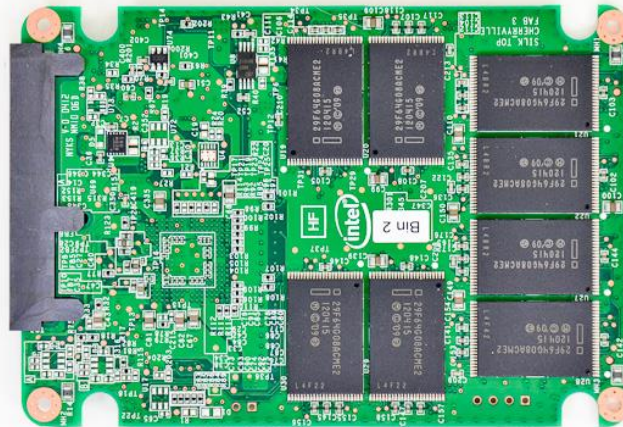
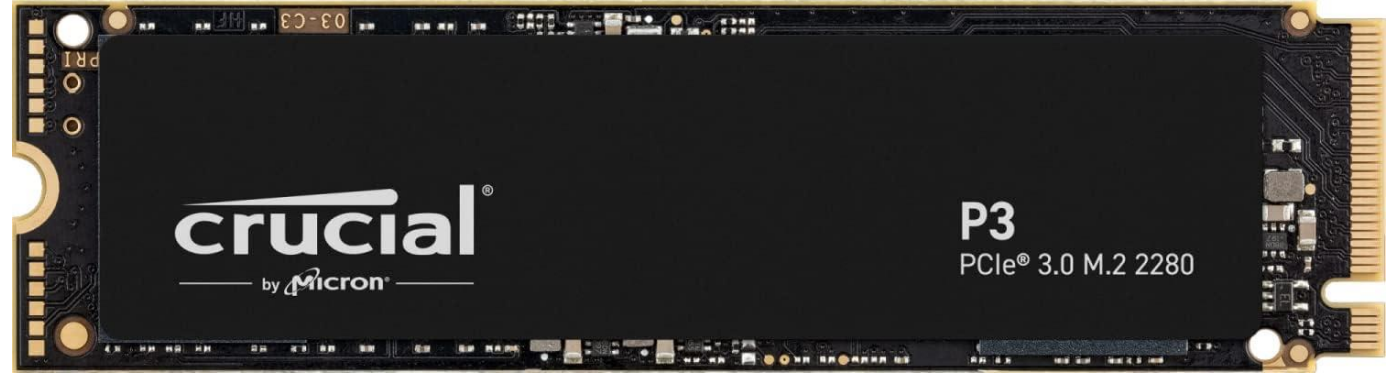


HIGH PERFORMANCE I/O

Hardware and Software interfaces



Storage



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

- Different ways of defining how inodes → blocks, and how directories, filenames, etc... are structured
 - As well as alternative approaches (i.e. log-structured filesystem)
 - Extended functionality (replication, distribution, backup, erasure coding, etc...)
- These are what differentiate filesystems, i.e.:
 - ext*: ext3, ext4
 - xfs
 - zfs
 - btrfs
 - etc...
- Maybe be important for performance or required functionality but the default can be used by most

Parallel filesystems

- Build on local filesystem but provide
 - Aggregated distributed local filesystem
 - Custom approach to define how inodes → blocks, and how directories, filenames, etc... are structured
 - Relaxed consistency (potentially) for concurrent writing
- i.e. Lustre:
 - Open-source parallel file system
 - Three main parts
 - Object Storage Servers (OSS)
 - Store data on one or more Object Storage Targets (OST)
 - The OST handles interaction between client data request and underlying physical storage
 - OSS typically serves 2-8 targets, each target a local disk system
 - Capacity of the file system is the sum of the capacities provided by the targets (roughly)
 - The OSS operate in parallel, independent of one another
 - Metadata Target (MDT)
 - One(ish) per filesystem
 - Storing all metadata: filenames, directories, permissions, file layout
 - Stored on Metadata Server (MDS)
 - Clients
 - Supports standard POSIX access

POSIX I/O

- Standard interface to files
 - Linux approach
 - Based on systems with single filesystem
 - open, close, write, read, etc...
- 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

- 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

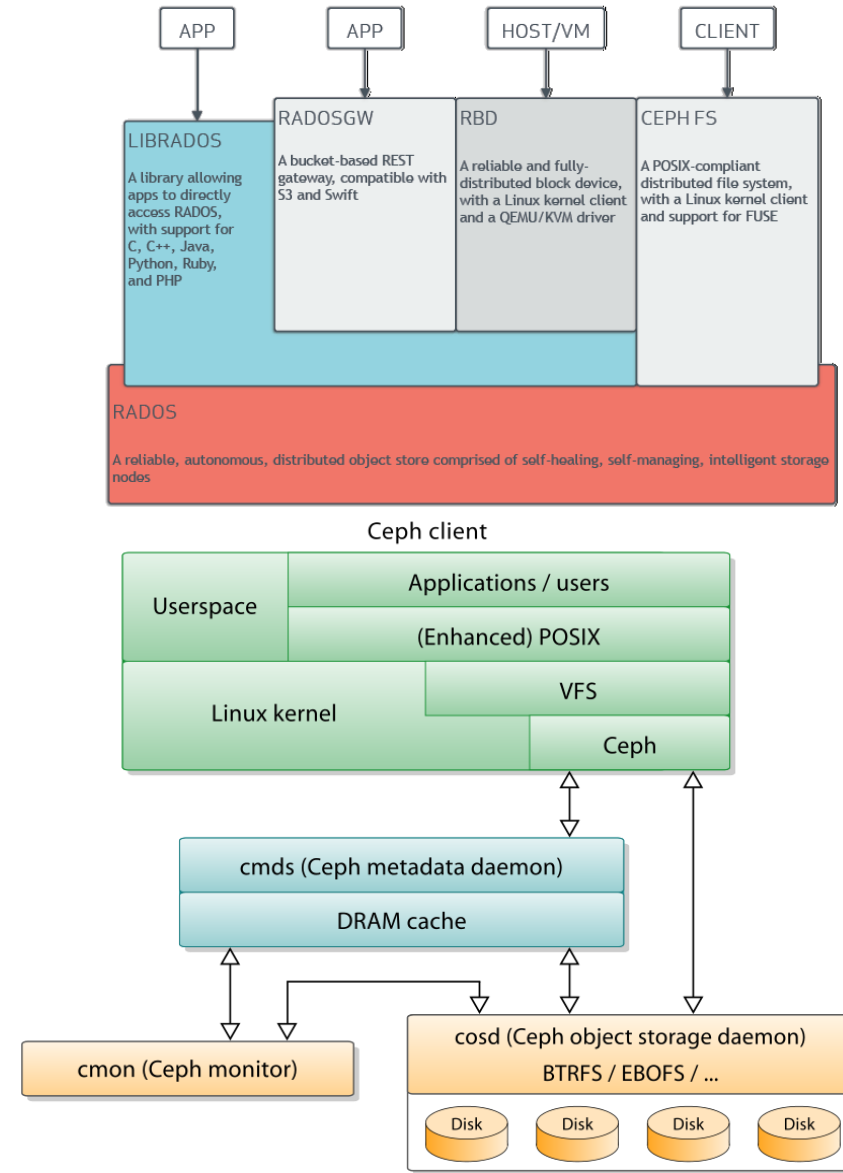
- Objects are combination of data and metadata
- Metadata is name – value pair (key) identifying the object
 - Default has some other information as well:
 - Date last modified
 - HTTP Content-Type
 - Version (if enabled)
 - Access Control List (if configured)
 - Can add custom metadata
- Data
 - An 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

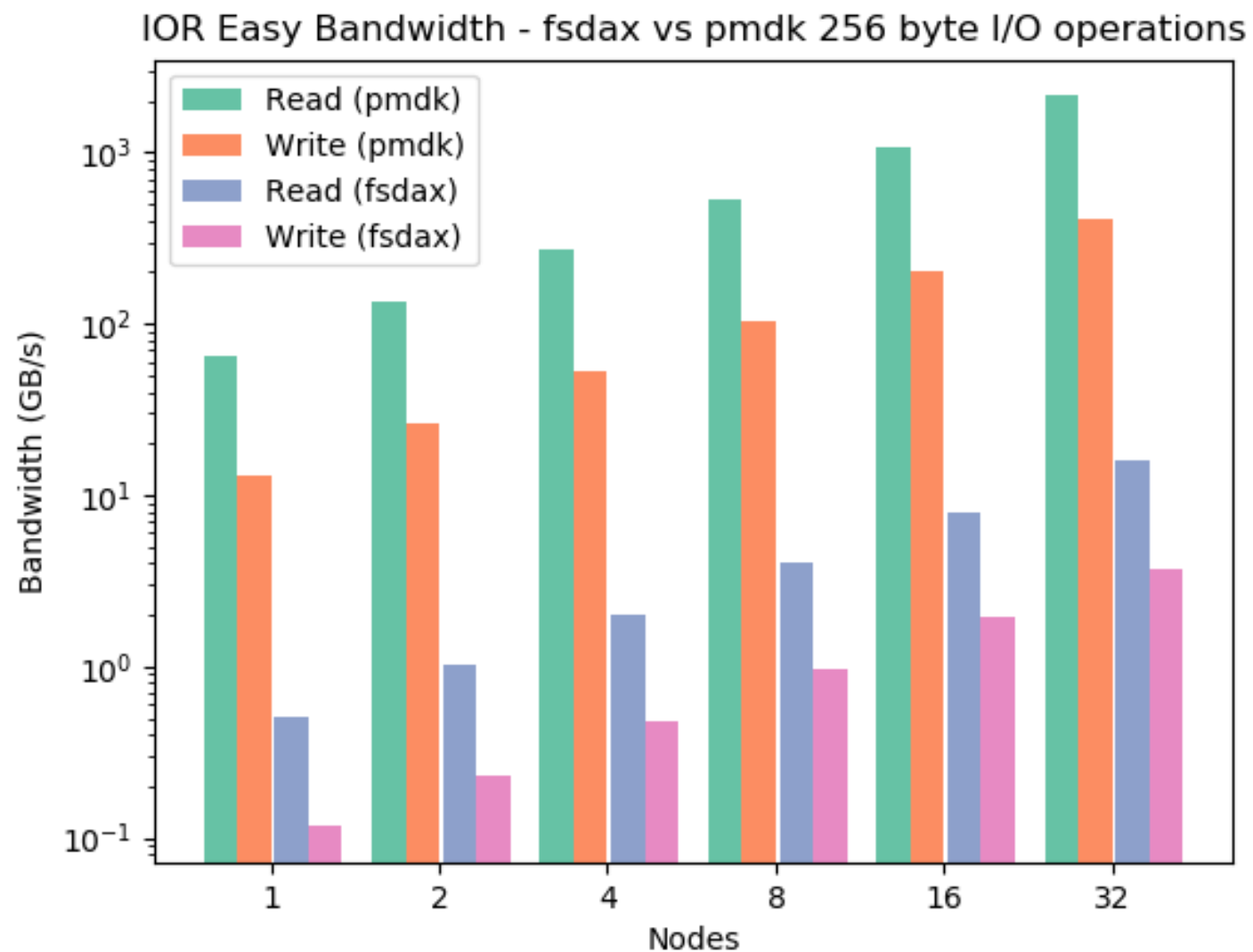
- Widely used object store from academic storage project
- Designed to support multiple targets
 - Traditional object store: RadosGW → S3 or Swift
 - Block interface: RBD
 - Filesystem: Ceph FS
 - Lower-level object store: LibRados
- Distributed/replicated functionality
 - Scale out by adding more Ceph 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



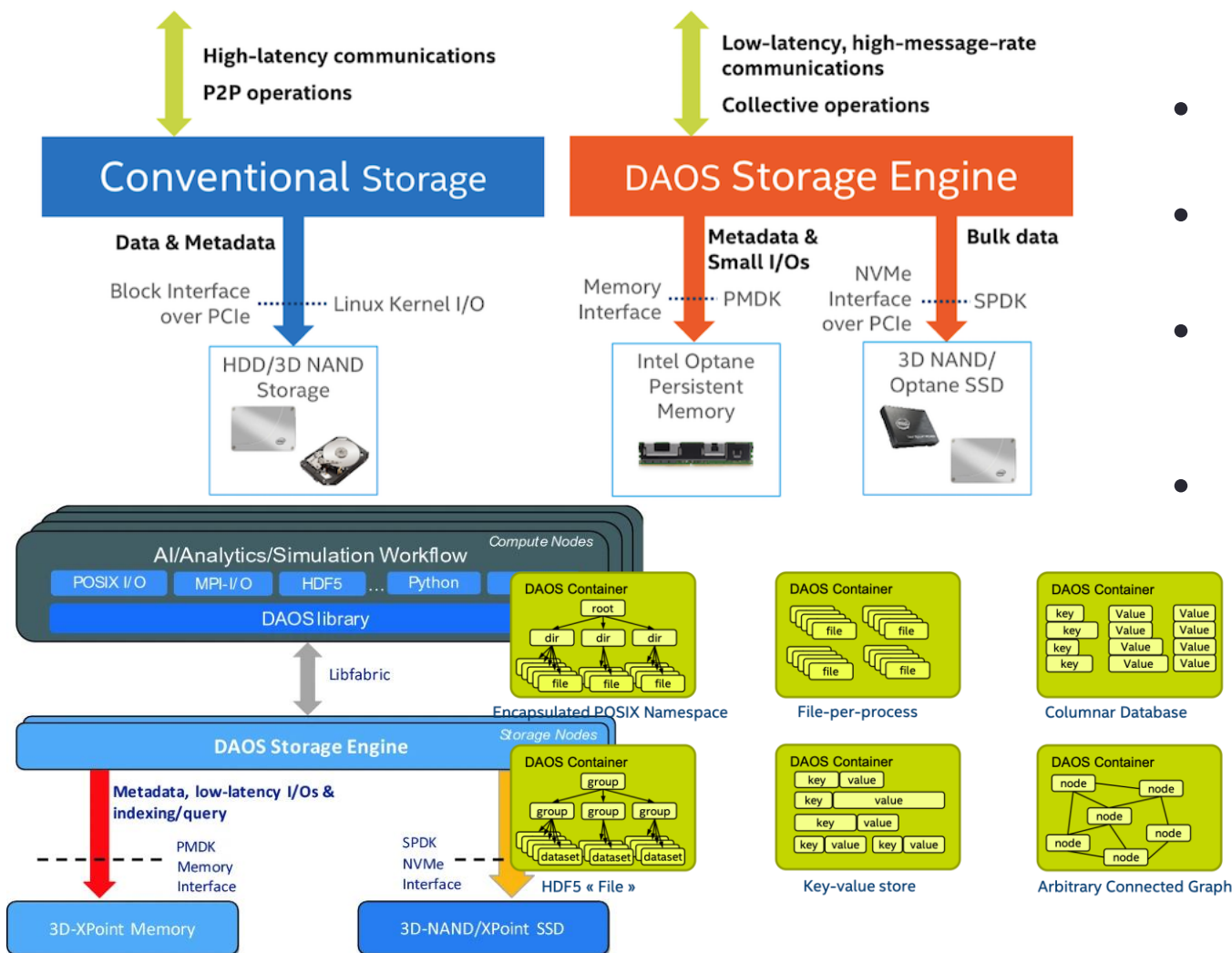
NVRAM



Optane



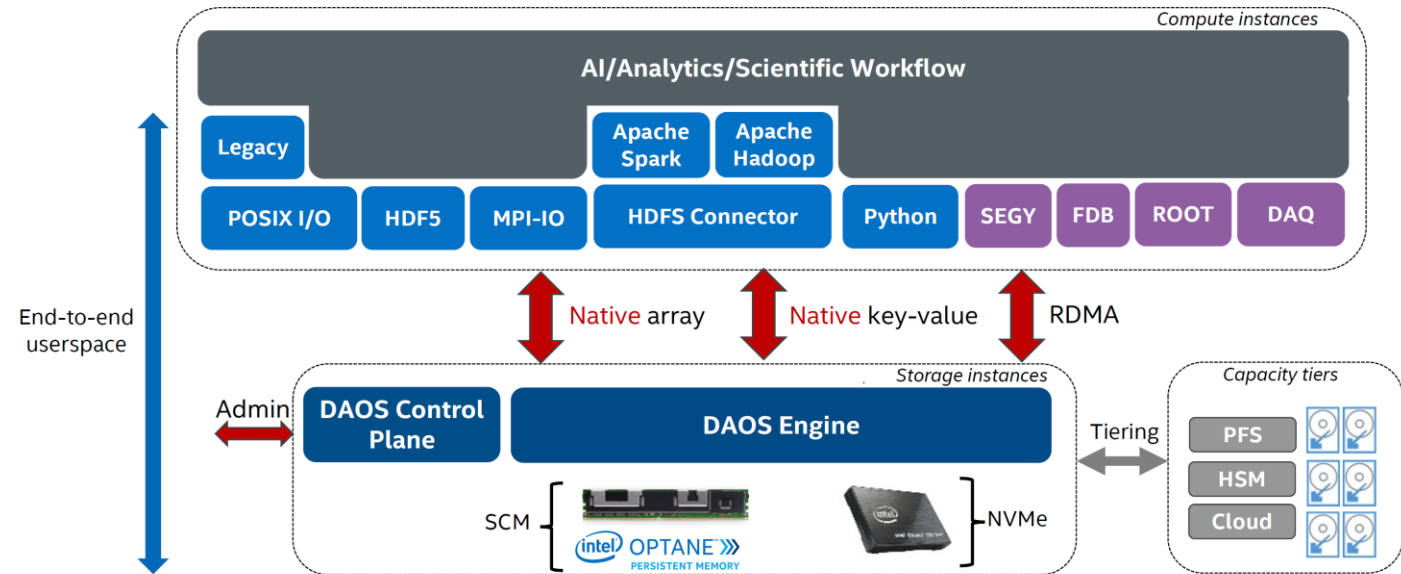
DAOS



- 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.

DAOS

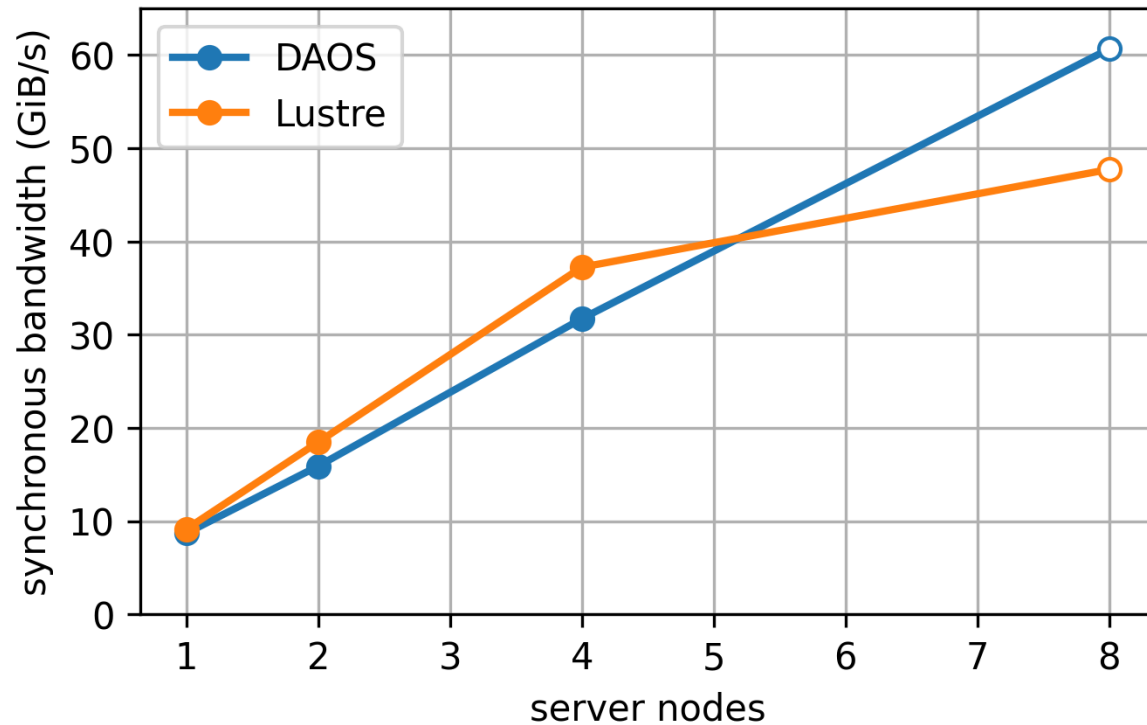
- Range of storage interfaces
 - Native object store (libdaos)
 - Filesystem (various approaches)
 - Raw block device
 - MPI-I/O (ROMIO)
 - HDF5
 - PyDAOS
 - Spark/Hadoop
 - TensorFlow I/O
- DAOS systems built from DAOS servers
 - One per socket, has own NVMe and NVRAM
 - Scale system by adding more servers (in node or across nodes)
 - Metadata and data entirely distributed/replicated (no metadata centralisation)
 - RAFT-approach used for consensus across servers



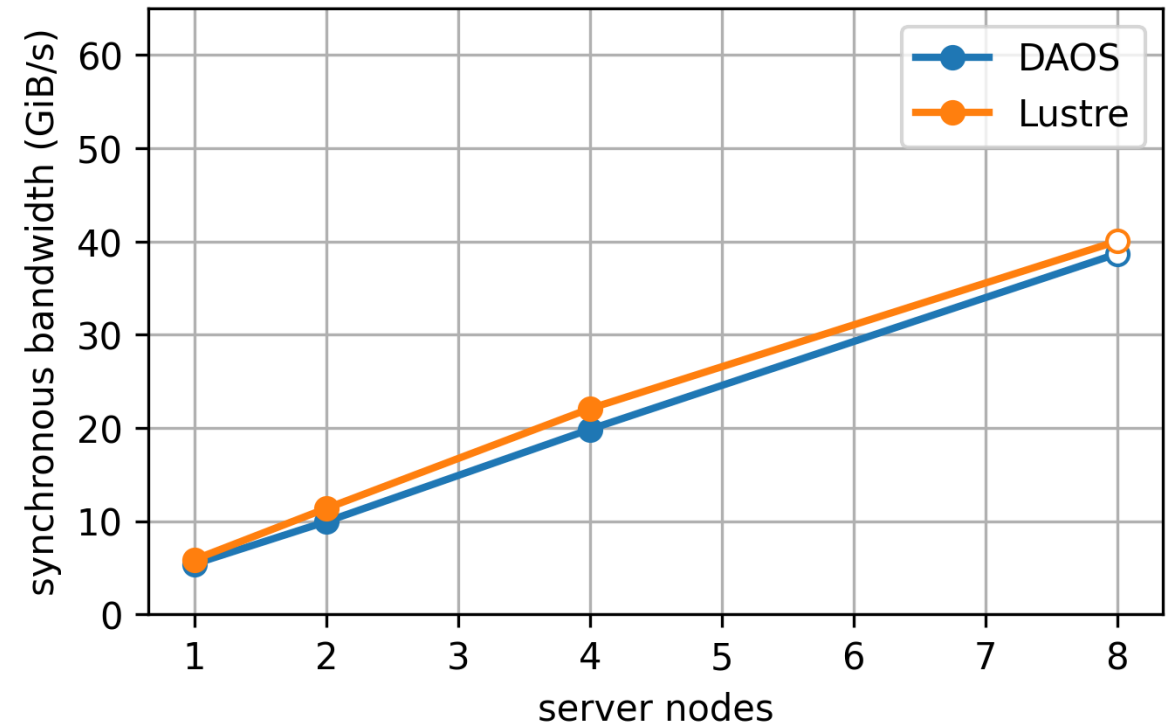
DAOS Performance

- Comparing Lustre and DAOS on the same hardware
 - IOR bulk synchronous I/O

Read Bandwidth

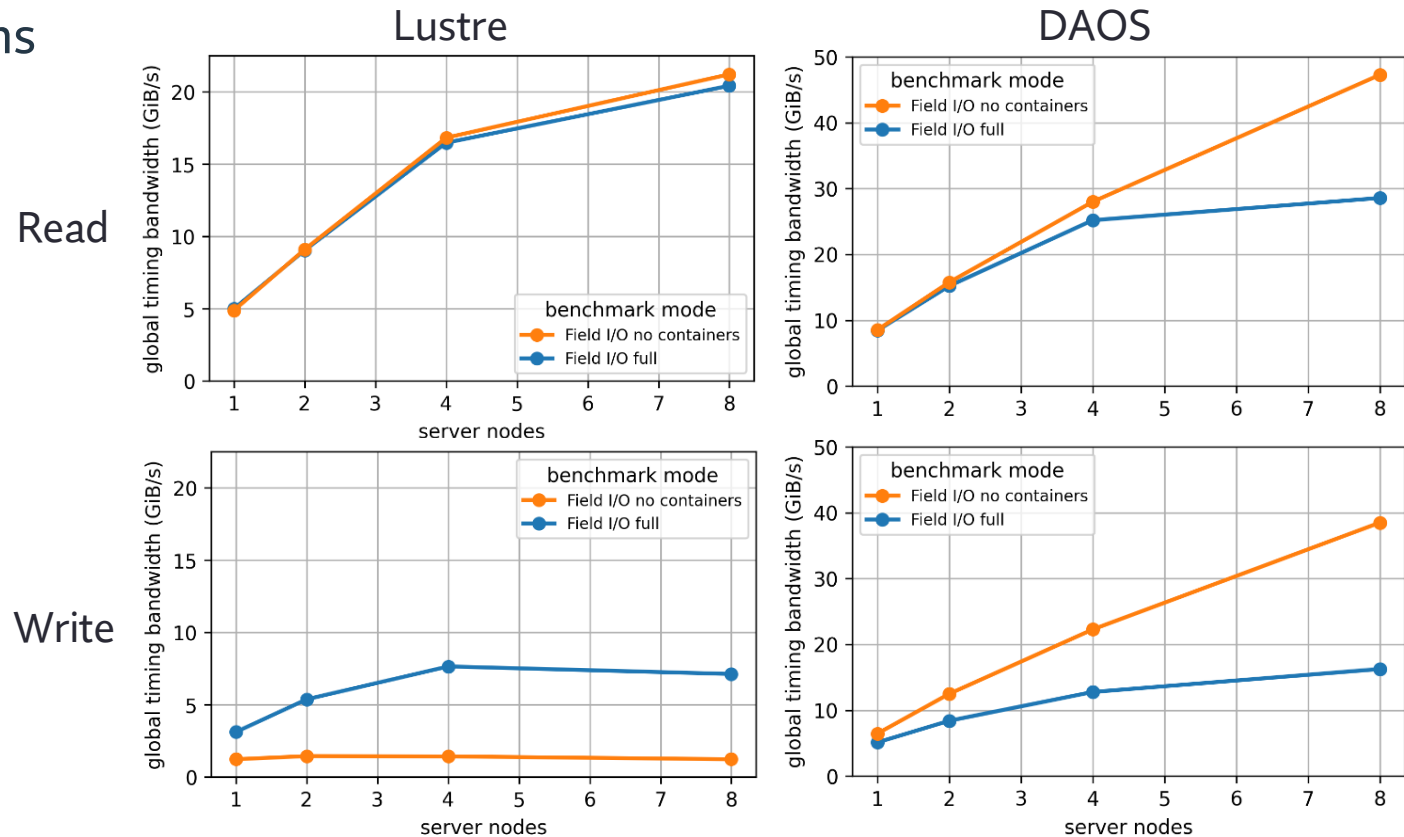
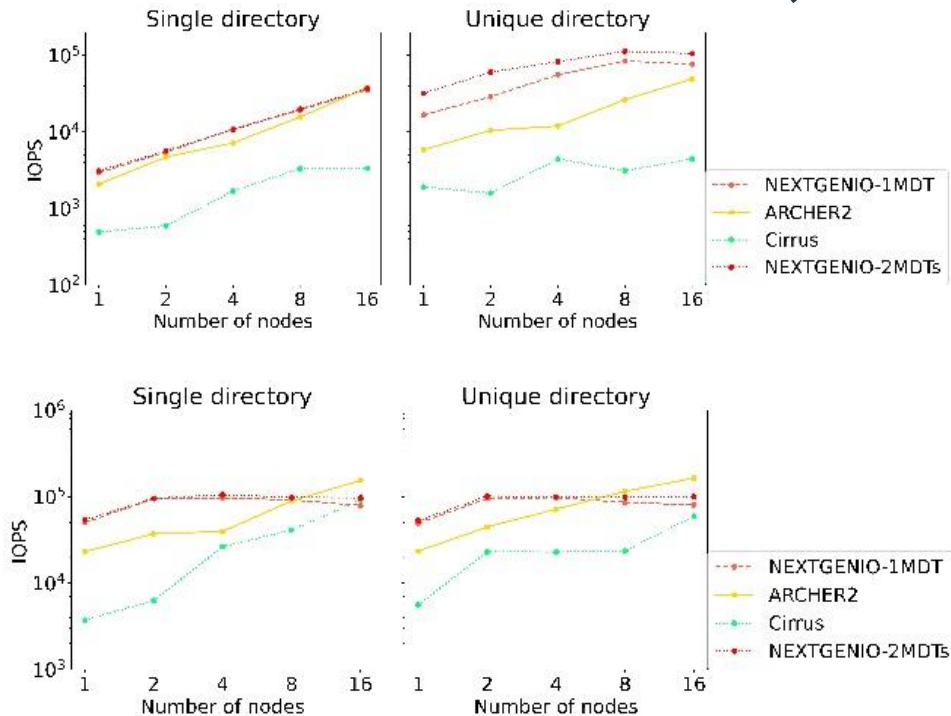


Write Bandwidth



DAOS performance

- Separate read and write steps
 - More “object like” access patterns
 - Weather field -> Object or file

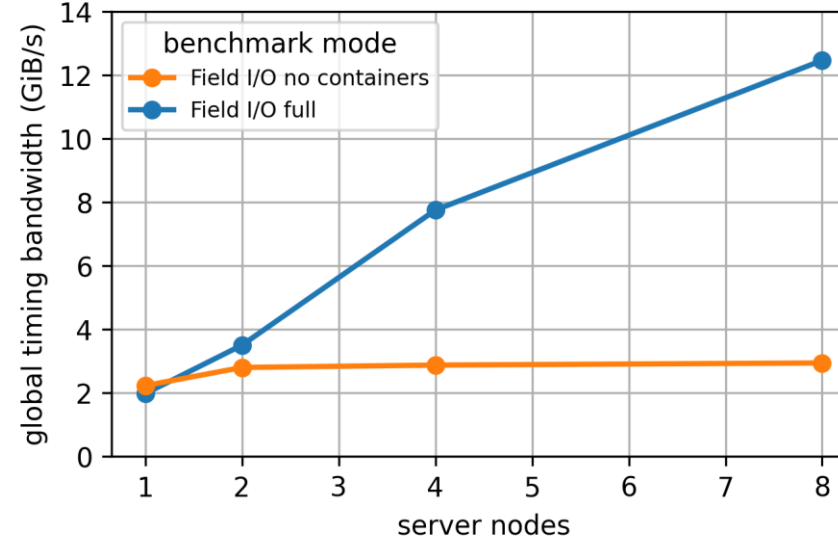


DAOS performance

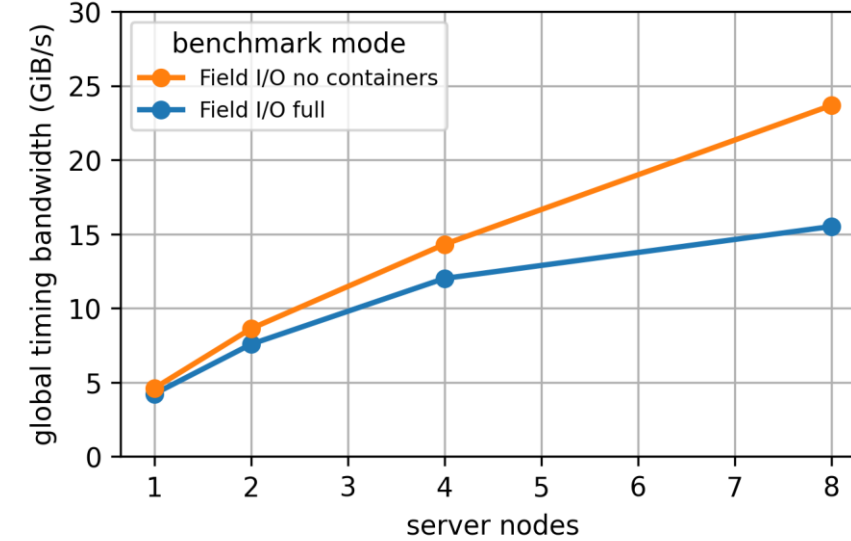
- Contending read and write workers
 - Containers represent filesystem or object store structure

Read

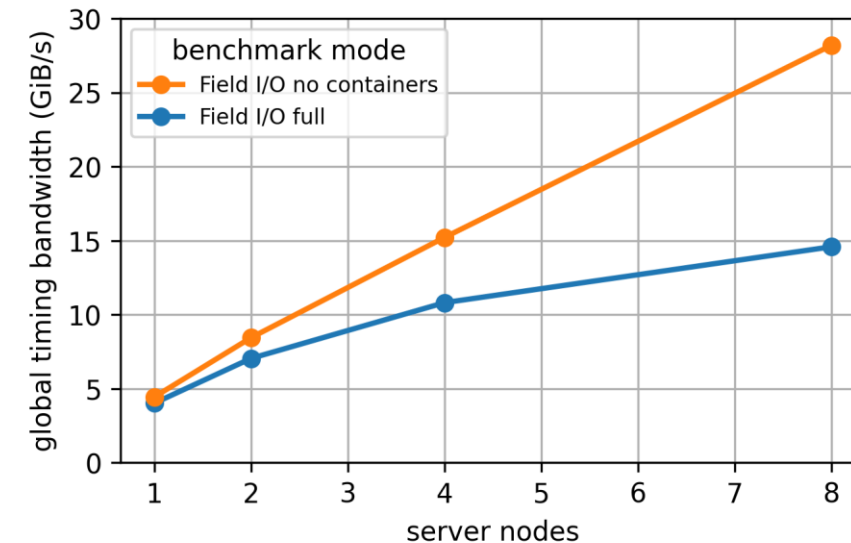
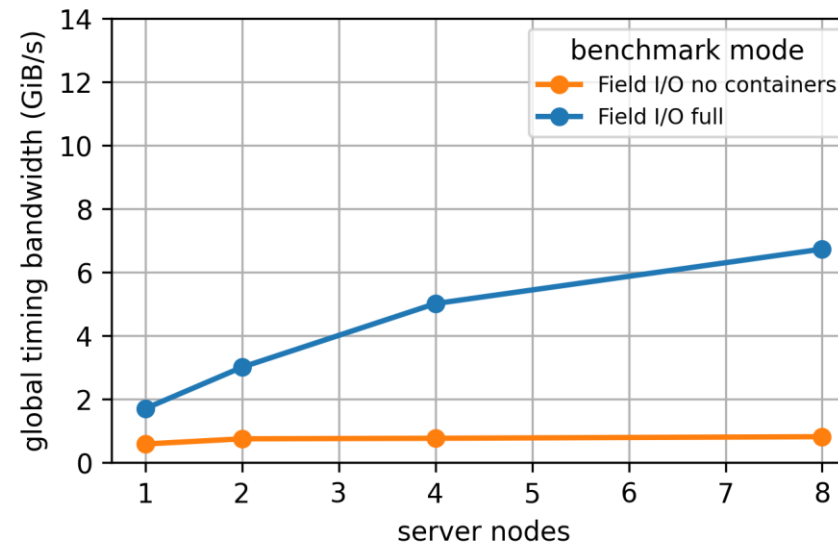
Lustre



DAOS



Write



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

- 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/ObjectStoreTutorial/Exercises/exercisesheet.pdf>
- Take IOR source code
- Run on the GCP system
- SSH to 34.29.249.155
- You will get a username
 - tuXXX
 - And an ssh key

