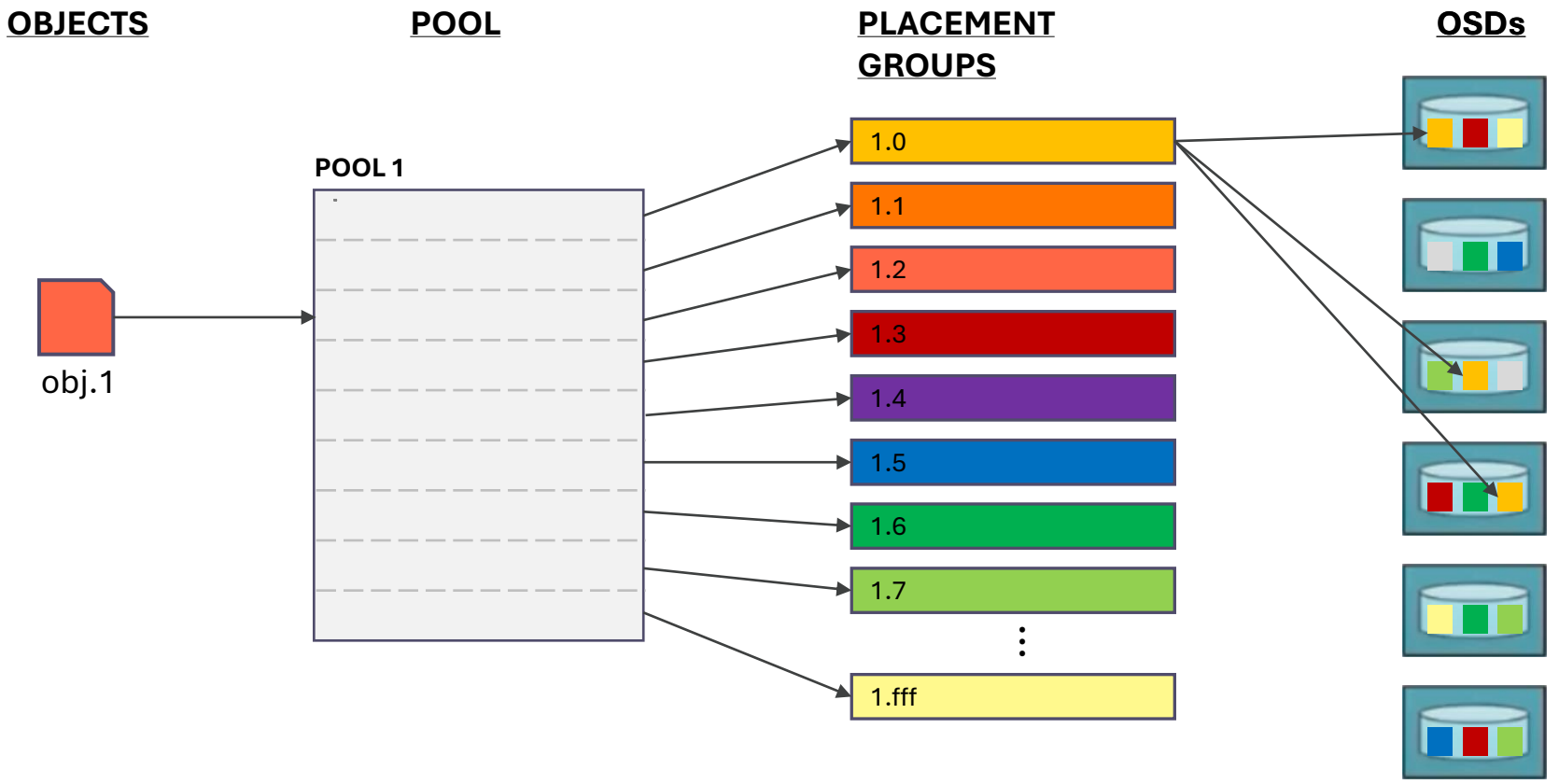


Placement Groups



pgid = hash(obj_name) % pg_num
Many GiB of data per PG

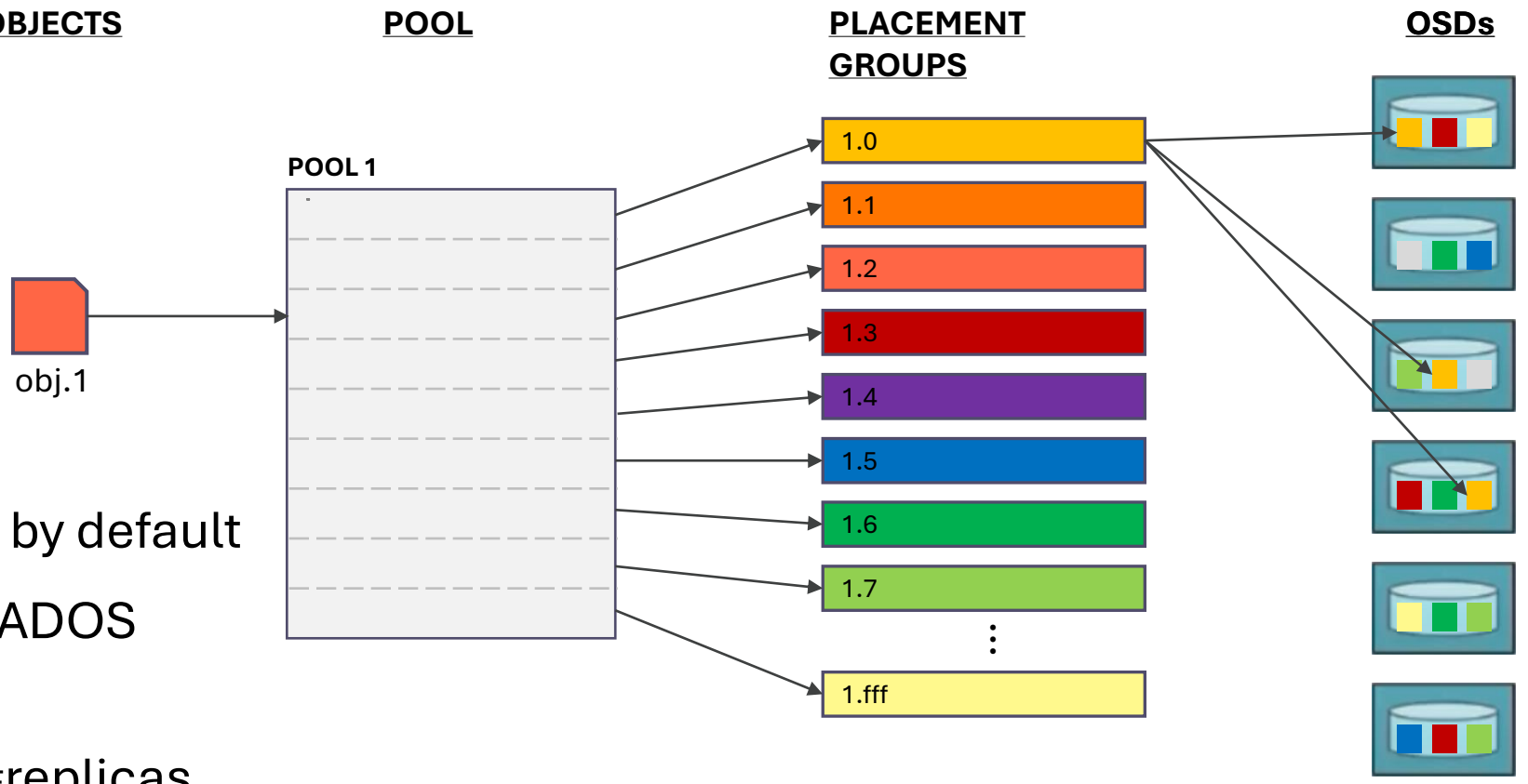
Placement Groups



pgid = hash(obj_name) % pg_num
Many GiB of data per PG

N replicas of each PG
10s of PGs per OSD

Placement Groups



- PGs are auto-adjusted by default
- As a rule of thumb, a RADOS system should have

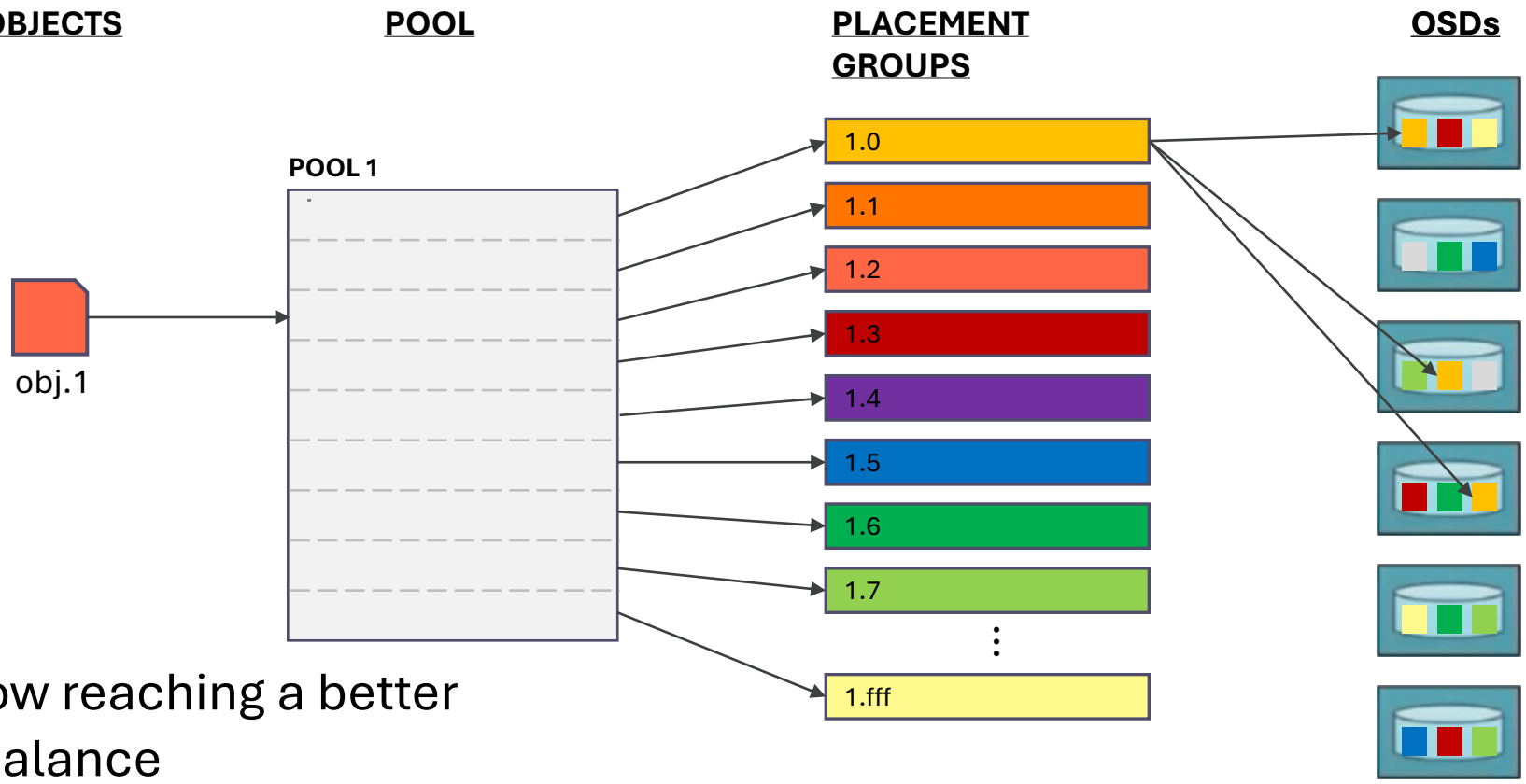
  #PG = 100 * #OSD / #replicas

https://docs.ceph.com/en/latest/rados/operations/pgcalc/

pgid = hash(obj_name) % pg_num
Many GiB of data per PG

N replicas of each PG
10s of PGs per OSD

Placement Groups



- Placement Groups allow reaching a better performance / safety balance

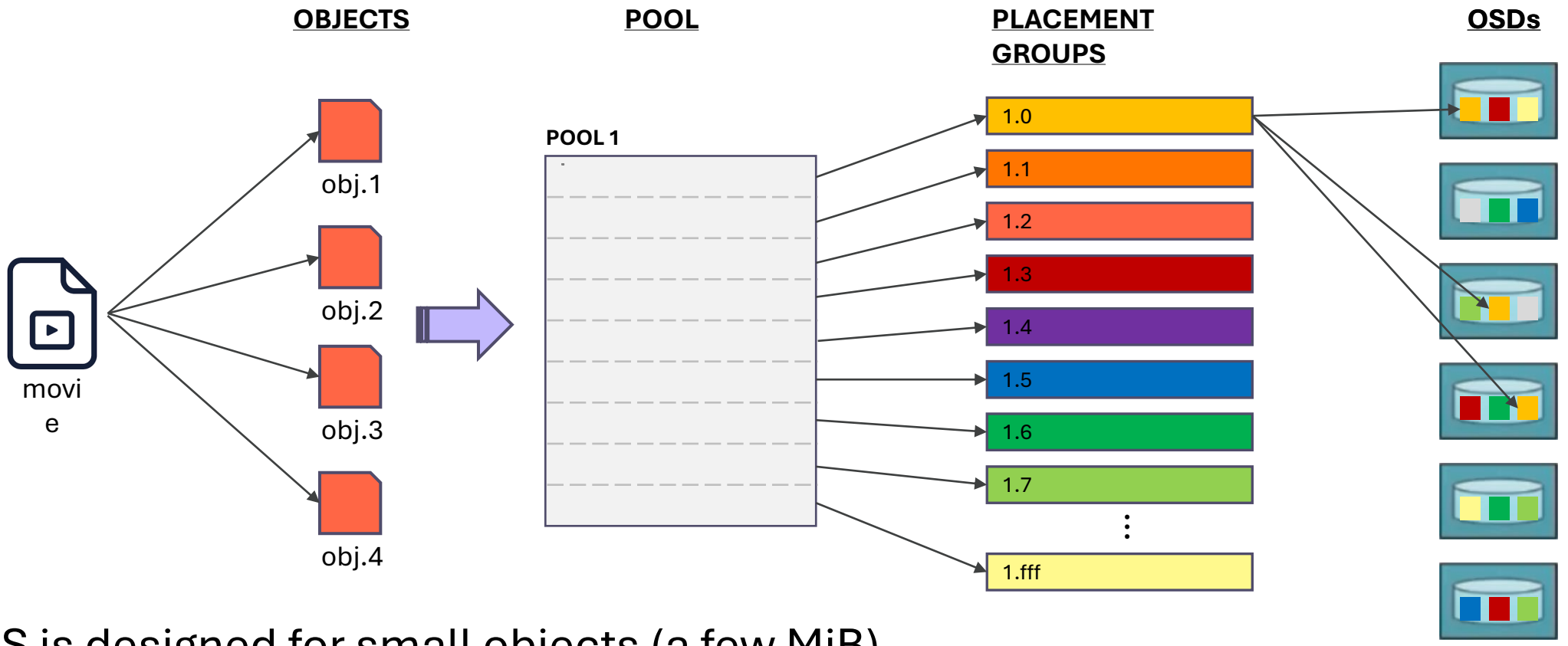
https://www.youtube.com/watch?v=PmLPbrf-x9g

https://ceph.io/assets/pdfs/weil-rados-pdsw07.pdf - section 2

pgid = hash(obj_name) % pg_num
Many GiB of data per PG

N replicas of each PG
10s of PGs per OSD

Placement Groups



- RADOS is designed for small objects (a few MiB)

pgid = hash(obj_name) % pg_num
Many GiB of data per PG

N replicas of each PG
10s of PGs per OSD

RADOS consistency and persistency

- Strong consistency guarantees
  - Write-read, read-write, write-write
  - No client-side caching
- Algorithm similar to MVCC
  - Placement calculation
  - Write to primary OSD
  - Replicate to peer OSDs in PG
  - Index object location and return
  - Reader always checks latest index entry on primary OSD

RADOS consistency and persistency

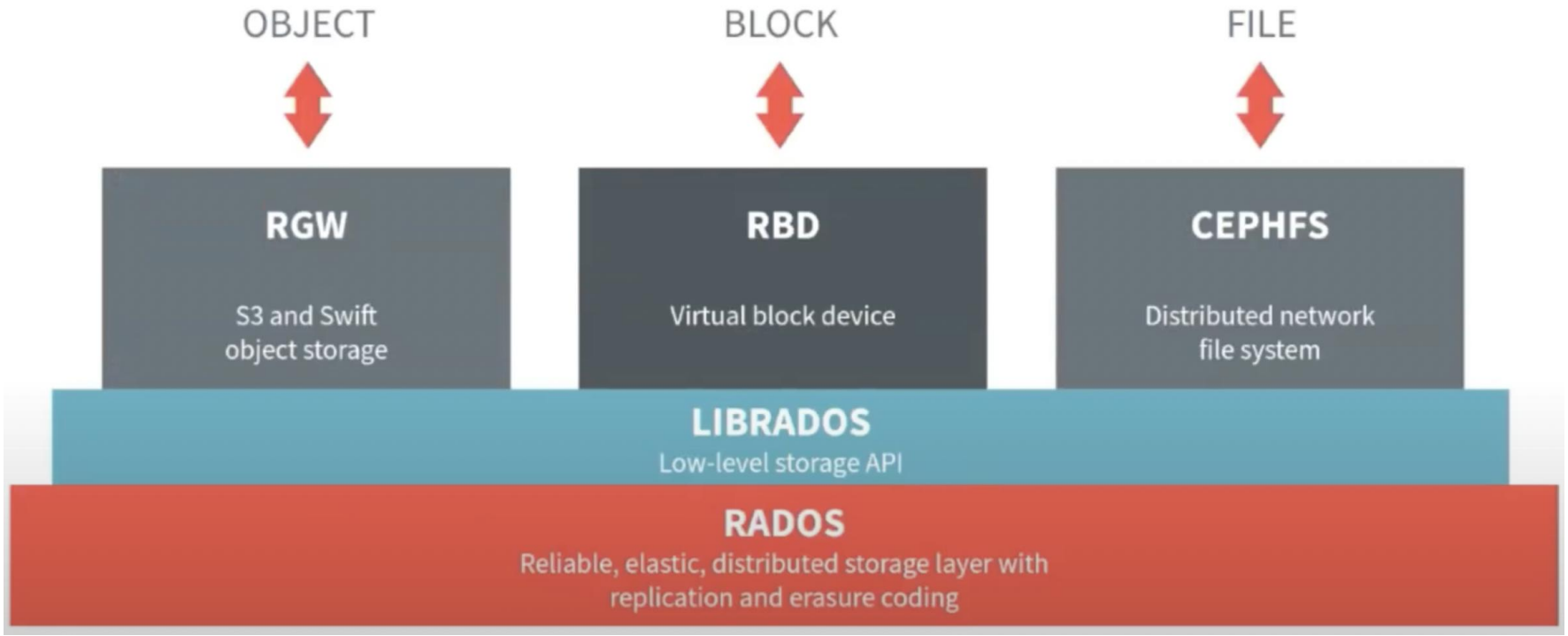
- Immediate persistency
  - Two-step persisting procedure
- Inefficiencies
  - Frequent RPCs to primary OSD
  - Copy on write for partial writes
  - Full transfer to client for partial reads
  - Two-step persistency increases latency

https://ceph.io/assets/pdfs/weil-rados-pdsw07.pdf - section 3

https://docs.ceph.com/en/latest/architecture/#interrupted-full-writes

https://docs.ceph.com/en/latest/dev/osd_internals/erasure_coding/enhancements/#partial-overwrites

Ceph interfaces on RADOS



- RBD and RGW have earned Ceph popularity in Cloud environments
- RGW and CephFS can eliminate the small object constrain in RADOS

https://www.youtube.com/watch?v=PmLPbrf-x9g

RADOS performance

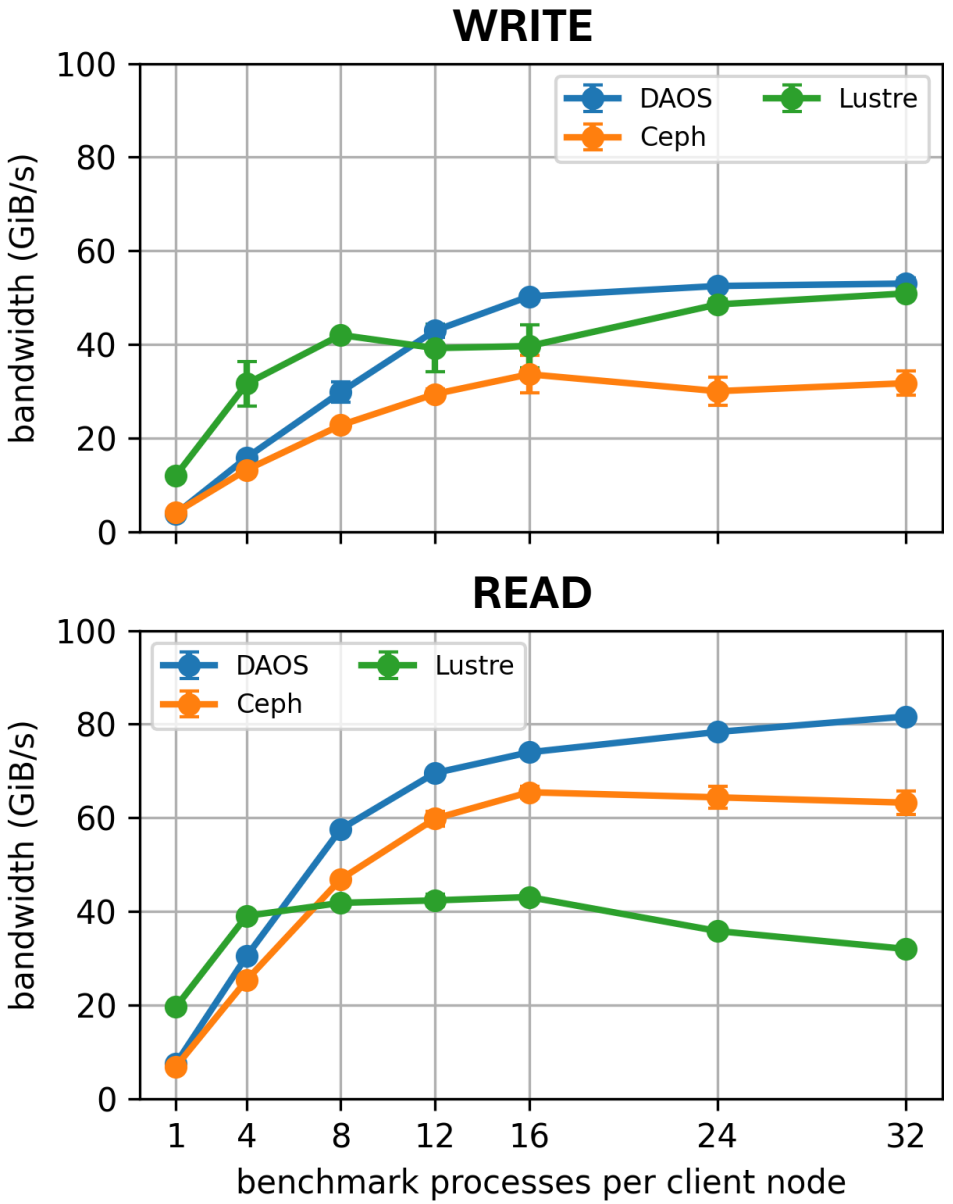
- Designed to scale
- Designed for safety
  - Overhead for RPCs to primary OSDs
  - Overhead for two-phase persistence
  - Partial write/read inefficiencies
- Performance analysis papers

https://sdm.lbl.gov/pdc/pubs/201811_PDSW2018-ObjEval.pdf
https://msstconference.org/MSST-history/2017/Papers/CephObjectStore.pdf
https://www.croit.io/blog/ceph-performance-benchmark-and-optimization
https://ceph.io/en/news/blog/2024/ceph-a-journey-to-1tibps/
https://arxiv.org/pdf/2409.18682

RADOS performance on NVMe

- RADOS/DAOS/Lustre deployments on 16x 6TiB nodes
- I/O benchmark (fdb-hammer) runs on 32 client nodes
- 10000 I/O operations of 1 MiB per process
- No replication or erasure-coding

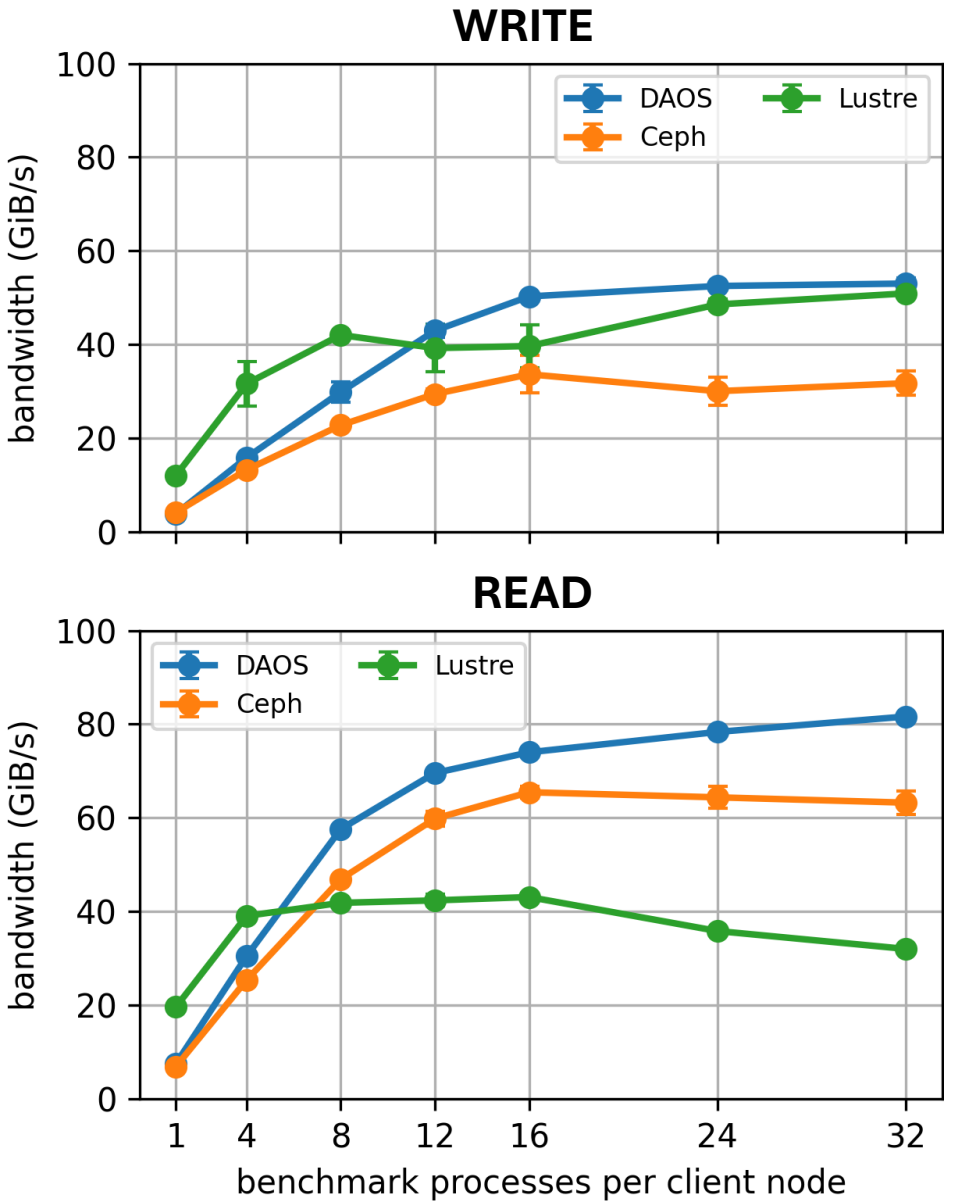
https://arxiv.org/pdf/2409.18682



RADOS performance on NVMe

- DAOS reaches close to hardware bandwidths for both write and read
- RADOS performance is lower but decent
- POSIX/Lustre struggles with metadata operations when managing small (1MiB) objects

https://arxiv.org/pdf/2409.18682



RADOS vs DAOS vs Lustre

Feature	RADOS	DAOS	Lustre
Algorithmic placement	○	○	
Client-side caching	●	●	○
Kernel involved	●		○
Centralised metadata			○
Strictly consistent	○	○	○
Immediately persistent	○	○	●
Provides POSIX files/directories	●	●	○
Provides objects	○	○	
Provides key-values	○	○	
Software-level data safety	○	○	

Feature	RADOS	DAOS	Lustre
High-performance networks		○	○
Byte-addressable		○	
Zero-copying		○	
Can exploit fast storage tier	○	○	○
Supports HDDs	○		○
Can scale to O(10k) nodes	○	○	○
Performs for GiB objects		○	○
Performs for MiB object	○	○	
Performs for KiB objects		○	

○ Yes ● Can do No

librados

- Provides functionality to manipulate all entities in RADOS
  - Daemons, Pools, Objects / Omaps
- Available in many programming languages, including
  - C, C++, Python, Java, PHP, …
  - The C API is the richest and best documented
  - Ceph distributes the `rados` command-line tool wrapping librados






https://docs.ceph.com/en/reef/rados/api/librados/

librados – cluster handle

```
int rados_create2(rados_t *pcluster, const char *const clustername,
                  const char *const name, uint64_t flags)
```

- Initialises a `ratos_t` struct given the cluster name and RADOS username to user for subsequent librados calls

librados – configuration

- The minimum configuration required by librados includes
  - address and port of one ceph-mon (ideally of all ceph-mon)
  - path to keyring file
- Can be specified in different ways
  - Manually item by item via `rados_conf_set()`
  - Via environment variables plus `rados_conf_parse_env()`
  - Via a configuration file plus `rados_conf_read_file()`

librados – configuration

```
int rados_conf_read_file(rados_t cluster, const char *path)
```

- If path is NULL, the following paths are checked:
  - $CEPH_CONF environment variable
  - /etc/ceph/ceph.conf
  - ~/.ceph/config
  - ceph.conf in the current working directory

- If a keyring path is not given in the configuration (“keyring” item), a keyring file with the name `ceph.<username>.keyring` is looked for in the same directories

librados – cluster connect

`int rados_connect(rados_t cluster)`

- Opens a connection with a RADOS cluster
- If succeeds (`rc == 0`), it must be released with `rados_shutdown()`

`int rados_shutdown(rados_t cluster)`

- Closes an open connection with a RADOS cluster

(RADOS pool create – admin only)

```
ceph osd pool create ${pool_name} ${pg_count} ${pgp_count} replicated
```

- Creates a replicated pool
- Defaults to 3 replicas

```
ceph osd pool set ${pool_name} size ${replica_count}
```

- Sets the replica count for a replicated pool

(RADOS pool create – admin only)

```
ceph osd erasure-code-profile set myprofile k=${k} m=${m} \
  crush-failure-domain=host
```

- Creates an erasure-code profile

```
ceph osd pool create ${pool_name} ${pgc} ${pgpc} erasure [myprofile]
```

- Creates an erasure-coded pool with a given EC profile
- Defaults to 2+2
- Does not support omap objects

librados – pool connect

```
int rados_ioctx_create(rados_t cluster, const char *pool_name,
                       rados_ioctx_t *ioctx)
```

- Initialises an IO context struct for an existing pool
- The rados_ioctx_t allows performing IO operations on a pool
- If succeeds (`rc == 0`), it must be released with `rados_ioctx_destroy()`

```
int rados_ioctx_destroy(rados_ioctx_t io)
```

- Signals librados that an IO context will no longer (and must not) be used
- The iocxt may not be destroyed immediately if it holds active async operations

librados – namespaces

```
void rados_ioctx_set_namespace(rados_ioctx_t io, const char *nspace)
```

- Sets the namespace to use for a given IO context
- nspace can be set to LIBRADOS_ALL_NAMESPACES to list all objects in a pool with `rados_nobjects_list_open()`

librados – I/O APIs

- Synchronous I/O
  - Blocks until operation is complete
  - Provides methods to perform I/O to regular objects
- Transactional API
  - Allows specifying a set of operations to be performed atomically
  - Blocks until transaction is complete
  - Provides methods to perform I/O to regular objects as well as Omaps
- Async API
  - Does not block unless `wait_for_complete()` or `flush()` are called
  - Provides the same I/O features as the transactional API

librados – synchronous I/O

```
int rados_write(rados_ioctx_t io, const char *oid, const char *buf,
                size_t len, uint64_t off)
```

- Write len bytes from buf into the oid object, starting at offset off
- Creates object if n.e.
- Returns 0 on success or a negative value on failure

```
int rados_write_full(rados_ioctx_t io, const char *oid,
                     const char *buf, size_t len)
```

- If the object exists, it is atomically truncated and then written

librados – synchronous I/O

```
int rados_append(rados_ioctx_t io, const char *oid, const char *buf,
                 size_t len)
```

- Append len bytes from buf into the oid object
- Returns 0 on success or a negative value on failure

```
int rados_read(rados_ioctx_t io, const char *oid, char *buf,
               size_t len, uint64_t off)
```

- Read len bytes starting at off from object into buf
- Returns number of read bytes on success or a negative value on failure

librados – synchronous I/O

```
int rados_remove(rados_ioctx_t io, const char *oid)
```

- Delete an object.
- Returns 0 on success or a negative value on failure.

```
int rados_trunc(rados_ioctx_t io, const char *oid, uint64_t size)
```

- Resize an object to size.
- If shrinking, the excess data is deleted. If enlarging, filled with zeros.

librados – transactional I/O

- A single “rados operation” can perform multiple operations on one object atomically
- The whole operation will succeed or fail, and no partial results will be visible
- Operations may be either reads, which can return data, or writes, which cannot
- The effects of writes are applied and visible all at once
  - E.g. an operation that sets an xattr and then checks its value will not see the updated value

create_write_op → write_op_XXX → write_op_YYY → write_op_operate → release_write_op

create_read_op → read_op_XXX → read_op_YYY → read_op_operate → release_read_op

librados – transactional I/O

```
int rados_write_op_operate(rados_write_op_t write_op,
        rados_ioctx_t io, const char*oid, time_t *mtime, int flags)

int rados_read_op_operate(rados_read_op_t read_op,
        rados_ioctx_t io, const char*oid, int flags)
```

- Execute a “rados operation” on an object

librados – transactional I/O - Omap

```
void rados_write_op_omap_set2(rados_write_op_t write_op,
        char const *const *keys, char const *const *vals,
        const size_t*key_lens, const size_t *val_lens, size_t num)
```

- Insert an array of key-value pairs into an Omap

```
void rados_read_op_omap_get_vals_by_keys2(rados_read_op_t read_op,
        char const *const *keys, size_t num_keys,
        constsize_t *key_lens, rados_omap_iter_t *iter, int *prval)
```

- Query the values associated to a given array of keys
- Returns an iterator with the values
- Any failures are signaled via prval

librados – transactional I/O - Omap

```
unsigned int rados_omap_iter_size(rados_omap_iter_t iter)
```

- Returns number of elements in Omap iterator

```
int rados_omap_get_next2(rados_omap_iter_t iter, char **key,
        char **val, size_t *key_len, size_t *val_len)
```

- Extracts the next key-value pair from an Omap iterator
- If the end of the list has been reached, key=val=NULL, and keylen=vallen=0
- key and val should be copied as rados_omap_get_end() releases them

```
void rados_omap_get_end(rados_omap_iter_t iter)
```

The MPI-IO Interface

Optimizing parallel accesses

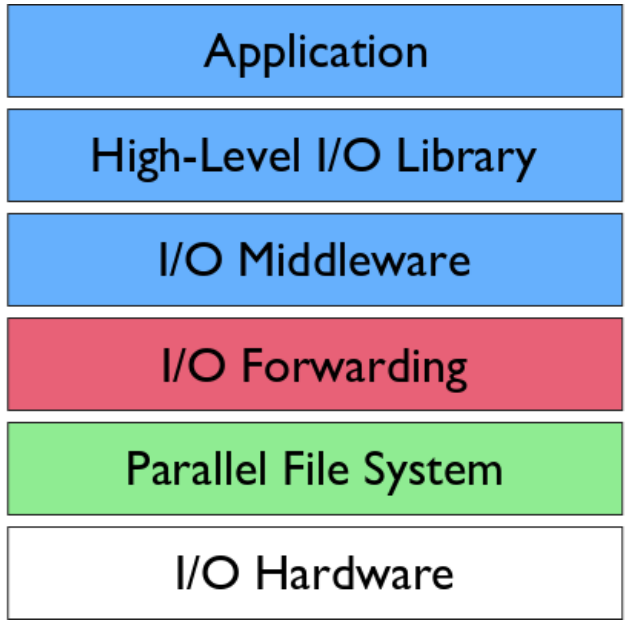
I/O for Computational Science

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

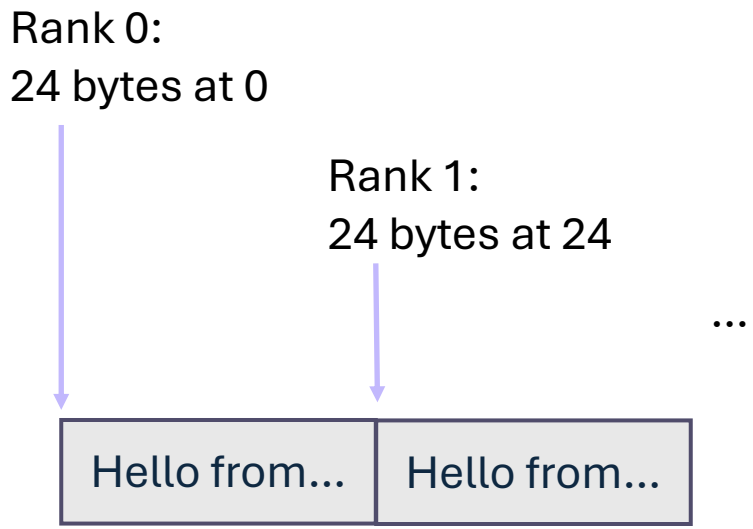
- I/O interface specification for use in MPI apps

- Data model is same as POSIX: stream of bytes in a file
- Like classic POSIX in some ways…
  - Open() → MPI_File_open()
  - Pwrite() → MPI_File_write()
  - Close() → MPI_File_close()
- Features many improvements over POSIX:
  - 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)

“Hello World” MPI-IO style

```
/* an "Info object":  these store key-value strings for tuning the
 * underlying MPI-IO implementation */
MPI_Info_create(&info);

snprintf(buf, BUFSIZE, "Hello from rank %d of %d\n", rank, nprocs);
len = strlen(buf);
/* We're working with strings here but this approach works well
 * whenever amounts of data vary from process to process. */
MPI_Exscan(&len, &offset, 1, MPI_OFFSET, MPI_SUM, MPI_COMM_WORLD);

MPI_CHECK(MPI_File_open(MPI_COMM_WORLD, argv[1],
            MPI_MODE_CREATE|MPI_MODE_WRONLY, info, &fh));

/* _all means collective.  Even if we had no data to write, we would
 * still have to make this call.  In exchange for this coordination,
 * the underlyng library might be able to greatly optimize the I/O */
MPI_CHECK(MPI_File_write_at_all(fh, offset, buf, len, MPI_CHAR,
            &status));

MPI_CHECK(MPI_File_close(&fh));
```



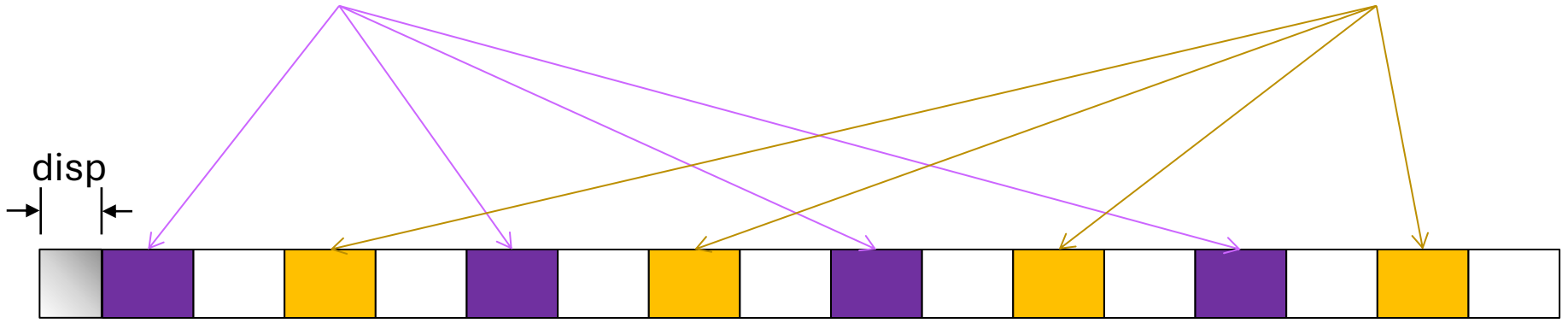
https://github.com/radix-io/hands-on/blob/main/hello-io/hello-mpiio.c

Simple MPI-IO

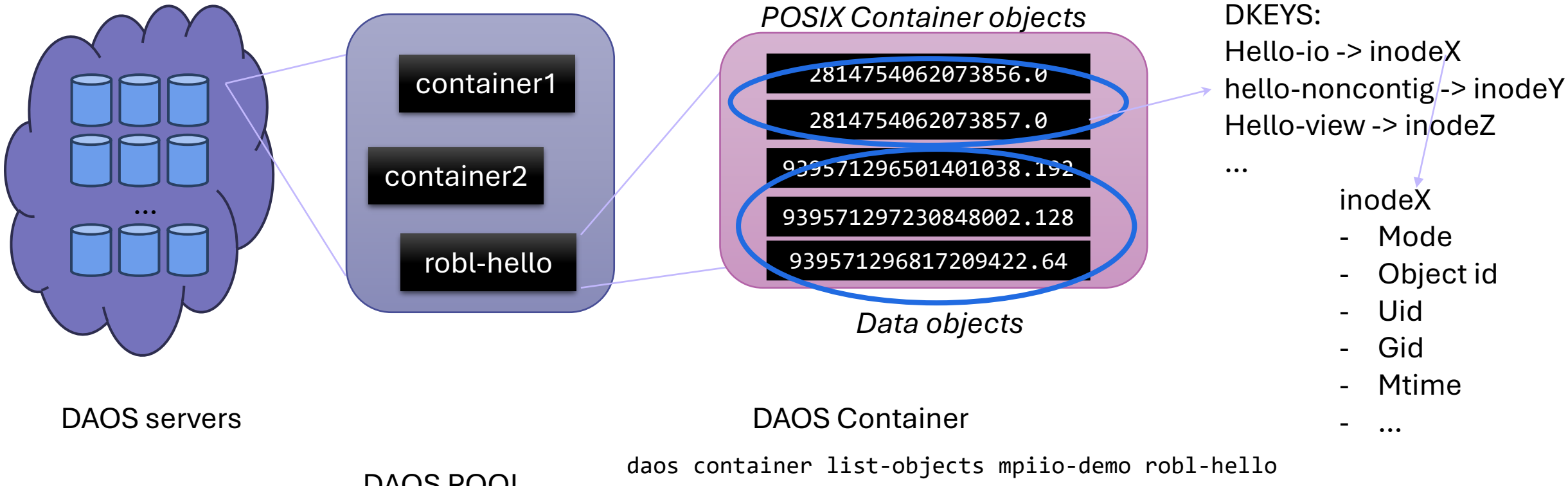
- Collective open: all processes in communicator
- File-side data layout with *file views*
- Memory-side data layout with *MPI datatype* passed to write

```
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);
```

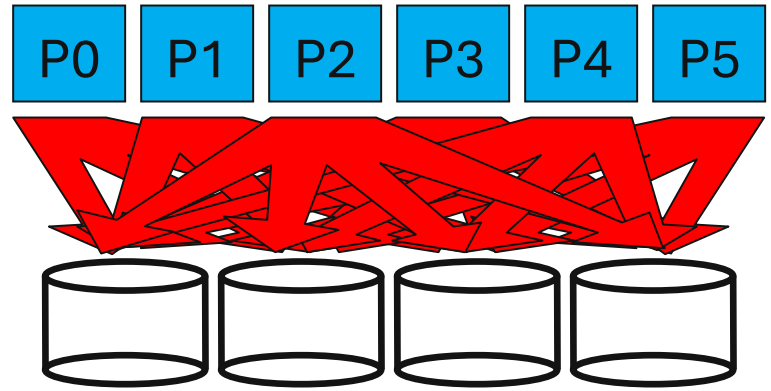


Under the hood: DAOS (essentially)

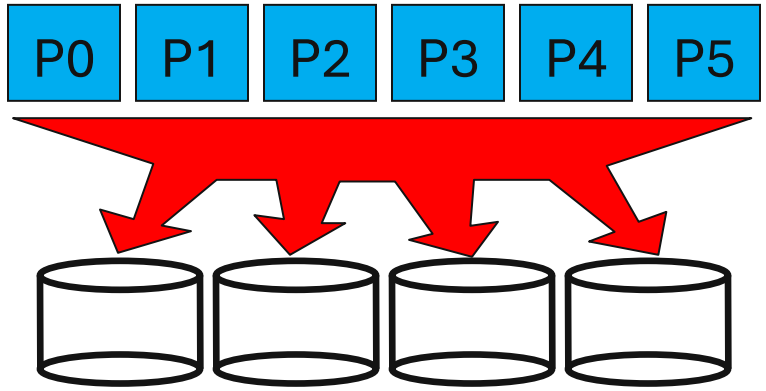


Independent and Collective I/O

- Independent I/O operations
  - Specify only what a single process will do
  - Do not pass on relationships between I/O on other processes
- Many applications have alternating 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



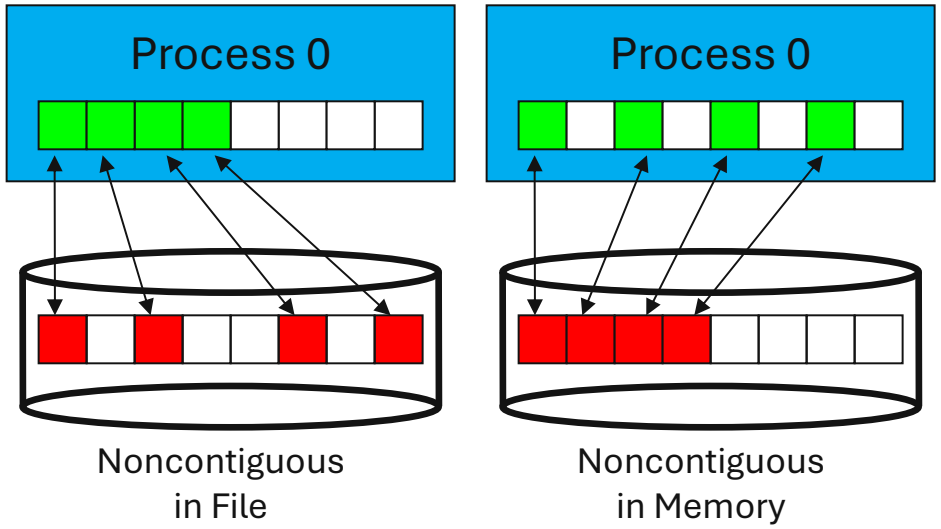
Independent I/O

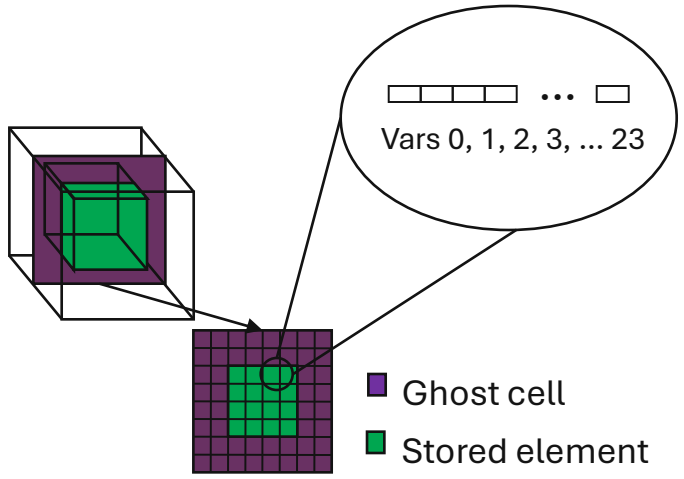


Collective I/O

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
  - 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



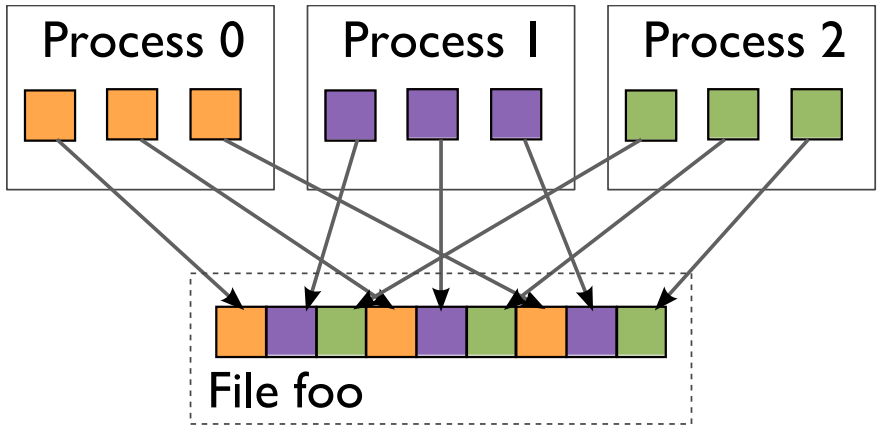


Extracting variables from a block and skipping ghost cells will result in noncontiguous I/O

I/O Transformations

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

- Goals of transformations:
  - Reduce number of I/O operations to PFS (avoid latency, improve bandwidth)
  - Avoid lock contention (eliminate serialization)
  - Hide huge number of clients from PFS servers
- "Transparent" transformations don't change the final file layout
  - File system is still aware of the actual data organization
  - File can be later manipulated using serial POSIX I/O



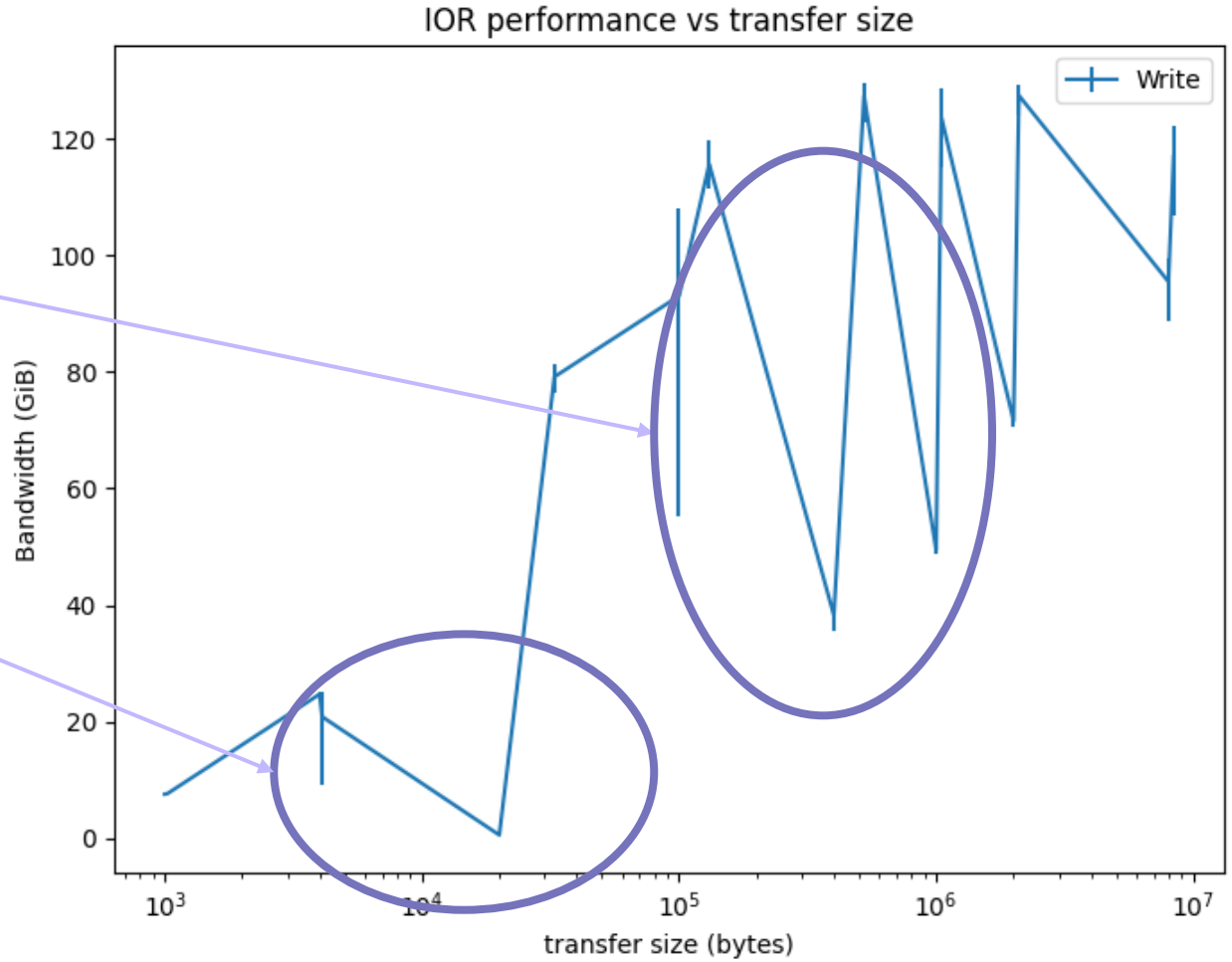
When we think about I/O transformations, we consider the mapping of data between application processes and locations in file

Request Size and I/O Rate

Sawtooth due to “power of 10” vs “power of 2” differences

Small I/O transfers dominated by network performance/ overhead

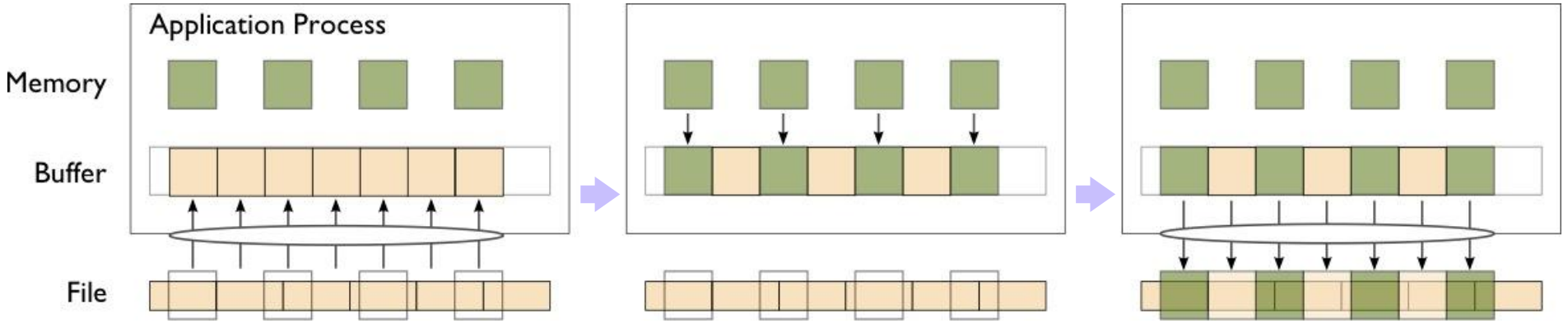
In general, larger requests better.



Tests run on 128 nodes, 16 process per node (2048 processes total) of HPE/Cray/Intel Auroa at Argonne, writing to DAOS

Reducing Number, Increasing Size of Operations

- **Because 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).

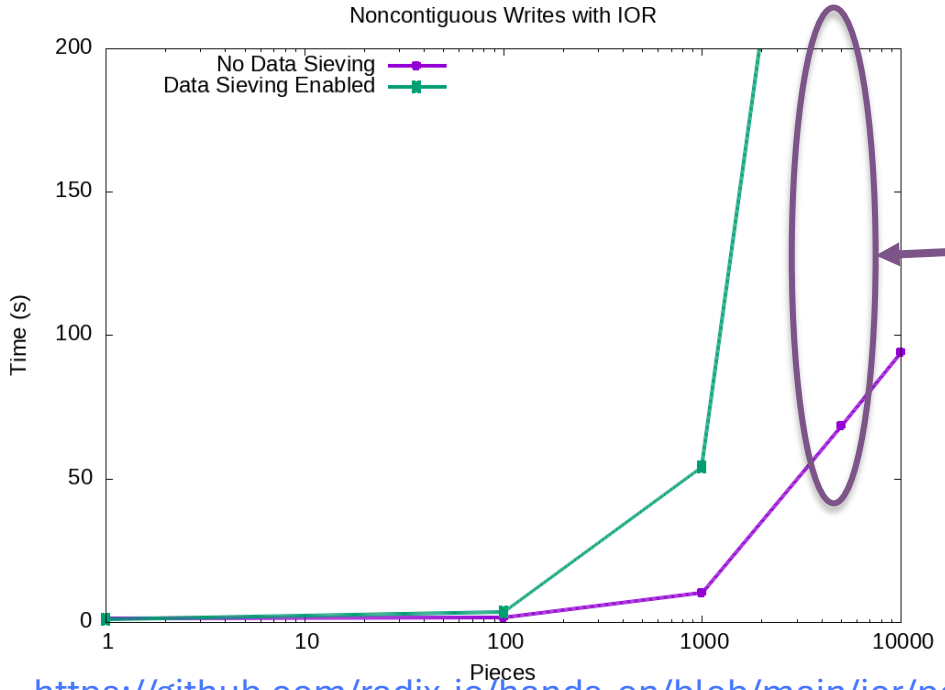
Step 2: Elements to be written to file are replaced in intermediate buffer.

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

Data Sieving in Practice

Not always a win, particularly for writing and this workload:

- IOR benchmark, fixed file size, increasing segments
- Enabling data sieving instead made writes slower: why?
  - Locking to prevent false sharing (not needed for reads)
  - Multiple processes per node writing simultaneously
  - Internal ROMIO buffer too small, resulting in write amplification [1]



[1]

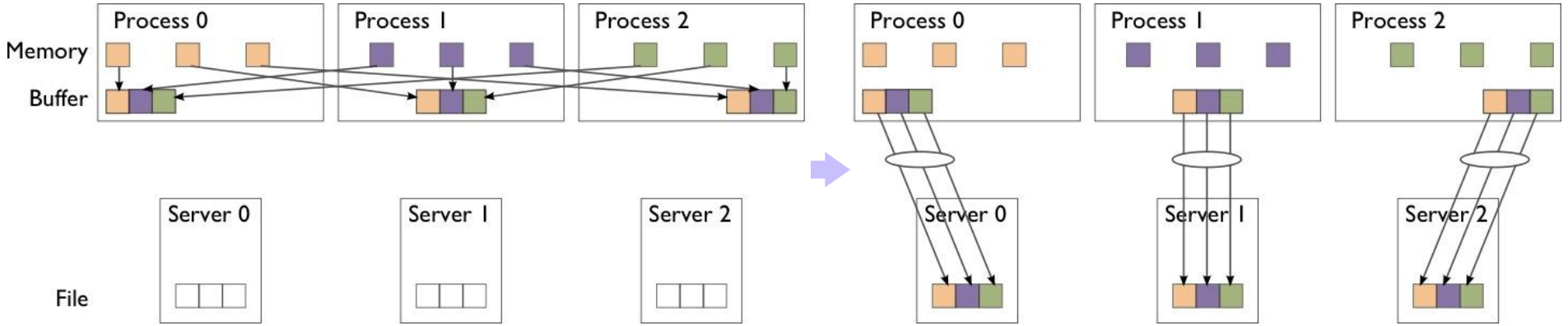
	Naiive	Data Sieving
MPI-IO writes	960	960
MPI-IO Reads	0	0
Posix Writes	4 800 000	4 800 000
Posix Reads	0	4 800 784
MPI-IO bytes written	8.9 GiB	8.9 GiB
MPI-IO bytes read	0	0
Posix bytes read	0	2334 GiB
Posix bytes written	8.9 GiB	2343 GiB
Runtime (sec)	68.8	404.2

Selected Darshan statistics for 5000 segments

https://github.com/radix-io/hands-on/blob/main/ior/polaris/ior-noncontig-segments.sh

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
  - 0th phase determines exchange schedule (not shown)

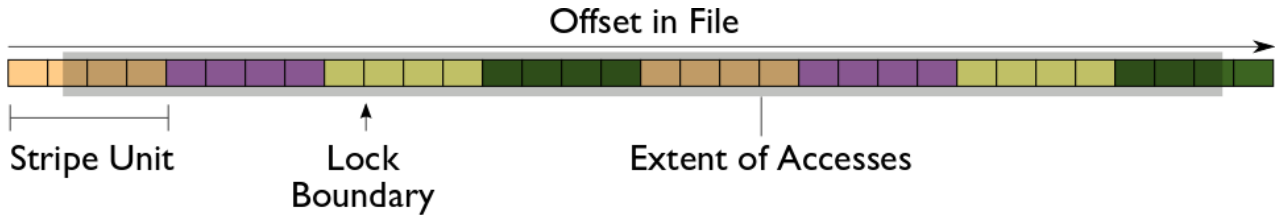


Phase 1: Data are exchanged between processes based on organization of data in file.

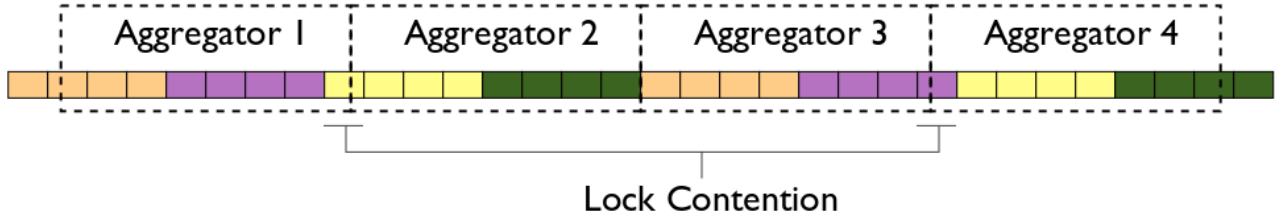
Phase 2: Data are written to file (storage servers) with large writes, no contention.

Two-Phase I/O Algorithms

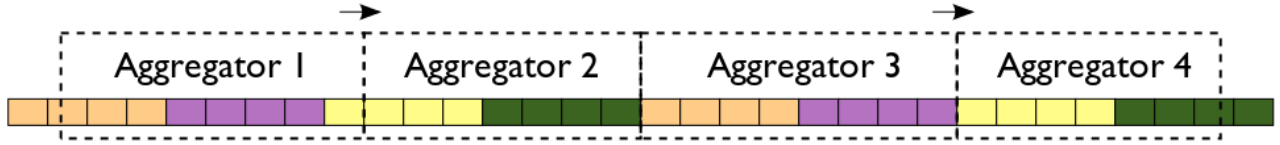
Imagine a collective I/O access using four aggregators to a file striped over four file servers (indicated by colors):



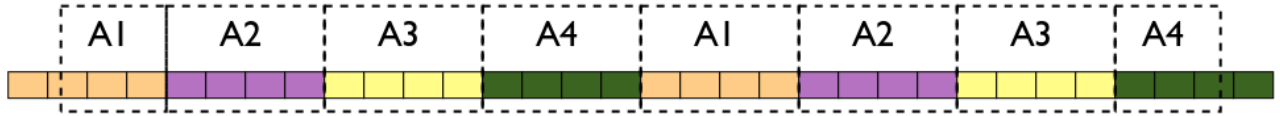
One approach is to evenly divide the region accessed across aggregators.



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).

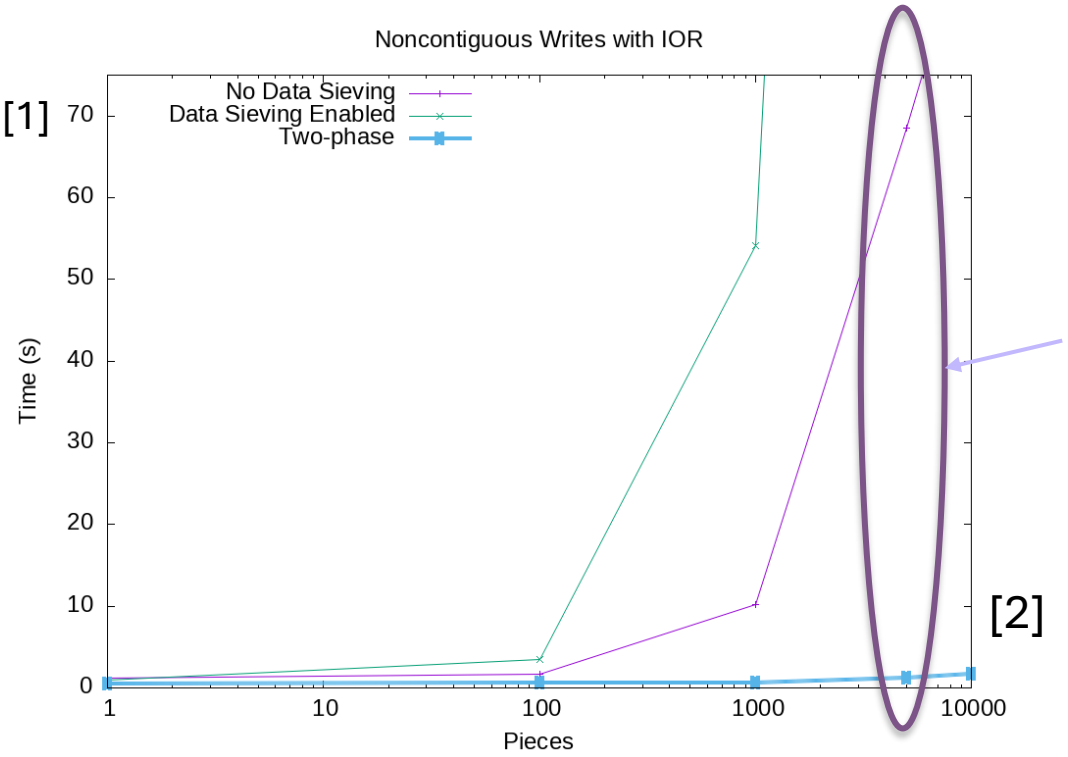


For more information, see W.K. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," SC2008, November 2008.

Two-phase I/O in Practice

- Consistent performance independent of access pattern
  - Note re-scaled y axis [1]
- No write amplification, no read-modify-write
- Some network communication but networks are fast
- Requires “temporal locality” -- not great if writes “skewed”, imbalanced, or some process enter collective late.

[1]



[2]

	Naiive	Data Sieving	Two-phase
MPI-IO writes	960	960	960
MPI-IO Reads	0	0	0
Posix Writes	4 800 000	4 800 000	9156
Posix Reads	0	4 800 784	0
MPI-IO bytes written	8.9 GiB	8.9 GiB	8.9 GiB
MPI-IO bytes read	0	0	0
Posix bytes read	0	2334 GiB	0
Posix bytes written	8.9 GiB	2343 GiB	8.9 GiB
Runtime (sec)	68.8	404.2	1.56

Selected Darshan statistics, 5000 segments

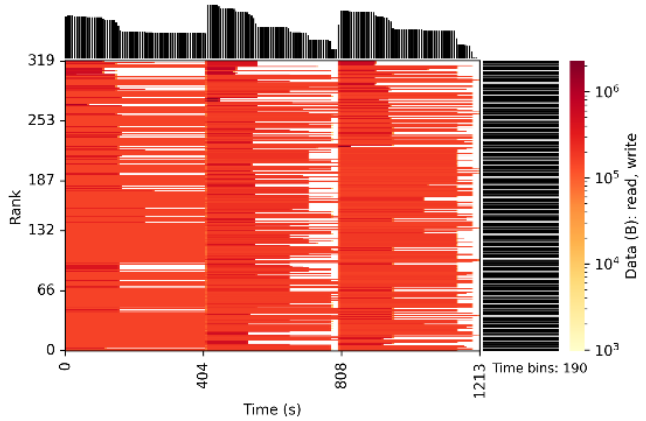
More investigation: Darshan heatmaps

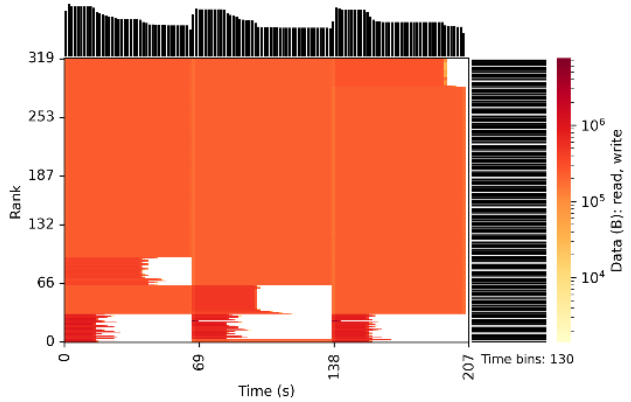
Data sieving

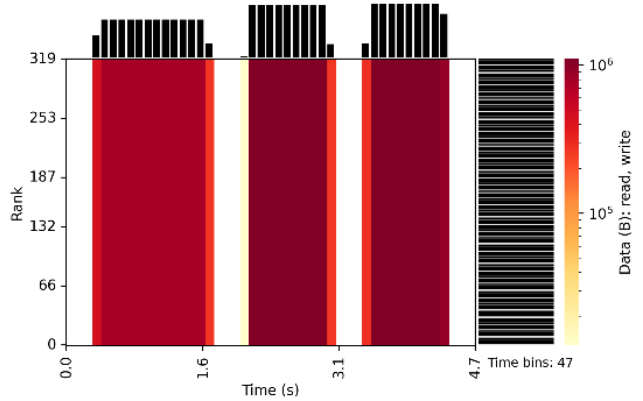
Data sieving disabled

Collective buffering

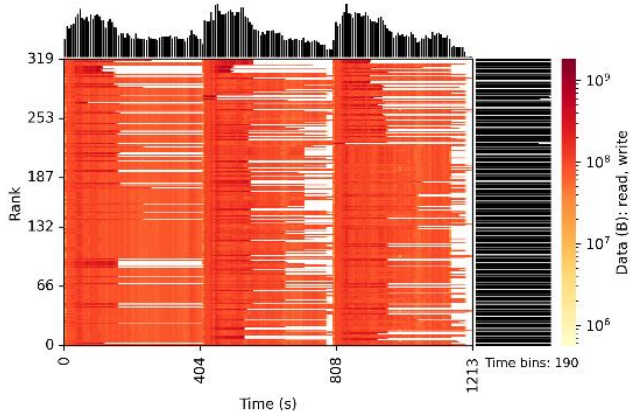
MPI-IO

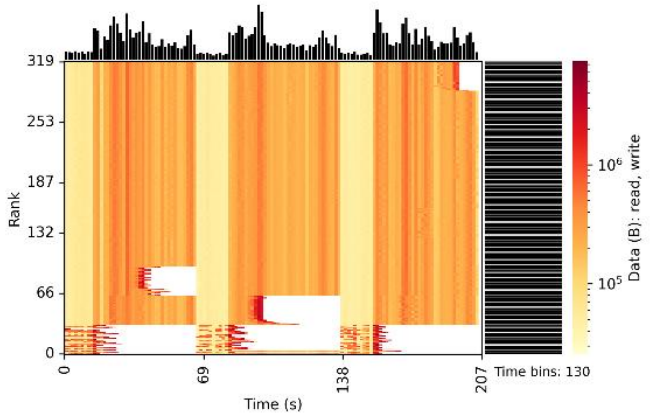


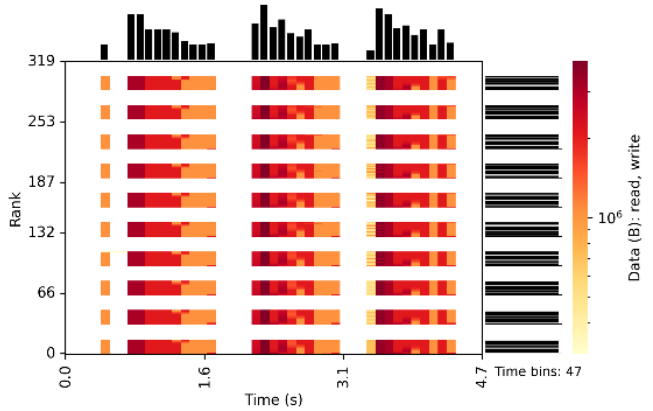




POSIX





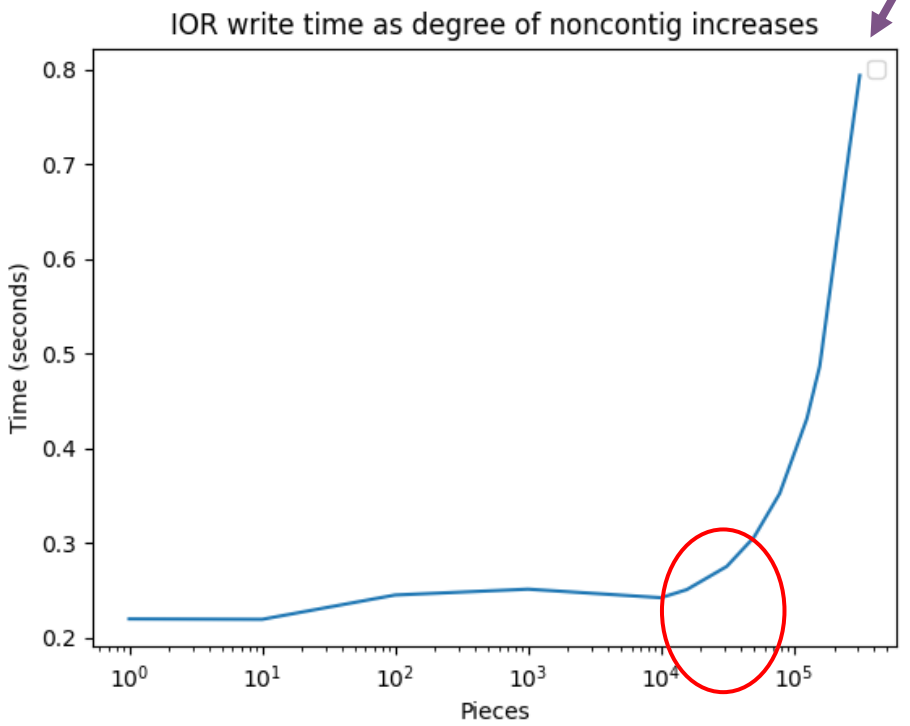


Effect of ROMIO optimizations on IOR benchmark: 5000 non-contiguous segments, three iterations. Note the x axis

Non-contiguous Alternative: scatter-gather (list-io)

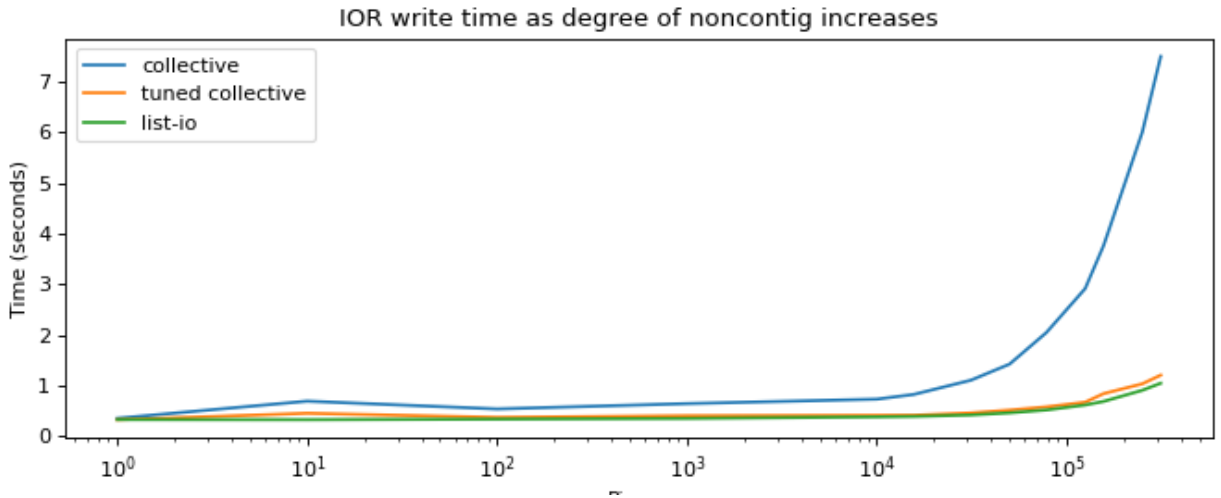
- Same IOR experiment, this time on Aurora's DAOS
- DAOS provides an alternative approach: describe the entire I/O request with a scatter-gather list (d_sg_list_t):
  - `int dfs_write(dfs_t *dfs, dfs_obj_t *obj, d_sg_list_t *sgl, daos_off_t off, daos_event_t *ev);`
- ROMIO driver does this for you
- Curve starts to bend at 50 000 elements:
  - note y axis – still under one second
  - We think due to server side processing of these very long lists
  - Some new optimizations in the pipeline as well

312 500 pieces
32 bytes each



DAOS: Collective I/O vs scatter-gather I/O

- Same IOR experiment but on Aurora this time
  - 2 nodes, 96 processes per node
- List-IO lets us avoid two sources of overhead
  - “rounds” of I/O – no buffering at intermediate aggregator
  - No network exchange of data
- tuned: – asking for more aggregators per node lets us use all 8 network cards
- Since List-IO does not aggregate, could be a problem at larger scale (evaluation “on my list”)
  - Obviously, combining both approaches would be great (that’s “on my list” now too…)

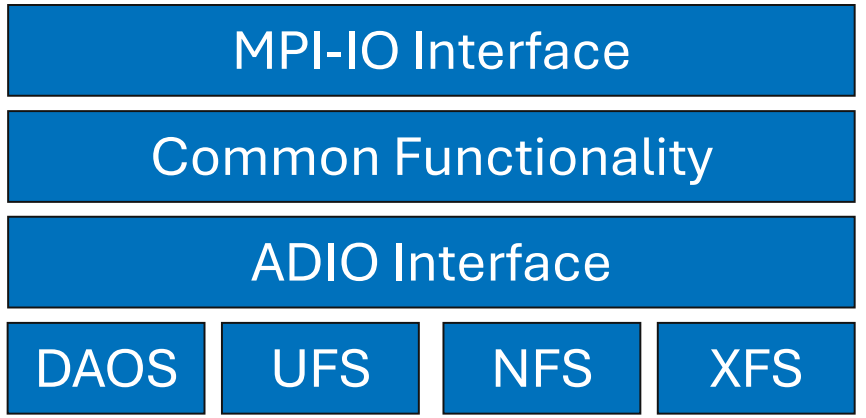


	Two-phase	Tuned Two-phase	List-IO
MPI-IO writes	1152	1152	1152
MPI-IO Reads	0	0	0
DAOS Writes	696	768	1152
DAOS Reads	0	0	0
MPI-IO bytes written	10.7 GiB	10.7 GiB	10.7 GiB
MPI-IO bytes read	0	0	0
DAOS bytes read	0	0	0
DAOS bytes written	10.7 GiB	10.7 GiB	10.7 GiB
Max MPI-IO write time	1.335 sec	0.35 sec	0.22 sec
Max DAOS write time	3.10 msec	3.485 msec	0.22 sec

Selected Darshan statistics, 5000 segments

MPI-IO Implementations

- Different MPI-IO implementations exist
- Today two 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 on GPFS, Lustre, and others
    - Includes data sieving and two-phase optimizations
    - Basis for many vendor MPI implementations
    - https://wordpress.cels.anl.gov/romio/
  - OMPIO from OpenMPI
    - Emphasis on modularity and tighter integration into MPI implementation
    - https://docs.open-mpi.org/en/v5.0.x/faq/ompio.html



ROMIO's layered architecture.

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
  - Object store? File system? Something else? Same interface to all of them
- Also forms solid basis for high-level I/O libraries
  - But they must take advantage of these features!

Understanding I/O Behavior and Performance

Thanks to the following for much of this material

Shane Snyder, Philip Carns, Kevin Harms, Charles Bacon, Sam Lang, Bill Allcock
Math and Computer Science Division and
Argonne Leadership Computing Facility
Argonne National Laboratory

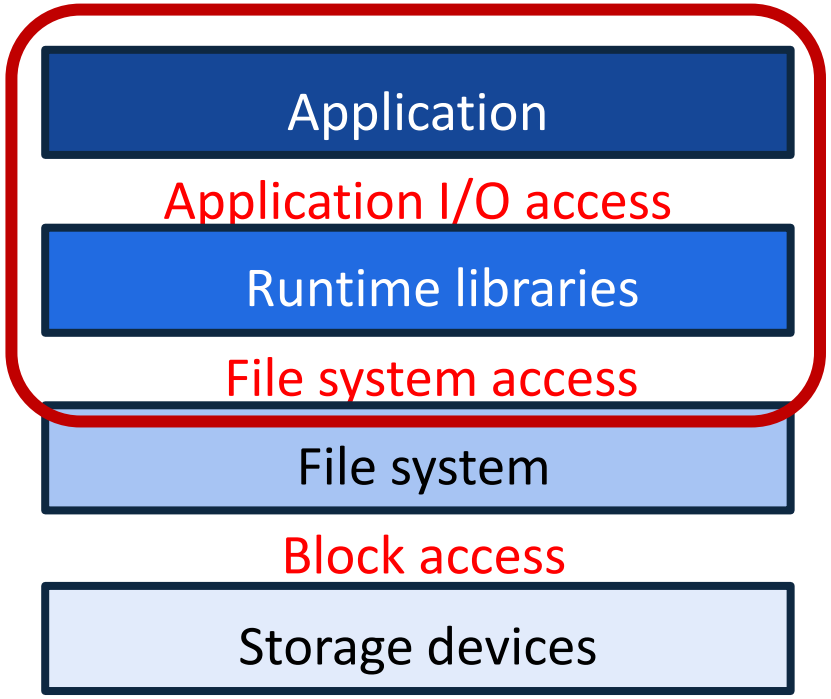
Katie Antypas, Jialin Liu, Quincey Koziol
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory

How is an application using the I/O system? How successful is it at attaining high performance?

Strategy: observe I/O behavior at the application and library level

- What did the application intend to do?
- How much time did it take to do it?
- What can be done to tune and improve?

Simplified HPC I/O stack

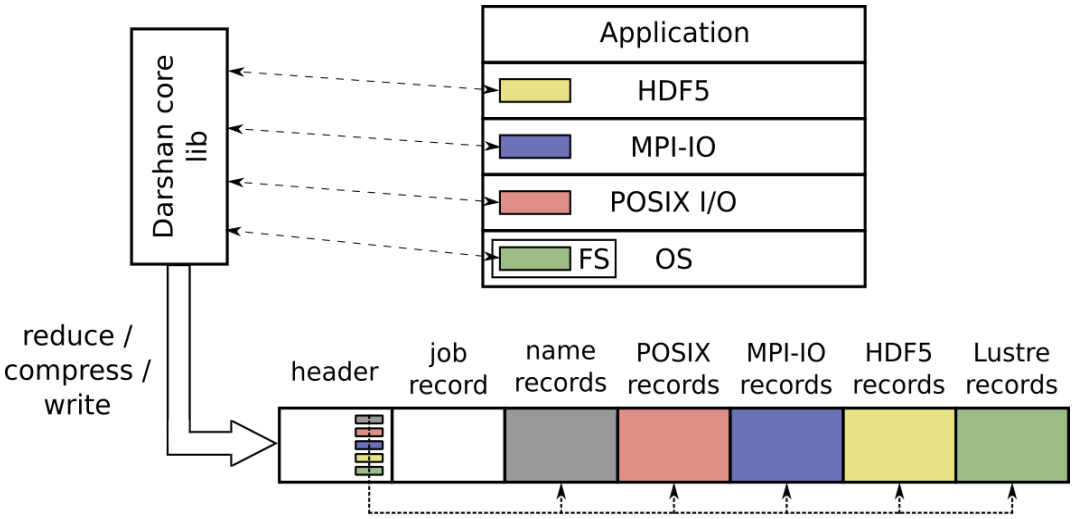


What is Darshan?

- Darshan is a lightweight I/O characterization tool that captures concise views of HPC application I/O behavior
  - Produces a summary of I/O activity for each instrumented job
    - Counters, histograms, timers, & statistics
    - If requested by user, full I/O traces
- Widely available
  - Deployed (and commonly enabled by default) at many HPC facilities around the world
- Easy to use
  - No code changes required to integrate Darshan instrumentation
  - Negligible performance impact; just “leave it on”
- Modular
  - Adding instrumentation for new I/O interfaces or storage components is straightforward

How does Darshan work?

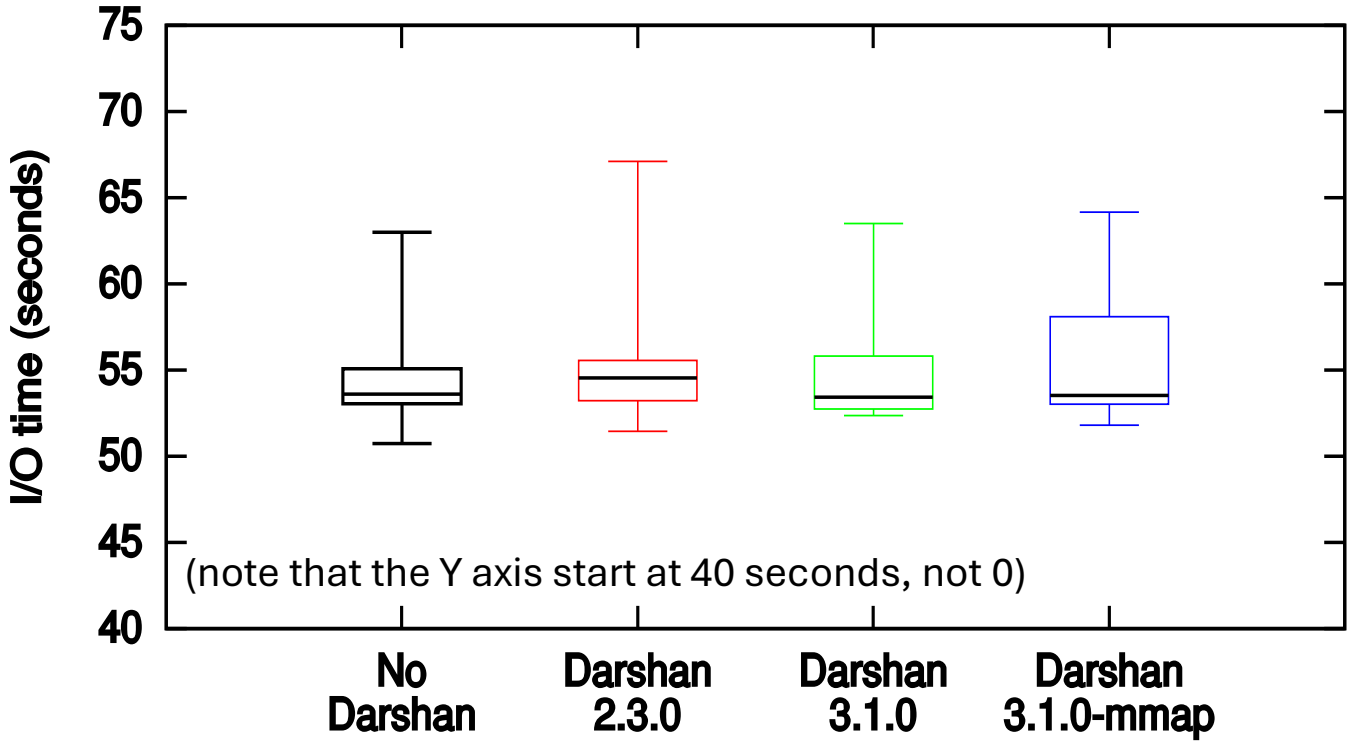
- Darshan records file access statistics independently on each process
- At app shutdown, collect, aggregate, compress, and write log data
- After job completes, analyze Darshan log data
  - darshan-parser - provides complete text-format dump of all counters in a log file
  - PyDarshan - Python analysis module for Darshan logs, including a summary tool for creating HTML reports



- Originally designed for MPI applications, but any dynamically-linked executable can be instrumented
  - In MPI mode, a log is generated for each *app*
  - In non-MPI mode, a log is generated for each *process*

What Is the Overhead of Darshan I/O Function Wrapping?

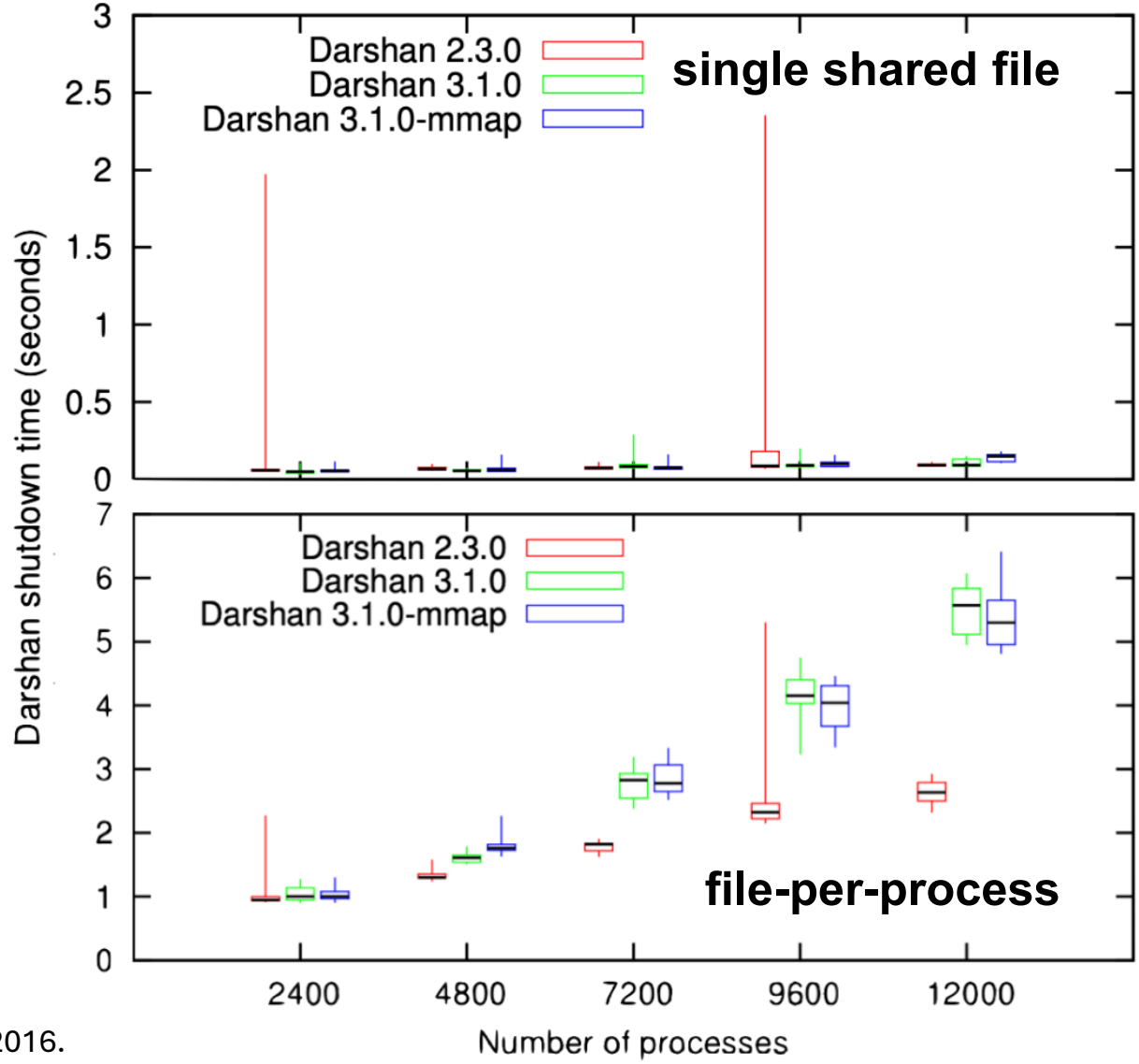
- < 2% increase in median I/O time
  - File-per-process workload
  - 6,000 MPI processes
  - >12 million instrumented calls
- Note use of box plots
  - Ran each test 15 times
  - I/O variation is a reality
  - Performance is a distribution, not a singular value



S. Snyder, et al., "Modular HPC I/O Characterization with Darshan," ESPT 2016. Nov 2016.

Does Shutdown Overhead Impact Walltime?

- Darshan aggregates, compresses, and collectively writes records at MPI_Finalize()
- Its added shutdown time scales well and is negligible
  - Shared-file
    - ~O(1) scaling
    - 100 ms additional overhead
  - File-per-process
    - ~O(N) scaling
    - 5-6 seconds overhead at 12K files



S. Snyder, et al., "Modular HPC I/O Characterization with Darshan," ESPT 2016. Nov 2016.

Analyzing I/O Performance Problems with Darshan

Lightweight nature of Darshan means it can be always on and help both **users** and HPC **operators**

- Users
  - Many I/O problems can be observed from these logs
  - Study applications on demand to debug specific jobs
- Operators
  - Mine logs to catch problems proactively
  - Analyze user behavior, misbehavior, and knowledge gaps

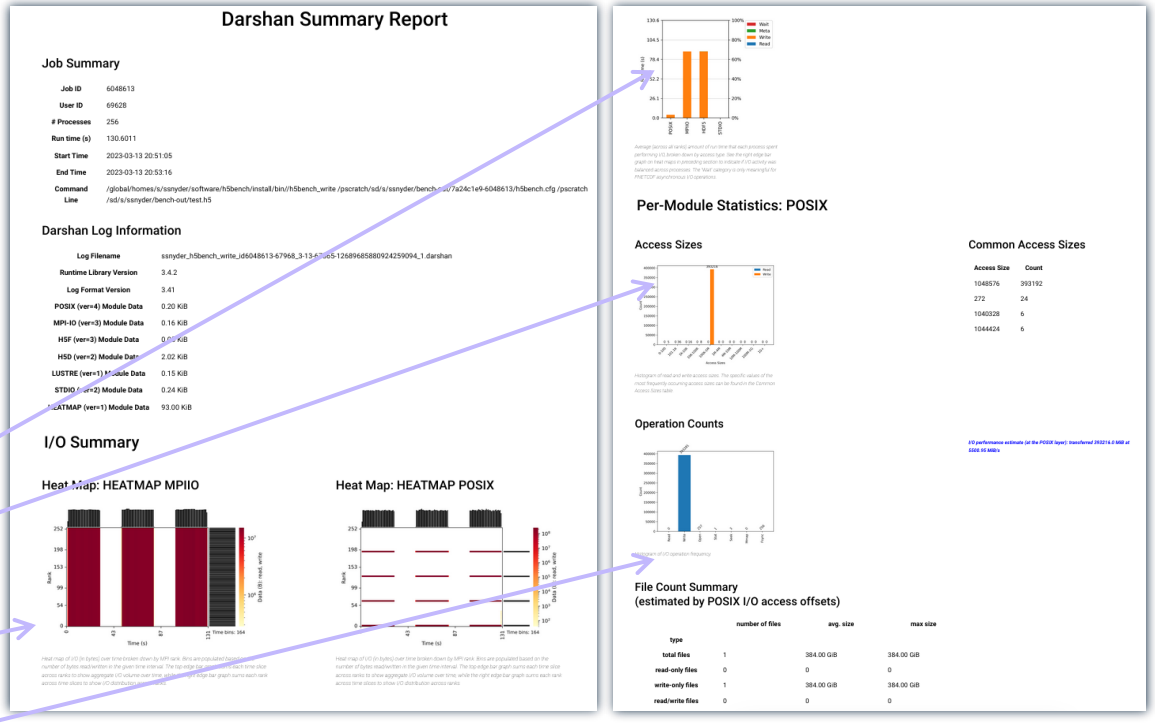
An Example of the Darshan Experience

```
# on the HPC system, load darshan, then compile and run the app
module load darshan
cc -o myapp myapp.c # on a Cray system, for example
srun -n 16 ./myapp

# find the log, then generate a report from it
darshan-config --log-path # figure out where your log went
python -m darshan summary /path/to/2024/7/16/glock_myapp_...darshan

# on your local laptop, copy the report and open it in a browser
scp mysupercomputer:/path/to/report.html .
open report.html
```

See https://tinyurl.com/darshan-tut for a more complete tutorial!

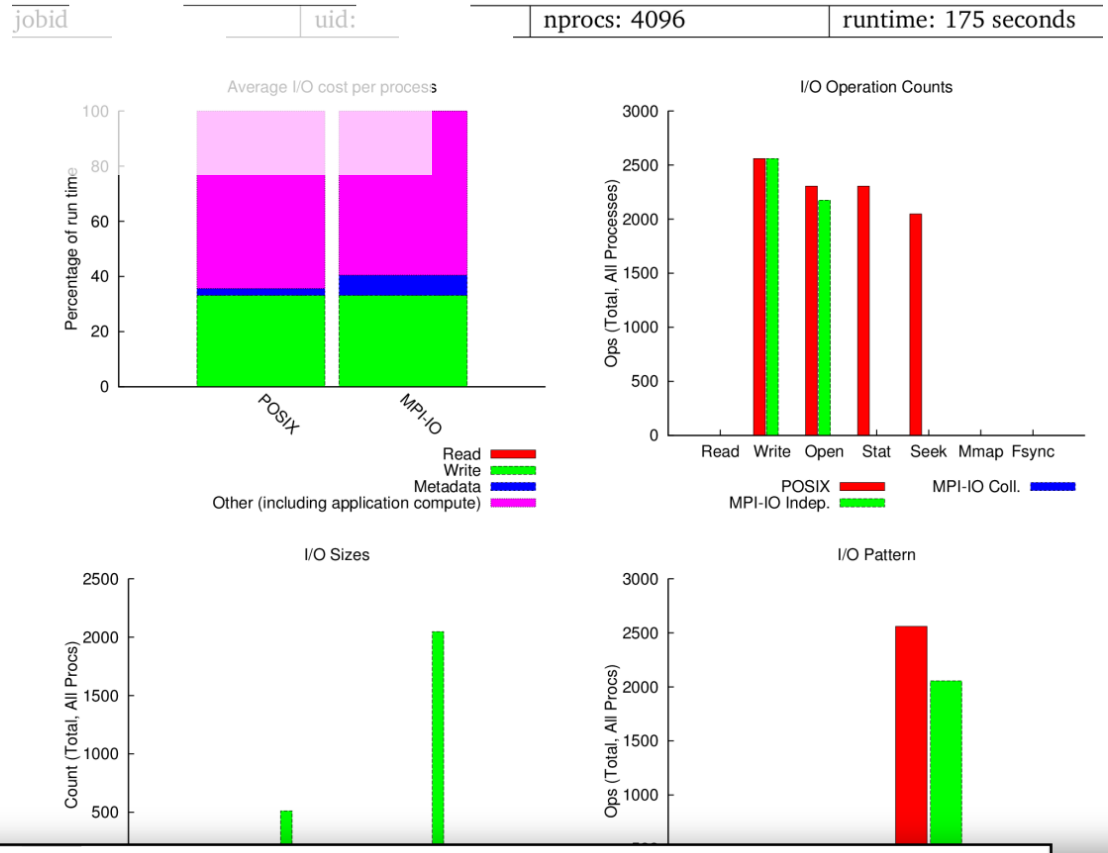
Job-Level Performance Analysis

- Darshan provides insight into the I/O behavior and performance of a job
- **Darshan Summary Report:** creates a web page summarizing various aspects of I/O performance
  - Percent of runtime spent in I/O
  - Access size histogram
  - “Heatmap”
  - Operation counts



Example: Checking User Expectations

- App opens 129 files (one “control” file, 128 data files)
- User *expected* one ~40 KiB header per data file
- Darshan showed 512 headers being written
- Code bug: header was written 4× per file



Most Common Access Sizes

access size	count
67108864	2048
41120	512
8	4
4	3

File Count Summary

type	number of files	avg. size	max size
total opened	129	1017M	1.1G
read-only files	0	0	0
write-only files	129	1017M	1.1G
read/write files	0	0	0
created files	129	1017M	1.1G

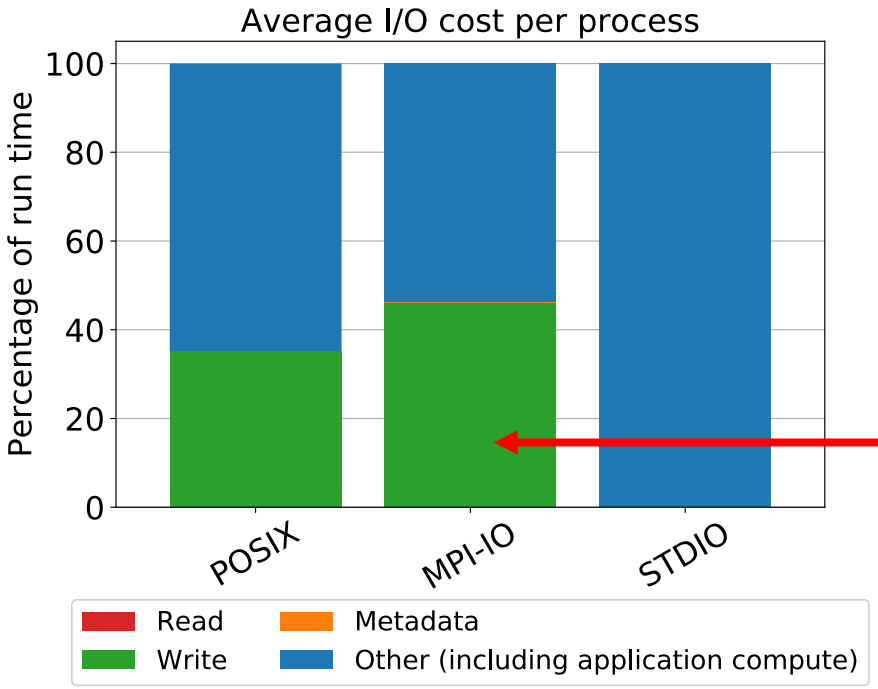
Example: When Doing the Right Thing Goes Wrong

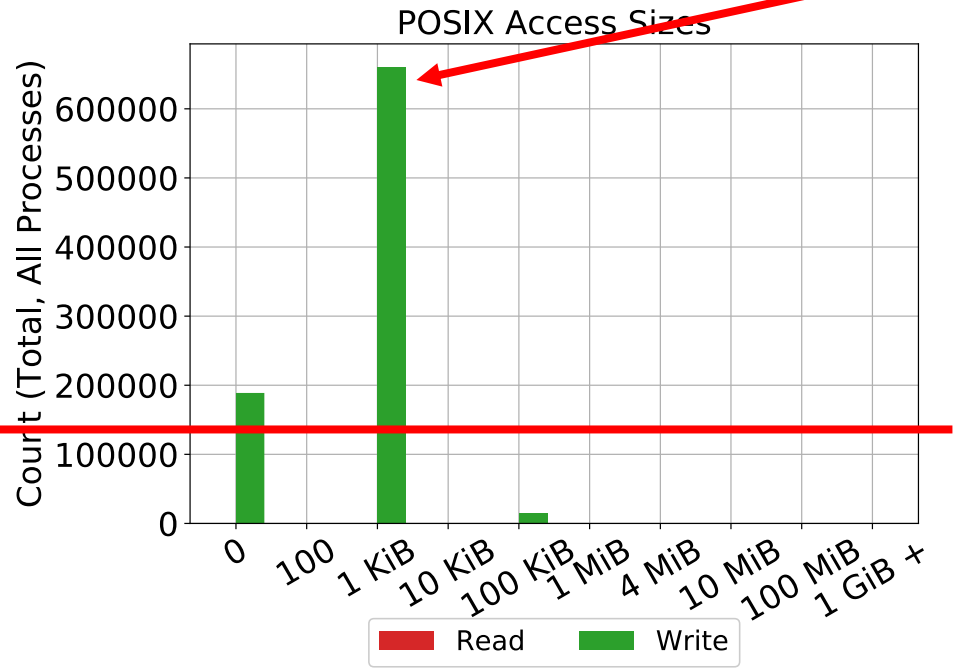
- Large-scale astrophysics application using MPI-IO
- Used 94,000,000 CPU hours at NERSC since 2015

jobid: 1950915785	uid: 81587417	nprocs: 4720	runtime: 1005 seconds

I/O performance *estimate* (at the MPI-IO layer): transferred 17443.1 MiB at 26.35 MiB/s
I/O performance *estimate* (at the STDIO layer): transferred 0.1 MiB at 1673.25 MiB/s

Wrote 17 GiB using 4 KiB transfers





40% of walltime spent doing I/O *even with MPI-IO*

Case study courtesy Jialin Liu and Quincey Koziol (NERSC)

Example: When Doing the Right Thing Goes Wrong

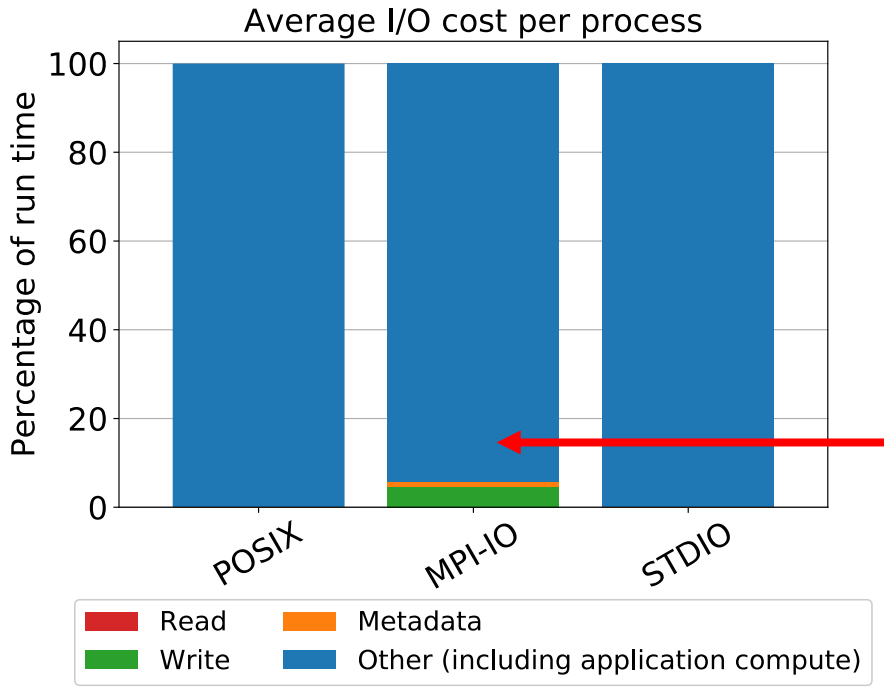
- Collective I/O was being disabled by middleware due to type mismatch in app code
- After type mismatch bug fixed, collective I/O gave 40× speedup

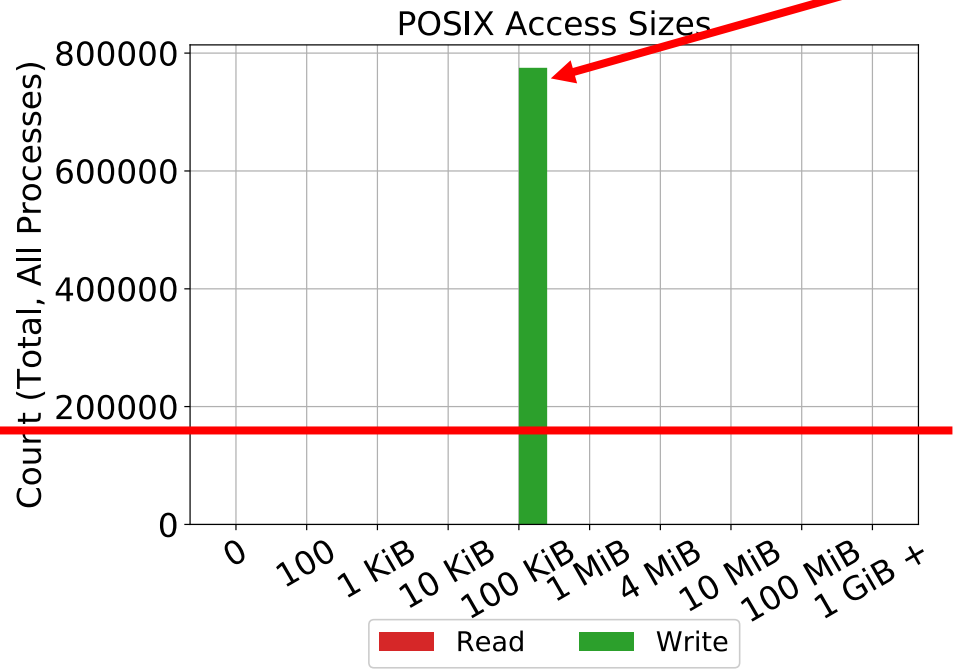
jobid: 2308034461	uid: 81587417	nprocs: 77312	runtime: 11680 seconds

I/O performance *estimate* (at the MPI-IO layer): transferred 774986.3 MiB at 1143.95 MiB/s

I/O performance *estimate* (at the STDIO layer): transferred 12298.7 MiB at 608.59 MiB/s

Wrote 756 GiB in 1 MiB transfers (>40× speedup)





6% of walltime spent doing I/O

Example: Redundant Read Traffic

- Applications sometimes read more bytes from a file than the file's size
  - Can cause disruptive I/O network traffic and storage contention
  - Good candidate for aggregation, collective I/O, or burst buffering
- Common pattern in emerging AI/ML workloads
- Example:
  - Scale: 6,138 processes
  - Run time: 6.5 hours
  - Avg. I/O time per proc: 27 minutes
- 1.3 TiB of file data
- 500+ TiB read!

File Count Summary
(estimated by I/O access offsets)

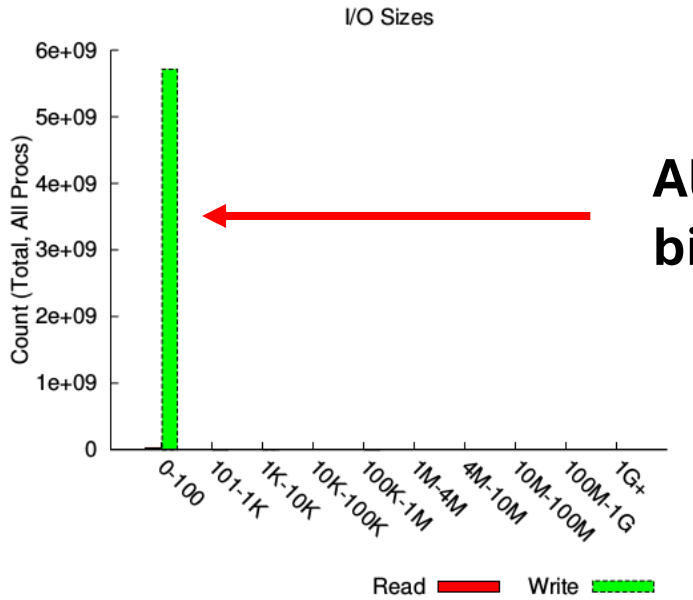
type	number of files	avg. size	max size
total opened	1299	1.1G	8.0G
read-only files	1187	1.1G	8.0G
write-only files	112	418M	2.6G
read/write files	0	0	0
created files	112	418M	2.6G

Data Transfer Per Filesystem

File System	Write		Read	
	MiB	Ratio	MiB	Ratio
/	47161.47354	1.00000	575224145.24837	1.00000

Example: Small Writes to Shared Files

- Scenario: Small writes can contribute to poor performance
  - Particularly when writing to shared files
  - Candidates for collective I/O or batching/buffering of write operations
- Example:
  - Issued 5.7 billion writes to shared files, each less than 100 bytes in size
  - Averaged just over 1 MiB/s per process during shared write phase



Almost 6 billion I/O!

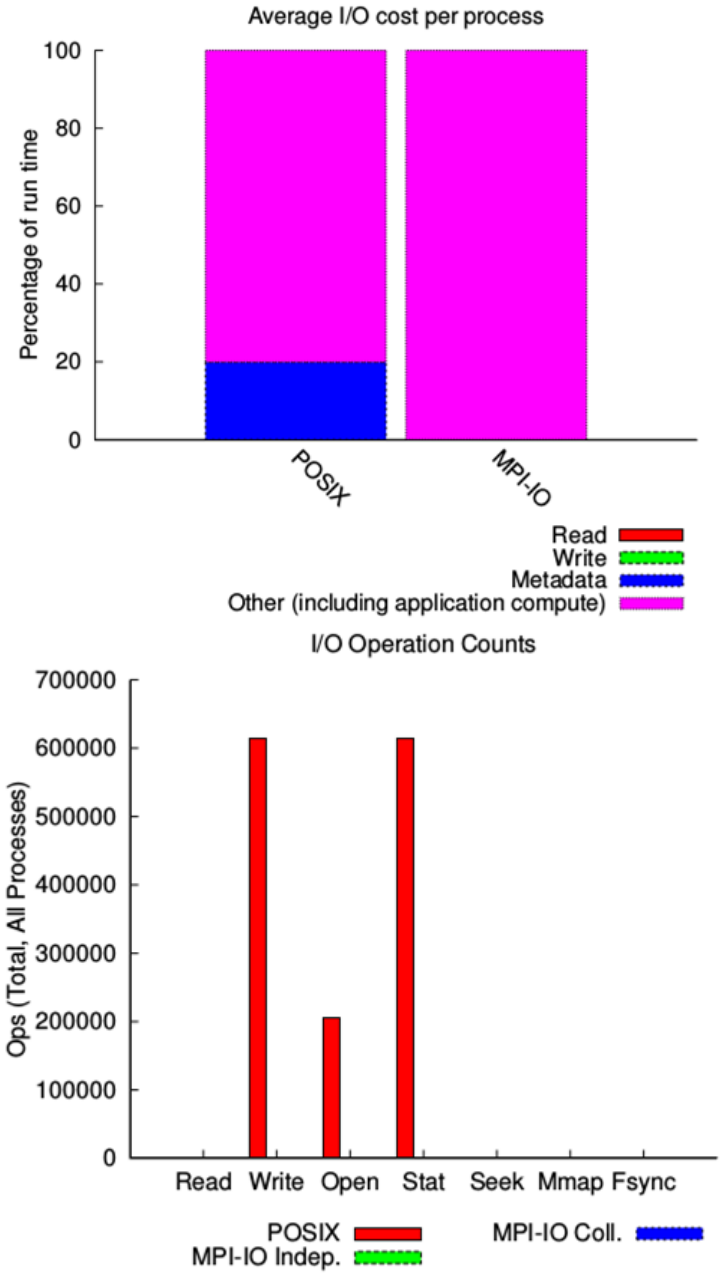
Most Common Access Sizes

access size	count
1	3418409696
15	2275400442
24	42289948
12	14725053

1-byte accesses!

Example: Excessive Metadata Overhead

- Scenario: Very high percentage of I/O time spent performing metadata operations (open, close, stat, seek)
  - Close() cost can be misleading due to write-behind cache flushing!
  - Candidates for coalescing files and eliminating extra metadata calls
- Example:
  - Scale: 40,960 processes, > 20% time spent in I/O
  - 99% of I/O time in metadata operations
  - Generated 200,000+ files with 600,000+ write and 600,000+ stat calls



Protip: Stat Is Not Cheap on Parallel File Systems

- Stat() requires a consistent size calculation for the file
  - Store a pre-calculated size on the metadata server(s)
    - IBM Spectrum Scale (GPFS)
    - Lustre with Lazy Size-on-MDT feature
    - BeeGFS
  - Calculate size on demand, triggering broadcast/incasts between metadata server and data servers
    - DAOS DFS
    - Lustre default behavior
- No present-day PFS implementations respond well when thousands of processes stat() the same file at once

System-Level Performance Analysis

- "Always-on" nature of Darshan enables system-wide I/O analysis
- Daily Top 10 I/O Users list at NERSC to identify users…
  - Running jobs in their home directory
  - Who might benefit from the burst buffer
- Can develop heuristics to detect anomalous I/O behavior
  - Highlight jobs spending a lot of time in metadata
  - Automated triggering/alerting

```
 #         Users  Read(GiB) Write(GiB)   # Jobs
==============================================
 1.         john     9727.3    10192.7    16432
 2.         mary     3672.1     3662.1      701
 3.         jane     6777.8      155.6        2

 #  File Systems  Read(GiB) Write(GiB)   # Jobs
==============================================
 1.     cscratch    18978.4    16940.1     4026
 2.        homes    10122.0    10692.7    16565
 3.    bb-shared      233.9        8.0        1

 #                 Applications  Read(GiB) Write(GiB)   # Jo
=============================================================
 1.                    vasp_std    10078.2    10528.6    168
 2.                        pw.x     3672.1     3662.1      7
 3.                    lmp_cori     6699.4        0.0

 #                  User/App/FS  Read(GiB) Write(GiB)   # Jo
=============================================================
 1.          john/vasp_std/homes     9727.3    10192.7    164
 2.           mary/pw.x/cscratch     3672.1     3662.1      7
 3.       jane/lmp_cori/cscratch     6699.4        0.0
```

Lockwood et al., "TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis," in *Proceedings of the Cray User Group* (CUG'18), 2018.

Available Darshan Analysis Tools

- Officially supported tools
  - **pydarshan**: python -m darshan …
    - summary: Creates HTML with graphs for initial analysis
    - to_json: Dump log to json format for building your own analysis
  - **darshan-parser**: Dumps all information into text format
- Documentation:
  - **PyDarshan**: https://www.mcs.anl.gov/research/projects/darshan/docs/pydarshan/
  - **darshan-util**: https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-util.html
- Third-party tools
  - **DXT Explorer**: Interactive tool to explore Darshan DXT traces https://github.com/hpc-io/dxt-explorer

I/O Understanding Takeaway

- Scalable tools (like Darshan) can yield useful insight
  - Identify characteristics that make applications successful (and those that cause problems)
  - Identify problems to address through I/O research
- Performance tools for scale require special considerations
  - Naïve approaches like strace are unuseable at scale
  - Target the problem domain carefully to minimize data and overhead
- For more information, see: https://www.mcs.anl.gov/research/projects/darshan
- Explore a collection of interesting Darshan logs: https://github.com/darshan-hpc/darshan-logs

The Parallel netCDF Interface and File Format

Thanks to Wei-Keng Liao, Alok Choudhary, Kaiyun Hou, and Kui Gao(NWU) for their help in the development of PnetCDF.

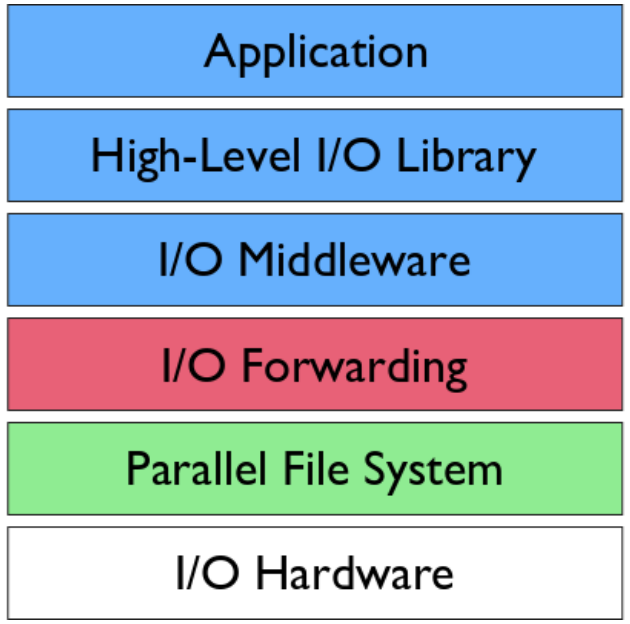
I/O for Computational Science

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 accessing the parallel file system directly. Reduces or (ideally) eliminates need for optimization in application codes.

Higher Level I/O Interfaces

- Provide structure to files
  - Well-defined, portable formats
  - Self-describing
  - Organization of data in file
  - Interfaces for discovering contents

- Present APIs are more appropriate for computational science
  - Typed data
  - Noncontiguous regions in memory and file
  - Multidimensional arrays and I/O on subsets of these arrays

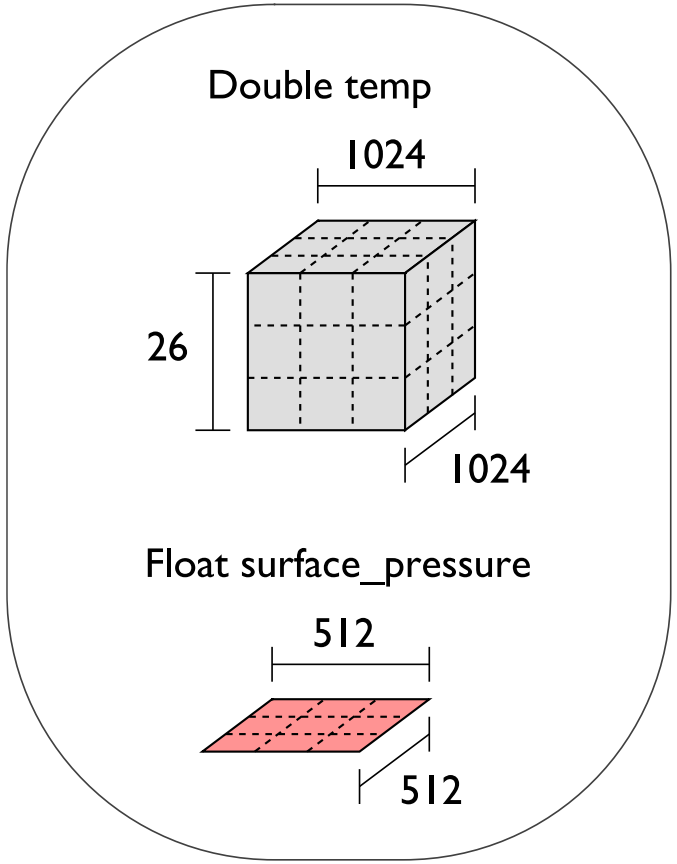
- Both of our example interfaces are implemented on top of MPI-IO

Parallel NetCDF (PnetCDF)

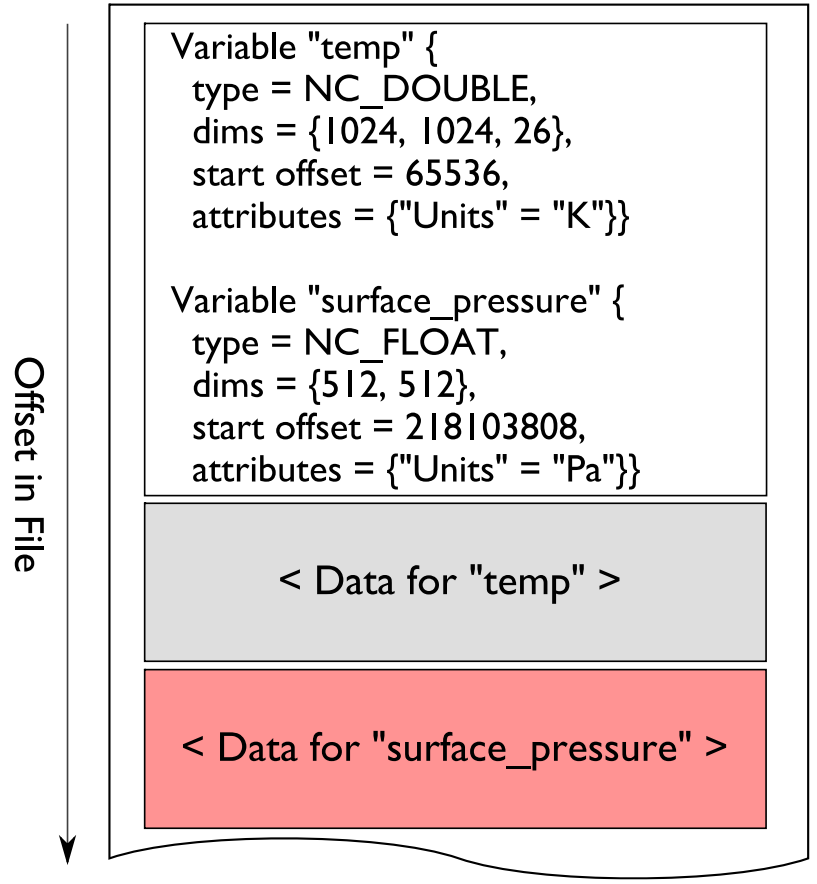
- Based on original “Network Common Data Format” (netCDF) work from Unidata
  - Derived from their source code
- Data Model:
  - Collection of variables in single file
  - Typed, multidimensional array variables
  - Attributes on file and variables
- Features:
  - C, Fortran, and F90 interfaces
  - Portable 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:
  - https://parallel-netcdf.github.io/wiki/QuickTutorial.html

Data Layout in netCDF Files

Application Data Structures



netCDF File "checkpoint07.nc"

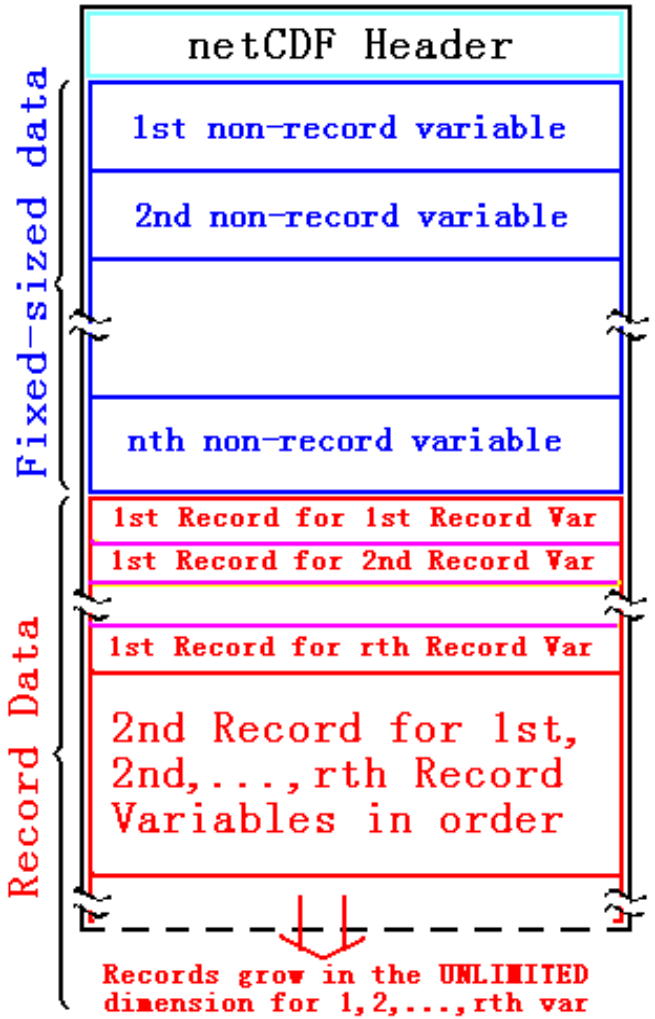


netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

Data for variables is stored in contiguous blocks, encoded in a portable binary format according to the variable's type.

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



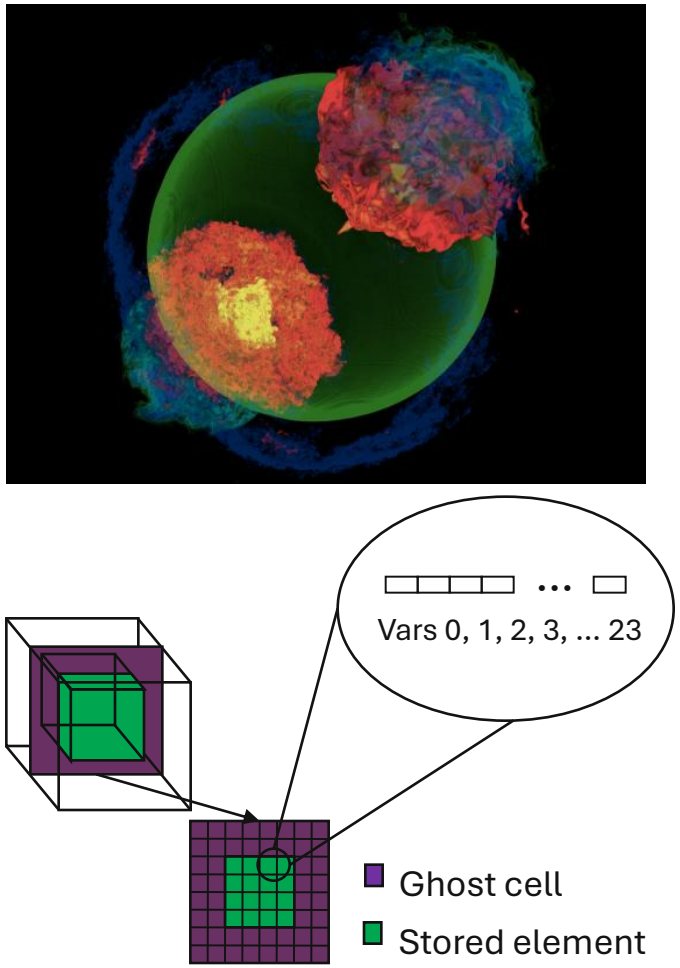
Storing Data in PnetCDF

- 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 Astrophysics

- FLASH is an astrophysics code for studying events such as supernovae
  - Adaptive-mesh hydrodynamics
  - Scales to 1000s of processors
  - MPI for communication

- Frequent checkpoints:
  - Large blocks of typed variables from all processes
  - Portable format
  - Canonical ordering (different than in memory)
  - Skipping ghost cells

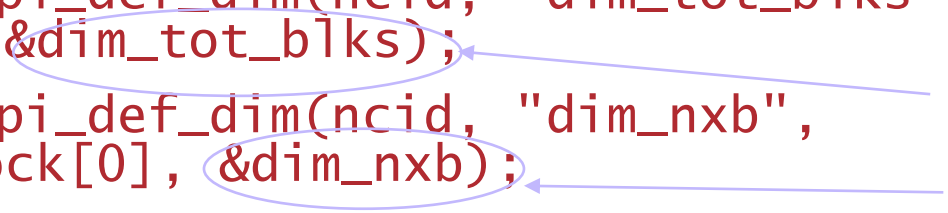


Example: FLASH with PnetCDF

- FLASH AMR structures do not map directly to netCDF multidimensional arrays
- Must create mapping of the in-memory FLASH data structures into a representation in netCDF multidimensional arrays
- Chose to:
  - Place all checkpoint data in a single file
  - Impose a linear ordering on the AMR blocks
    - Use 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_labe][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 memory datatype “mpi_type” describing location,
     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

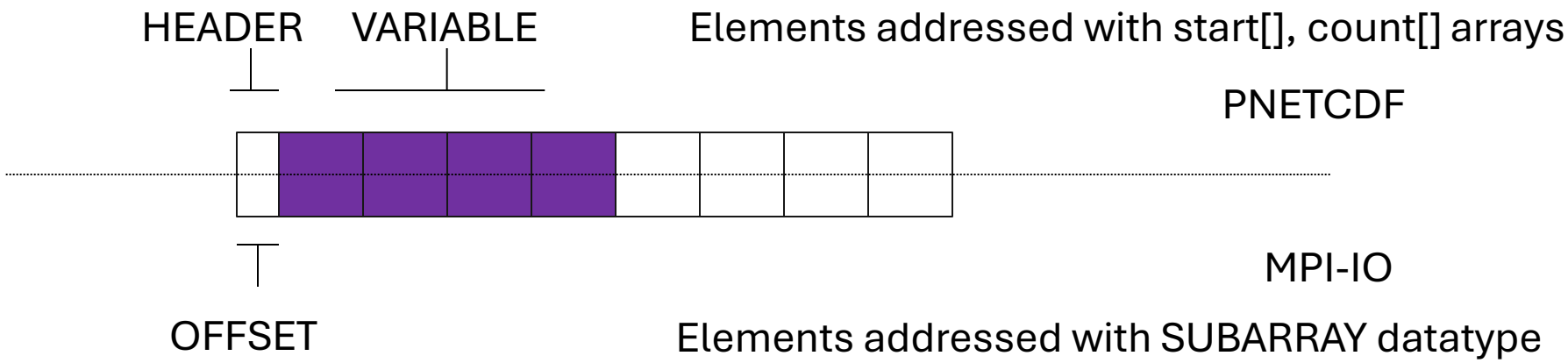
- At ncmpi_enddef
  - Process 0 writes header with `MPI_File_write_at`
  - `MPI_Bcast` result to others
  - Everyone has header data in memory, understands placement of all variables
    - No need for any additional header I/O during data mode!

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
  - Possibly applies two-phase when writing variables
- MPI-IO makes PFS calls
  - PFS client code communicates with servers and stores data

Parallel-NetCDF and MPI-IO

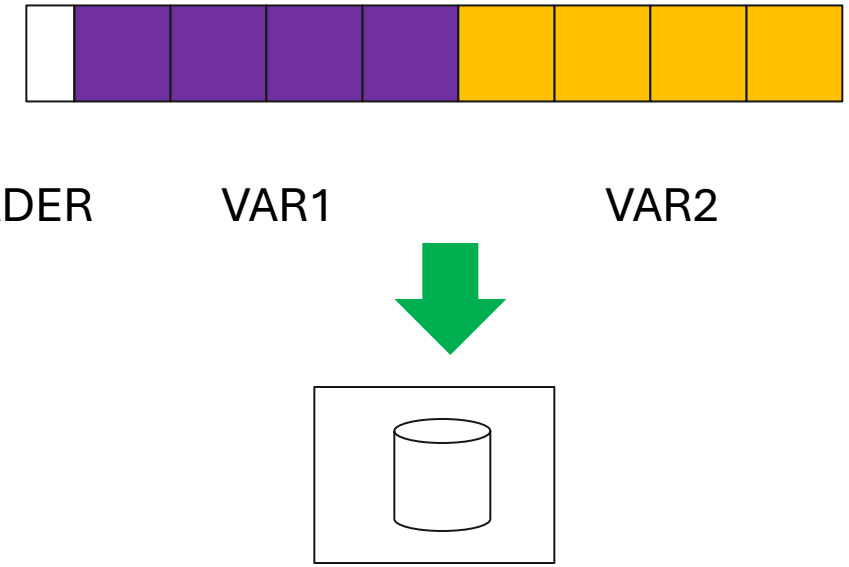


- `ncmpi_put_vara_all` describes access in terms of arrays, elements of arrays
  - For example, “Give me a 3x3 subcube of this larger 1024x1024 array”
- Library translates into MPI-IO calls
  - `MPI_Type_create_subarray`
  - `MPI_File_set_view`
  - `MPI_File_write_all`

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);
```

- 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



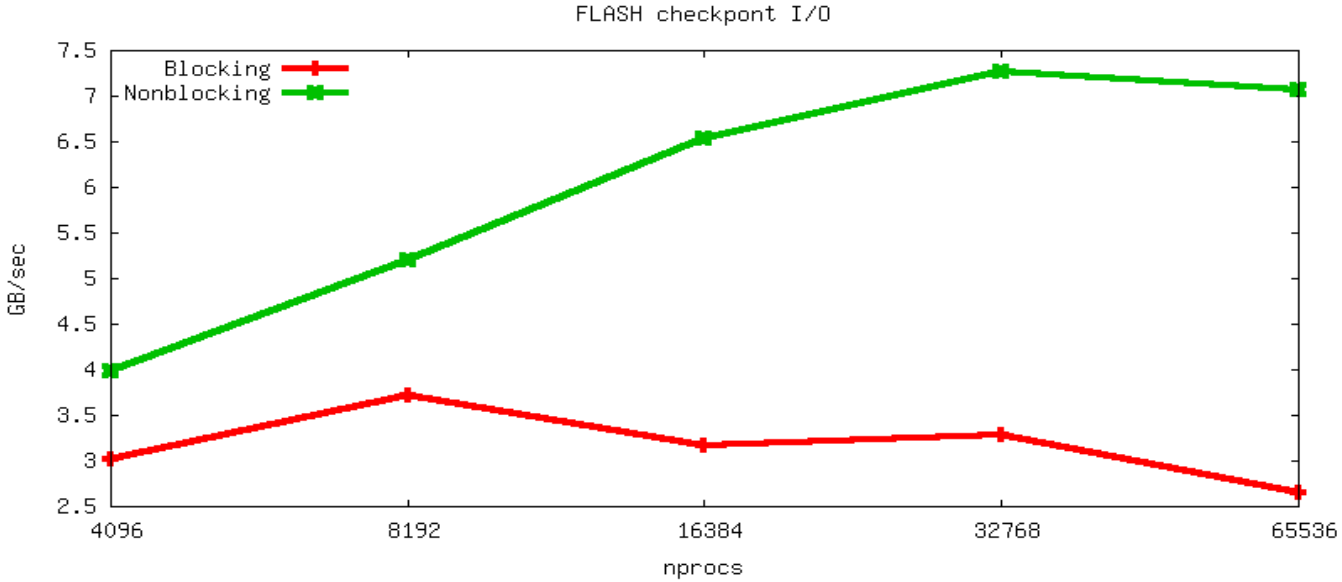
	Separate	Combined
POSIX writes	161	2
POSIX reads	0	1
MPI-IO indep writes	1	1
MPI-IO coll. writes	160	16

Selected Darshan stats from 16 processes each writing 10 variables to a dataset: Note effects of data sieving and deciding against collective I/O

https://github.com/radix-io/hands-on/blob/main/array/solutions/array-pnetcdf-op-combine-compare.c

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, good performance is likely to result
  - 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)

Pnetcdf and Object Stores

- Actually not much to say here…
- MPI-IO layer responsible for transport, optimizations

The HDF5 Interface and File Format

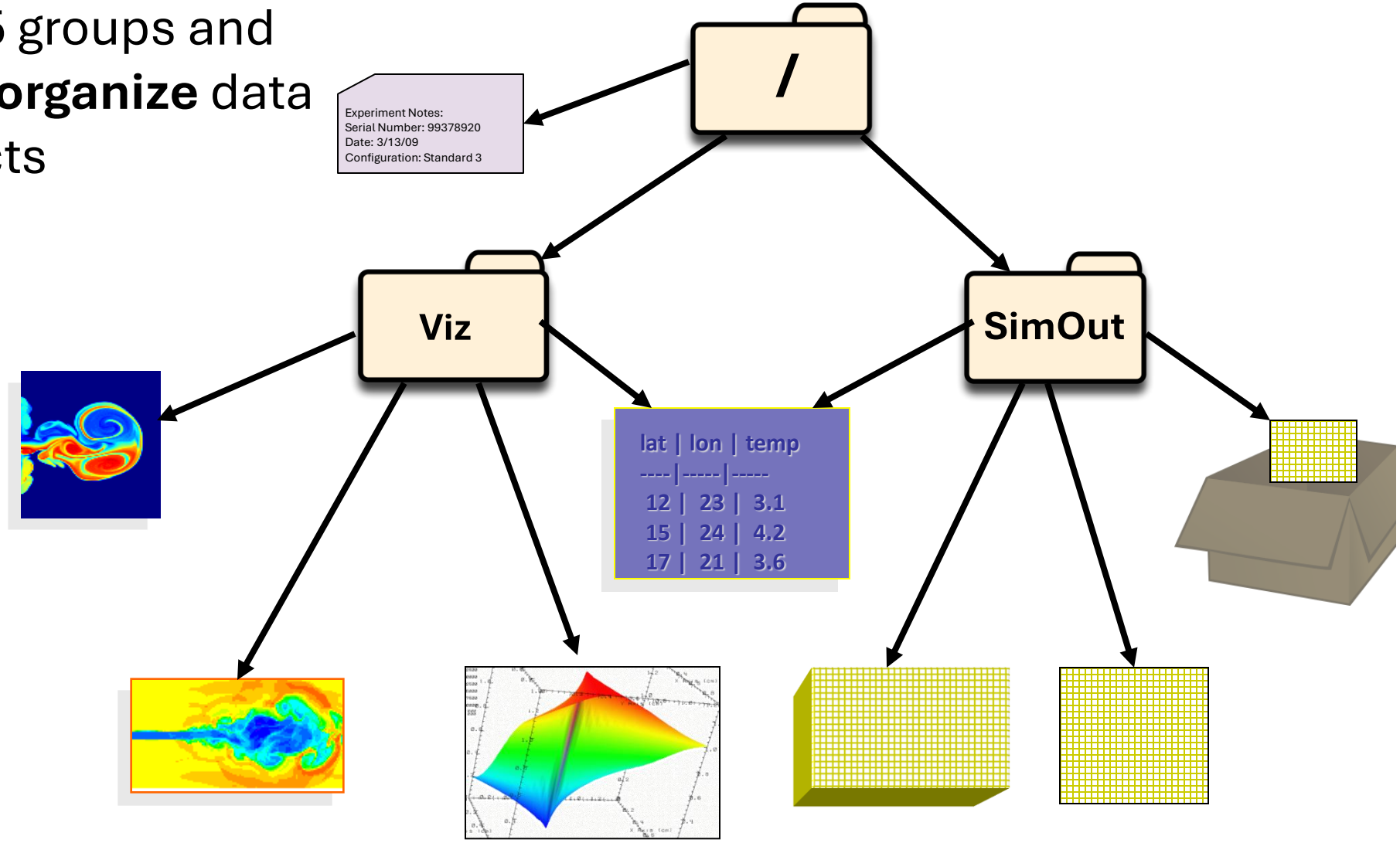
Thanks to our friends at The HDF Group for materials and feedback!

HDF5

- Hierarchical Data Format, from The HDF Group (formerly of NCSA)
  - https://www.hdfgroup.org/
- Data Model:
  - Hierarchical data organization in single file
  - Typed, multidimensional array storage
  - Attributes on any HDF5 "object" (dataset, data, groups)
- Features:
  - C, C++, Fortran, Java (JNI) interfaces
    - Community-supported Python, Lua, R
  - Portable data format
  - Optional compression (even in parallel I/O mode)
  - Chunking: efficient row or column oriented access
  - Noncontiguous I/O (memory and file) with hyperslabs
- Parallel HDF5 tutorial:
  - https://portal.hdfgroup.org/display/HDF5/Introduction+to+Parallel+HDF5

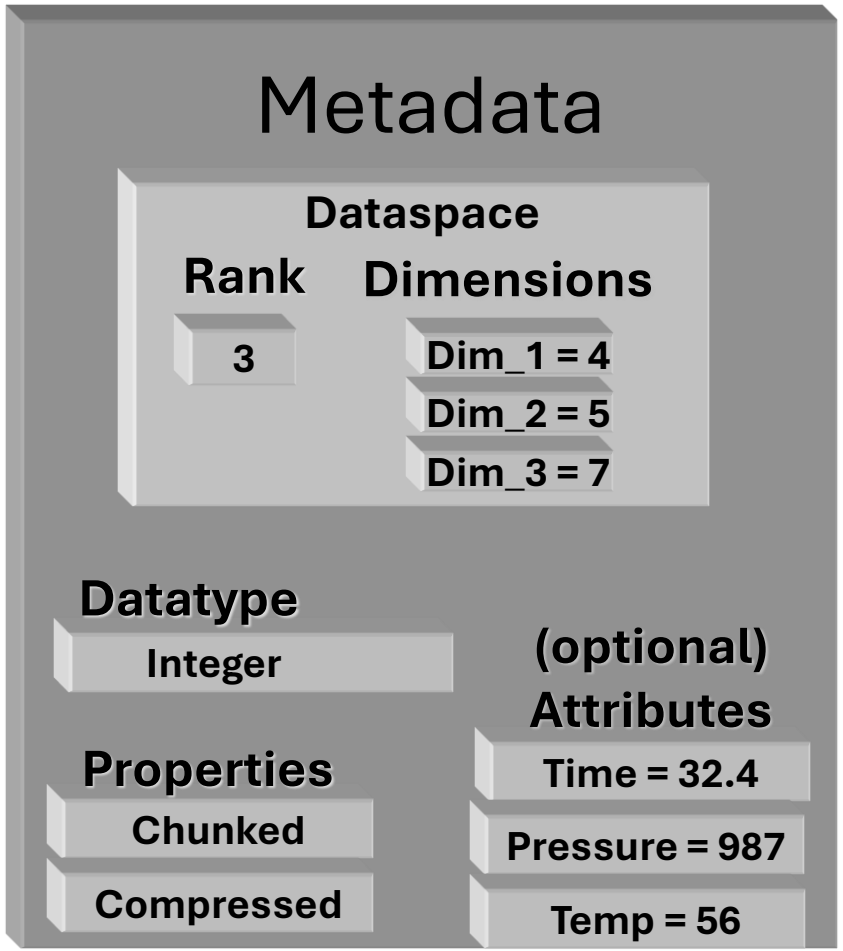
HDF5 Groups and Links

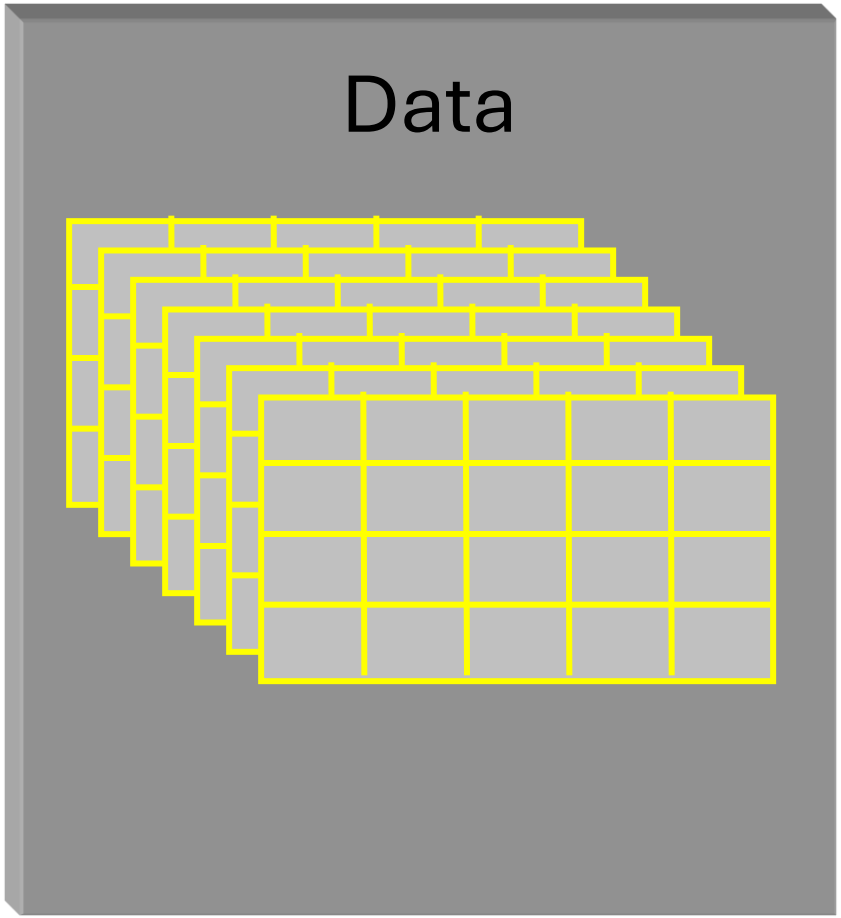
HDF5 groups and links **organize** data objects



The HDF Group

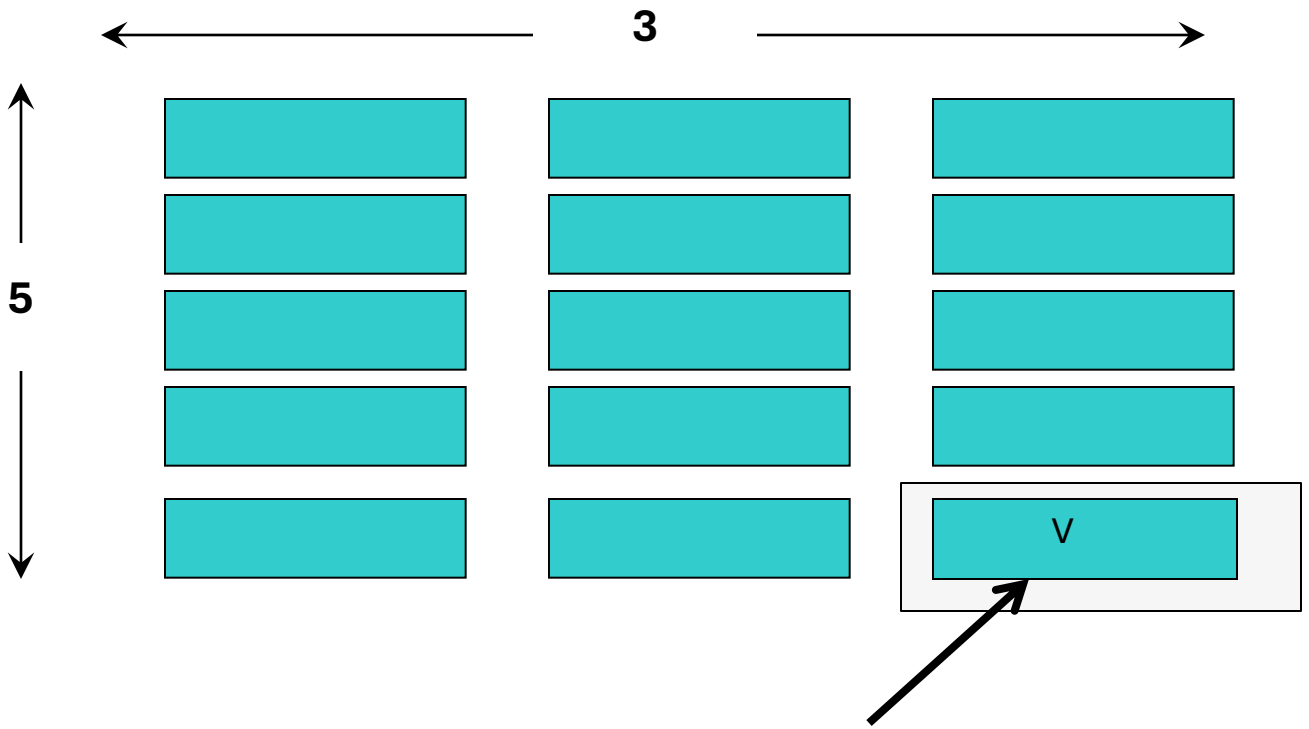
HDF5 Dataset





The HDF Group

HDF5 Dataset



Datatype: 16-byte integer

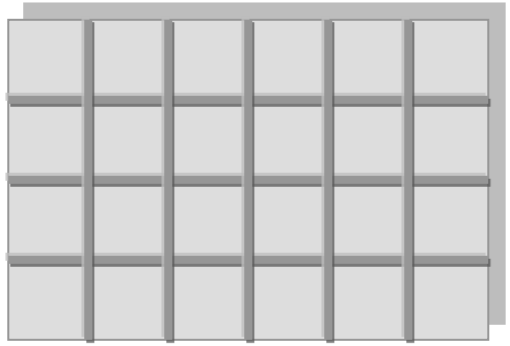
Dataspace: Rank = 2
Dimensions = 5 x 3

HDF5 Dataspaces

Two roles:

Dataspace contains spatial information (logical layout) about a dataset stored in a file

- Rank and dimensions
- Permanent part of dataset definition



Rank = 2

Dimensions = 4x6

Subsets: Dataspace describes application's data buffer and data elements participating in I/O

Rank = 1

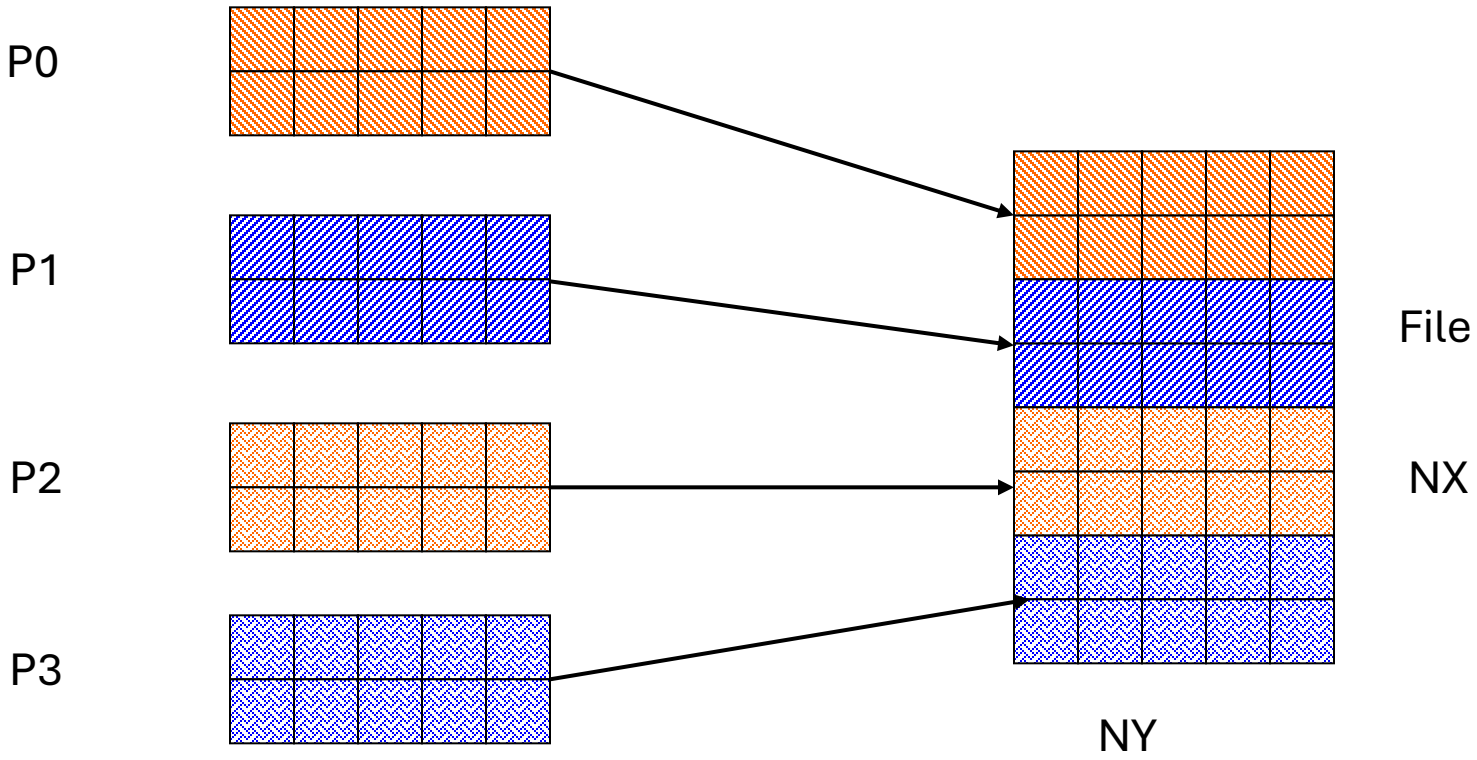
Dimension = 10

Basic Functions

H5**F**create (H5**F**open)	*create (open) File*
H5**S**create_simple/H5**S**create	*create dataSpace*
H5**D**create (H5**D**open)	*create (open) Dataset*
H5**S**select_hyperslab	*select subsections of data*
H5**D**read, H5**D**write	*access Dataset*
H5**D**close	*close Dataset*
H5**S**close	*close dataSpace*
H5**F**close	*close File*

NOTE: Order not strictly specified

Example: Writing Dataset by Rows



Writing by Rows: Output of h5dump

```
HDF5 "grid_rows.h5" {
GROUP "/" {
 DATASET "dataset1" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 8, 5 ) / ( 8, 5 ) }
      DATA {
        18, 18, 18, 18, 18,
        18, 18, 18, 18, 18,
        19, 19, 19, 19, 19,
        19, 19, 19, 19, 19,
        20, 20, 20, 20, 20,
        20, 20, 20, 20, 20,
        21, 21, 21, 21, 21,
        21, 21, 21, 21, 21
      }
   }
}
```

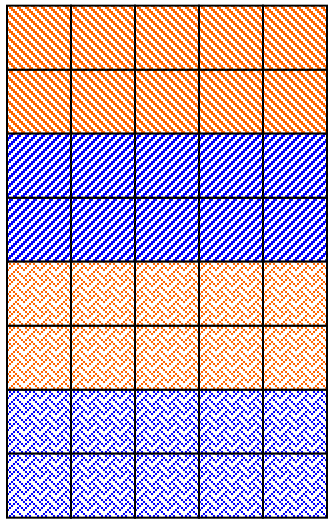
The HDF Group

Initialize the File for Parallel Access

```
/* first initialize MPI */

/* create access property list */
plist_id = H5Pcreate(H5P_FILE_ACCESS);

/* necessary for parallel access */
status = H5Pset_fapl_mpio(plist_id,
MPI_COMM_WORLD, MPI_INFO_NULL);

/* Create an hdf5 file */
file_id = H5Fcreate(FILENAME, H5F_ACC_TRUNC, H5P_DEFAULT,
plist_id);

status = H5Pclose(plist_id);
```

Create File Dataspace and Dataset

```
/* initialize local grid data */

/* Create the dataspace */


dimsf[0] = NX;
dimsf[1] = NY;


filespace = H5Screate_simple(RANK, dimsf,NULL);

/* create a dataset */
dset_id = H5Dcreate(file_id, "dataset1”,
    H5T_NATIVE_DOUBLE, filespace, H5P_DEFAULT,
    H5P_DEFAULT, H5P_DEFAULT);
```

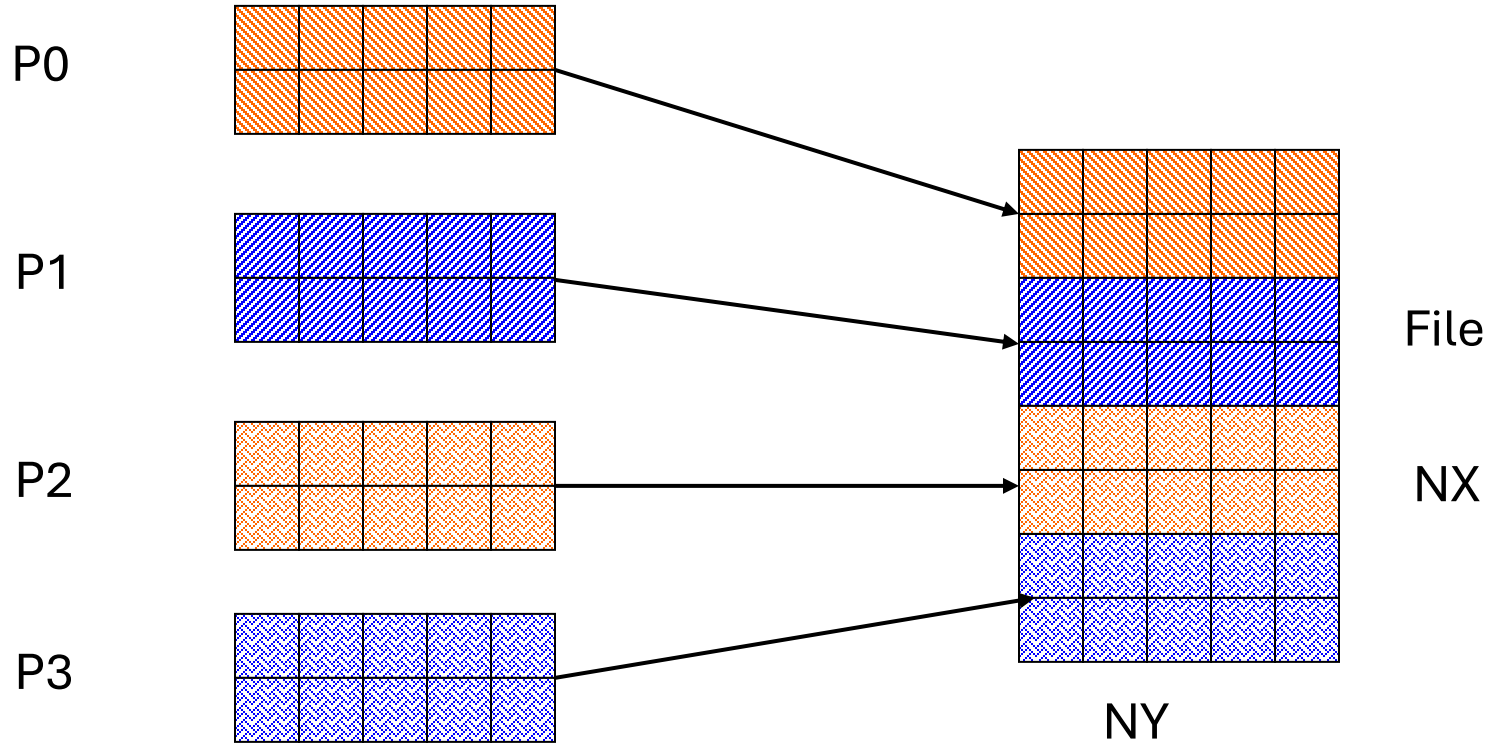


The HDF Group

Create Property List

```
/* Create property list for collective dataset write. */

plist_id = H5Pcreate(H5P_DATASET_XFER);

/* The other option is HDFD_MPIO_INDEPENDENT */
H5Pset_dxpl_mpio(plist_id,H5FD_MPIO_COLLECTIVE);
```

Calculate Offsets



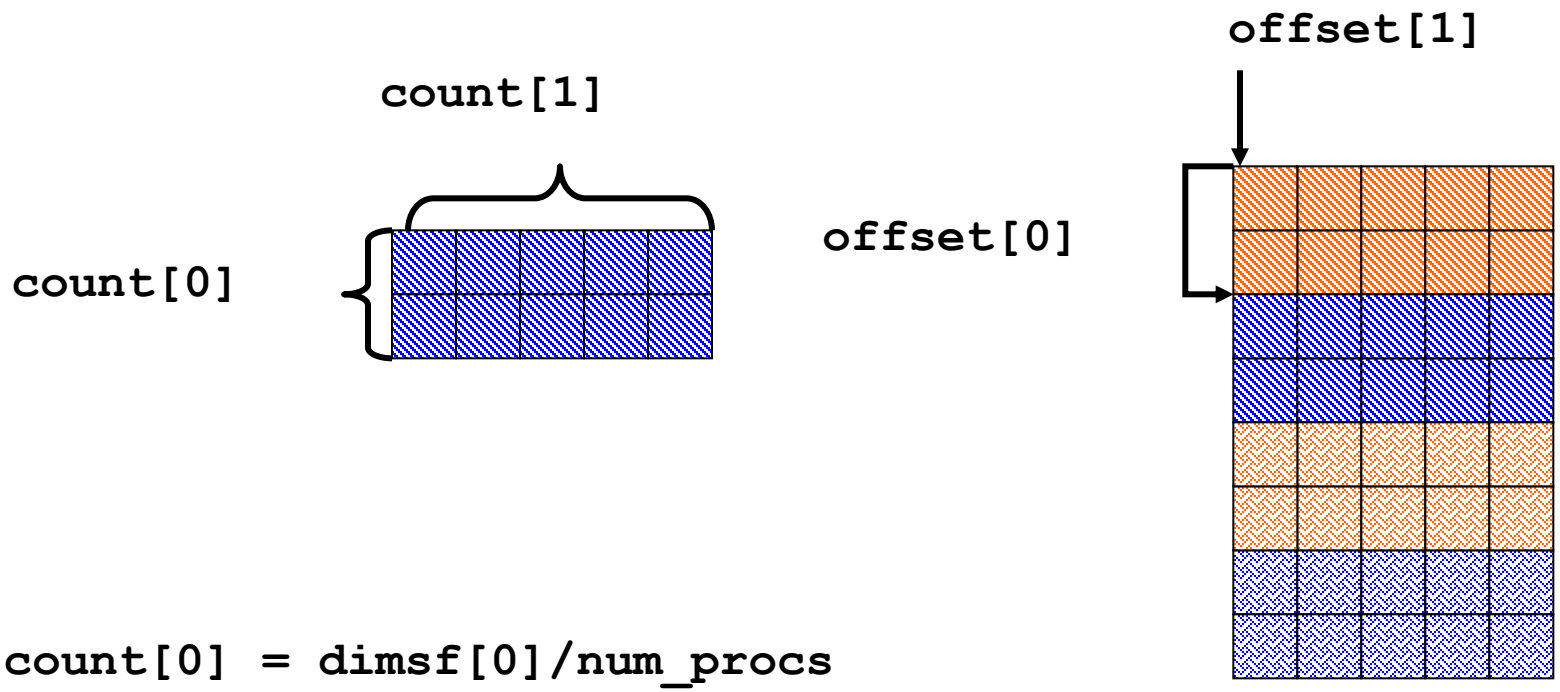
Every processor has a 2d array, which holds the number of blocks to write and the starting offset

Example: Writing Dataset by Rows

Process 1

Memory

File



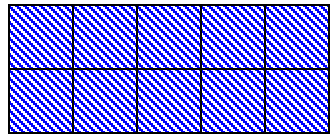

```
count[0] = dimsf[0]/num_procs
count[1] = dimsf[1];
offset[0] = my_proc * count[0];   /* = 2 */
offset[1] = 0;
```

The HDF Group

Writing and Reading Hyperslabs

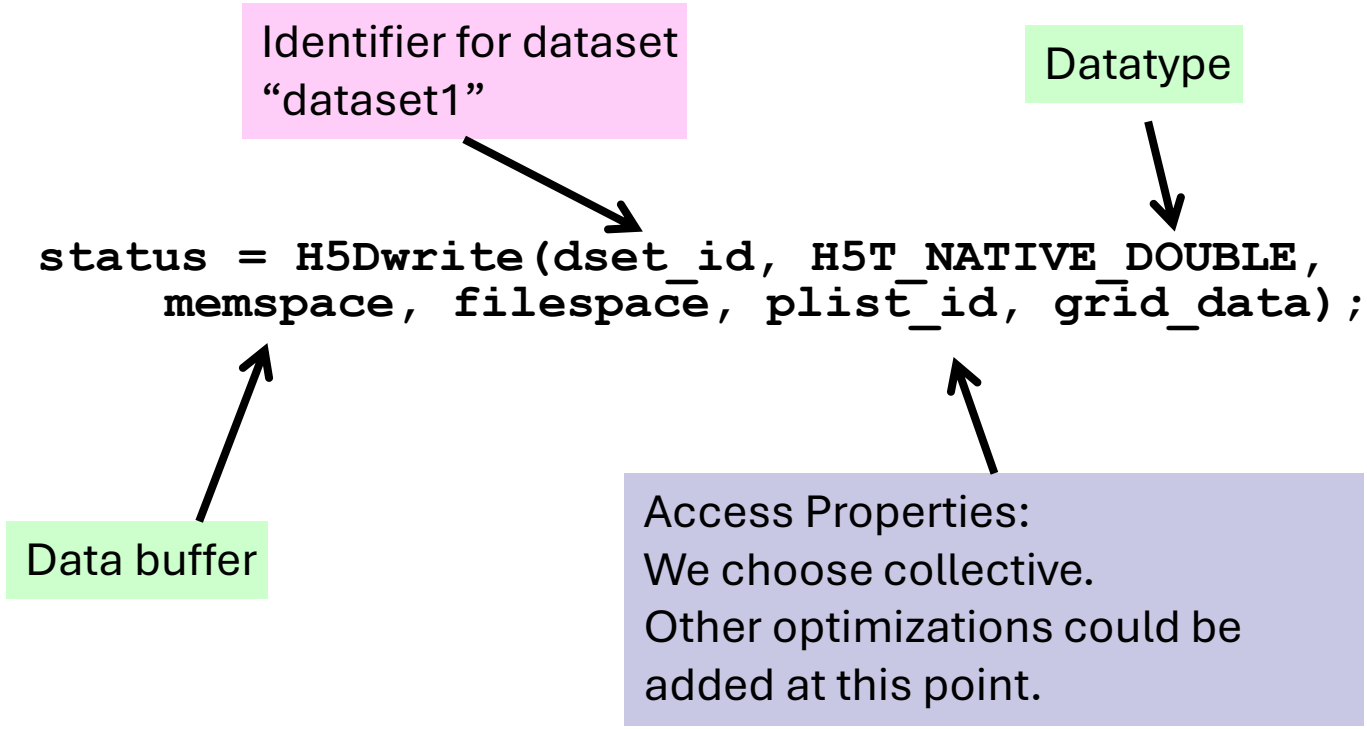
- Distributed memory model: data is split among processes
- PHDF5 uses HDF5 hyperslab model
- Each process defines memory and file hyperslabs
- Each process executes partial write/read call
  - Collective calls
  - Independent calls

Create a Memory Space Select Hyperslab



```
/* Create the local memory space */
memspace = H5Screate_simple(RANK, count, NULL);


filespace = H5Dget_space (dset_id);

/* Create the hyperslab -- says how you want to lay out data */

status = H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset,
NULL, count, NULL);
```

Write Data



Then, close every dataspace and file space that was opened

Example: FLASH's HDF5 Requirements

- FLASH: Astrophysics code covered in more detail in Pnetcdf section
- 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);
```

Effect of HDF5 Tuning

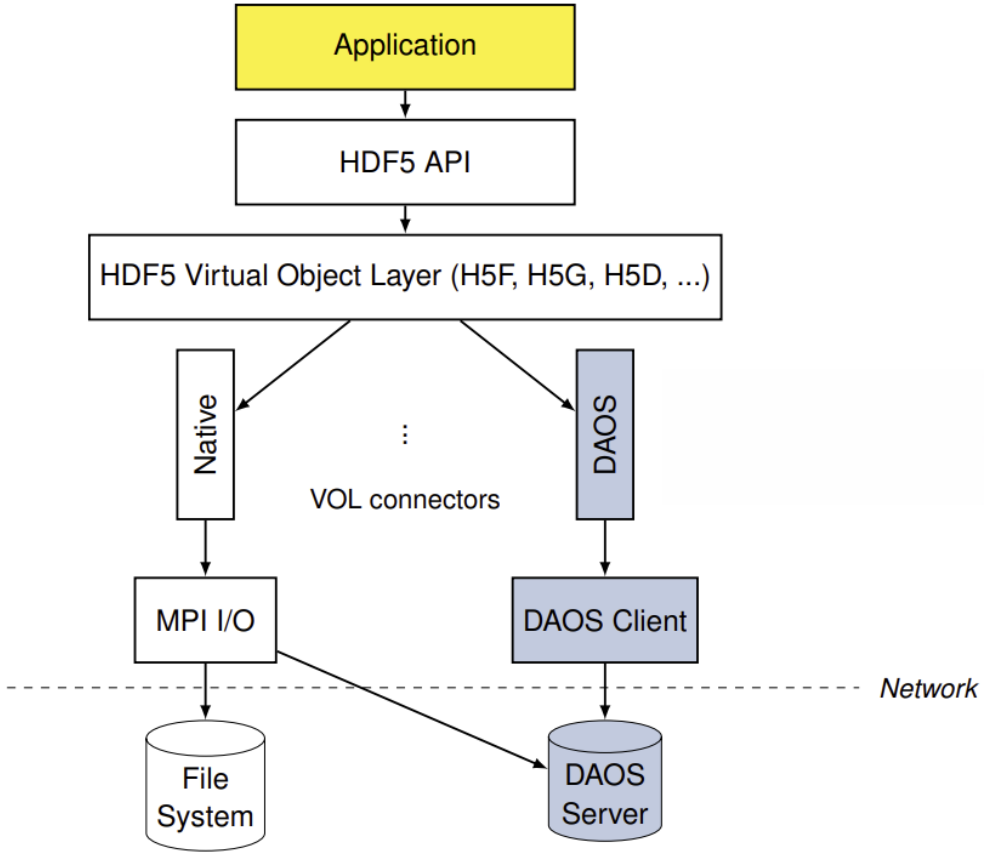
- HDF5 property lists can have big impact on internal operations
- Collective I/O vs. Independent I/O
  - Huge reduction in operation count
  - Implies all processes hit I/O at same time
- Collective metadata (new in 1.10.2)
  - Further reduction in op count, especially reads (reading HDF5 internal layout information)
  - Big implications for performance at scale

Operation counts	Independent	Coll. I/O	Coll. MD
POSIX Write	3680007	9	9
MPI-IO Indep write	3680007	7	0
MPI IO Collective Write	0	16	48
POSIX Read	3680113	115	10
MPI-IO indep read	3680113	113	8
MPI-IO collective read	0	16	16

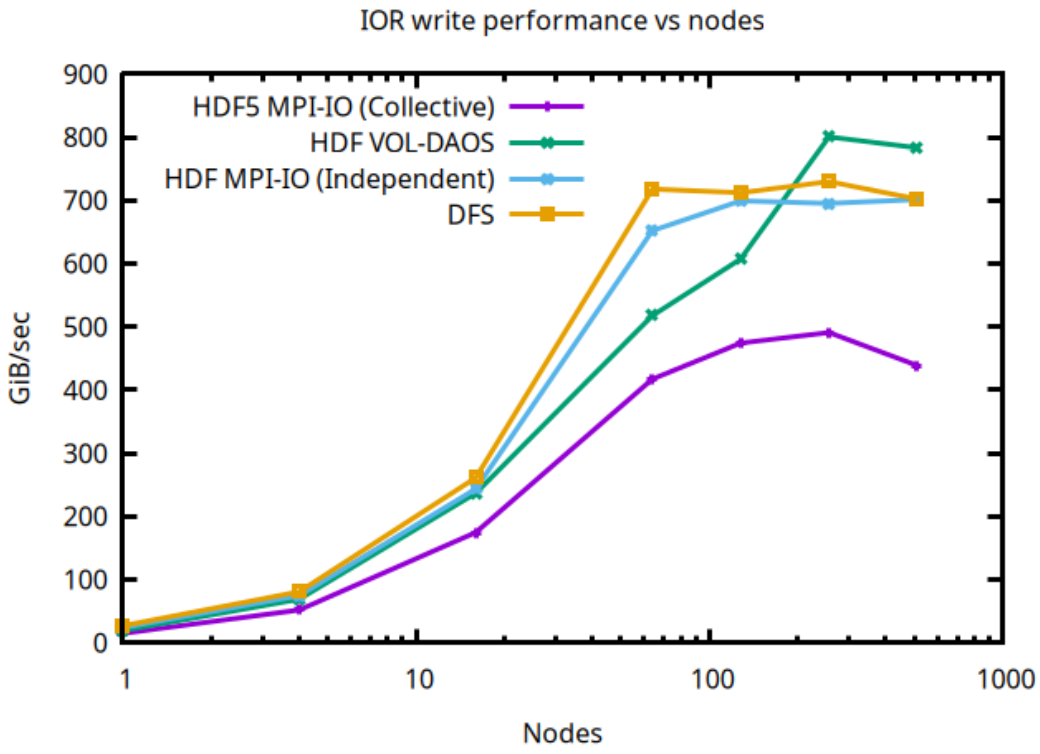
Selected Darshan statistics for 16 MPI processes writing 230 K doubles to HDF dataset, reading back same.
https://github.com/radix-io/hands-on/blob/main/hdf5/h5par-comparison.c

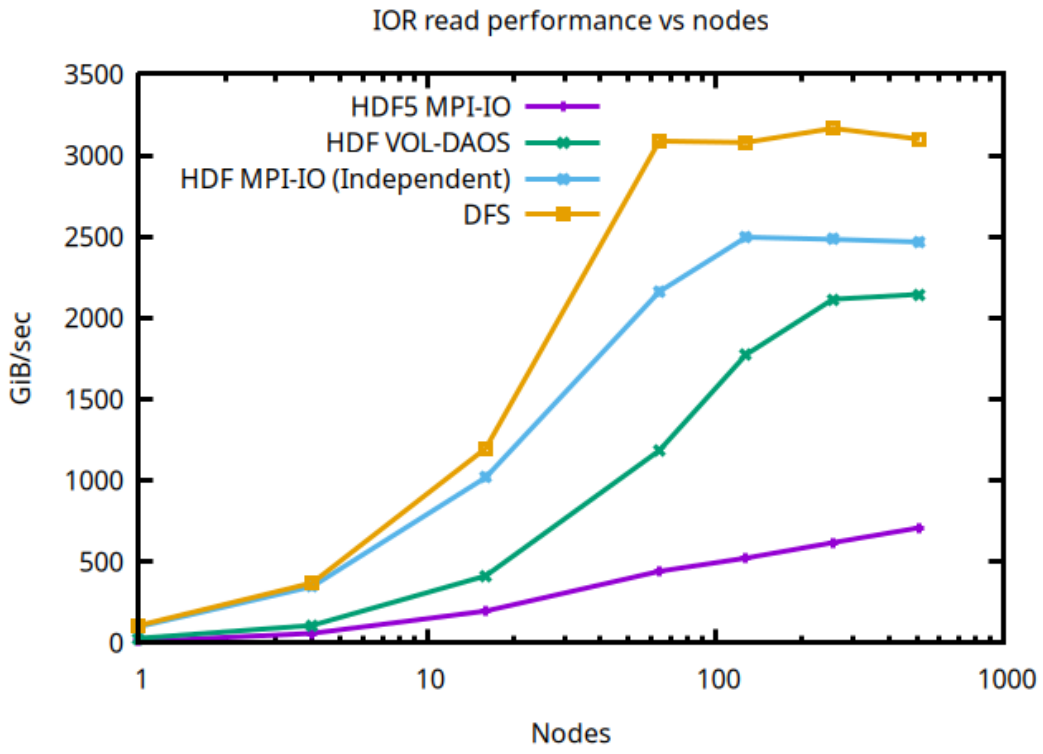
HDF5 and DAOS: the Virtual Object Layer (VOL)

- VOL: keep the HDF5 interface; allow alternate backend storage approaches
- Can opt-in with HDF5 function calls or environment variables (no code change needed)
- DAOS vol bypasses MPI machinery for direct access to DAOS



VOL demonstration: Aurora + DAOS





- Experiment: IOR – every process writes/reads a big, unaligned, contiguous region
  - Compare HDF5 three ways: MPI-IO collective, MPI-IO independent, DAOS-VOL and straight DFS
  - Note: not the most typical HDF5 workload : a “worst-case” for collective I/O

HDF5 Wrap-up

- Tremendous flexibility: 300+ routines
- "H5 High Level" routines for common cases
  - https://portal.hdfgroup.org/display/HDF5/High-level+Library
- Tuning via property lists
  - “use MPI-IO to access this file”
  - “read this data collectively”
- Extensive on-line documentation, tutorials (see “On Line Resources” slide)
- New efforts:
  - Fast appends (finance-motivated)
  - Multiple-dataset I/O: similar to Parallel-NetCDF operation-combining optimization
  - Virtual Object Layer (VOL): extend HDF5 to e.g. object or cloud storage

Other High-Level I/O Libraries

- NetCDF-4: https://www.unidata.ucar.edu/software/netcdf
  - netCDF API with HDF5 back-end
- ADIOS: https://adios2.readthedocs.io/
  - Configurable (xml) I/O approaches
- SILO: https://wci.llnl.gov/simulation/computer-codes/silo
  - A mesh and field library on top of HDF5 (and others)
- H5part: https://gitlab.psi.ch/H5hut/src/-/wikis/home
  - simplified HDF5 API for particle simulations
- GIO: https://svn.pnl.gov/gcrm
  - Targeting geodesic grids as part of GCRM
- SCORPIO: https://e3sm.org/scorpio-parallel-io-library/
  - 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

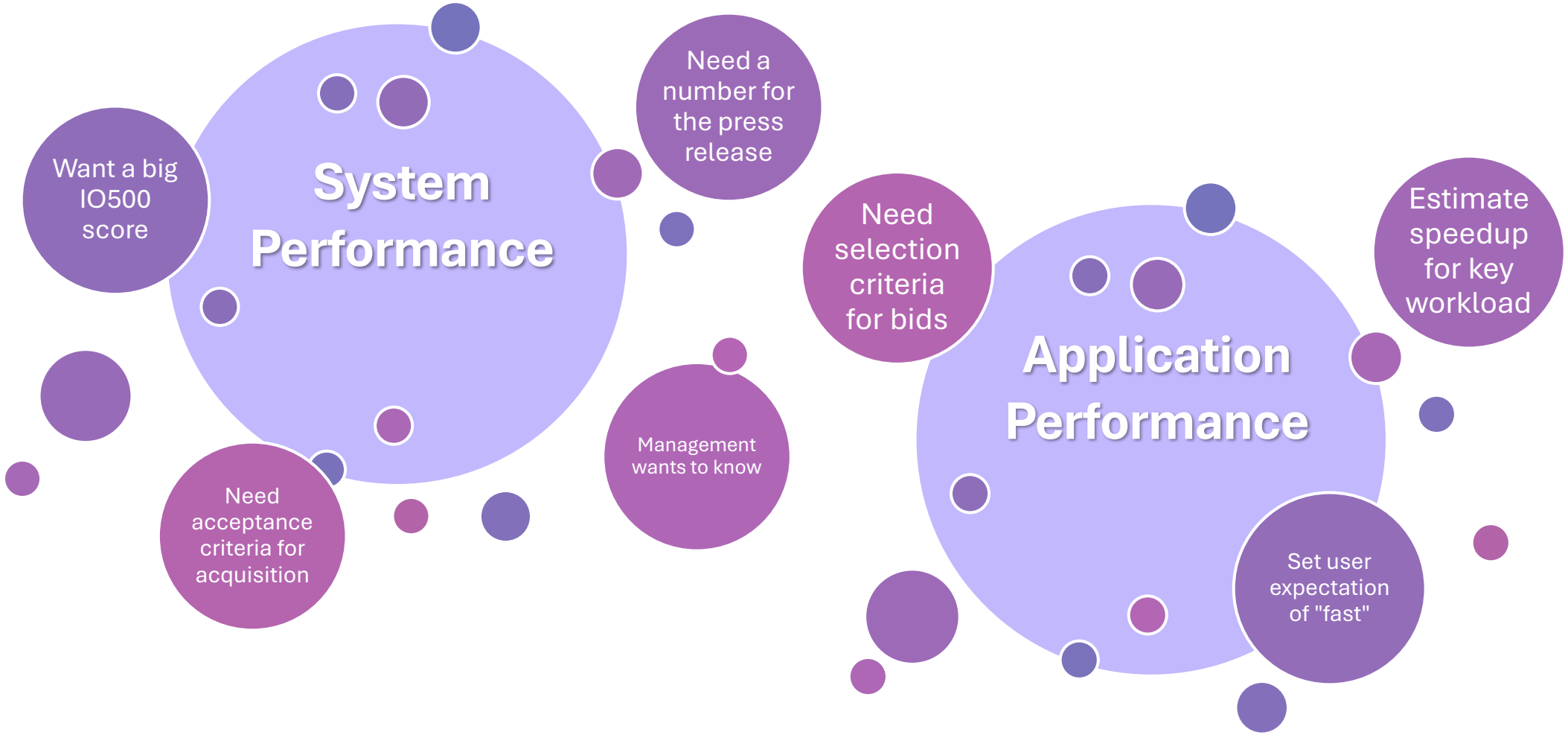
Parallel I/O Wrap-up

- 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
  - No cost to performance; may sometimes improve it
  - Gains include 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
- Software stack makes it easy to migrate to new (object store) back ends

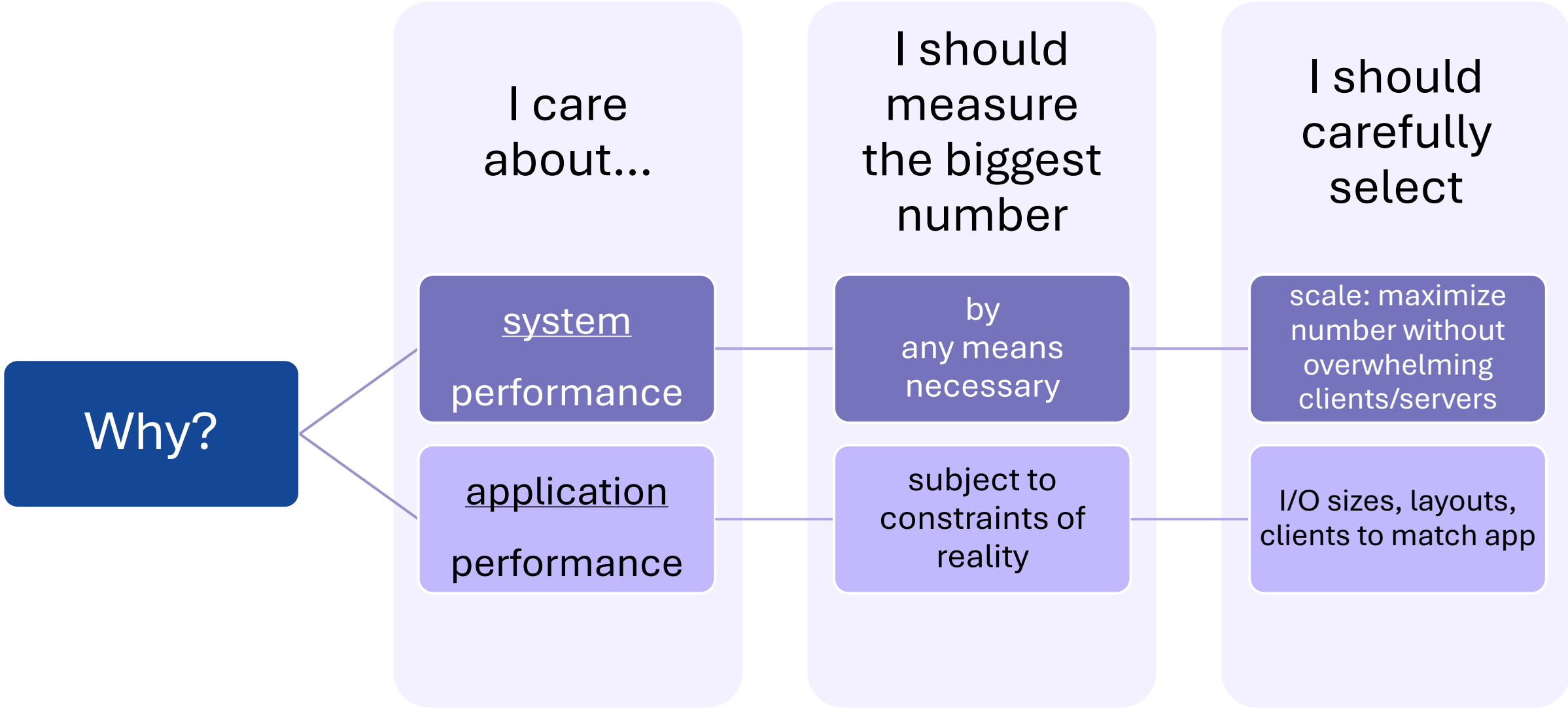
Modeling I/O Behavior and Performance

A fancy way to say “benchmarking”

Why do you want to benchmark storage?



How this shapes your approach to benchmarking



~~Experimental Design~~ Benchmarking Process

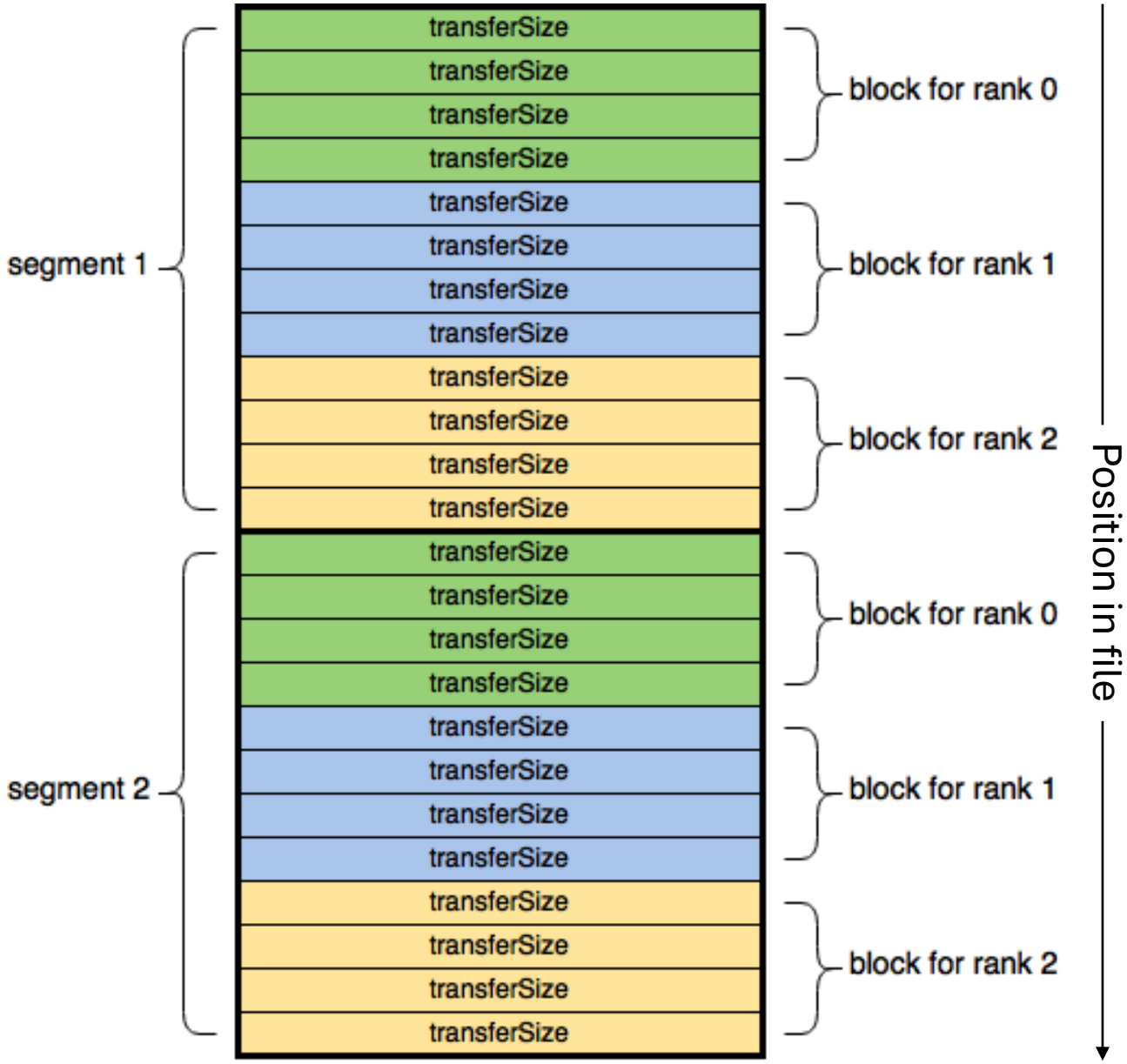
Why are you benchmarking?	How will you benchmark?	What did you just measure?	What is the uncertainty?

• To understand system
  • Bandwidth
  • IOPS
  • Metadata
• To understand apps
  • Parallel checkpoint
  • Ensembles
	• IOR, elbencho, …?
	• mdtest, md-workbench, …?
	• IO500?
		• Client DRAM?
		• Network?
		• Server DRAM?
		• Server HDD or SSD?
			• Std. deviation?
			• Multimodality?
			• Distribution shape?

Basic throughput benchmarking with IOR

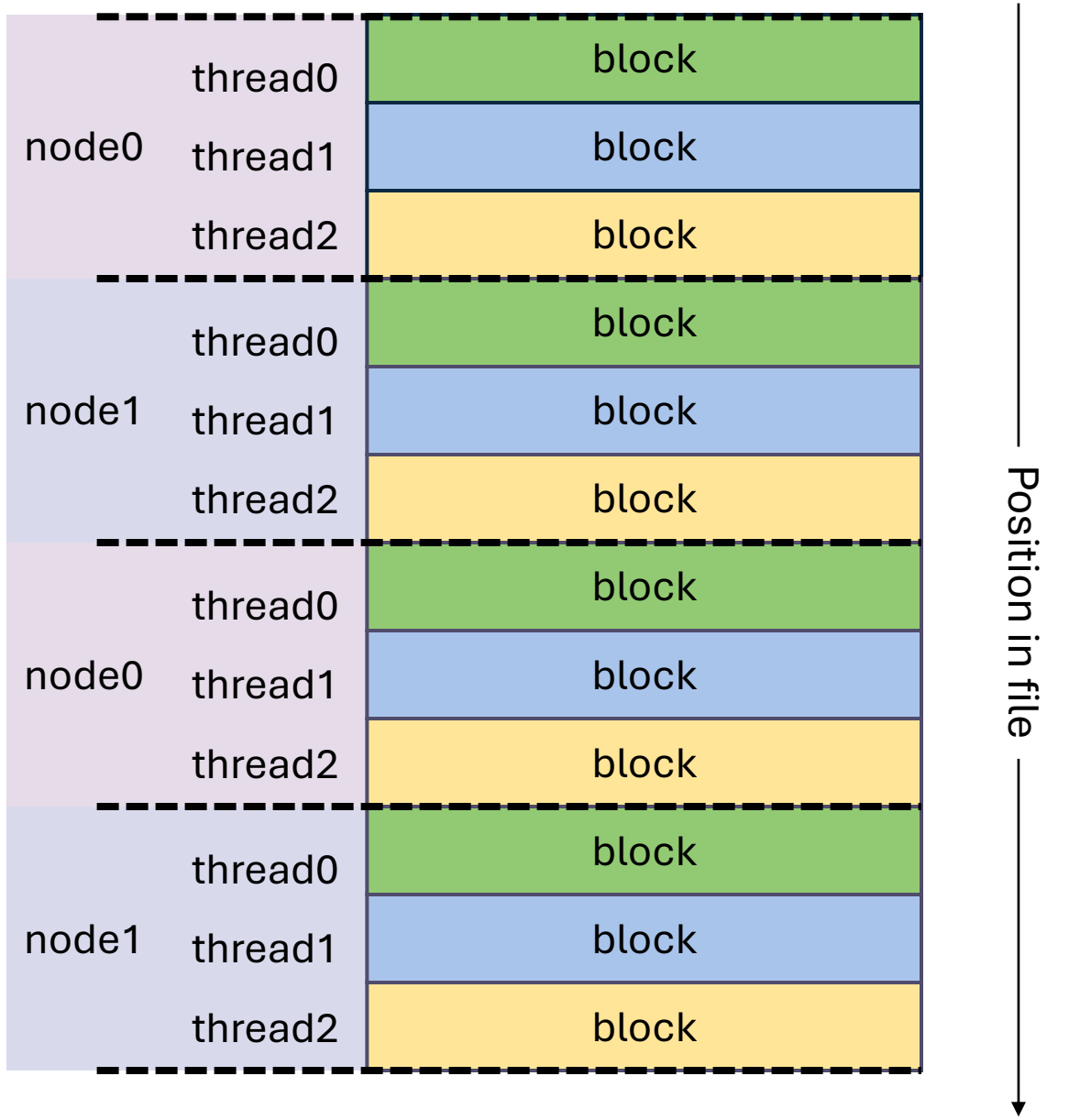
The IOR benchmark tool

- MPI application benchmark
  - reads and writes data in configurable ways
  - I/O pattern can be interleaved or random
- Input:
  - transfer size, block size, segment count
  - interleaved or random
- Output: Bandwidth and IOPS
- Configurable backends
  - POSIX, STDIO, MPI-IO
  - HDF5, PnetCDF, S3, rados
- https://github.com/hpc/ior



The elbencho benchmark tool

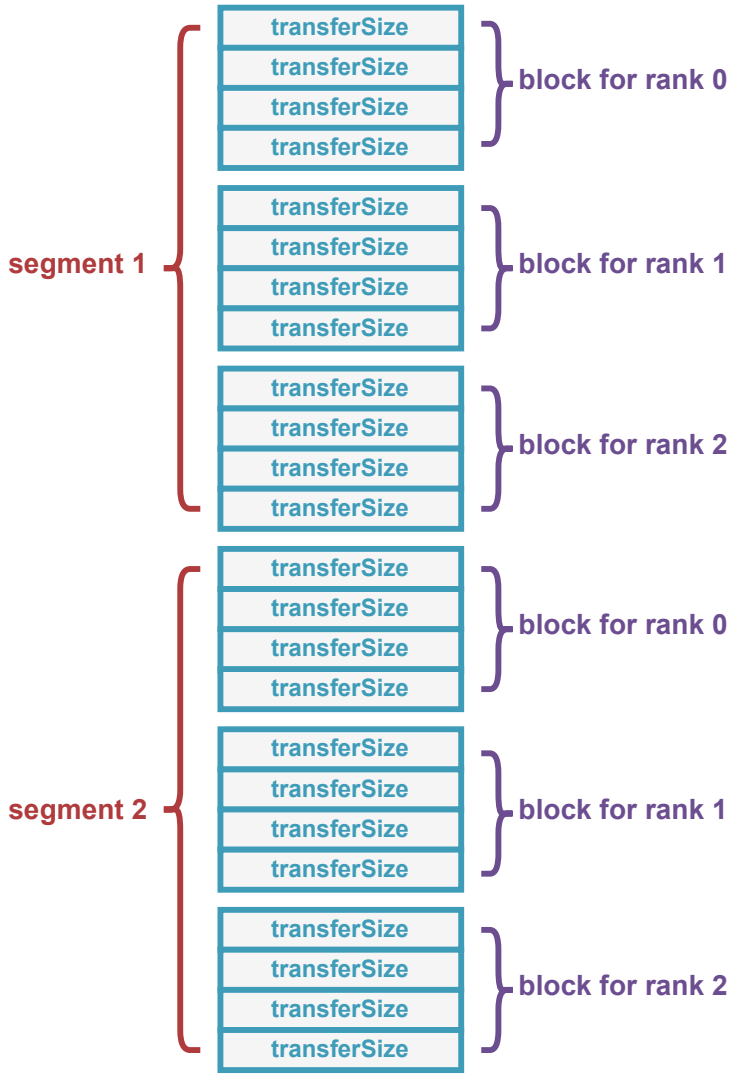
- **Non-MPI** application benchmark
  - reads and writes data in configurable ways
  - I/O pattern can be interleaved or random
- Input:
  - transfer size, file size
  - **strided** or random
- Output: Bandwidth and IOPS
- Configurable backends
  - POSIX, GPUDirect Storage
  - S3
- https://github.com/breuner/elbencho



First attempt at benchmarking an I/O pattern

- 120 GB/sec Lustre file system
- 4 compute nodes, 16 ppn, 200 Gb/s NIC
- Performance makes no sense
  - write performance is awful
  - read performance is mind-blowingly good

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64
...
Operation   Max(MiB)
write         9539.38
read        492123.04
```



Try breaking up output into multiple files

- IOR provides -F option to make each rank read/write to its own file instead of default single-shared-file I/O
  - Reduces lock contention within file
  - Can cause metadata load at scale
- Problem: > 400 GB/sec from 4 OSSes is faster than light

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F
...
Operation   Max(MiB)
write        72852.83
read        481168.60
```

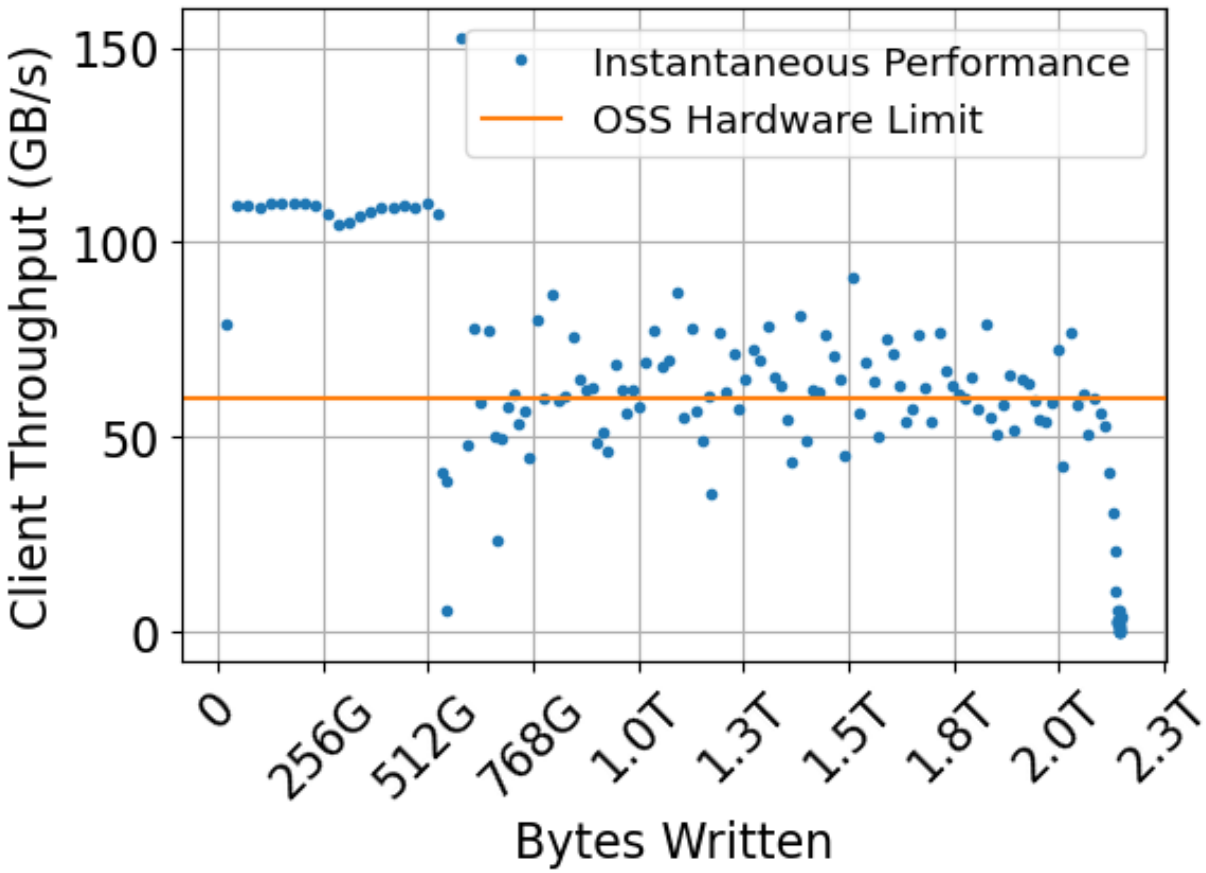
Effect of page cache on measured I/O bandwidth

- Unused compute node memory to cache file contents
- Can dramatically affect I/O
  - Writes:
    - only land in local memory at first
    - reordered and sent over network later
    - `max_dirty_mb` and `max_pages_per_rpc` (Lustre)
    - `dirty_background_ratio` and `dirty_ratio` (NFS)
  - Reads:
    - come out of local memory if data is already there
    - read-after-write = it's already there
    - readahead also exists

Time-resolved write bandwidth

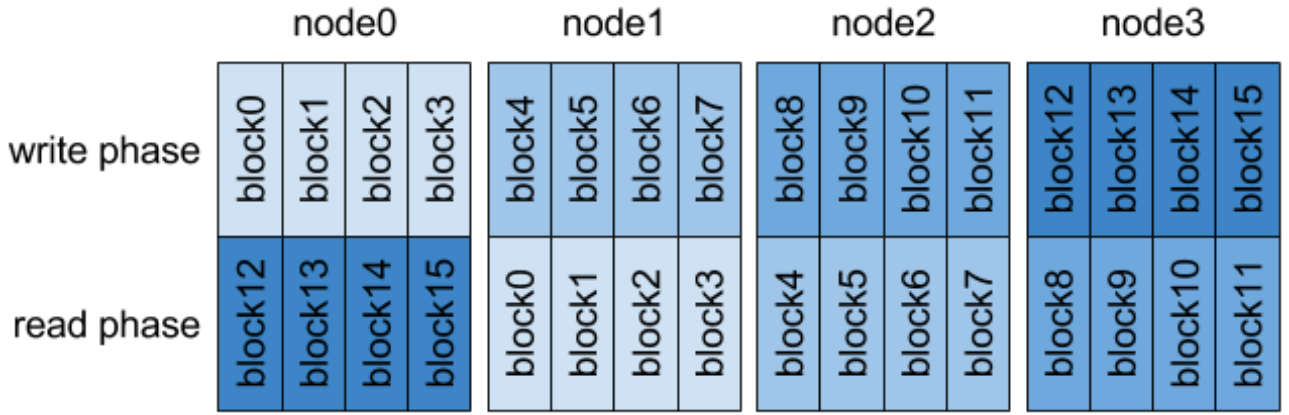
4 clients / 256 GiB DDR each

2 OSTs / 60 GB/s write spec



Avoid reading from cache with rank shifting

- Use -C to shift MPI ranks by one node before reading back
- Read performance looks reasonable
- But what about write cache?

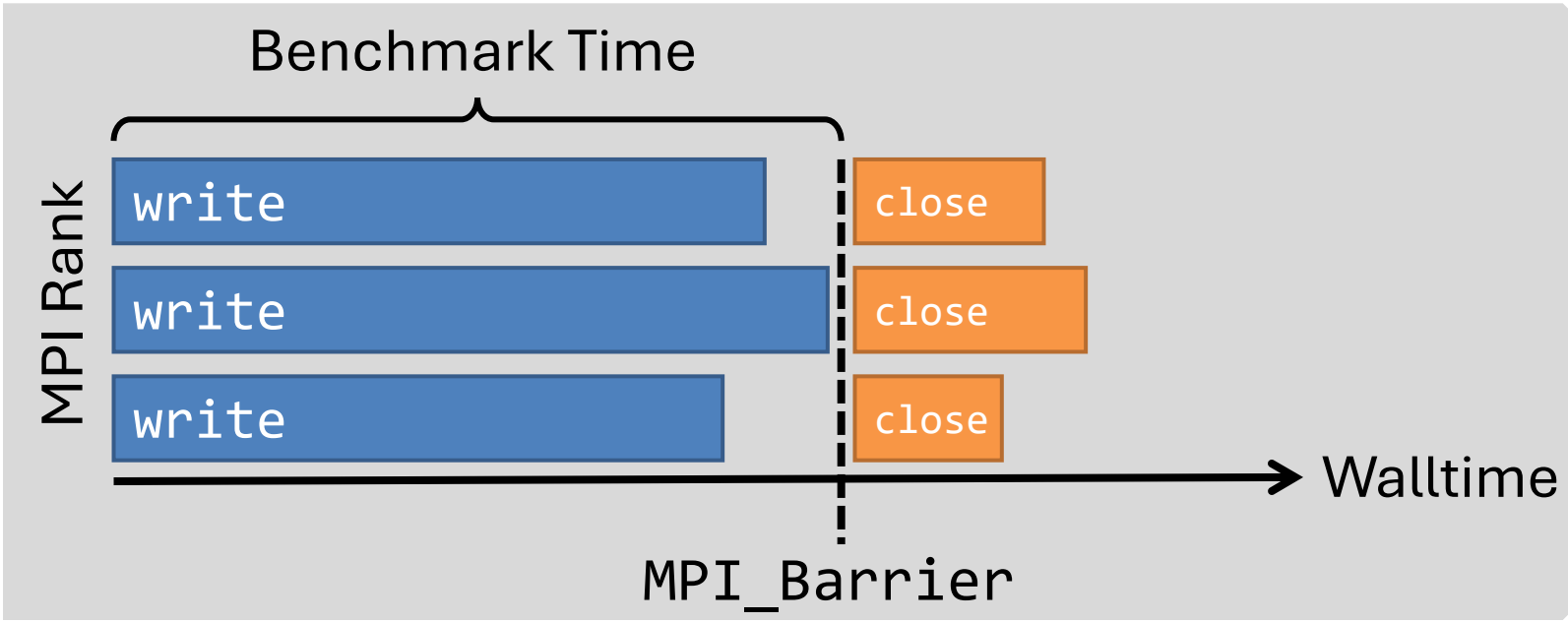



```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F -C
...
Operation   Max(MiB)
write       63692.33
read        28303.09
```

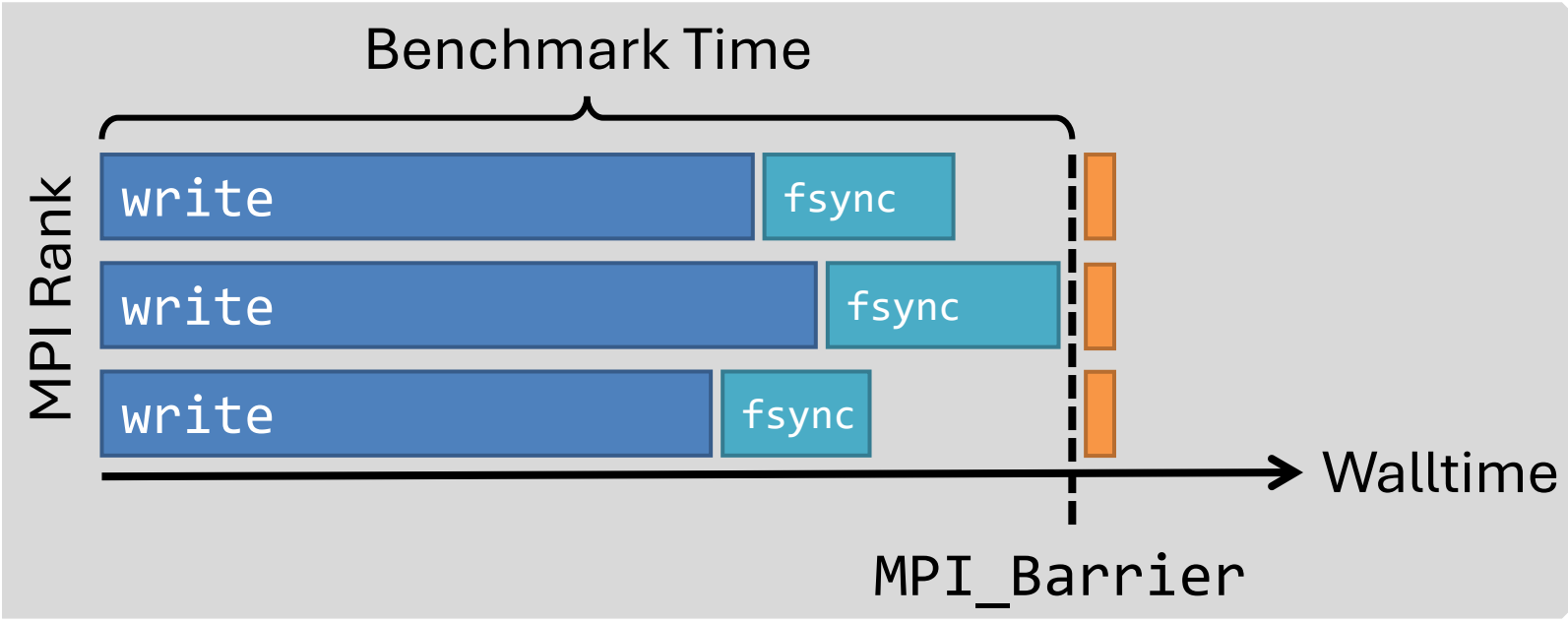
Force sync to account for write cache effects

- Default: benchmark timer stops when last write completes
- Desired: benchmark timer stops when all data reaches OSSes
  - Use `-e` option to force `fsync(2)` and write back all "dirty" (modified) pages
  - Measures time to write data to durable media—not just page cache
- Without fsync, `close(2)` operation may include hidden sync time

```
$ srun -N 4 -n 64 ./ior -t 1m -b 64m -s 64 -F -C -e
...
Operation   Max(MiB)
write       70121.02
read        30847.85
```

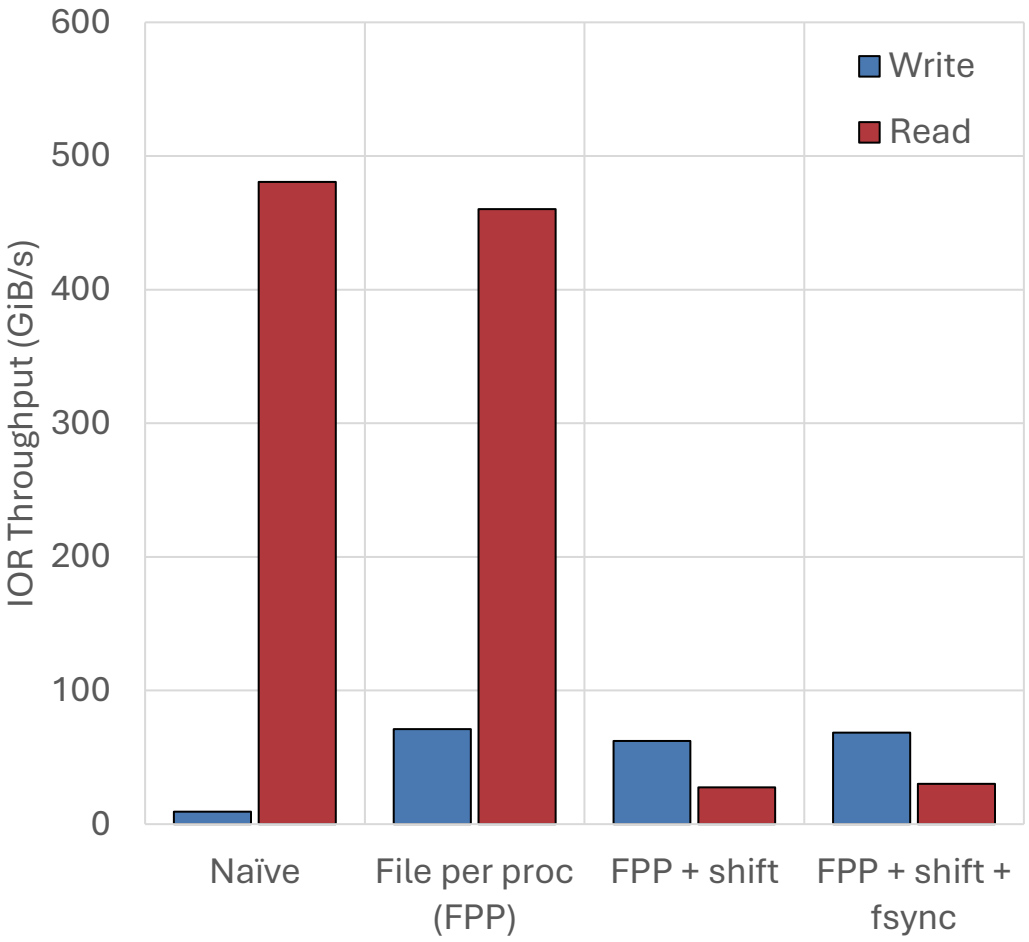


By default, benchmark may appear to hang at the end when files are being closed



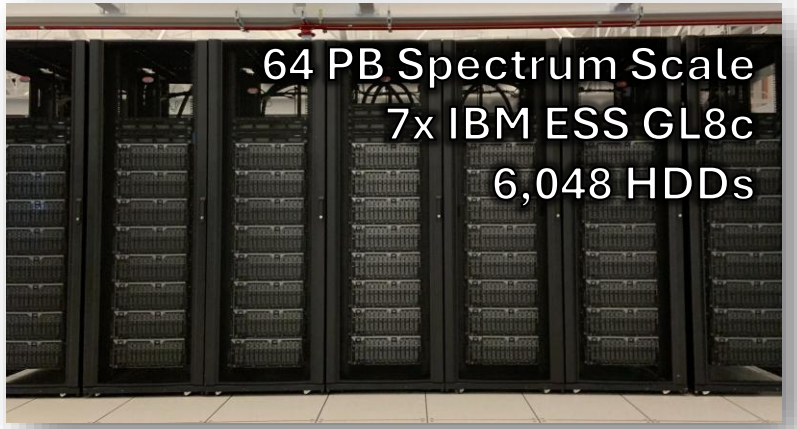
With -e / fsync, time to write dirty pages to file system is included

Measuring bandwidth can be complicated



- 100x difference from same file system!
  - Client caches and sync
  - File per proc vs. shared file
  - Usual Lustre stuff (e.g., striping)
- For system benchmarking, start with `-F -C -e`
- Page cache is not part of POSIX!
  - Every file system does it differently
  - Understanding its effects requires expert knowledge

IOR Acceptance Tests
Spectrum Scale Bandwidth



8 ppn used

```
$ srun -N 51 -n 408 ./ior -F -C -e -b 32g -t 1m
```

Standard args:

- -F File-per-process
- -C Shift ranks
- -e include fsync(2) time

Every rank writes (1 X 32) GiB total, 1 MiB at a time
(note: -s not given, so default is 1)

Results:

- 193,717 MB/s write (max)
- 162,753 MB/s read (max)

IOR Acceptance Tests
Lustre Bandwidth

Only 4 ppn needed

```
$ srun -N 960 -n 3840 ./ior -F -C -e -g -b 4m -t 4m -s 1638 -w -k
$ srun -N 960 -n 3840 ./ior -F -C -e -g -b 4m -t 4m -s 1638 -r
```

Standard args:

- -F File-per-process
- -C Shift ranks
- -e include fsync(2) time

- -w Perform write benchmark only
- -k Don't delete written files
- -r Perform read benchmark only

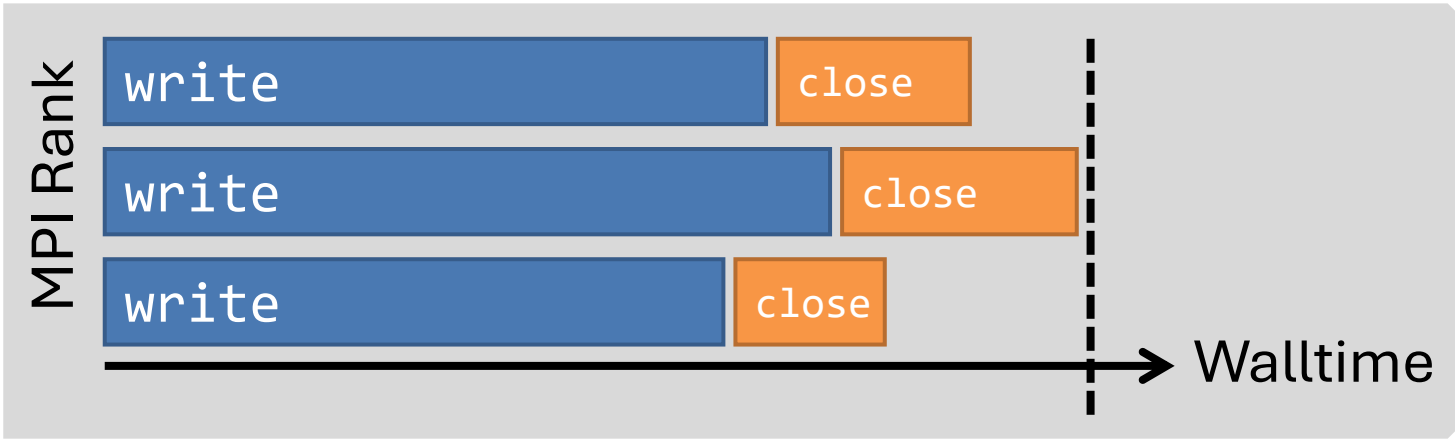
Separate sruns drop client caches

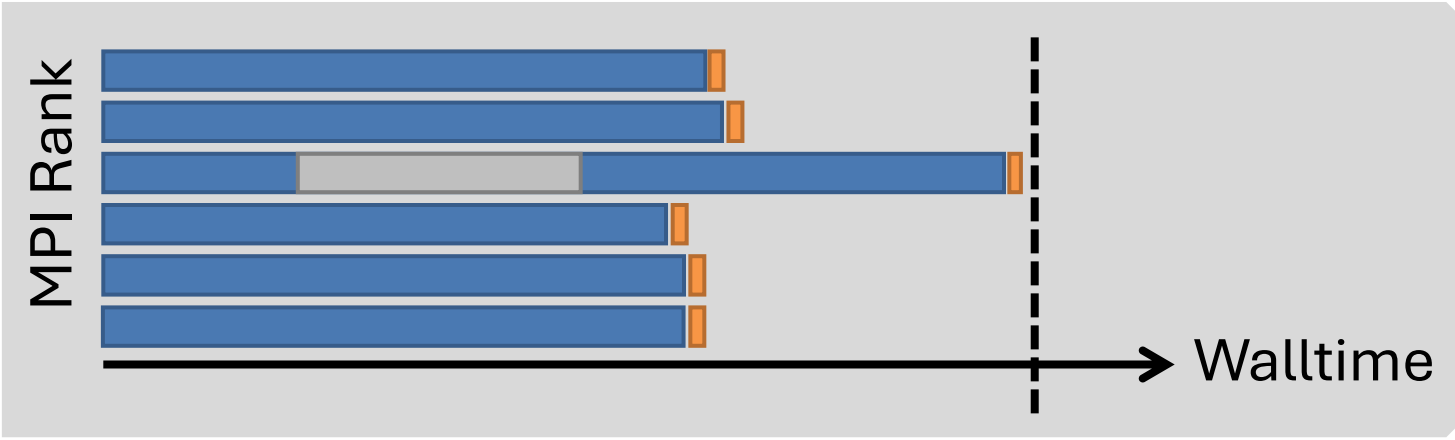
Every rank writes 4 MiB × 1,638
4 MiB at a time
Total ~25 TB

Results:

- 751,709 MB/s write (max)
- 678,256 MB/s read (max)

Running afoul of “wide” benchmarking





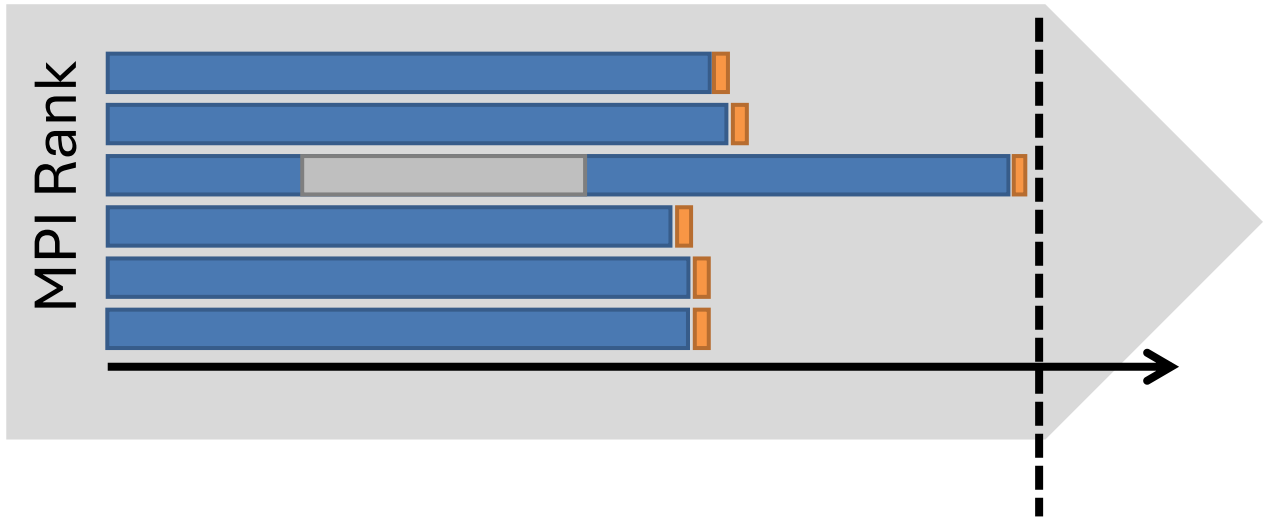
How much -b/-s?

- More is better: overrun cache effects
- More is worse: increase likelihood of hiccup
- Preference: run for 30-60s

What is realistic for you?

- do you want the big number?
- do you want to emulate user experience?
- small = fast
- big = realistic

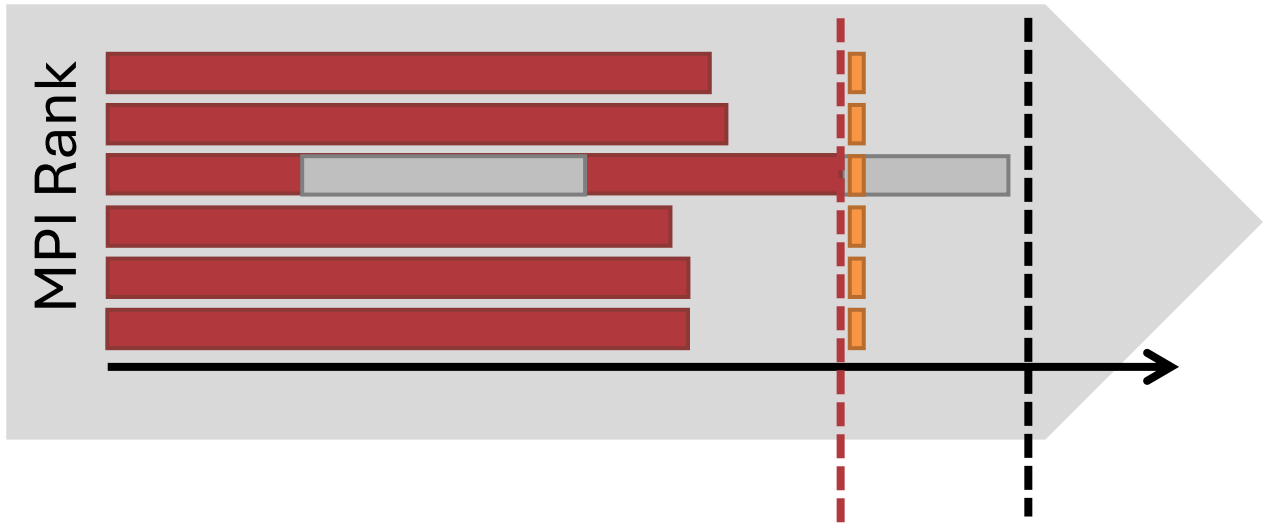
Stonewalling to reduce penalty of stragglers



Default behavior

- all ranks write same total bytes
- timer stops when slowest rank finishes

Stonewalling to reduce penalty of stragglers



Default behavior

- all ranks write same total bytes
- timer stops when slowest rank finishes

Stonewalling (`-D 30`)

- stop all writes after 30 sec, add up bytes written
- bandwidth = total bytes written / 30 sec [+ fsync time]
- *not* what apps do – apps don't give up if I/O is slow!
- *shows best-case system capability despite hiccups* – recommended for system acceptance

IOR Acceptance Tests
Lustre Bandwidth – Writes with time limit

4 ppn used

```
$ srun -N 1382 -n 5528 ./ior -F -C -e -g -b 1m -t 1m -s 100000 -D 45 -w
```

Standard args:

- -F File-per-process
- -C Shift ranks
- -e include fsync(2) time

1 × 100,000 MiB/rank
1 MiB at a time
~527 TiB total

-or-

for 45 seconds (-D 45),
whichever happens first

-w Write-only benchmark

Results: 3,593,657 MB/s write (max)

IOR Acceptance Tests
Lustre Bandwidth – Reads with time limit


```
$ srun -N 1382 -n 5528 \
          ./ior -F -C -e -g -b 1m -t 1m -s 100000 -D 90 -w -k -O stoneWallingWearOut=1
$ srun -N 1382 -n 5528 \
          ./ior -F -C -e -g -b 1m -t 1m -s 100000 -D 30 -r
```

1. Write data to files for 90 seconds
   - `-O stoneWallingWearOut=1`: every rank writes the same number of bytes even if stonewalling (`-D 90`) cuts the run short
   - Generates uniform files - avoid EOF when ranks are shifted and read back
2. Read back files for 30 seconds

Results: 4,003,761 MB/s read (max)

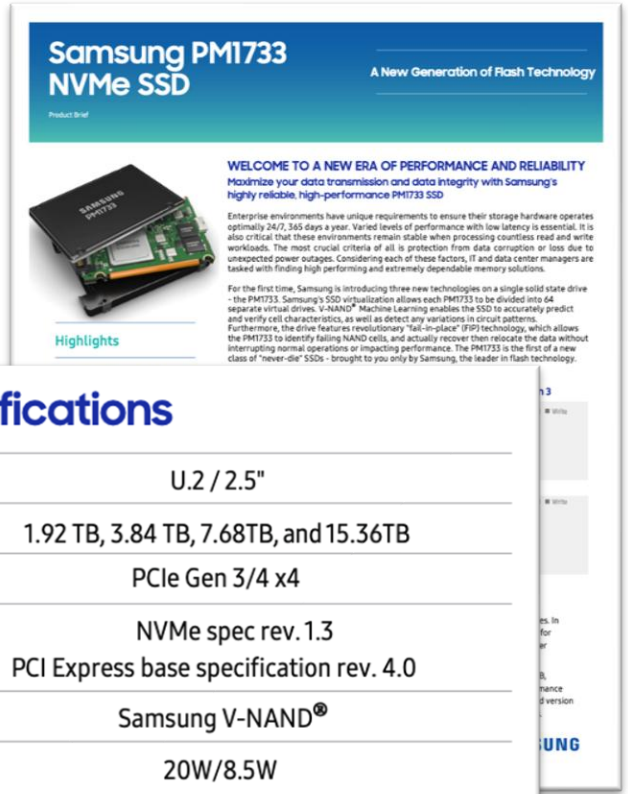
Advanced benchmarking

IOPS and metadata

Measuring random I/O performance - IOPS

I/O Operations Per Second

- Move smallest unit of storage from/to arbitrary location
- “smallest” usually 4 KiB (memory page in Linux)
- 1 IOP = 4 KiB I/Os per sec from random offsets
- Historically block-level

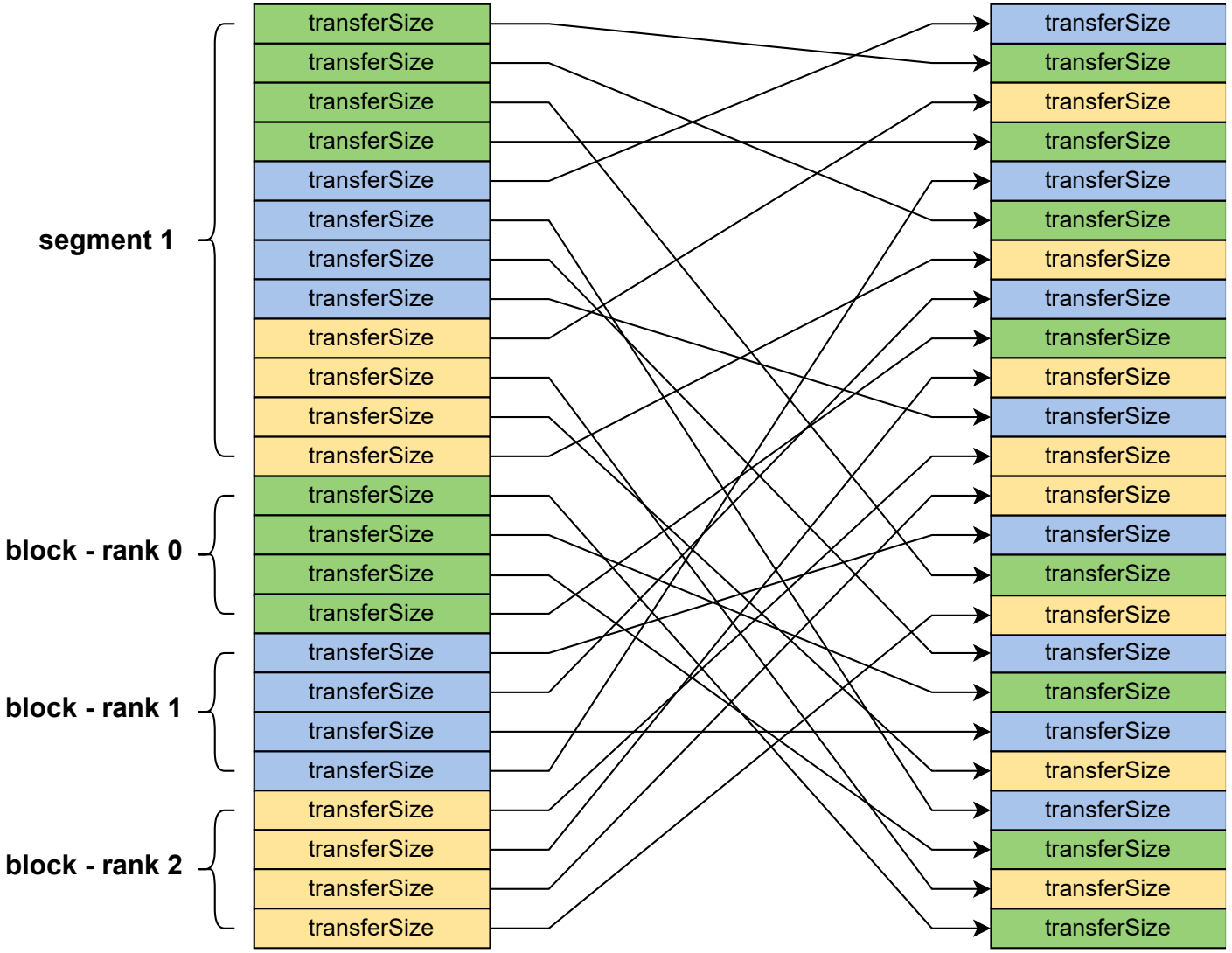

Samsung PM1733
NVMe SSD
A New Generation of Flash Technology
Product Brief
WELCOME TO A NEW ERA OF PERFORMANCE AND RELIABILITY
Maximize your data transmission and data integrity with Samsung's highly reliable, high-performance PM1733 SSD
Highlights

Samsung PM1733 specifications

Form factor	U.2 / 2.5"
Capacity	1.92 TB, 3.84 TB, 7.68TB, and 15.36TB
Host interface	PCIe Gen 3/4 x4
Spec Compliance	NVMe spec rev. 1.3 PCI Express base specification rev. 4.0
NAND flash memory	Samsung V-NAND®
Power consumption (Active/Idle)	20W/8.5W
Uncorrectable Bit Error Rate (UBER)	1 sector per10^17 bits read
Mean Time Between Failure (MTBF)	2,000,000 hours
Endurance	1 DWPD for 5 years
Sequential read	**Gen 3:** 3,500 MB/s, **Gen 4:** 7,000 MB/s
Sequential write	**Gen 3:** 3,200 MB/s, **Gen 4:** 3,500 MB/s
Random read	**Gen 3:** 800K IOPS, **Gen 4:** 1.5M IOPS
Random write	**Gen 3 & 4:** Up to 135,000 IOPS

Measuring IOPS is generally dumb;
see https://glennklockwood.blogspot.com/2021/10/iops-are-dumb.html

Switching IOR from “interleaved” to “random”

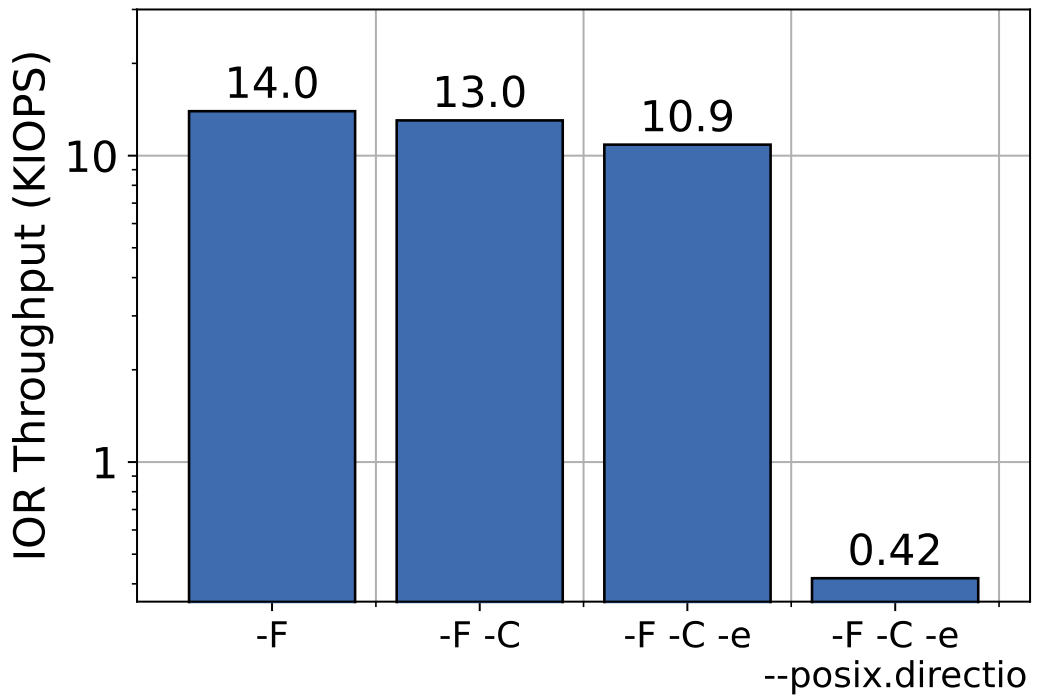


- `-z` randomizes order of each transfer (`-t`)
- `-b` and `-s` still set dataset size
- `-t  4K` sets size of each read/write

Lessons learned still apply (plus a new gotcha for IOPS)

- **File per process (-F) or shared file?**
  - Do you want your IOPS number to reflect lock contention?
  - What are you trying to measure?
- **How to cope with client page cache?**
  - Read – IOR will never re-read the same transfer from cache with -C
  - Write – client caching will reorder/coalesce random writes
- **Unique to IOPS - write vs. rewrite**
  - Re-writing files randomly has much higher overhead
  - Consider RAID read-modify-write impacts

Overcoming caches for random writes



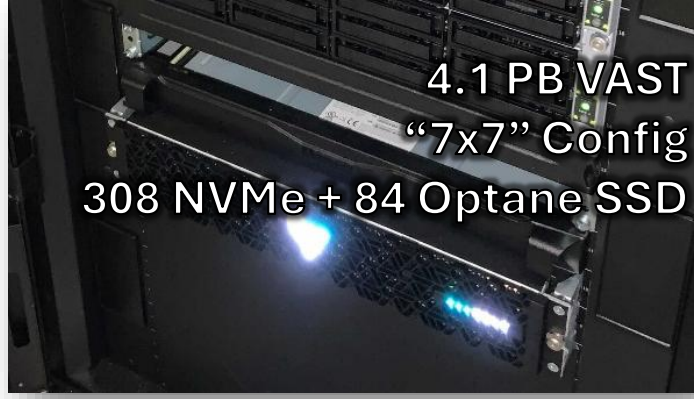
IOR Write IOPS tests

4 clients, 16 ppn

274x NVMe OSTs

- `--posix.odirect`
  - forces file I/O to bypass page cache entirely
  - reduces apparent write IOPS
- Which is “true performance?”
  - True random writes are rare
  - Random, direct I/O is rarer
- Application performance should include write-back
- System performance is better measured with `O_DIRECT`

IOR Hero Test
VAST IOPS – Writes with time limit




```
$ srun -N 32 -n 4096 ./ior -F -C -e -g -b 1m -t 4k -s 372736 \
                           -D 45 -w -z --posix.odirect -l random
```

- Using `--posix.odirect`
  - Force random pattern to cross network without reordering
  - Oversubscribed clients (128 ppn) to make up for O_DIRECT loss
- Other parameters are standard
  - `-D 45` for stonewalling
  - `-w` is write-only test (and no -k means don’t bother keeping files after)
  - `-z` for random instead of interleaved

Results: 640,713 IOPS write (max)

IOR Hero Test
Lustre IOPS – Reads with time limit


```
$ srun –N 1024 –n 32768 ./ior –F    –e –g -b 64g -t 64m       -D 90 -w -k -O stoneWallingWearOut=1

$ srun -N 1024 -n 32768 ./ior -F    -e -g -b 64g -t 64m       -D 45 -w    -o tempfiles.dat

$ srun –N 1024 –n 32768 ./ior –F -C –e –g -b  1g -t  4k -s 20 -D 45 -r -k -z
```

1. Write data to files for 45 seconds
   - `-O stoneWallingWearOut=1` to generate uniform file sizes (uppercase O)
   - `-t 64m` for big bandwidth - generate big files for random read test
2. Cleanse the palate
   - write (`-w`) and discard (no `-k`) data to flush out caches (clients + servers) just in case
   - `-o tempfiles.dat` specifies name of files IOR creates (lowercase o)
   - don't want to overwrite dataset generated in Step 1
3. Read back files for 45 seconds
   - Avoid EOF by sizing `-b` and `-s` to match file sizes generated in step 1 from `-D 90 -O stoneWallingWearOut=1`

Results: 117,349,729 IOPS read (max)

The mdtest benchmark

- MPI application benchmark – included with IOR
- Performs metadata operations on configurable directory hierarchies
- Input:
  - # files/dirs to test
  - how deep/wide tree should be
- Output: metadata ops per second
- Configurable backends
  - POSIX, STDIO, MPI-IO
  - same support as IOR

https://github.com/hpc/ior/releases

- tree.0/
  - tree.1/
    - tree.3/
      - tree.7/
        - file.0
        - file.1
      - tree.4/
        - file.2
        - file.3
    - tree.2/
      - tree.5/
        - file.4
        - file.5
      - tree.6/
        - file.6
        - file.7

Step 1. Measure metadata performance

```
$ srun -n 2 ./mdtest -n 100000
...

2 tasks, 200000 files/directories

SUMMARY rate: (of 1 iterations)
   Operation                     Max            Min
   ---------                     ---            ---
   Directory creation        13873.461      13873.461
   Directory stat            23070.131      23070.131      23070.131        0.000
   Directory rename           6807.317       6807.317       6807.317        0.000
   Directory removal          9581.667       9581.667       9581.667        0.000
   File creation              8798.513       8798.513
   File stat                 10907.440      10907.440
   File read                 12079.891      12079.891
   File removal              14142.063      14142.063      14142.063        0.000
   Tree creation              2100.302       2100.302       2100.302        0.000
   Tree removal                236.739        236.739        236.739        0.000
```

Operate on 100,000 “things” per MPI process

“thing” = directories

“thing” = files

Operations per second

Step 2. Figure out what it's really doing

What mdtest does:

1. Create directory tree
2. Test directory performance
    1. create
    2. stat
    3. rename
    4. unlink
3. Test file performance
    1. create and write
    2. stat
    3. read and close
    4. unlink
4. Destroy directory tree

`MPI_Barrier()`

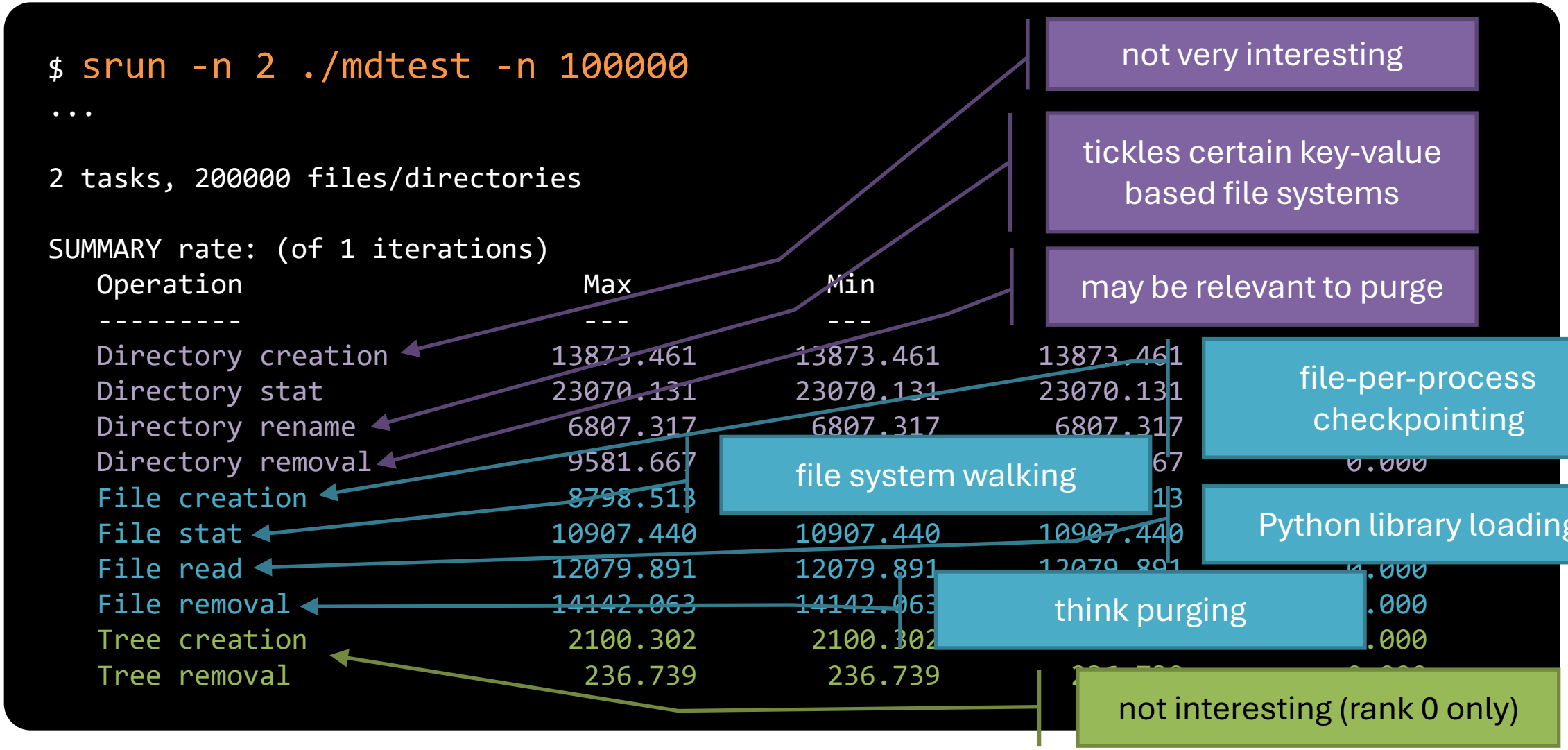
What mdtest tells you:

- how fast one operation *and nothing else* can be sustained
- bulk-synchronous performance (think: file-per-process checkpoint)
- cost of different metadata operations

What mdtest does not tell you:

- compile/untar/python performance
- when a file system will tip over
- how laggy a file system will feel

What these numbers represent



Selecting which tests to run (default: all)

```
$ srun -n 2 ./mdtest -n 100000
...

2 tasks, 200000 files/directories

SUMMARY rate: (of 1 iterations)
   Operation                     Max
   ---------                     ---
   Directory creation        13873.461
   Directory stat            23070.131
   Directory rename           6807.317
   Directory removal          9581.667
   File creation              8798.513
   File stat                 10907.440
   File read                 12079.891
   File removal              14142.063
   Tree creation              2100.302
   Tree removal                236.739
```

Option	Effect
-D	Run only **directory** tests
-F	Run only **file** tests
-C	Run only **create** phase
-T	Run only **stat** phase
-E	Run only **read** phase
-r	Run only **removal** phase

mdtest Acceptance Tests
Lustre on HDDs - No DNE

32 ppn

```
$ srun -N 1620 -n 51840 ./mdtest -n 20 -F -C -T -r
```

Create 20 files/dirs per MPI rank

- -F Only run file tests
- -C Only run create phase
- -T Only run stat phase
- -r Only run removal phase

Results:

- 45,945 creates/sec (max)
- 147,502 stats/sec (max)
- 28,213 unlinks/sec (max)

So don't run:

- directory tests
- read phase

mdtest Acceptance Tests
Lustre on HDDs - DNE Phase 1


```
$ srun -N 1620 -n 51840 ./mdtest -n 20 -F -C -T -r -u \
    -d /lus/mdt0@/lus/mdt1@/lus/mdt2@/lus/mdt3@/lus/mdt4
```

- `-d` specifies output directories
  - Multiple directories can be separated by `@`
  - Makes mdtest evenly stripe data across many dirs
  - Can repeat directories if, e.g., one MDT is “better” than others

Results:

- 112,349 creates/sec (max)
- 453,902 stats/sec (max)
- 111,286 unlinks/sec (max)

mdtest Acceptance Tests
Lustre on NVMe - DNE Phase 2


```
$ lfs mkdir -c 4 -D /lus/striped
$ srun -N 64 -n 1024 ./mdtest -n 2441 -F -C -r -d /lus/striped
```

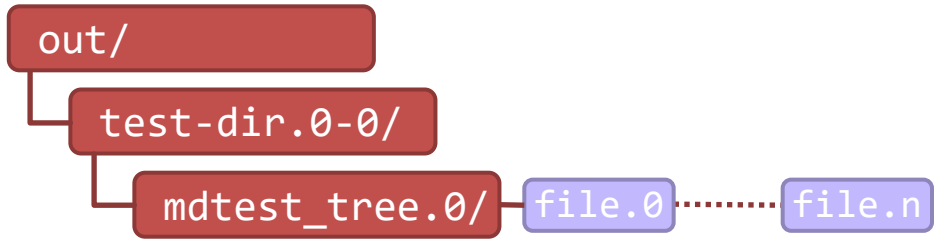
- Create striped metadata directory (`/lus/striped`)
- Use 2,441 files per MPI process
- Run only file tests (`-F`), create (`-C`) and unlink (`-r`) phases
- Work in our newly created striped dir (`-d /lus/striped`)

Results:

- 217,396 creates/sec (max)
- 187,845 unlinks/sec (max)

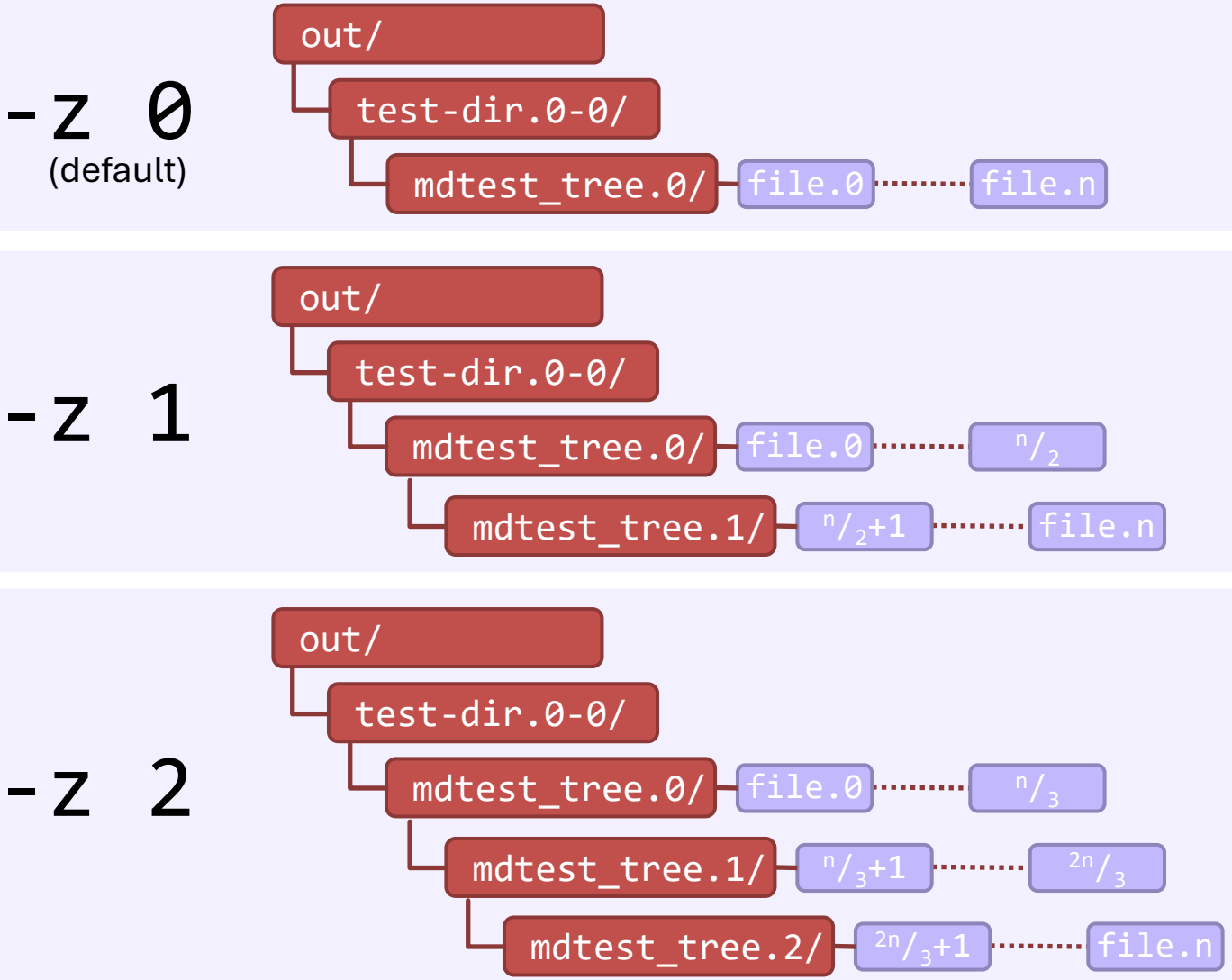
Controlling the directory hierarchy

- Depth factor (`-z`) controls depth
- Branching factor (`-b`) controls breadth
- Files are either
  - spread evenly throughout every directory (default)
  - spread evenly at deepest directories (leaf mode (`-L`))
- Default
  - zero depth (`-z 0`), zero breadth (`-b 1`)
  - dumps all files into one giant directory



Changing depth factor

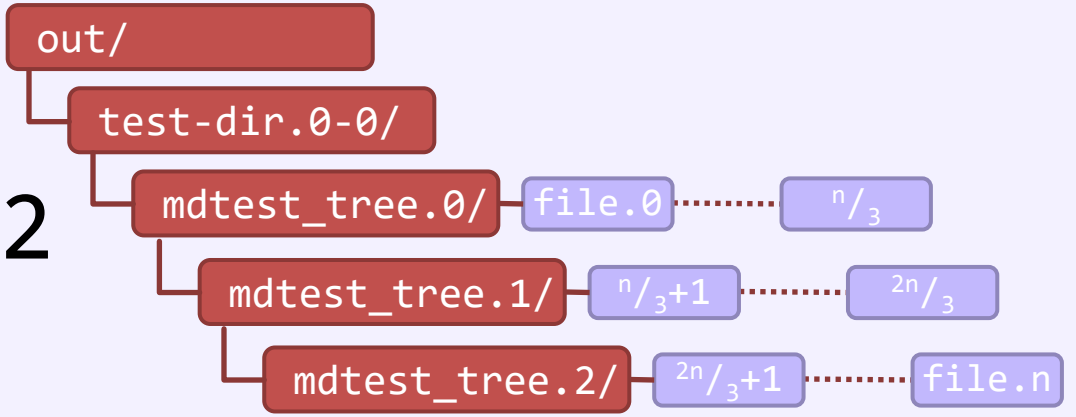
- Creates skinny trees
- Always the same file count in each dir
- Rounds n down if not evenly divisible by (depth+1)



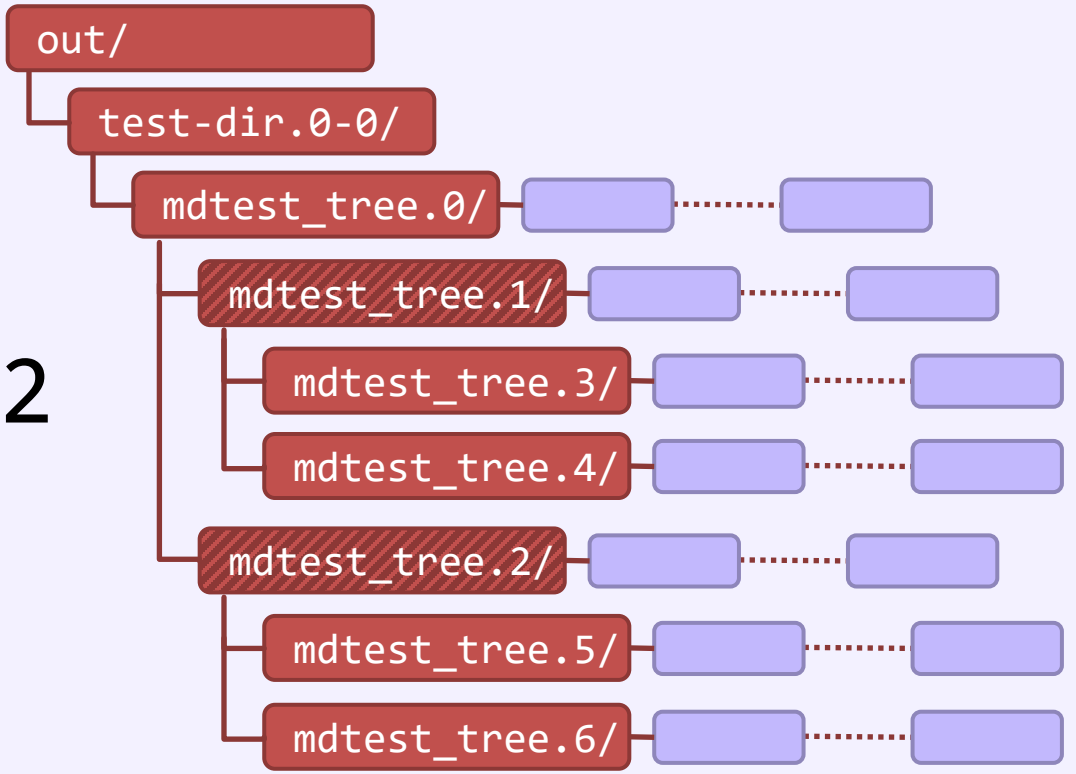
Changing branching factor

- Exponential branching
  - Files evenly spread, rounded down
  - Need high number of files (-n) to get lots of files/dir
- Realistic fs complexity
- Realistic workload?
  - Parallel file transfers?
  - Anything else?

-z 2



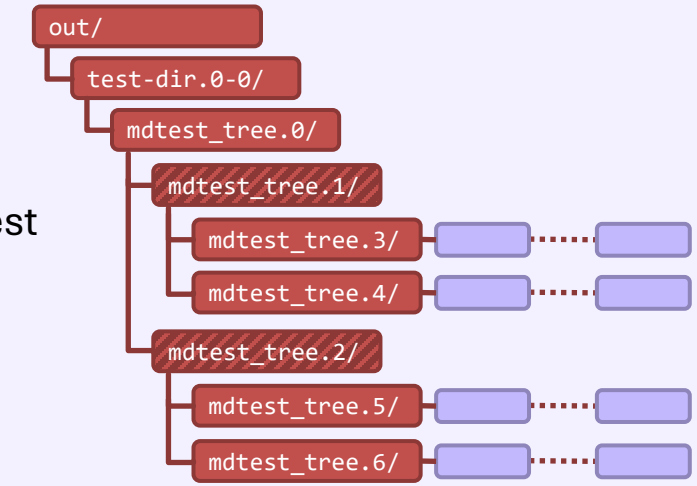
-z 2 -b 2

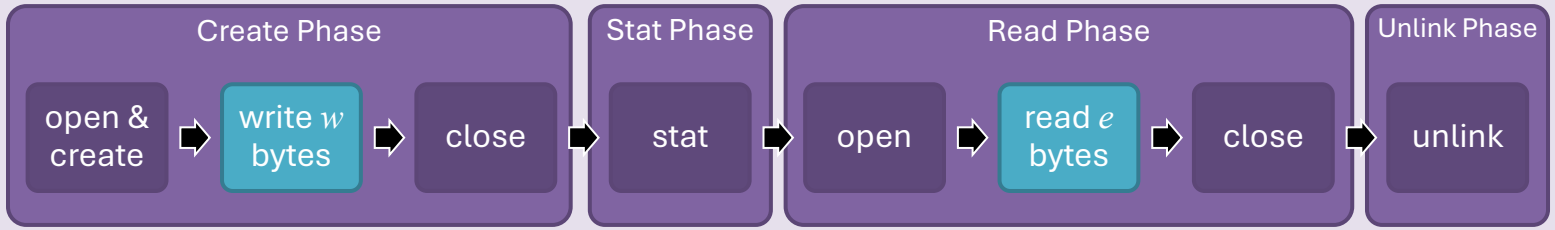


Other practical options

Leaf mode (-L)

- `-L -z 2 -b 2`
- Create files at deepest directories only
- Closer to some real datasets





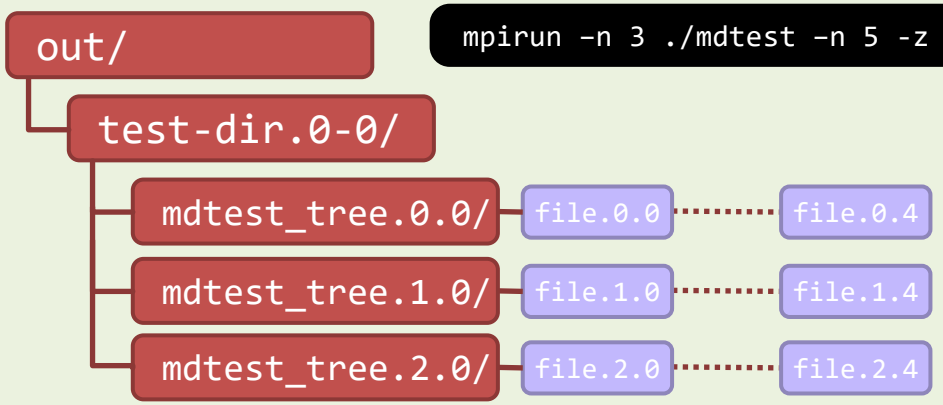
Perform I/O to files

- `-w` and `-e` write/read to each file
- `-w 4096 -e 4096` to create and read 4 KiB files
- Try using with data-in-inode

Directory-per-MPI rank (-u)

- Each MPI proc makes own directory for its files within tree
- Reduces directory locking
- Like file-per-proc in IOR

```
mpirun -n 3 ./mdtest -n 5 -z 0 -b 1 -u
```



mdtest Acceptance Tests
Lustre on NVMe - Purge performance


```
$ srun -N 1 -n 32 ./mdtest -n 93750 -F -C -u -d /lus/mdt0@/lus/mdt1 \
    -z 7 -b 3 -w 1048576

$ srun -N 1 -n 32 ./mdtest -n 93750 -F -r -u -d /lus/mdt0@/lus/mdt1 \
    -z 7 -b 3
```

Create 7-deep, 3-wide tree to approximate messiness of user scratch directories

Create and unlink in separate runs

Create 1 MiB files to unlink - because purging empty files is not realistic

Results:

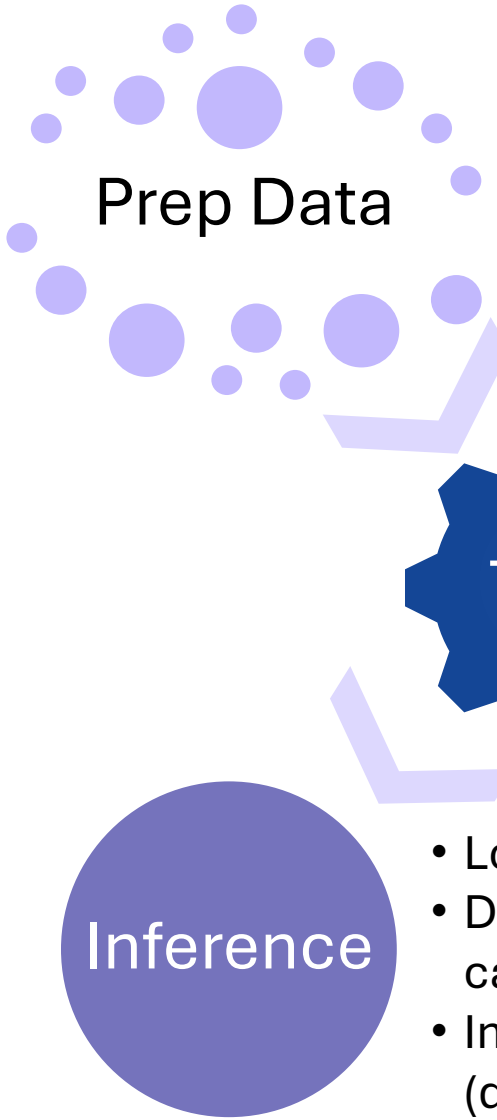
- 6,828 creates/sec (max)
- 70,546 unlinks/sec (max)

AI workloads

How HPC I/O principles apply to AI (and where they don’t!)

AI is the same as HPC *and* a whole separate tutorial

- Most principles for HPC still apply
  - Leverage parallelism
  - Avoid small I/Os
- But just like “HPC” isn’t one workload, “AI” isn’t either
  - Typically multi-stage, complex workflows
  - Training LLMs is different than training models for science

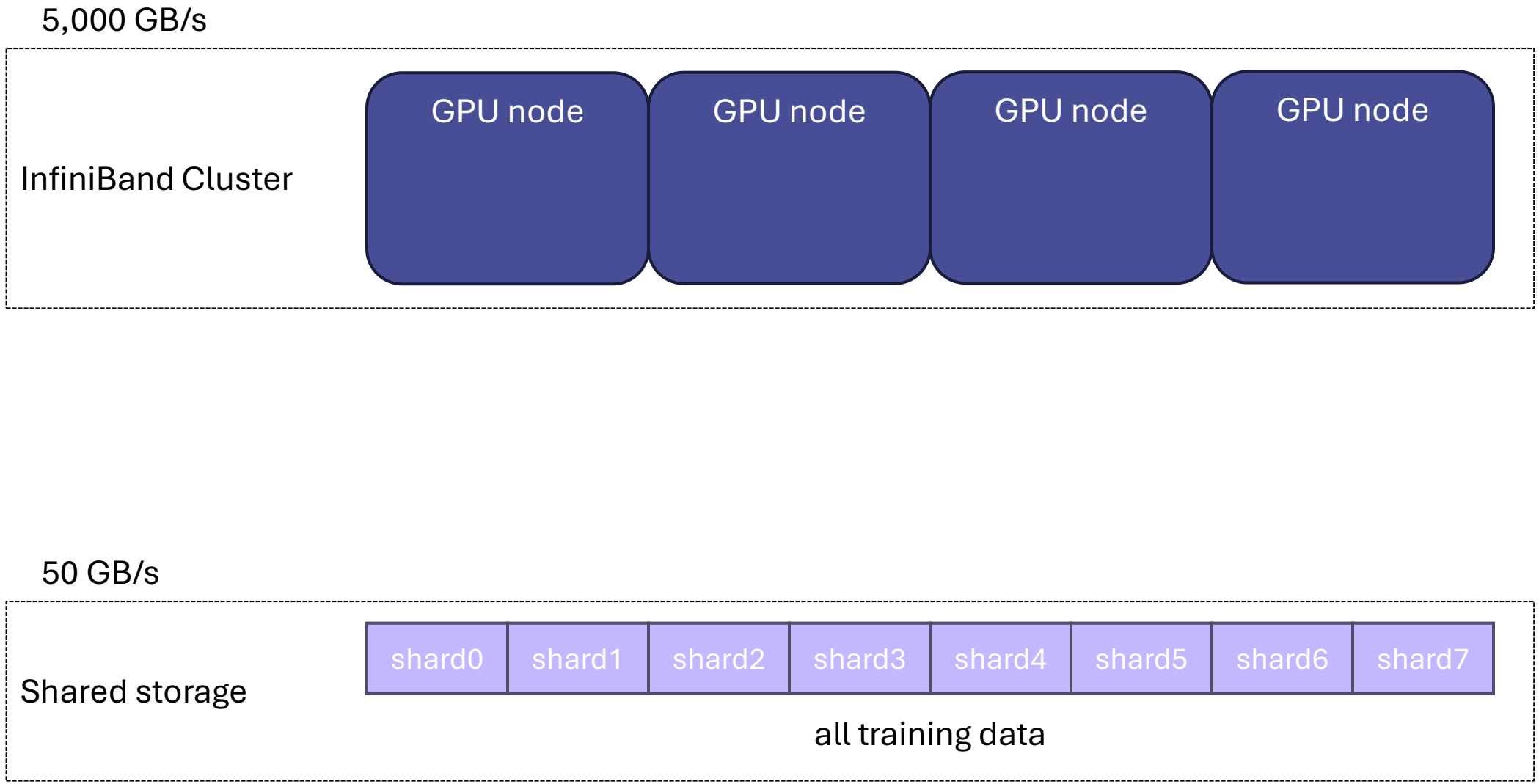


- Inputs: text, HDF5, FIT, …
- Resembles data analytics (Spark, Dask, etc)
- Outputs: Tokens (HDF5, Python Pickles)

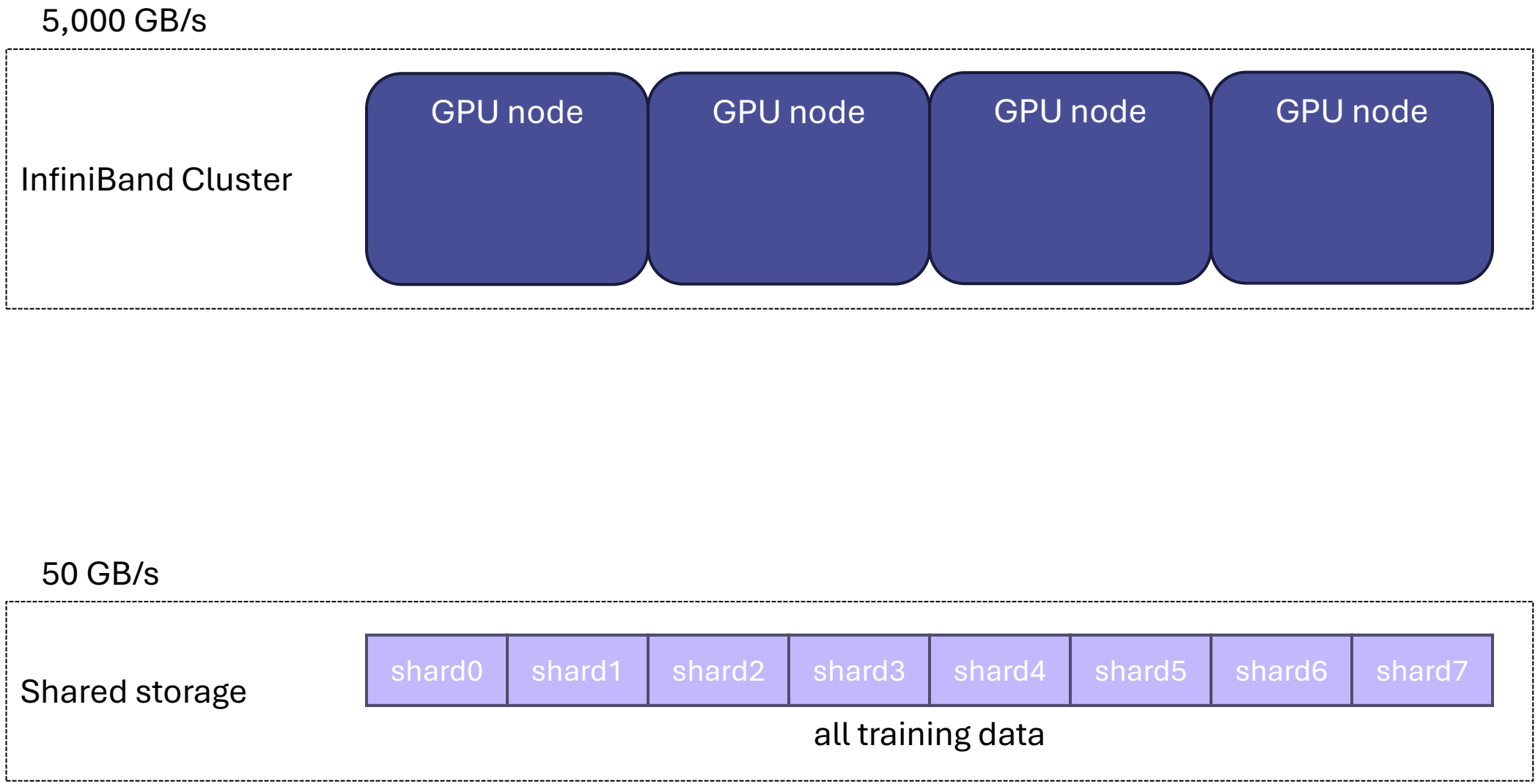
- Load data into GPU
- Checkpoint, restart model state

- Load once, inference many
- Domain-specific patterns (RAG, KV caches)
- Integrated with stateful services (databases, etc)

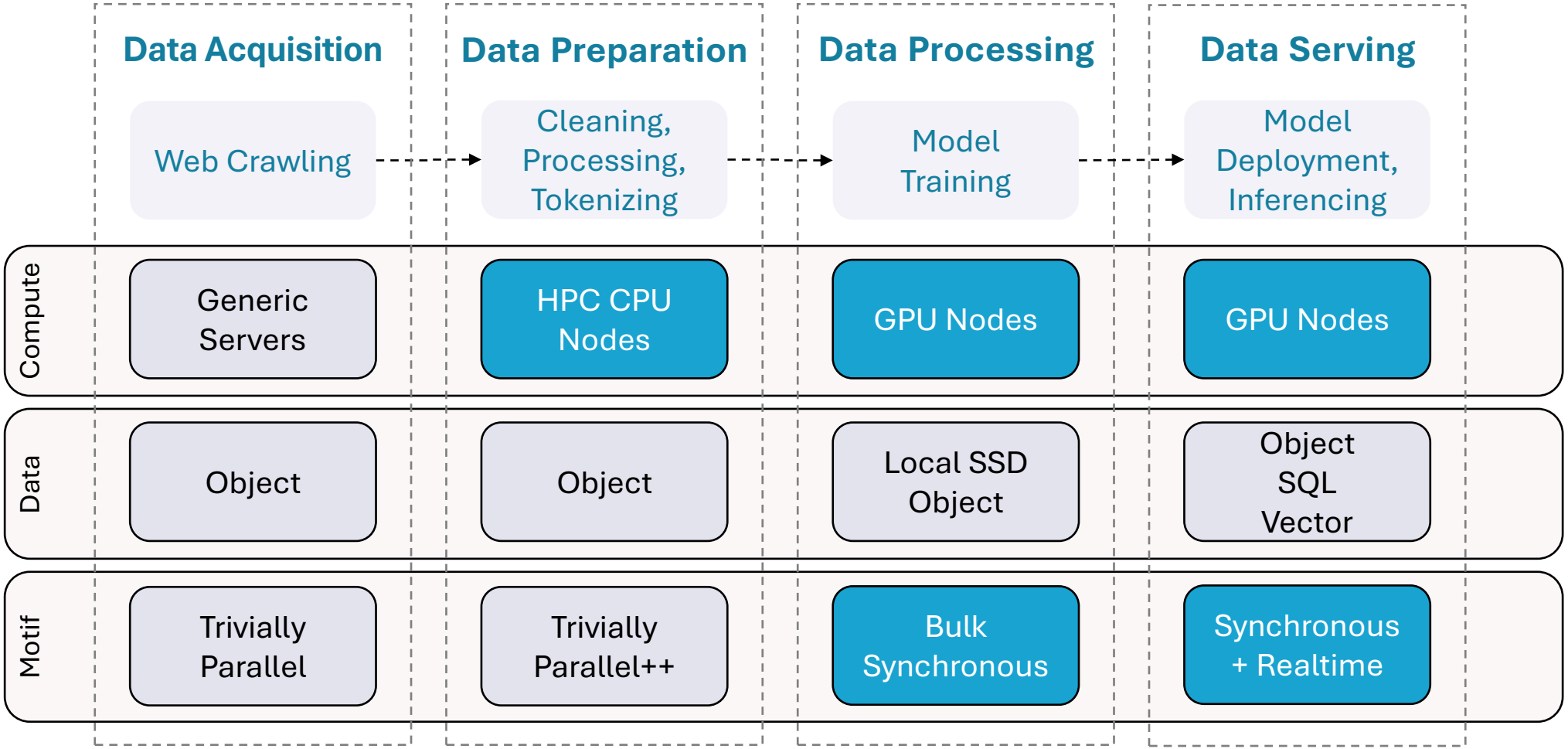
Loading training data the “HPC way”



Loading training data the “AI way”

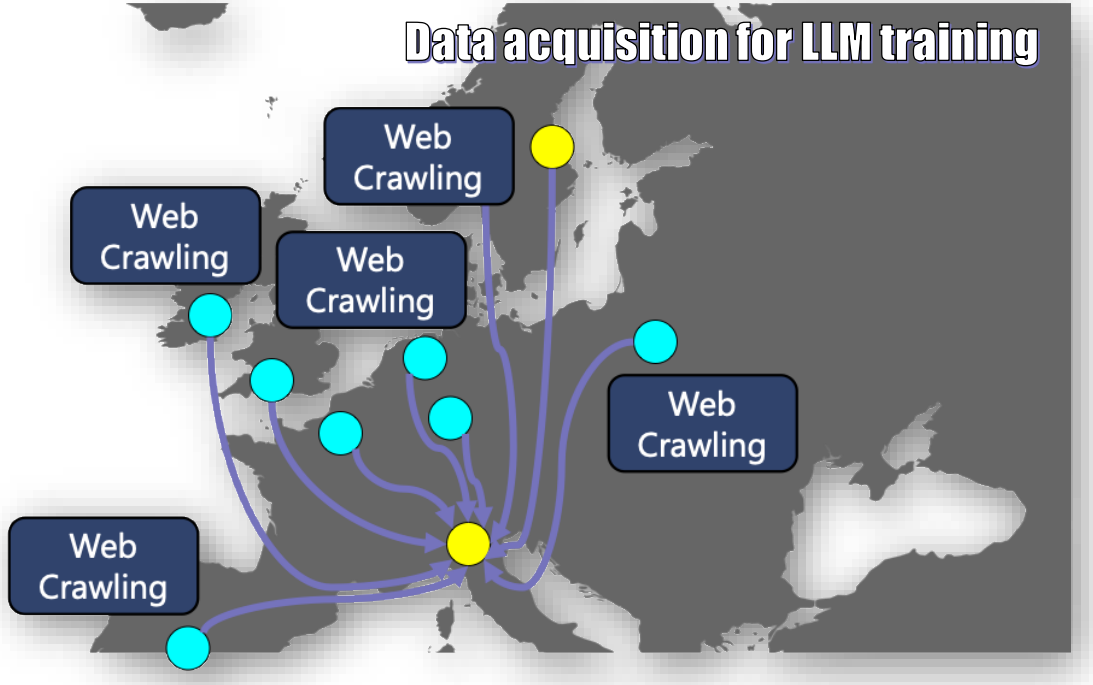


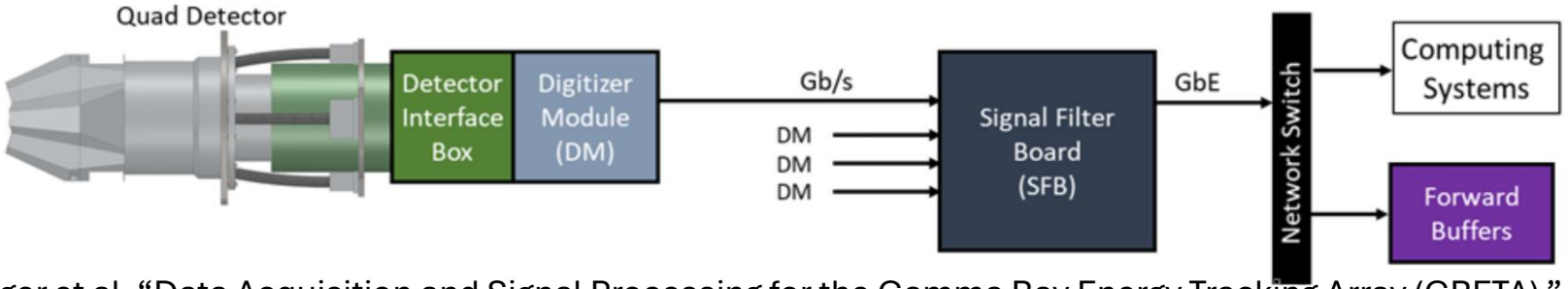
Each AI phase stresses storage differently



Data acquisition

- Domain-specific but conceptually similar
  - Parallel producers, one object store
  - Lightweight inline filtering
  - Trivially parallel, bandwidth-bound
- Examples:
  - LLMs: web crawling → object blobs.
  - Science: instruments or simulations streaming images or datasets.

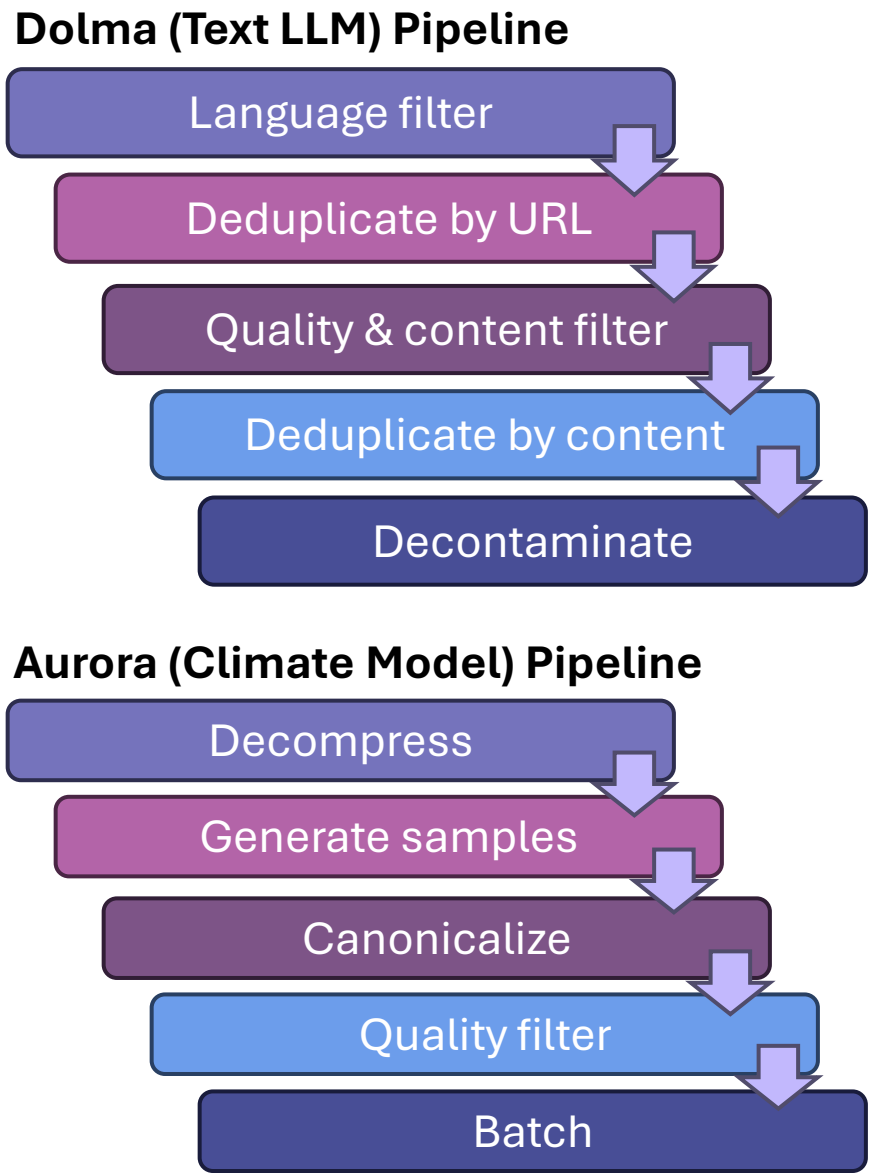




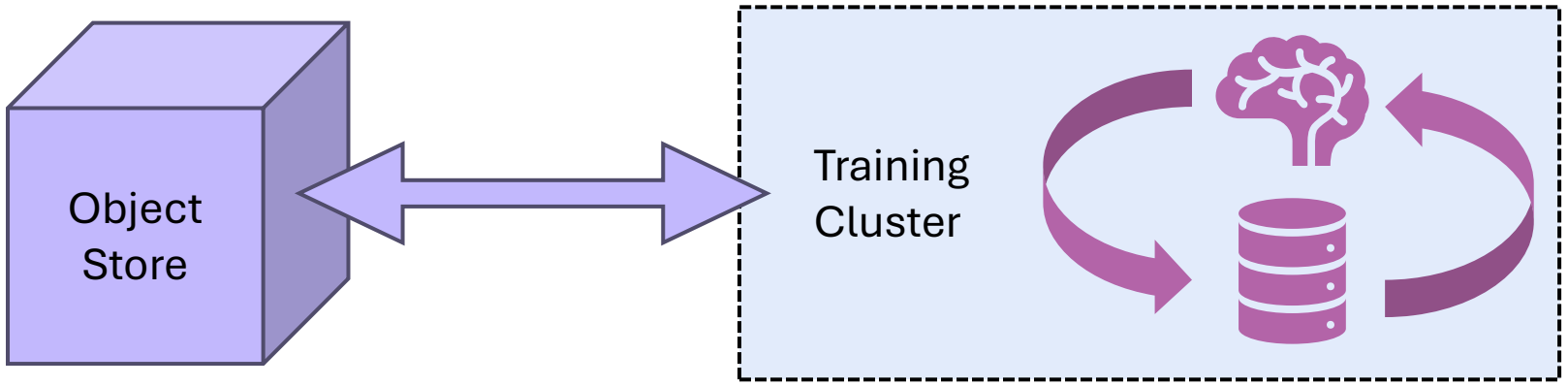
Stezelberger et al, "Data Acquisition and Signal Processing for the Gamma Ray Energy Tracking Array (GRETA)," arXiv preprint arXiv:2011.00129, Oct. 2020.

Data preparation

- Transform unstructured input into sharded, model-ready datasets
- Resembles large-scale analytics:
  - Embarrassingly parallel
  - Read large shards, process, output smaller shards
- Examples:
  - LLMs: Dolma (see https://arxiv.org/abs/2402.00159)
  - Science: Aurora (see https://www.nature.com/articles/s41586-025-09005-y)

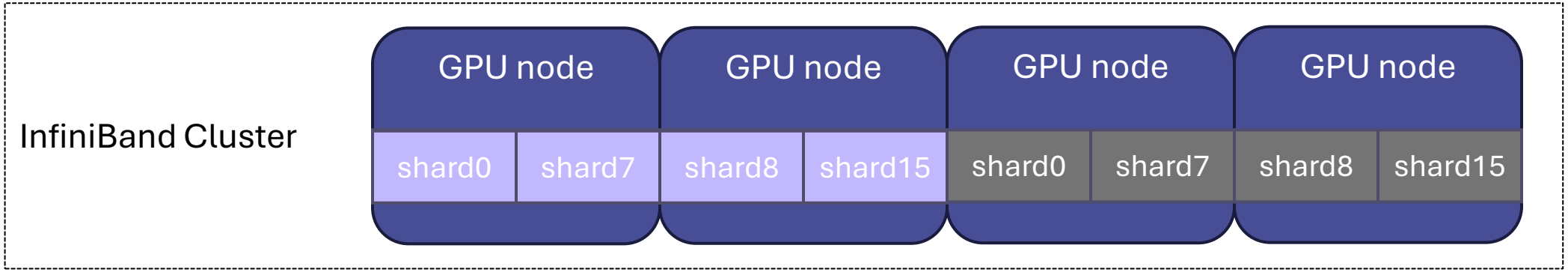


Data processing (training)



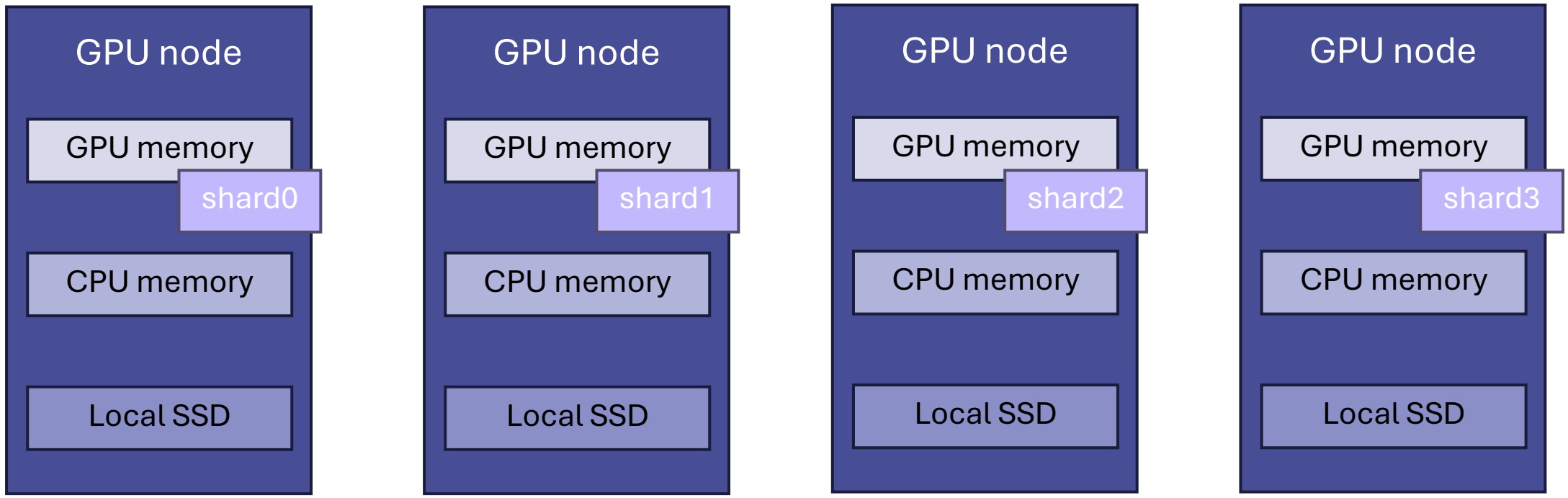
- Bulk-synchronous compute, asynchronous I/O
  - Within the training loop, I/O is localized
  - At the boundaries, I/O is asynchronous and hierarchical
- Examples:
  - Stage-in
  - Async checkpoints
- Some aspects are domain-specific
  - Transformers: GPUs process inputs at ~10 MB/s → high-compute architecture
  - CNNs: GPUs process inputs at ~1 GB/s → low-compute architecture

Checkpointing the “HPC way”



Shared Storage (PFS or object store)

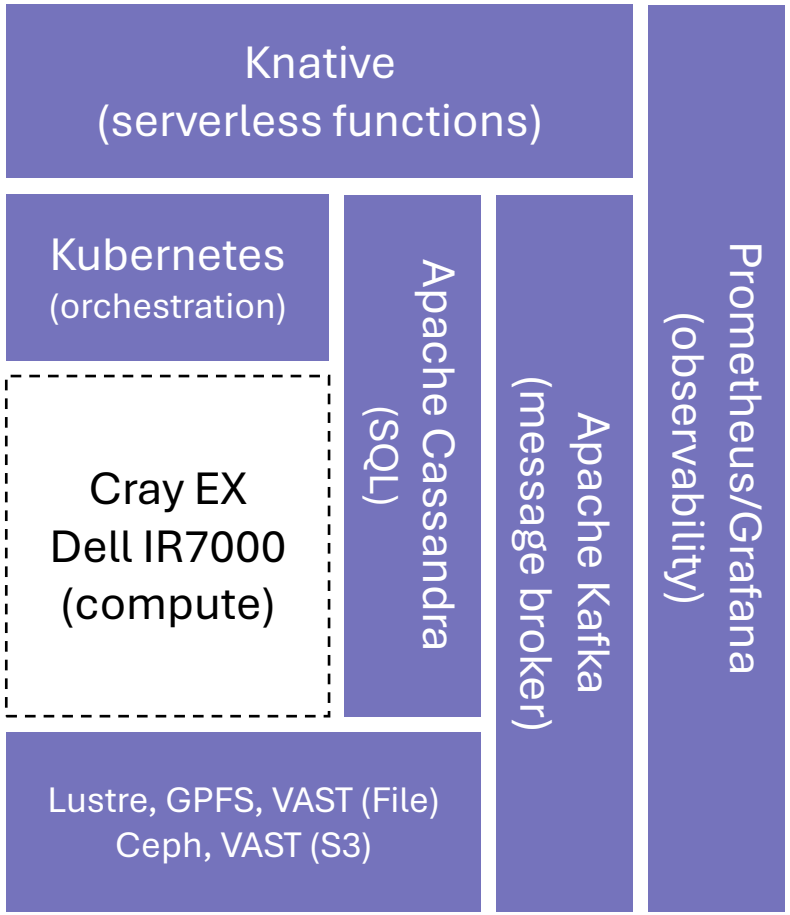
Checkpointing the “AI way” - hierarchically and asynchronously



Data serving (inference)

- Inference can be *synchronous and real-time*
  - Optimization goal becomes *latency*, not bandwidth
  - LLMs: Interactive users waiting for response
  - Science: Simulation or experiment waiting for response
- Inference can be *offline and batch-oriented*
  - Resembles standard HPC postprocessing
  - LLMs: Reinforcement learning
  - Science: Image reconstruction
- Both may require significant non-HPC service infrastructure
  - LLMs: benefit from KV caches, vector databases
  - Science models: databases and message buses attached to supercomputers

Data serving infrastructure



Principles of I/O apply equally to HPC and AI

- Most principles for HPC still apply
  - Leverage parallelism
  - Avoid small I/Os
- Domain-specific optimizations exist
  - Localize synchronous I/O in inner compute loops
  - Asynchronous I/O at the boundaries of compute
- Don’t over-index on HPC vs. AI
- Don’t over-index on LLMs vs. scientific models