TUTORIAL(ISH)

Exploring high performance object storage using DAOS

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

Slides and results borrowed from
Nicolau Manubens
ECMWF
Mohamad Chaarawi
Intel

epcc

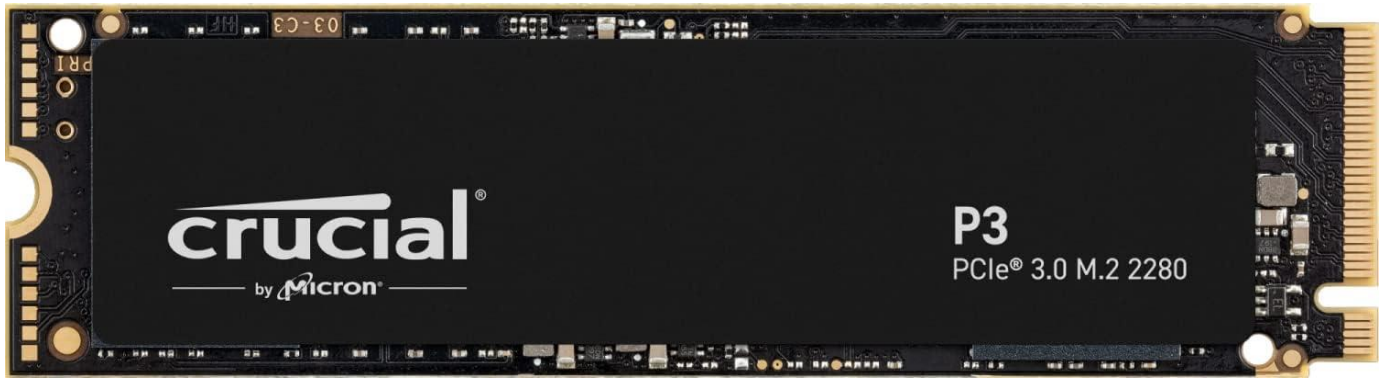
Aims

- Understand object storage vs filesystem storage
- Learn about DAOS
- Learn about DAOS APIs and exploiting DAOS from your applications
- Understand performant ways to exploit DAOS

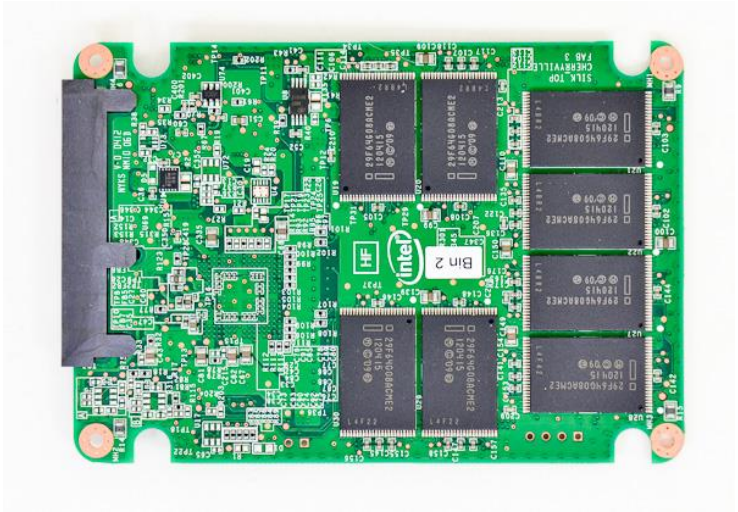
Aims cont.

- Understand data movement and think about application data requirements
- Thinking about different ways you undertake I/O or storing data
- Move beyond bulk, block-based, I/O paradigms

Storage


crucial
by Micron
P3
PCIe® 3.0 M.2 2280


crucial
2.5-inch SOLID STATE DRIVE


intel

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

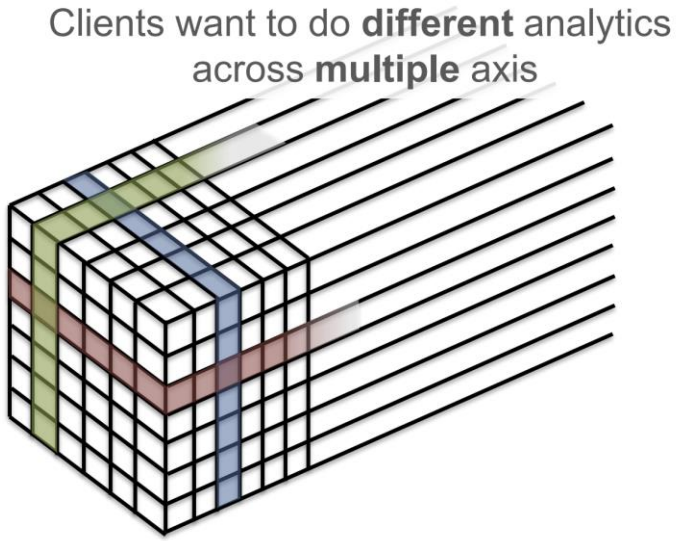
Parallel filesystems

- 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

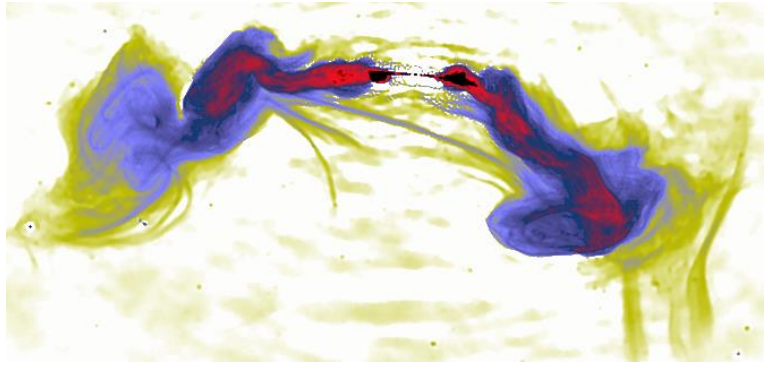
Object storage

- 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

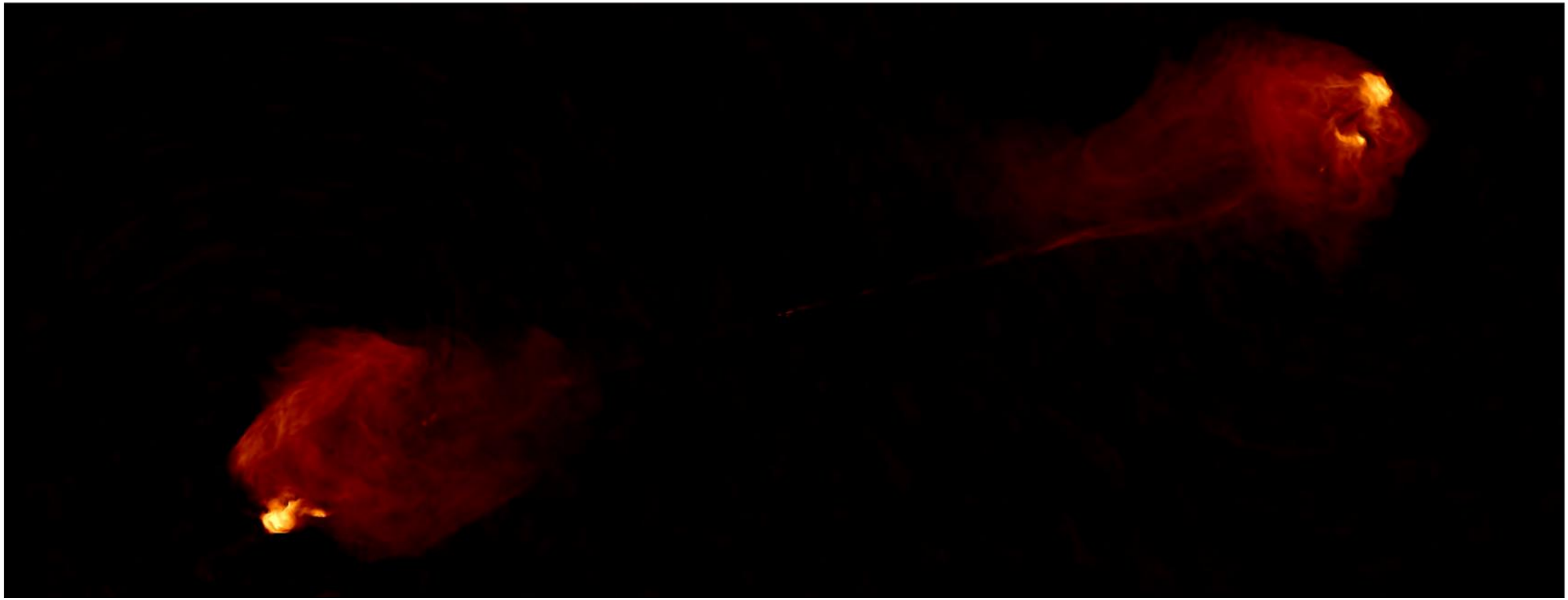


- Originally designed for unstructured data sets
  - Bunch of data with no specific hierarchy required
- Can also enable efficient/fast access to data in different structures
  - Supports different creation, querying, analysis, and use patterns
- Granular storage with rich metadata
  - Data retrieval leverages metadata
  - **Build structure on the fly**
- 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 in dimension from production workloads

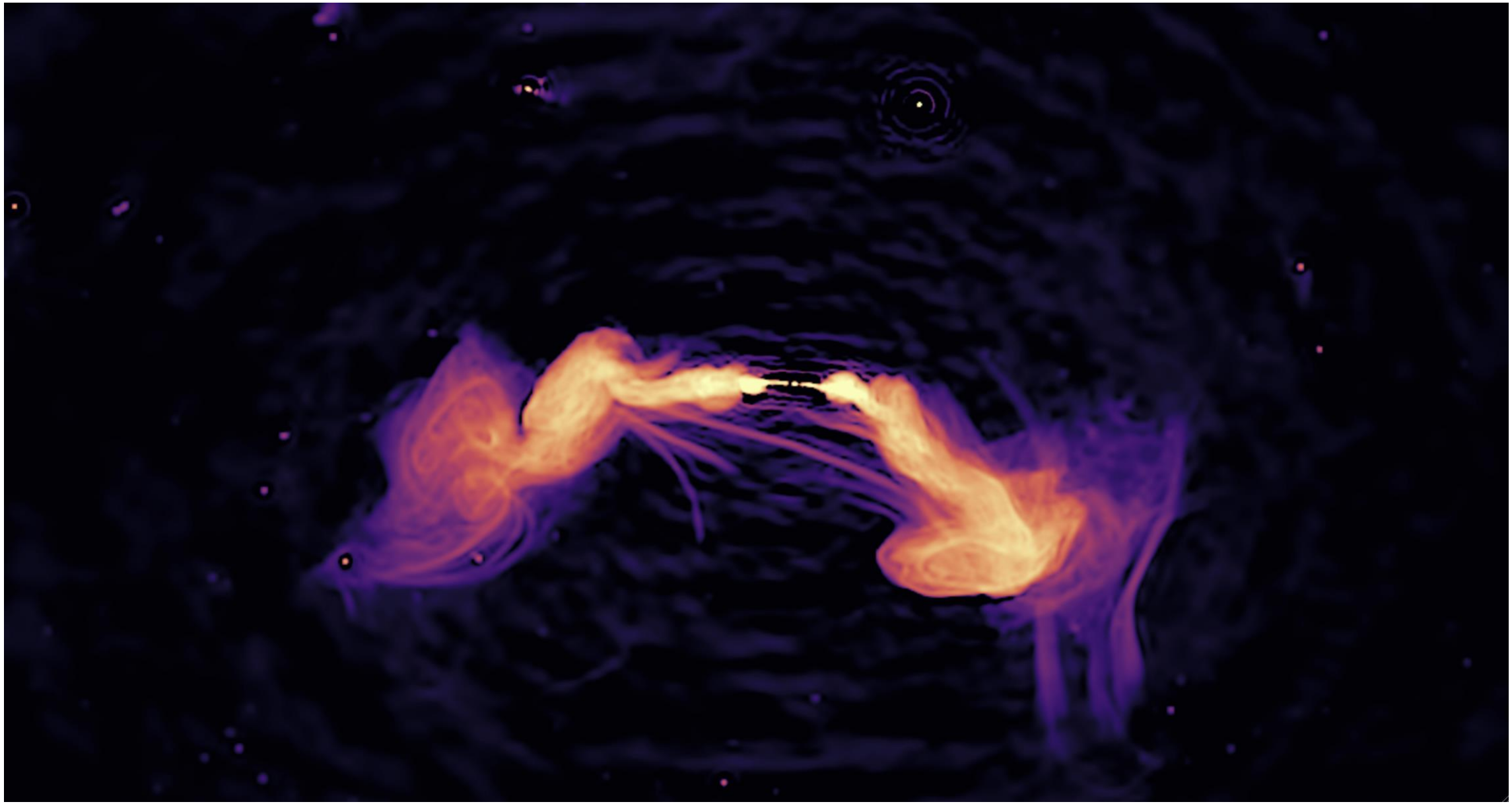


- Radio astronomy
  - Data collected and stored by antenna (frequency and location) and capture time
  - Reconstruction of images done in time order
  - Evaluation of transients or other phenomenon undertaken across frequency and location

epcc



epcc



epcc

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 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

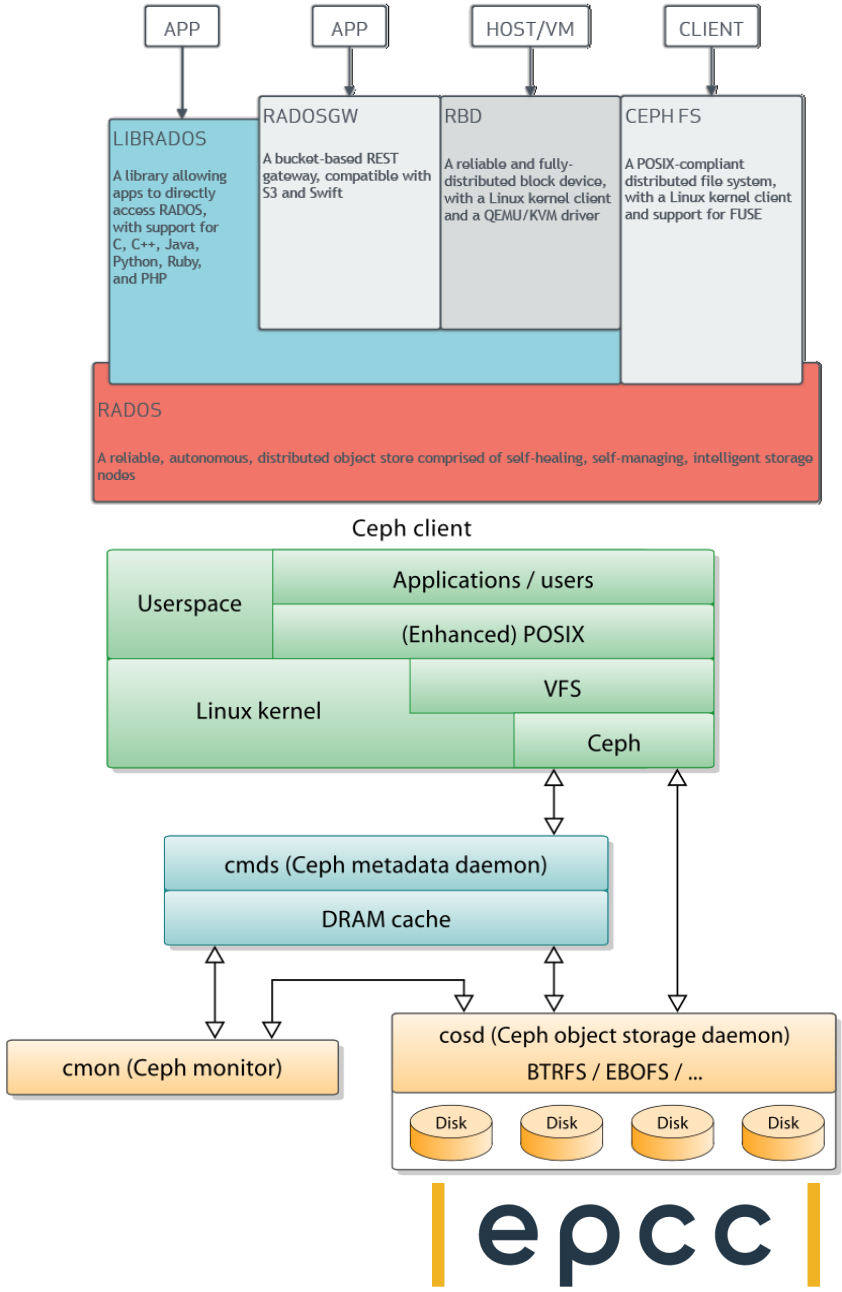
- Objects are combination of data and metadata
- Metadata is name – value pair (key) identifying the object
  - Default has some other information as well:
    - Date last modified
    - HTTP Content-Type
    - Version (if enabled)
    - Access Control List (if configured)
  - Can add custom metadata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

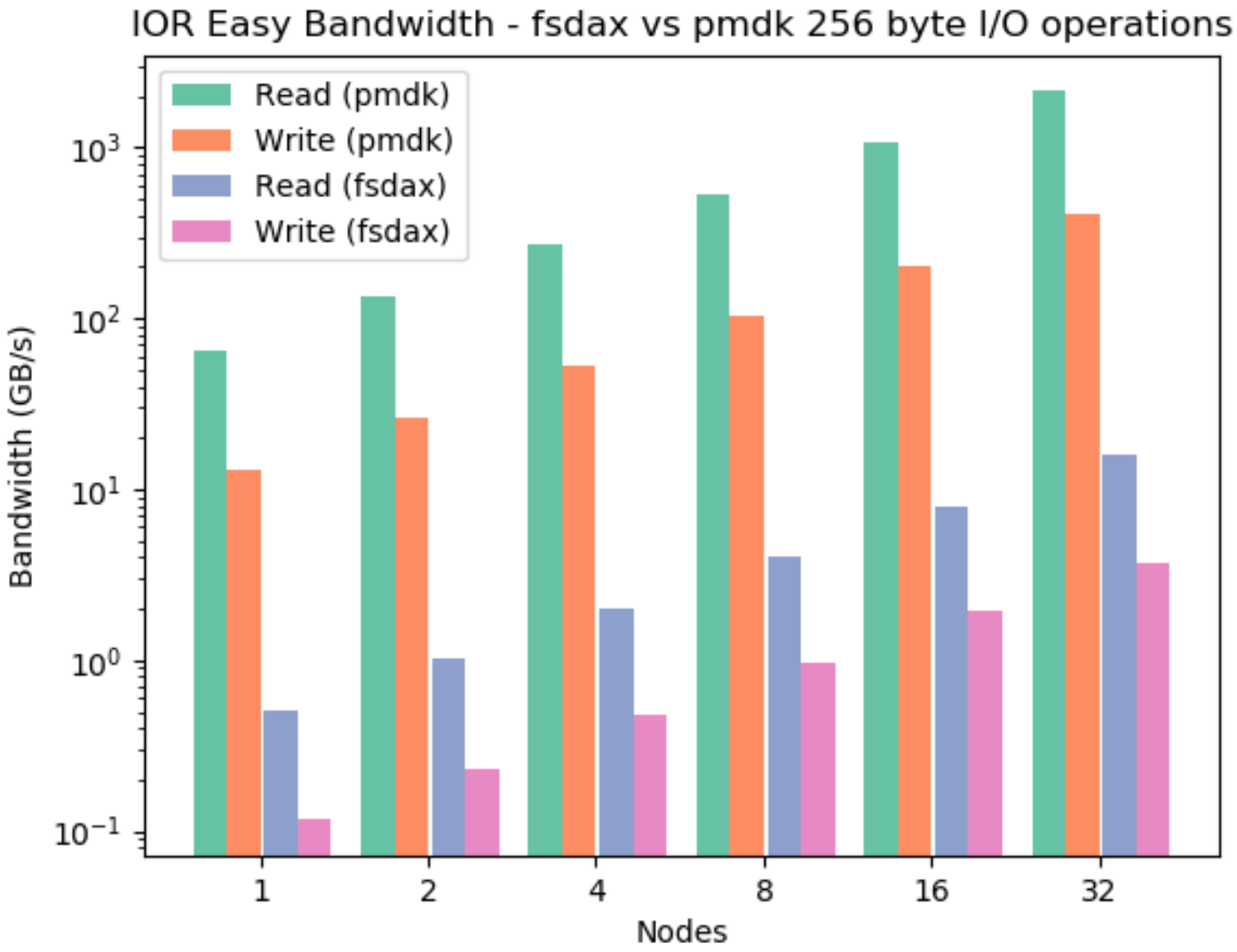


epcc

NVRAM

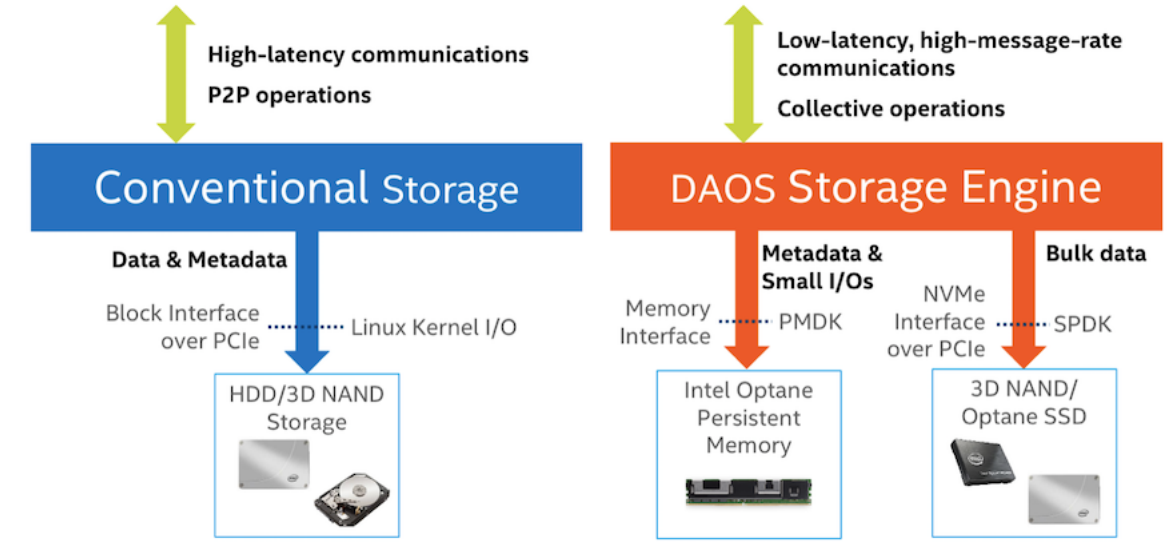

Assembled in Taiwan
NMA1XXD128GPS
128GB
Warranty Void if Label Removed
SN:8089A2221000000293D
PN:100-586-613-00

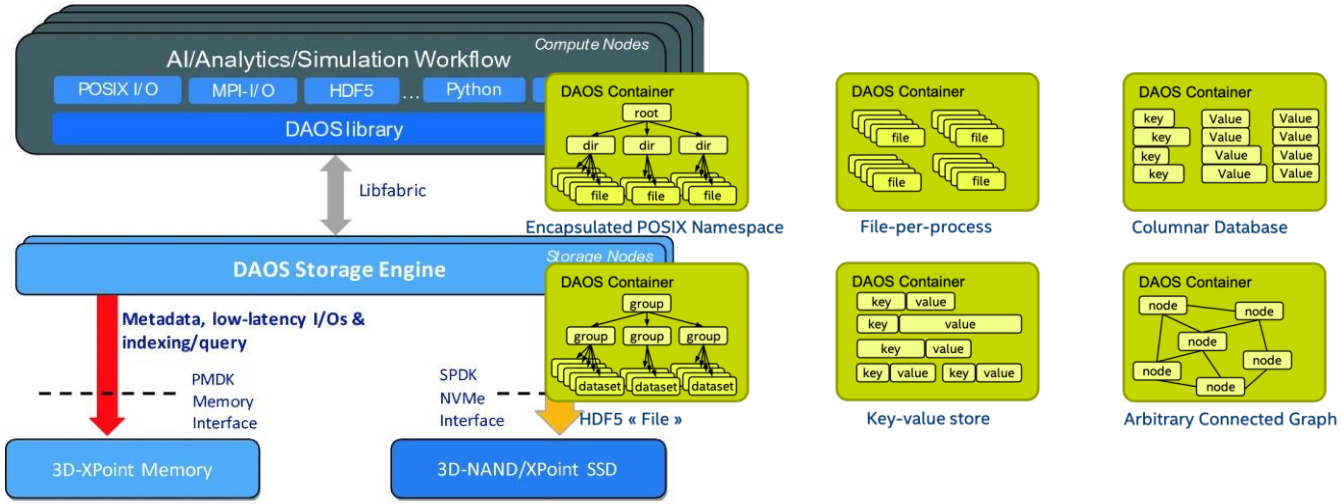
Optane


IOR Easy Bandwidth - fsdax vs pmdk 256 byte I/O operations
Read (pmdk)
Write (pmdk)
Read (fsdax)
Write (fsdax)
Bandwidth (GB/s)
10^3
10^2
10^1
10^0
10^{-1}
1
2
4
8
16
32
Nodes

epcc

DAOS



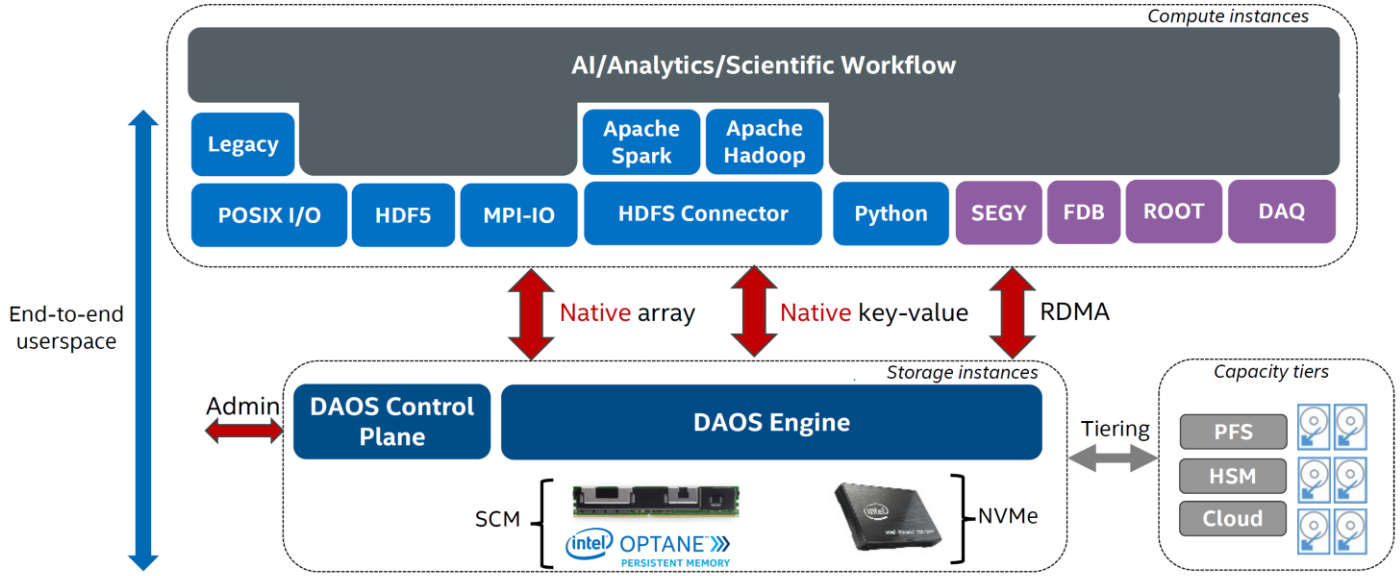


- Native object store on non-volatile memory and NVMe devices and designed for HPC
- Pools
  - Define hardware range of data
- Containers
  - User space and data configuration definitions
- Objects
  - **Multi-level key-array** API is the native object interface with locality
  - **Key-value** API provides 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.

epcc

DAOS

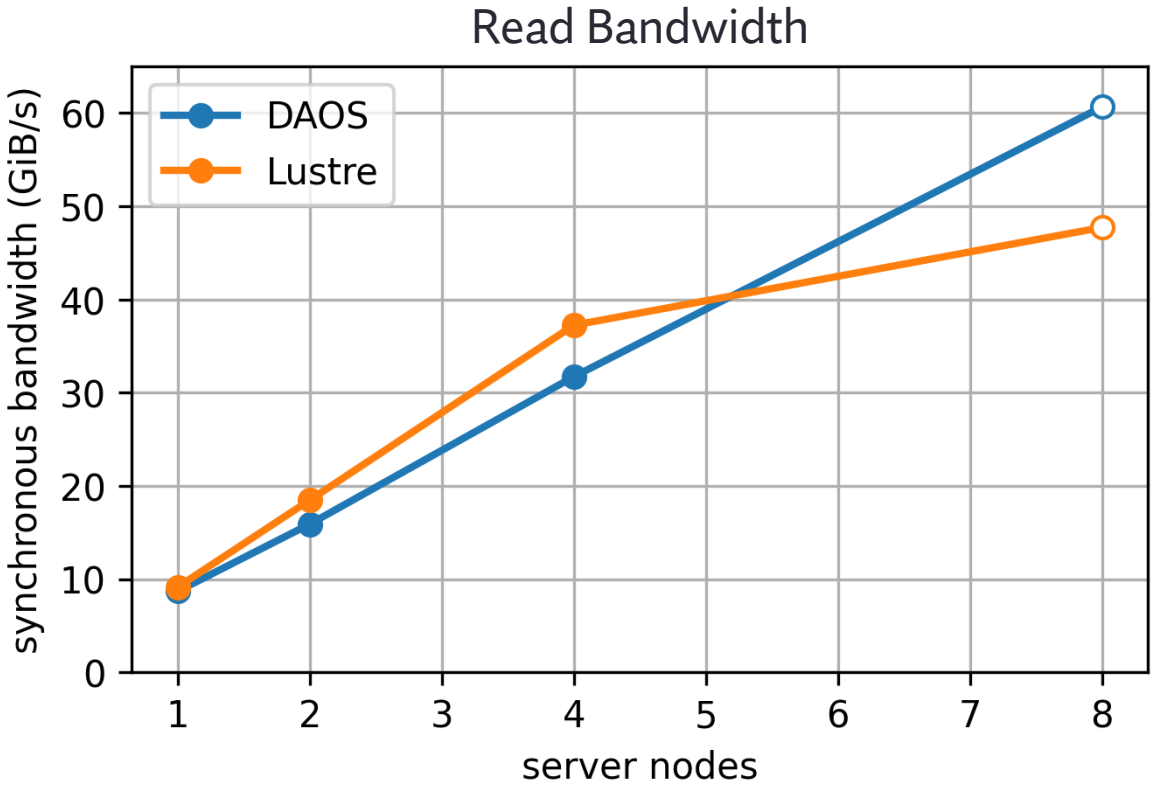
- Range of storage interfaces
  - Native object store (libdaos)
  - Filesystem (various approaches)
  - Raw block device
  - MPI-I/O (ROMIO)
  - HDF5
  - PyDAOS
  - Spark/Hadoop
  - TensorFlow I/O
- DAOS systems built from DAOS servers
  - One per socket, has own NVMe and NVRAM
  - Scale system by adding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

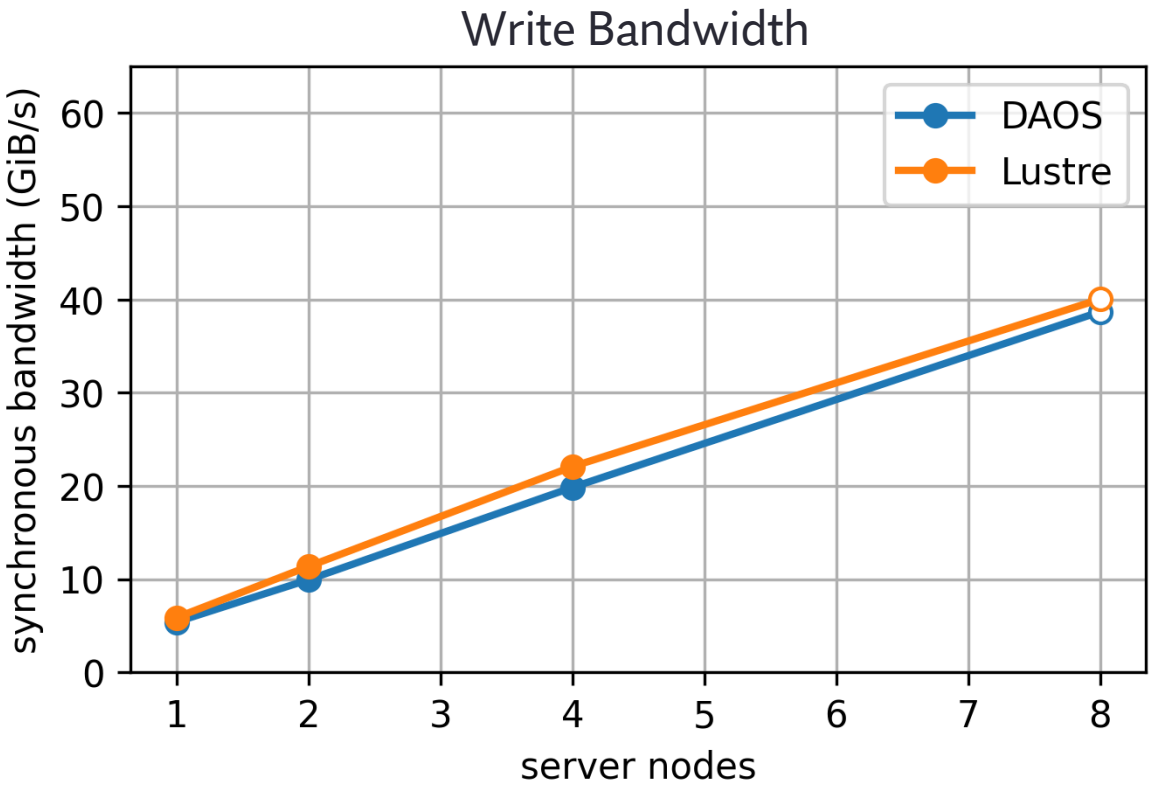


epcc

DAOS Performance

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

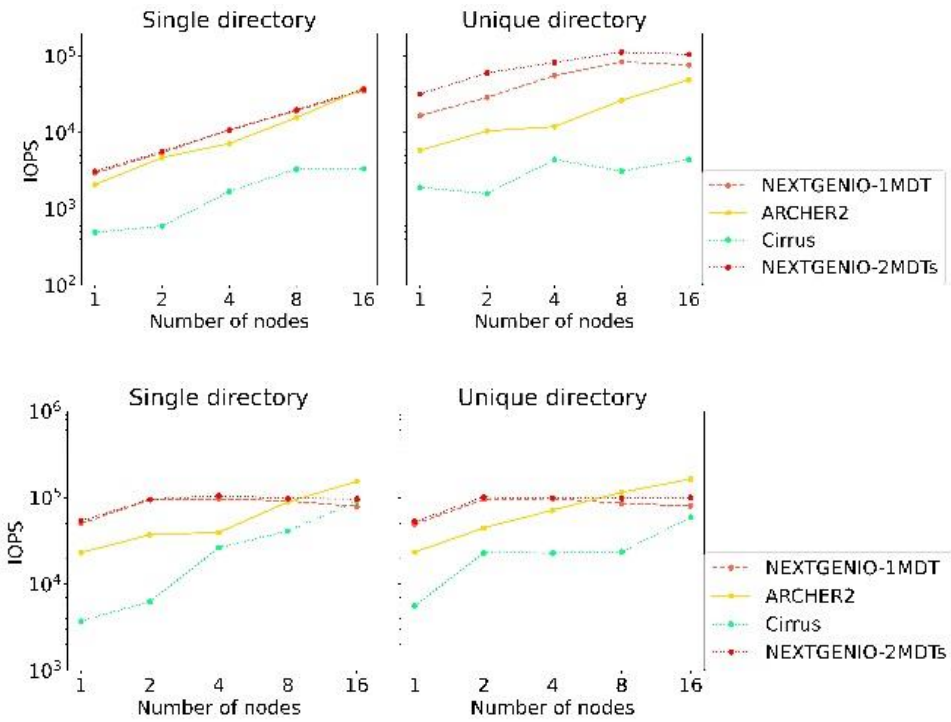




epcc

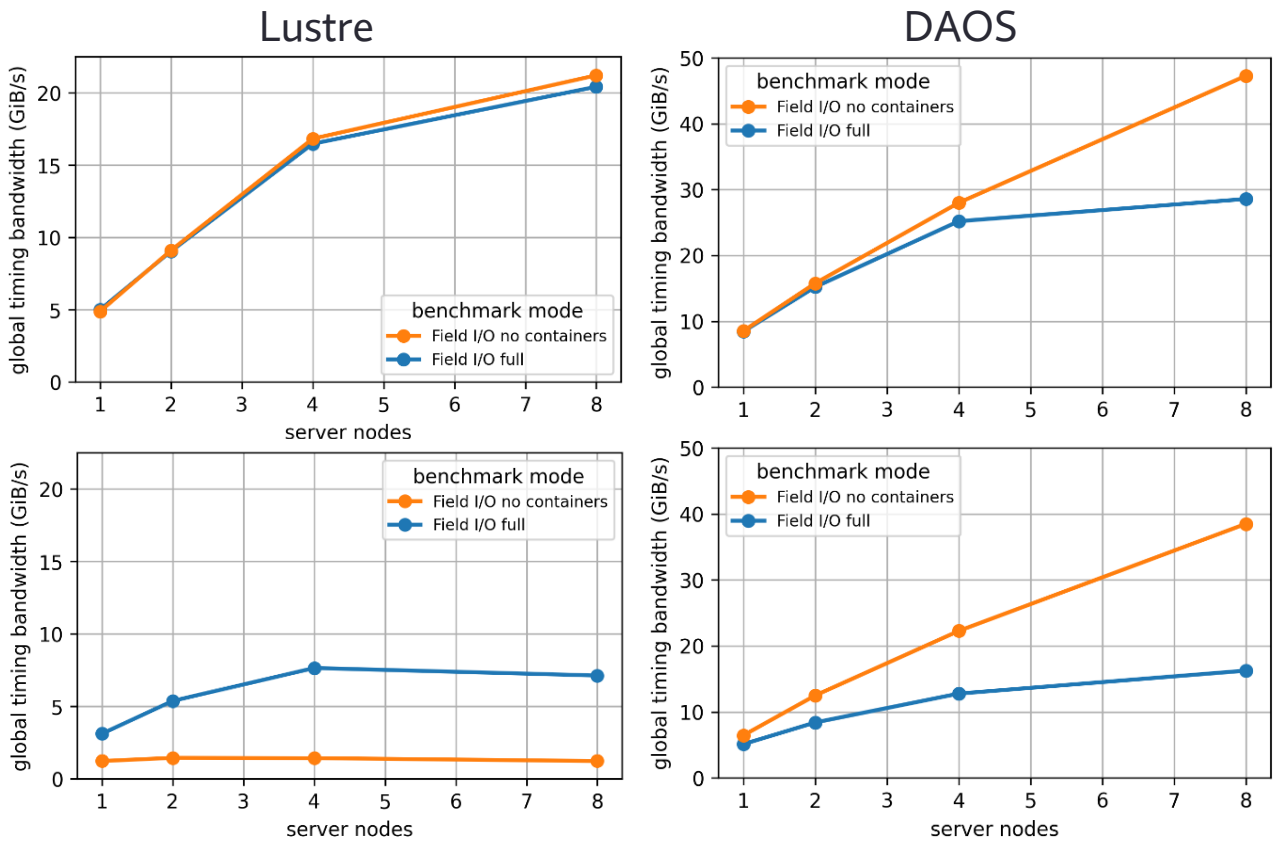
DAOS performance

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



Read

Write

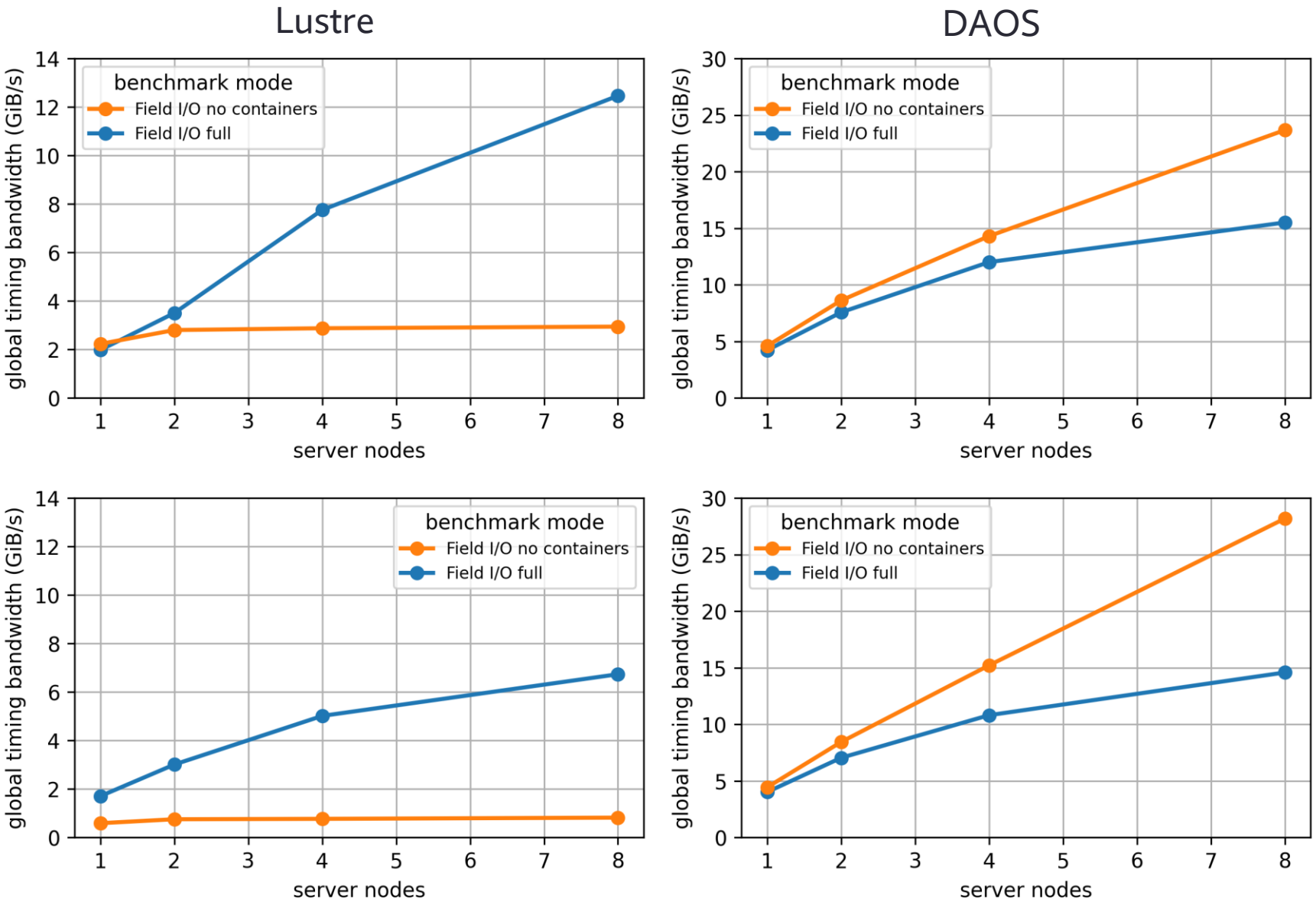


DAOS performance

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

Read

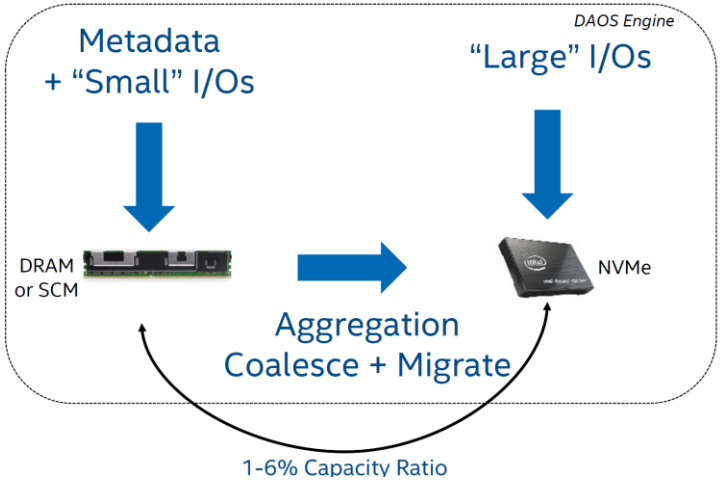
Write



epcc

VOS – Versioning Object Store

- DAOS is implemented as a VOS
  - Epochs and transactions built into the storage
- Updates create new version of the *value* in the kv
- Epochs used to provide consistency on updates
- Can use transactional APIs to group data updates atomically in applications
- Small I/O operations stay at the metadata level
  - Larger I/O operations hit the main storage
  - Dynamically managed



epcc

Non-NVRAM DAOS

- Path to solution with no persistent memory
  - Metadata in RAM
  - Write Ahead Log (WAL) on NVMe
  - Synchronous WAL updates
  - Asynchronous WAL checkpointing
- Reads faster as from DRAM
- Currently limited by DRAM capacity (async checkpointing done but not leveraged for lower resident set yet
  - Restricts the size of DAOS or requires expensive DRAM setup
  - Allows a wide range of hardware for DAOS servers
- Google are using it for high performance storage in GCP
  - ParallelStore
  - Volatile deployment, Lustre alternative
  - “Parallelstore is based on Intel DAOS and delivers up to 6.3x greater read throughput performance compared to competitive Lustre scratch offerings.”

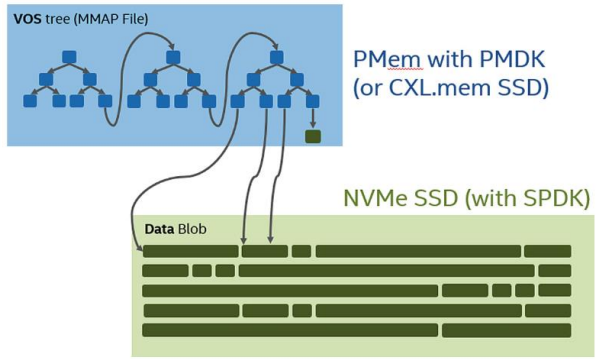


Fig. 2. DAOS Backend using Persistent Memory.

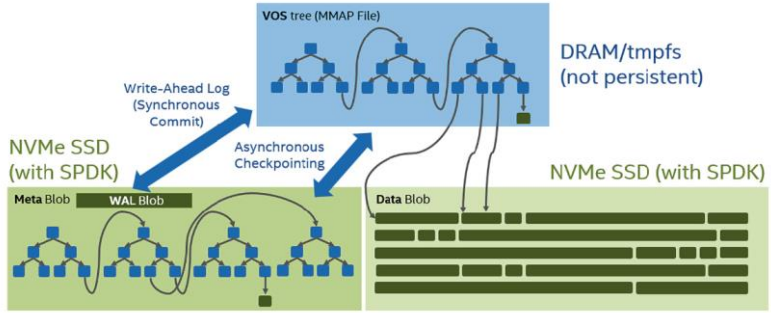


Fig. 3. DAOS Backend using Volatile Memory.

DAOS Beyond Persistent Memory: Architecture and Initial Performance Results
https://doi.org/10.1007/978-3-031-40843-4_26

epcc

Using DAOS

- DAOS provides “legacy” approaches
  - Requires some config to get high performance (i.e. number of workers/mount points)
  - https://github.com/johannlombardi/daos-kernel

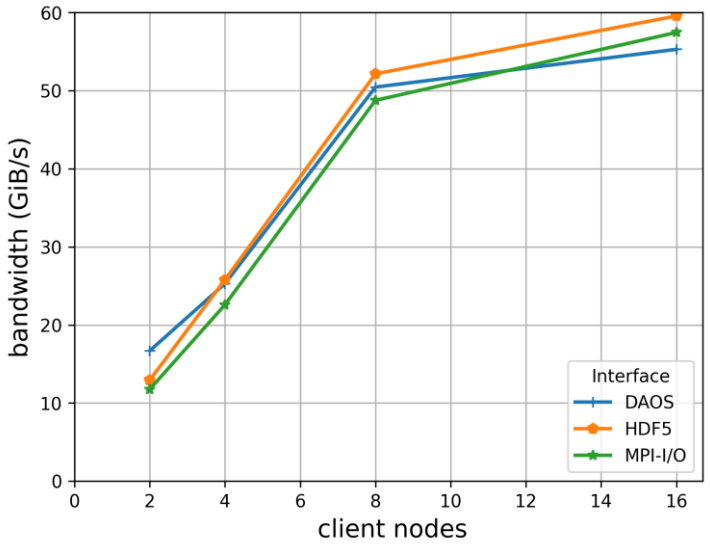
```
mkdir /tmp/my_filesystem
dfuse -m /tmp/my_filesystem --pool tutorial --cont adrians

…

fusermount3 -u /tmp/my_filesystem
```

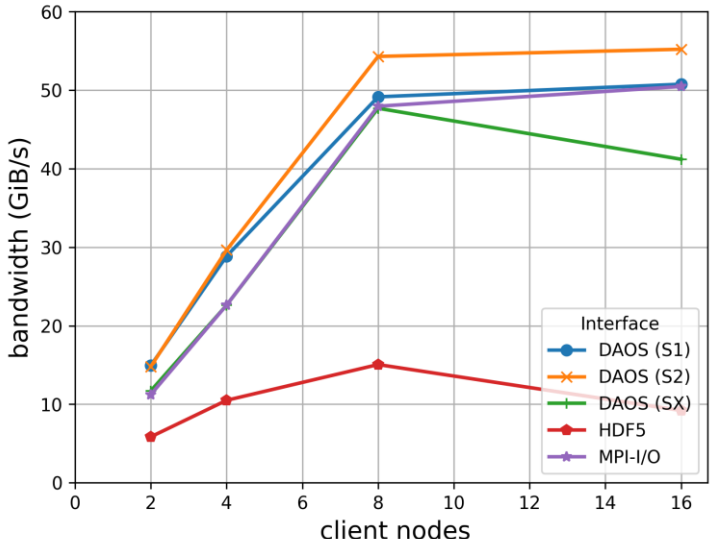
- Benchmarking interface
  - DAOS api
  - Fuse for HDF5/MPI-I/O
- Wide range of configs

Hard

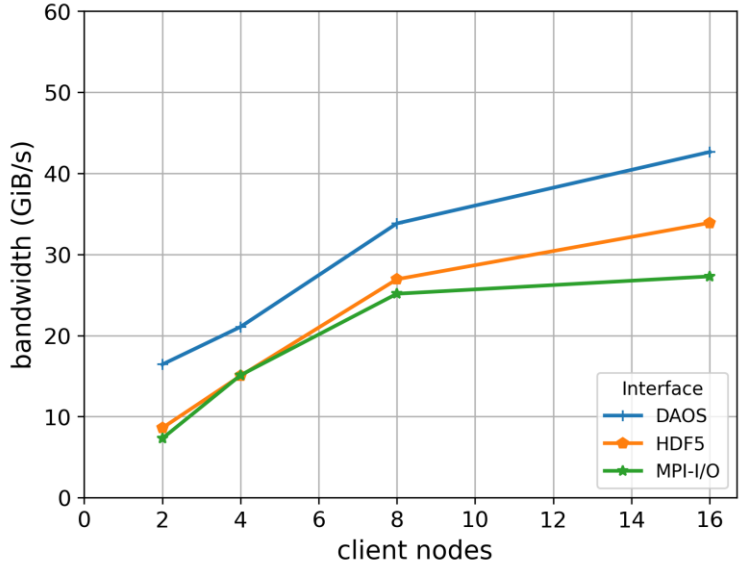
Easy

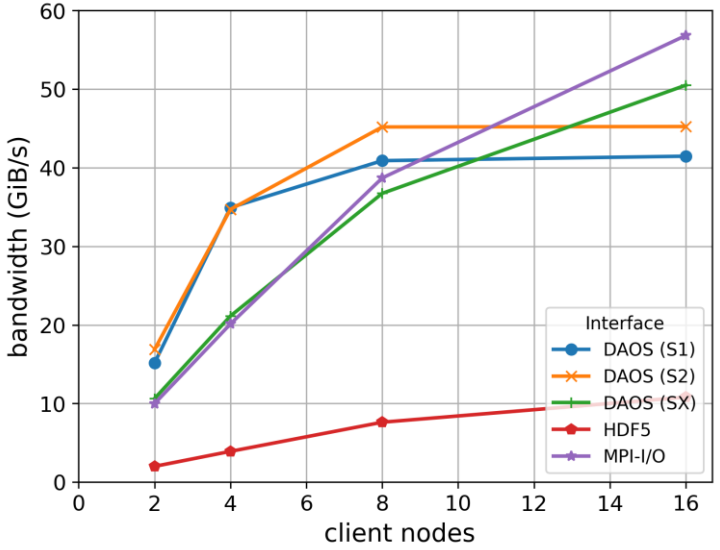
Read





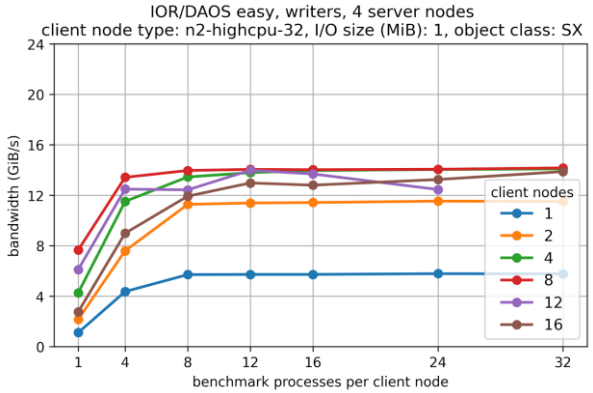
Write

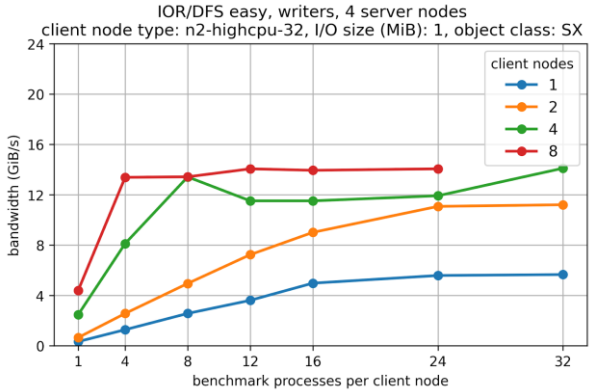


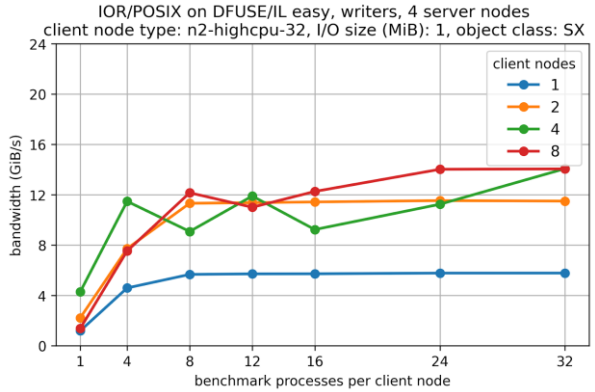


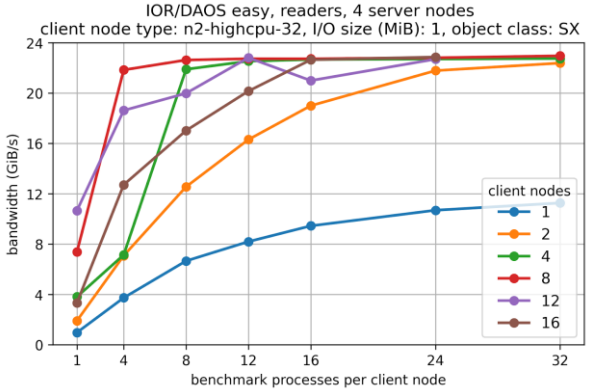
epcc

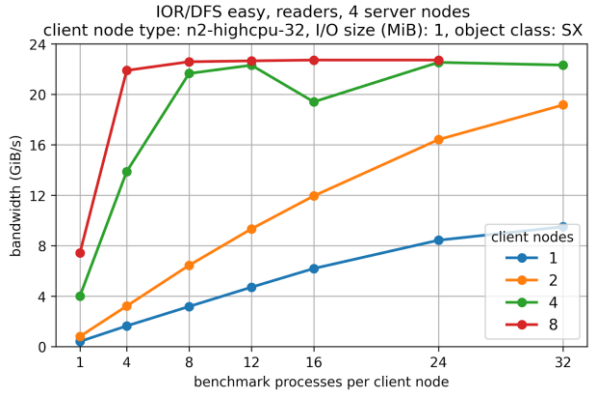
FUSE vs DAOS

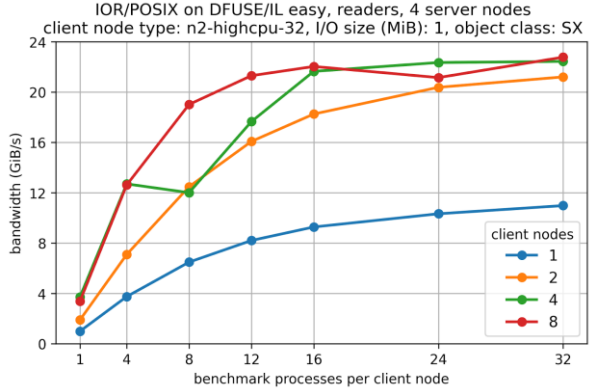










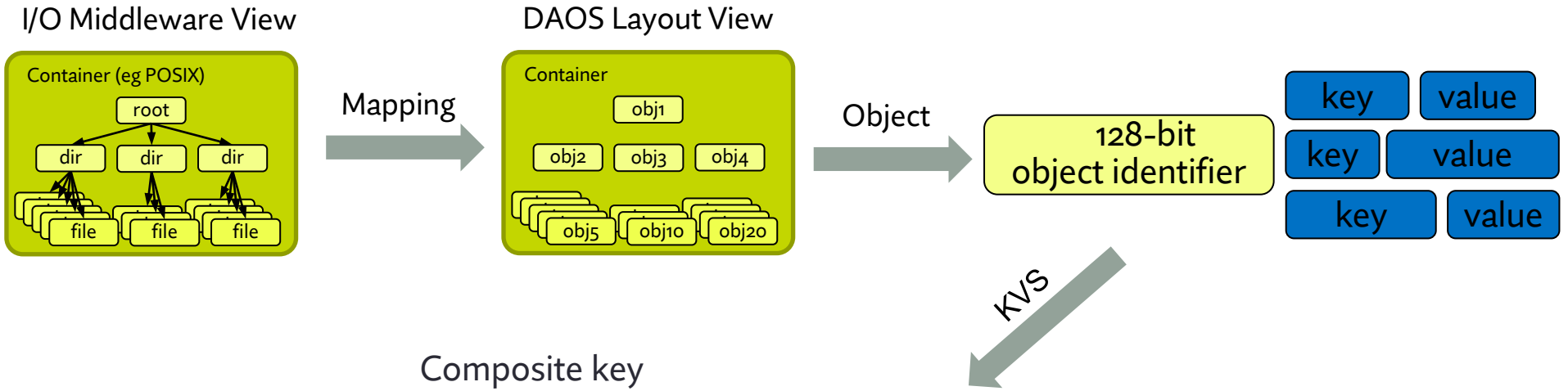


epcc

DAOS Objects

- DAOS Object Types:
  - DAOS KV
    - Simplifies the interface to just give what is needed for key-value interactions
    - i.e. put/get
  - DAOS multi-level KV:
    - Extended key functionality
    - Two parts to allow distribution and data to be separately managed
    - distribution (dkey): All entries under the same dkey are guaranteed to be collocated on the same target
    - attribute (akey): Enables multiple values to be stored under the same key (or can be single value)
  - DAOS Array
    - Single dimension array referenced by a DAOS key
    - Specialised version of the multi-level KV, dkey and akey managed for you

DAOS objects



Distribution key
Variable length
Integer, string, data structure, ...
Hashed/distributed across engines

Attribute key
Variable length
Integer, string, data structure, ...
Co-located on same engines
All akeys can be retrieved in a single RPC

Index
64-bit Integer
Address array of value
Support operation over range
of indices (= extent)

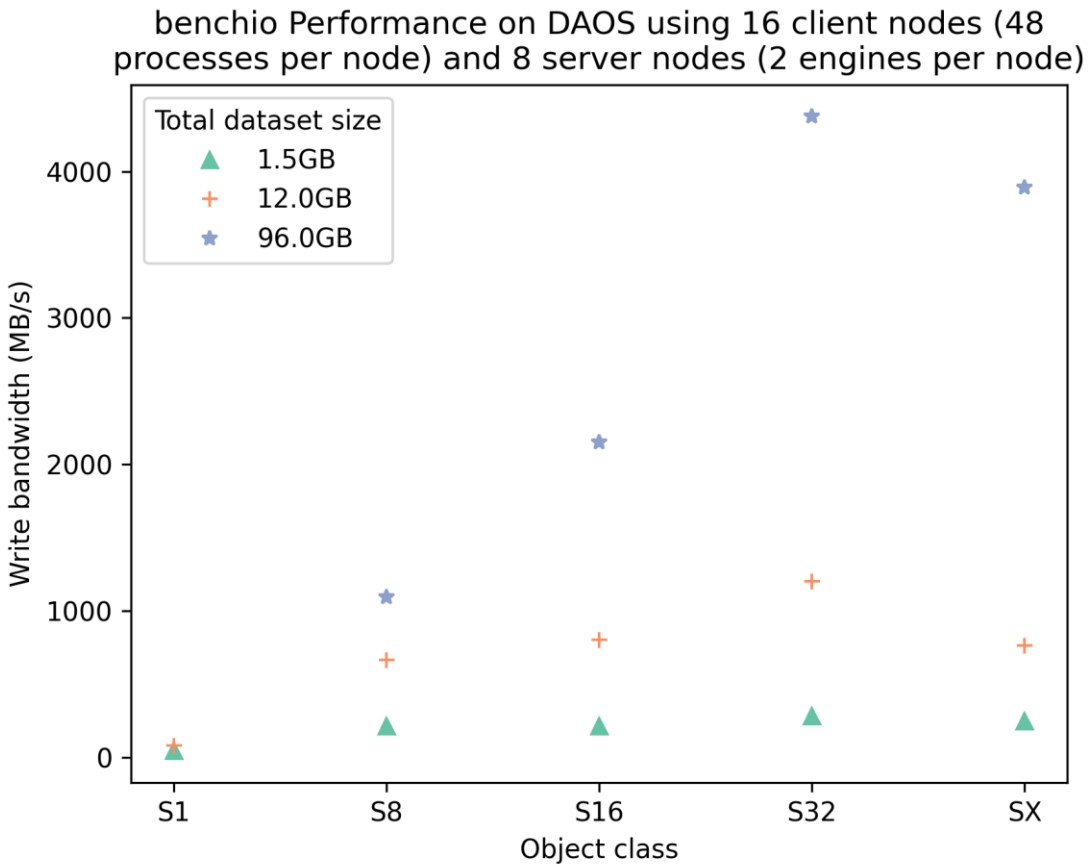
Record
Variable length value
From 1 byte to GB's

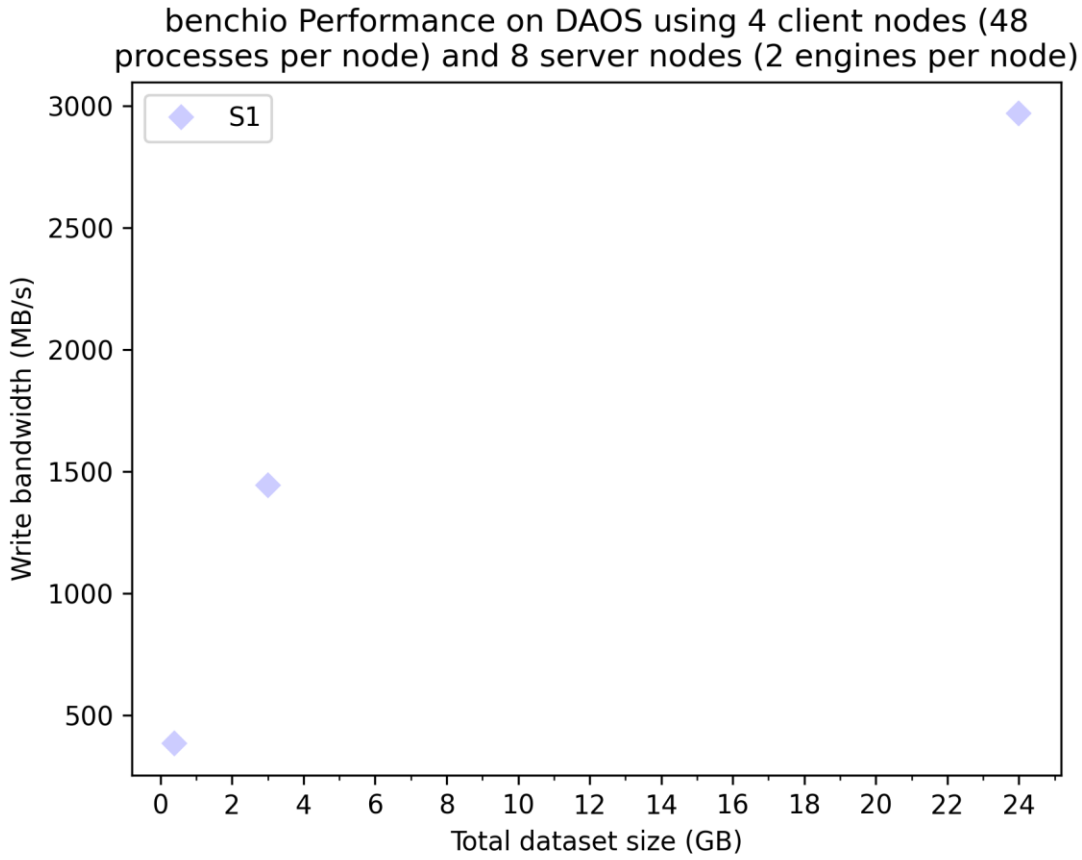
epcc

Object classes

- Object ID 128-bit space:
  - Lower 96 bits set by user
  - Unique OID allocator available in API for convenience
  - OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)
- DAOS allows configuration of *object class*
  - Sharding: Think Lustre striping
    - S1, S2, S4, S8, ..., S32, SX
  - Replication + Sharding: Think Lustre striping and object copying
    - RP_2G1, RP_2G2, RP_2G4, RP_2G8, ..., RP_2G32, RP_2GX
    - Can vary the replication higher than 2 (first number)
  - Erasure coding + Replication + Sharding:
    - OC_EC_2P1G1, OC_EC_2P1G2, ..., OC_EC_2P1GX
    - Can also vary the replication

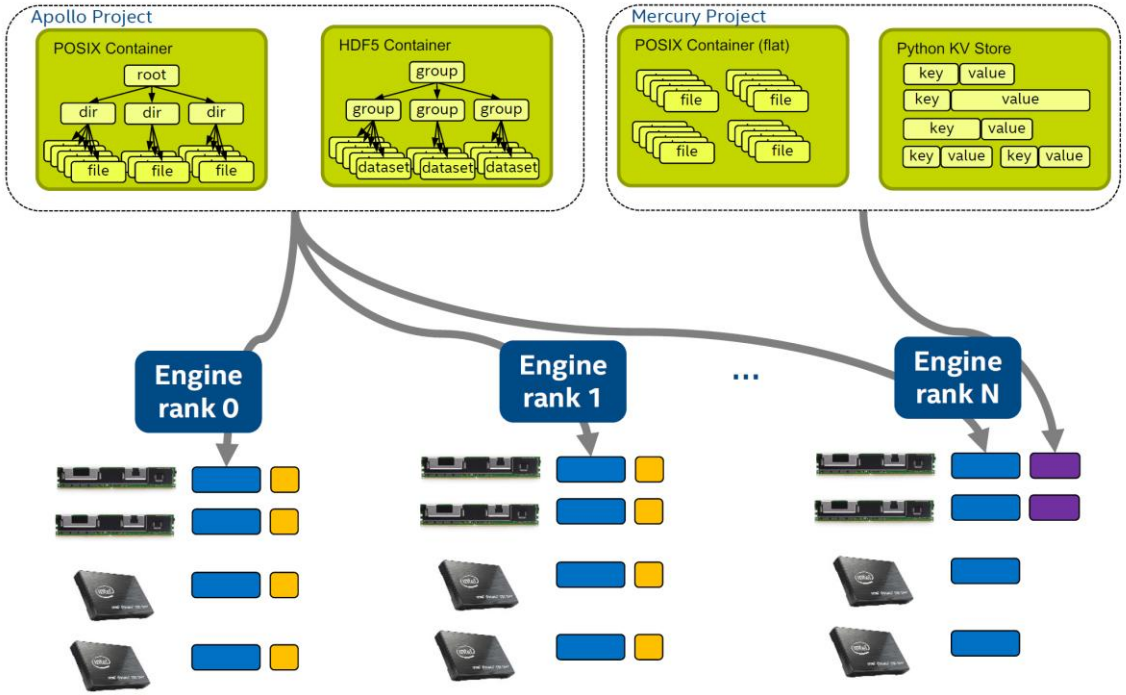
Object class





epcc

System Hierarchy

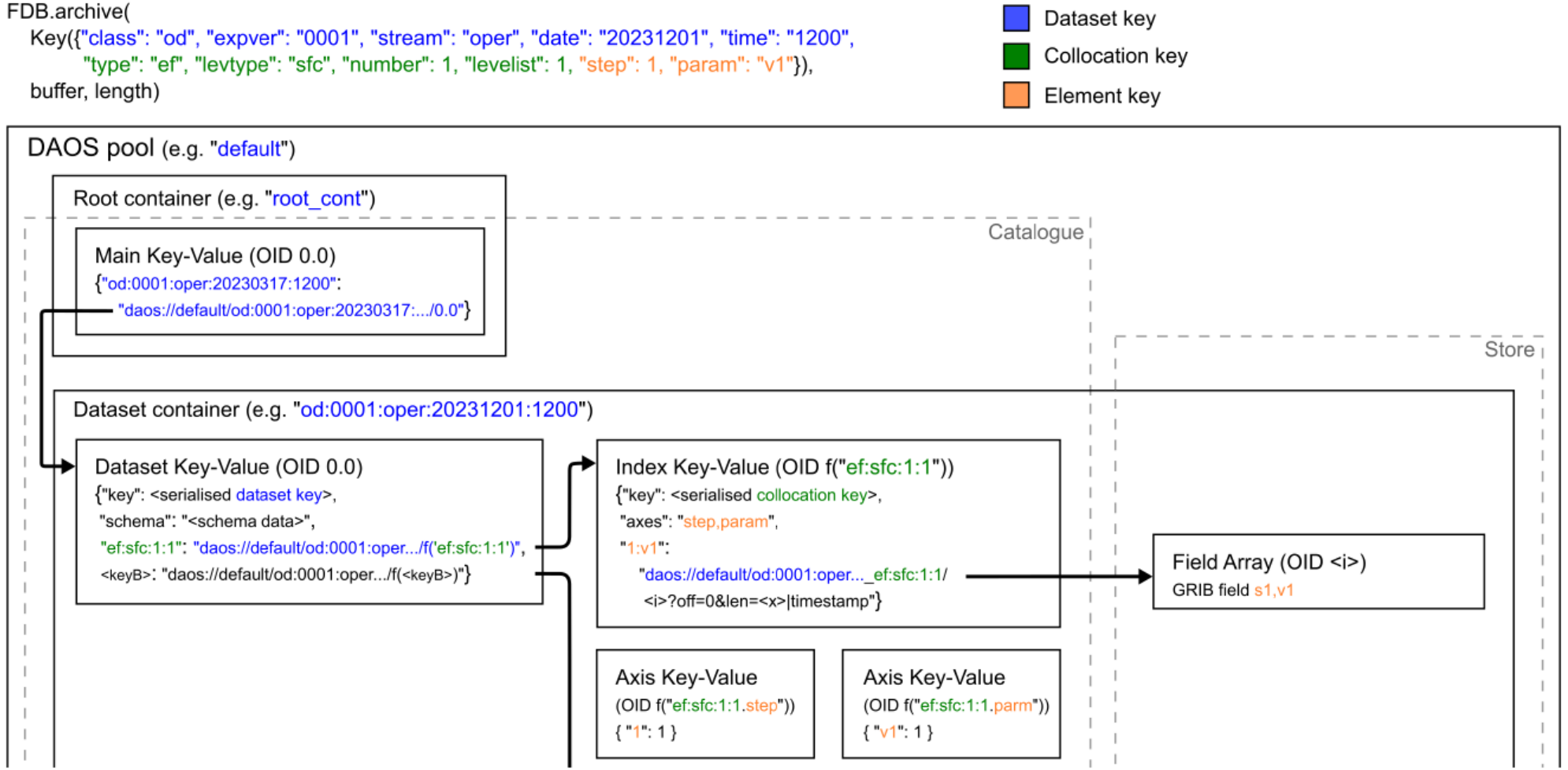


- Pools
  - Highest level construct in the system
  - Possibly analogous to a filesystem
  - Maybe more like a root of a tree
  - Maps to servers/storage hardware
  - Orders of ones per system
- Containers
  - Namespace/logical space inside a pool
  - Group together objects
  - Orders of hundred per pool
- Can specify configurations on both pools and containers
  - Access control, replication, redundancy, etc...
- Objects can inherit configurations from containers
  - And can define their own requirements
  - Any number

epcc

Example DAOS configuration/usage

```
FDB.archive(
   Key({"class": "od", "expver": "0001", "stream": "oper", "date": "20231201", "time": "1200",
        "type": "ef", "levtype": "sfc", "number": 1, "levelist": 1, "step": 1, "param": "v1"}),
   buffer, length)
```



DAOS API programming

```
rc = daos_init();
daos_pool_info_t pool_info;
daos_cont_info_t co_info;
rc = daos_pool_connect(o->pool, o->group, DAOS_PC_RW, &poh,
&pool_info, NULL);
rc = daos_cont_open(poh, o->cont, DAOS_COO_RW, &coh, &co_info, NULL);
rc = dfs_mount(poh, coh, O_RDWR, &dfs);
rc = dfs_write(dfs, obj, &sgl, off, NULL);
rc = dfs_read(dfs, obj, &sgl, off, &ret, NULL);
rc = dfs_umount(dfs);
rc = daos_cont_close(coh, NULL);
rc = daos_cont_destroy(poh, o->cont, 1, NULL);
rc = daos_pool_disconnect(poh, NULL);
rc = daos_fini();
```

DAOS C API - Pools

- Initialise and finalise the API

```
int daos_init(void);
int daos_fini(void);
```

- Connect to an existing pool

```
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);
```

- Disconnect from a pool to ensure all I/O has concluded

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

- Large scale parallel programs want to avoid every client opening the pool
  - Costly operation at the server side
  - Share the pool handle opened by a single client via MPI

```
daos_pool_local2global
daos_pool_global2local
```

Containers

- Can manually create/destroy containers at the command line

```
daos cont create mypool --label=mycont
  Container UUID : 5d33d6e0-6c8b-4bf5-bb49-c8723bf30c91
  Container Label: mycont
  Container Type : unknown

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

- Can do this programmatically as well:

```
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, daos_event_t *ev);
```

- Need to open/close the container from the program

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

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

- As with the pool, want to avoid all clients opening manually

```
daos_cont_local2global

daos_cont_global2local
```

epcc

Minimal example

```
#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);
  /** do things */
  daos_cont_close(coh, NULL);
  daos_pool_disconnect(poh, NULL);
  daos_fini();
  return 0;
}
```

Arrays

- Current DAOS supports 1-D arrays
  - Future support for multi-dimensional arrays
- Create/open/close arrays

```
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, daos_event_t *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_open_with_attr(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);
```

- `th` is a transaction handle
- `ev` is a completion event hand
- Both can be NULL

epcc

Arrays

- Chunk size: The striping chunk size
  - When to move on to another dkey
- Cell size: Size of the datatype of the array in bytes
- Open call will do create if it doesn't exist
  - This can be expensive to check
  - Open with attributes is cheaper if you know it exists
  - Bypasses checks (i.e. cell and chunk sizes)
- Data operations:

```
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);
```

Arrays

- Specify range of data to read/write and location of data source/destination

```
daos_array_iod_t *iod, d_sg_list_t *sgl
```

  - `iod` specifies the location of the data in the DAOS array
  - `sgl` specifies the location of the data in memory
- Multi-process writing/reading to DAOS array enabled
  - Can specify offsets and access patterns using the `iod`
- Support functions also available:

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

- sdf

Arrays example

```
/* create array - if array exists just open it */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 1048576, &array, NULL);
daos_array_iod_t iod;
d_sg_list_t sgl;
daos_range_t rg;
d_iov_t iov;
/* set array location */
iod.arr_nr = 1; /** number of ranges / array iovec */
rg.rg_len = BUFLEN; /** length */
rg.rg_idx = rank * BUFLEN; /** offset */
iod.arr_rgs = &rg;
/* set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;
d_iov_set(&iov, buf, BUFLEN);
sgl.sg_iovs = &iov;
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);
```

Key-Value 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_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, 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, daos_event_t *ev);
```

- Open returns a key handle, creates the key if required
  - Flag can be: `DAOS_OO_RO`, `DAOS_OO_RW`
- Put and Get work with the value associated with the key handle (`oh`)

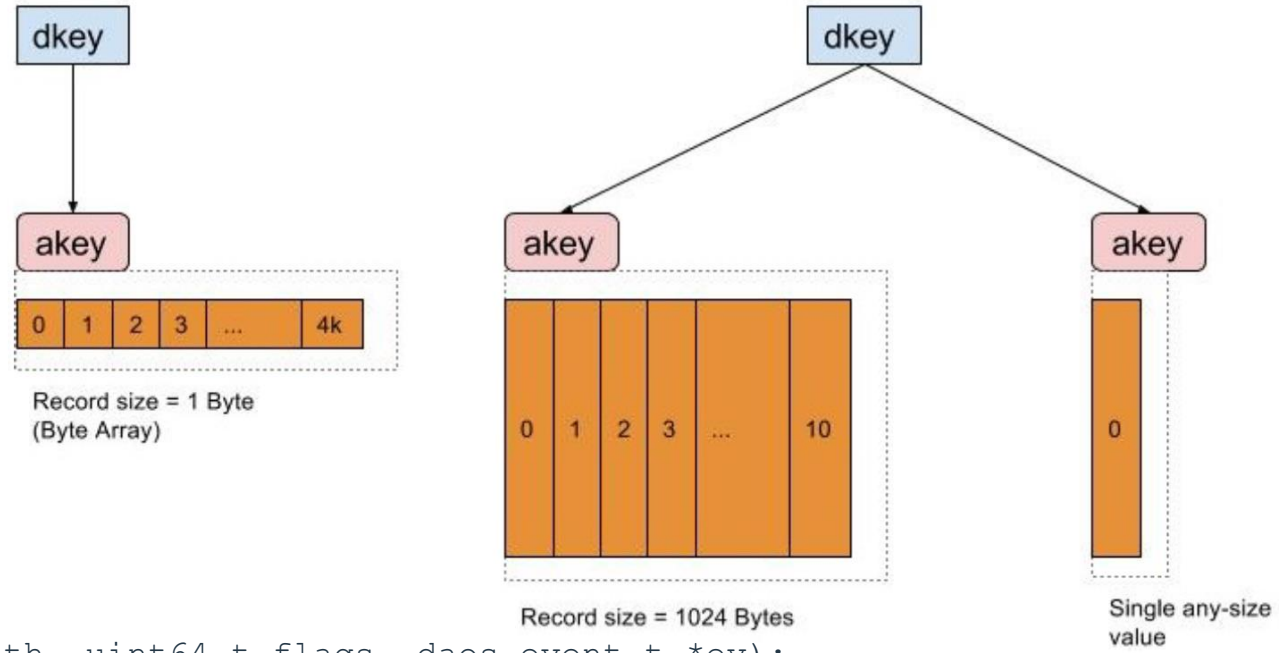
epcc

Key-Value example

```
oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid, DAOS_OF_KV_FLAT, 0, 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);
```

Multi-level KV

- Array and KV are specialised version
  - Multi-level provides full functionality




```
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, daos_event_t *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, daos_event_t *ev);

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);

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);
```

epcc

Multi-level KV example

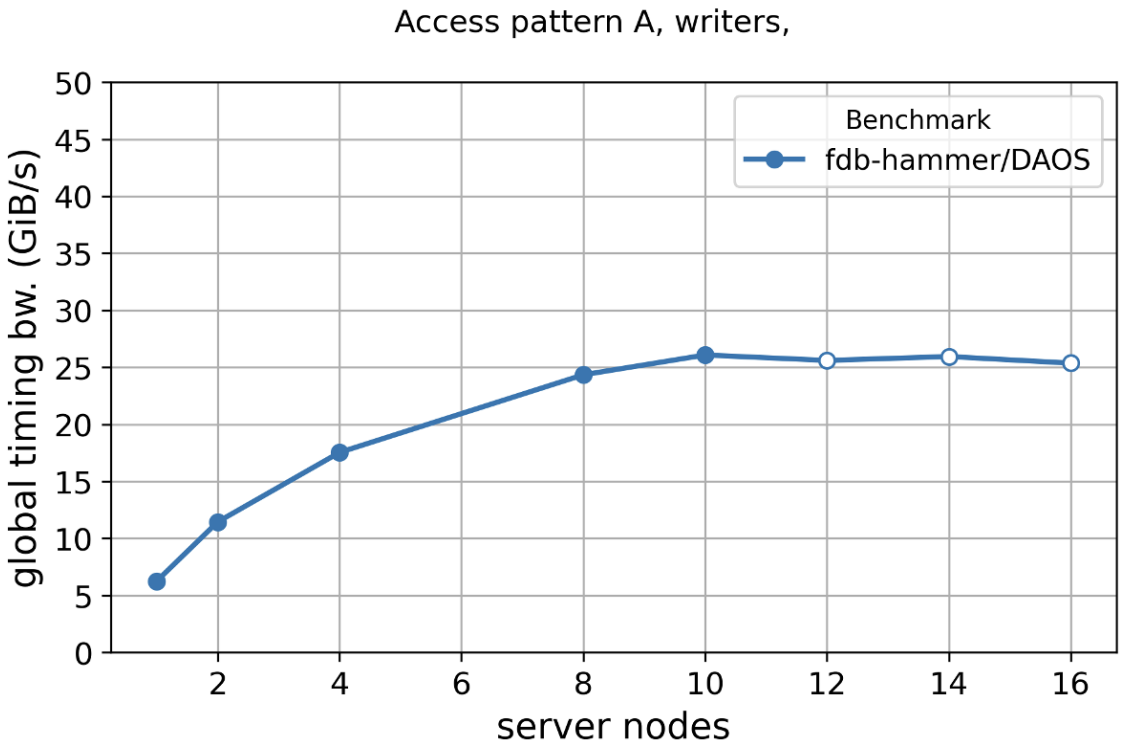
```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;
d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", 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);
```

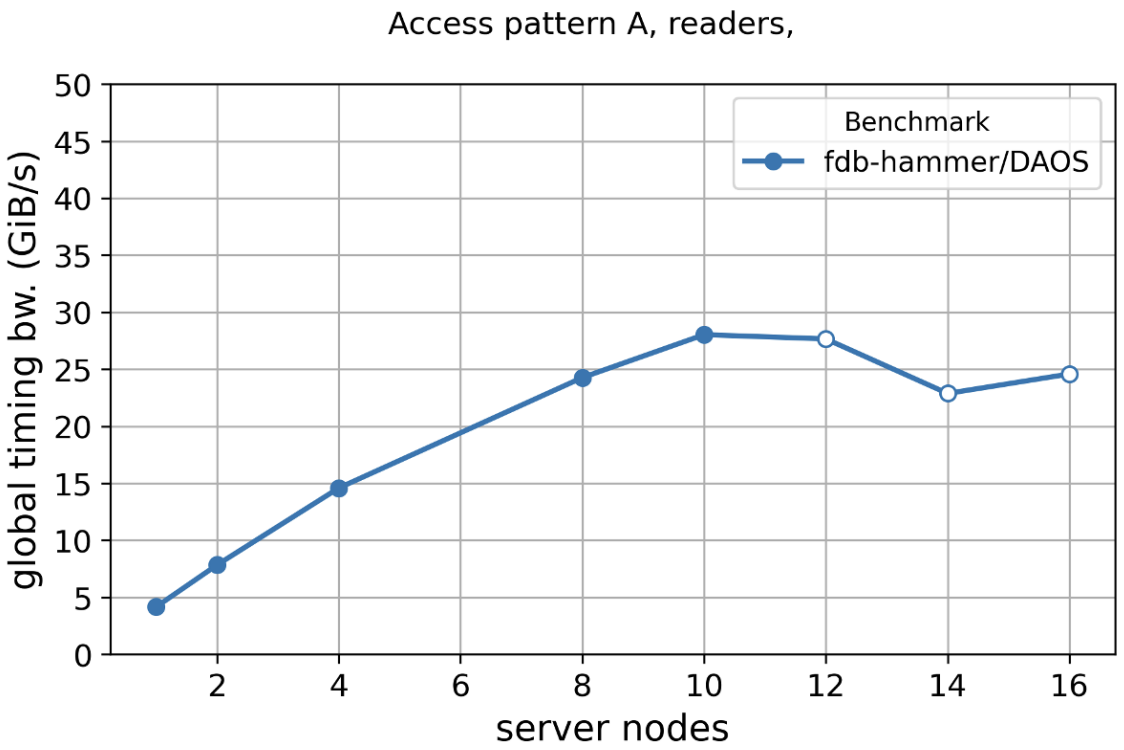
Fortran interfacing

```
call daos_initialise(pool_name_c, cartcomm)
call daos_write_array(ndim, arraysize, arraygsize, arraysubsize, arraystart, out_data, object_class_c, blocksize, check_data, daosconfig, cartcomm)
call daos_write_object(ndim, arraysize, arraygsize, arraysubsize, arraystart, out_data, object_class_c, blocksize, check_data, daosconfig, cartcomm)
call daos_read_array(ndim, arraysize, arraygsize, arraysubsize, arraystart, read_data, object_class_c, daosconfig, cartcomm)
call daos_read_object(ndim, arraysize, arraygsize, arraysubsize, arraystart, read_data, object_class_c, daosconfig, cartcomm)
call daos_finish(iocomm)

array_obj_id.hi = 0;
array_obj_id.lo = 0;

uuid_generate_md5(array_uuid, seed, array_name, strlen(array_name));

memcpy(&(array_obj_id.hi), &(array_uuid[0]) + sizeof(uint64_t), sizeof(uint64_t));
memcpy(&(array_obj_id.lo), &(array_uuid[0]), sizeof(uint64_t));

daos_array_generate_oid(container_handle, &array_obj_id, DAOS_OT_ARRAY_BYTE, array_obj_class, 0, 0);

ierr = daos_array_open(container_handle, array_obj_id, DAOS_TX_NONE, DAOS_OO_RW, &cell_size, &local_block_size, &array_handle, NULL);

total_size = sizeof(double);
for(i=0; i<num_dims; i++){
  total_size = total_size * arraysubsize[i];
}

iod.arr_nr = 1;
rg.rg_len = total_size;
rg.rg_idx = 0;
iod.arr_rgs = &rg;

sgl.sg_nr = 1;
d_iov_set(&iov, &output_data[0], total_size);
sgl.sg_iovs = &iov;

ierr = daos_array_read(array_handle, DAOS_TX_NONE, &iod, &sgl, NULL);

ierr = daos_array_destroy(array_handle, DAOS_TX_NONE, NULL);
```

epcc

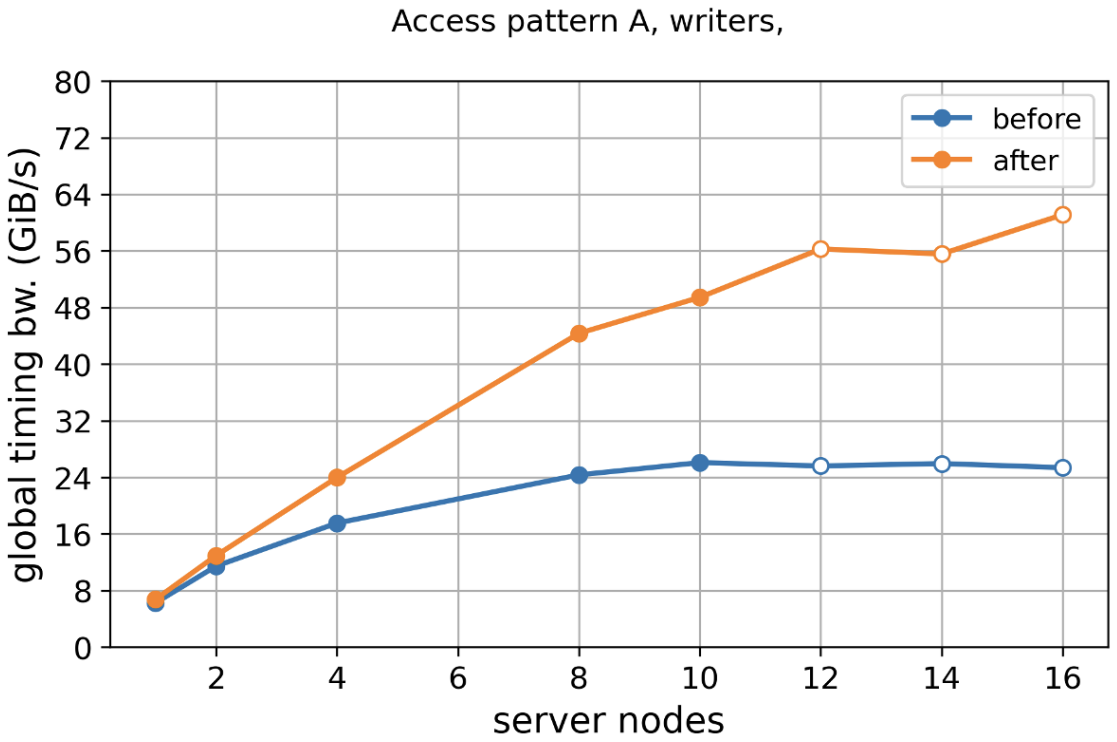
Evaluate performance/approach

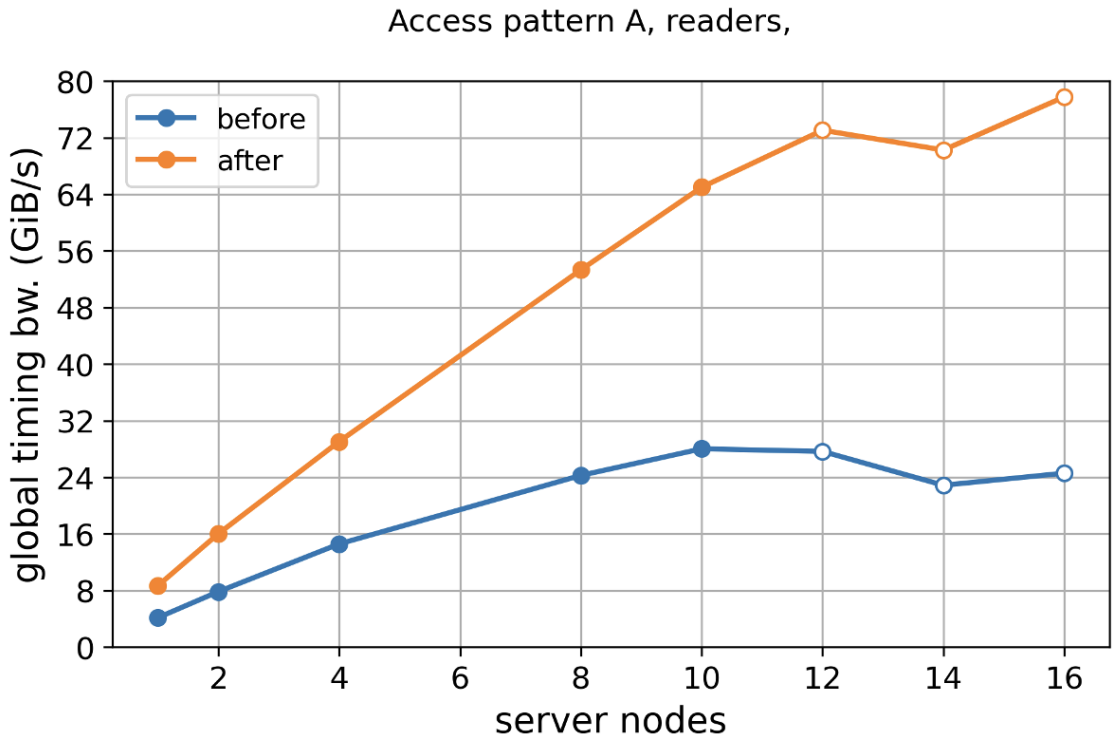




epcc

Optimised performance

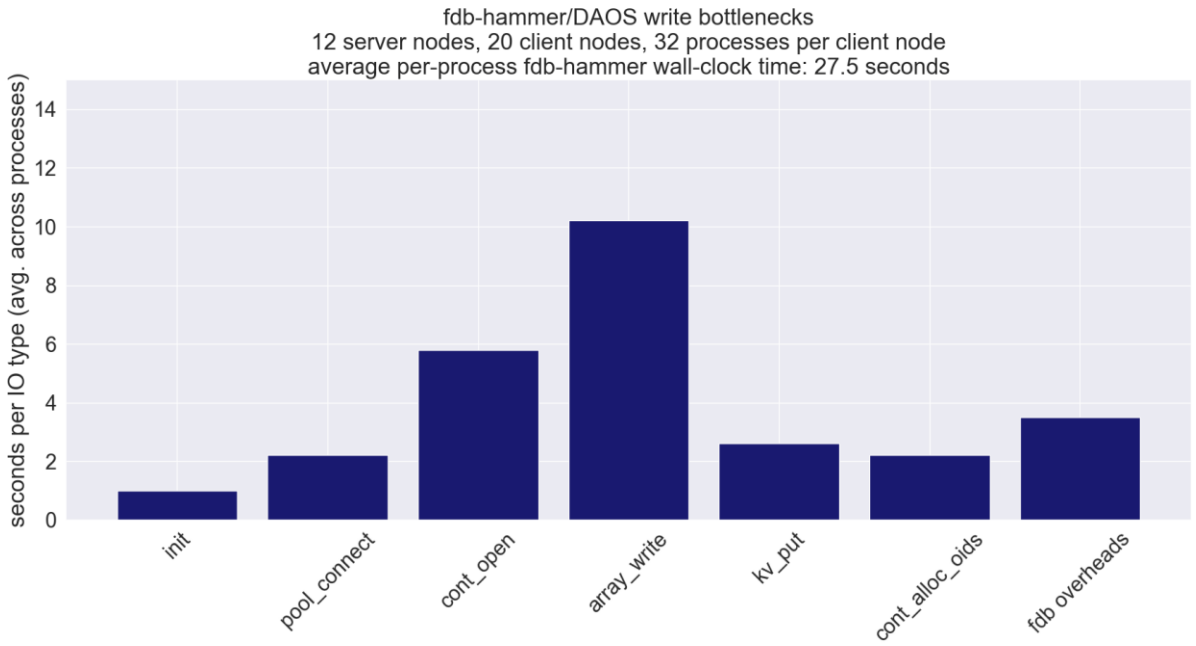


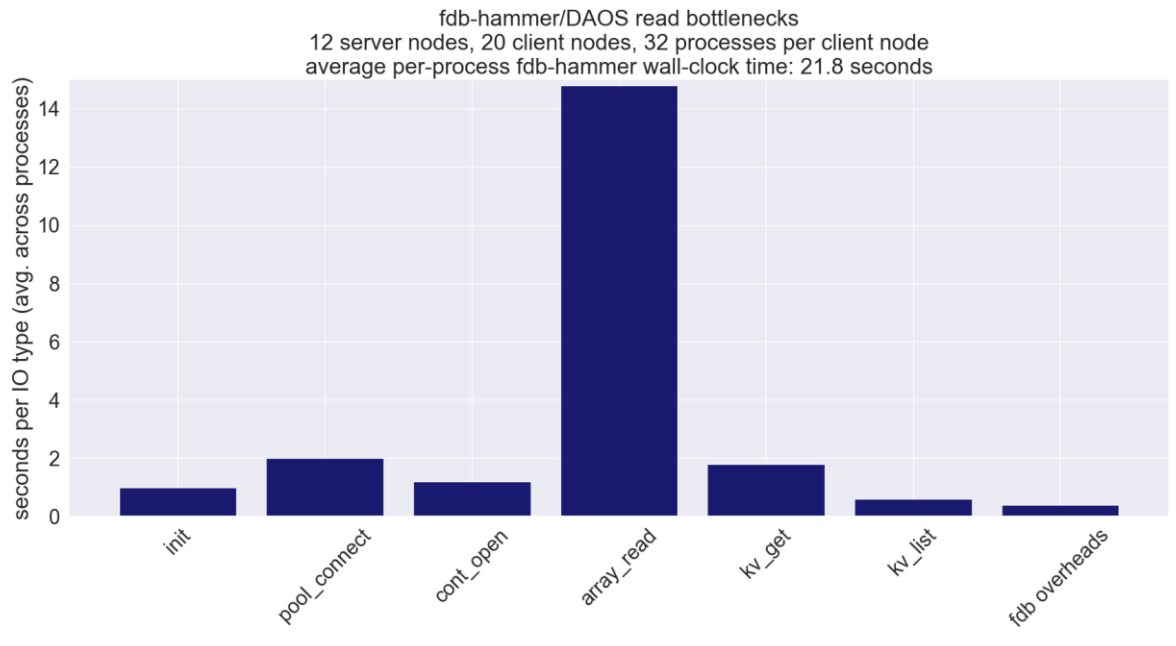


epcc

Profiling

- Example breakdown of where time is being spent
  - Manual profiling





epcc

Approach/recommendations

- Key-Value contention
- For a specific benchmark run configured with contention across processes on indexing Key-Values:
  - 20 GiB/s write
  - 13 GiB/s read
- Tweaking the benchmark configuration to have all processes operate on a separate Key-Values:
  - 35 GiB/s write
  - 68 GiB/s read
- This may not be trivial or possible for all applications, but if design can achieve it then this improves performance

Approach/recommendations

- Avoid communications on/with the server where possible
- Cache objects locally in DRAM if possible
- Use `daos_array_open_with_attr` to avoid `daos_array_create` calls
  - Only supported for `DAOS_OT_ARRAY_BYTE`, not for `DAOS_OT_ARRAY`
  - Warning: the cell size and chunk size attributes need to be provided consistently on any future `daos_array_open_with_attr` to avoid data corruption
- `daos_array_get_size` calls can be expensive
  - Can store array size in our indexing Key-Values
  - Can manually calculate
  - Also possible to infer the size by reading with overallocation:
    - use `DAOS_OT_ARRAY_BYTE`, over-allocate the read buffer, and read without querying the size. The actual read size (`short_read`) will be returned
- `daos_cont_alloc_oids` is expensive, call it just once per writer process
  - Required to generate object ideas to use in calls but can generate many at one

epcc

Approach/recommendation

- Creating several containers (starting at ~300) in a DAOS pool reduces performance
- Opening the same container from all processes is expensive
  - this happens even if only a few containers exist in the DAOS pool
  - e.g. out of 20 seconds taken by a process to write 2000 fields, 1.5 seconds were spent just to open one container
  - we observed this starting at ~200 parallel processes
  - Sharing handles using MPI is the way to fix this
- Opening more than one container per process is very expensive
  - e.g. out of 30 seconds taken by a process to read 2000 fields, 6 seconds were spent just to open two containers

Approach/recommendations

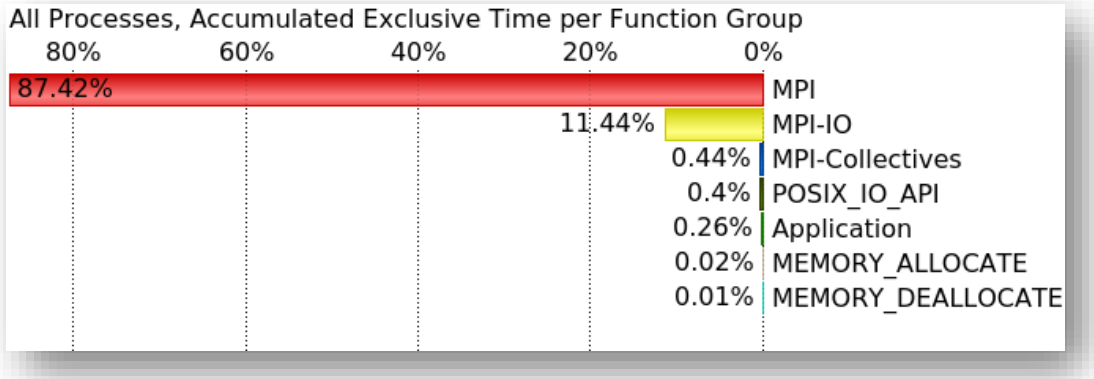
- `daos_key_value_list` is expensive
- `daos_array_open_with_attrs`, `daos_kv_open` and `daos_array_generate_oid` are very cheap (no RPC)
- Normal `daos_array_open` is expensive
- `daos_cont_alloc_oids` is expensive
- `daos_kv_put` and `_get` are generally cheap
  - Value size impacts this
- `daos_obj_close`, `daos_cont_close` and `daos_pool_disconnect` are cheap
- Server configuration to use available networks/sockets/etc… important for performance
  - Just like any storage system or application

epcc

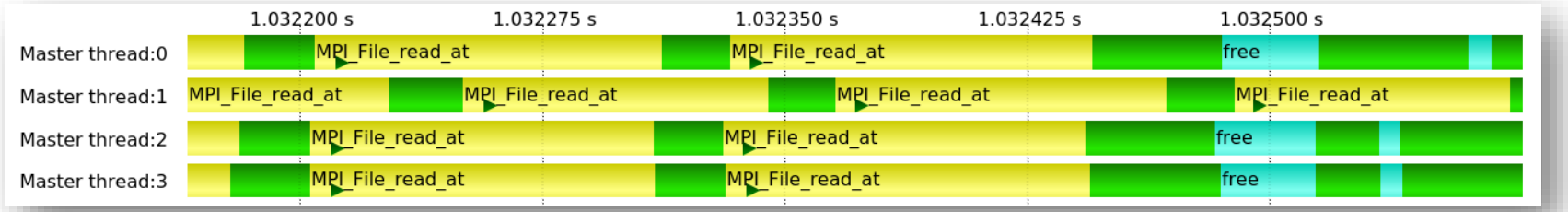
DAOS usage design

- Mapping data structures to KV and Array objects is key to getting good performance functionality
- We suggest mapping contiguous chunks of arrays to be stored to single DAOS array object
  - Collect multiple arrays with associated KV to make the whole array
- Can be as extreme as having a single value per KV
  - Significant overheads in this
- Depends on your application data structures you may want to aggregate less data for I/O
  - Group based on meaningful/scientific dimensions
- HDF5 or similar hierarchies could map well to Keys with Arrays

Summary



- DAOS provides high performance storage
  - 90+ GB/s per server is possible
  - Hardware and configuration dependent, just like all I/O
- Built in replication and redundancy under you/user control
- Different interfaces available
  - Filesystem for zero cost porting
  - Simple file like access for slightly improved performance at little effort
  - DAOS APIs for full functionality
- DAOS interface enables changing I/O granularity/patterns for bigger benefits



epcc