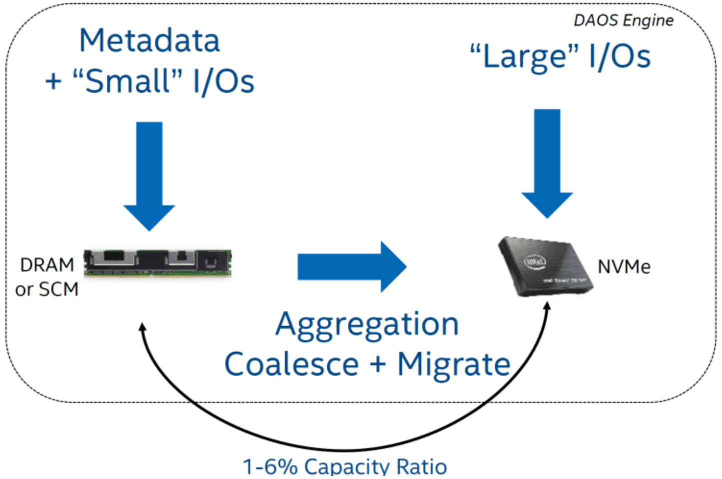
DAOS

HPC Object Store

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



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. ”

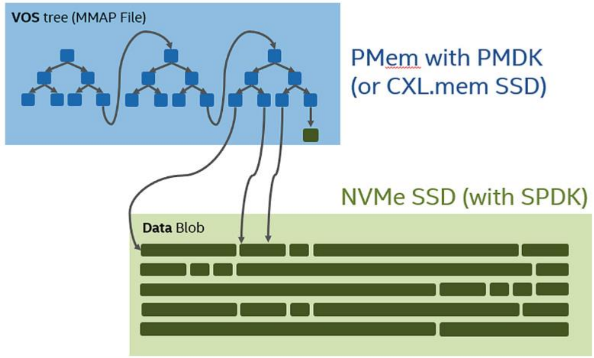


Fig. 2. DAOS Backend using Persistent Memory.

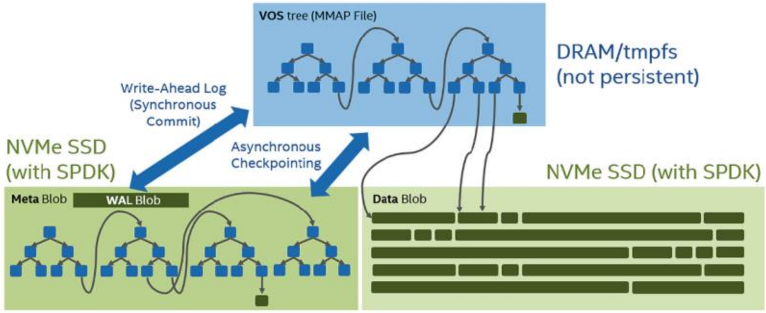


Fig. 3. DAOS Backend using Volatile Memory.

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

intel ECMWF 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

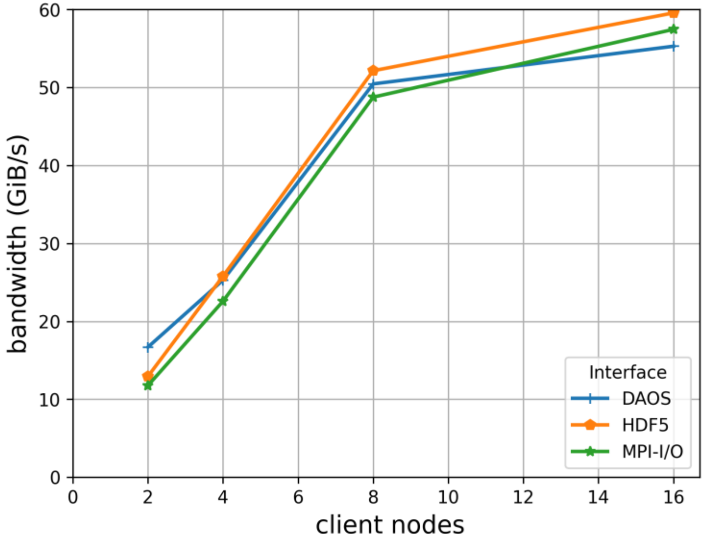
```
mkdir /tmp/my_filesystem
dfuse -m /tmp/my_filesystem --pool tutorial --cont adrians

…

fusermount3 -u /tmp/my_filesystem
```

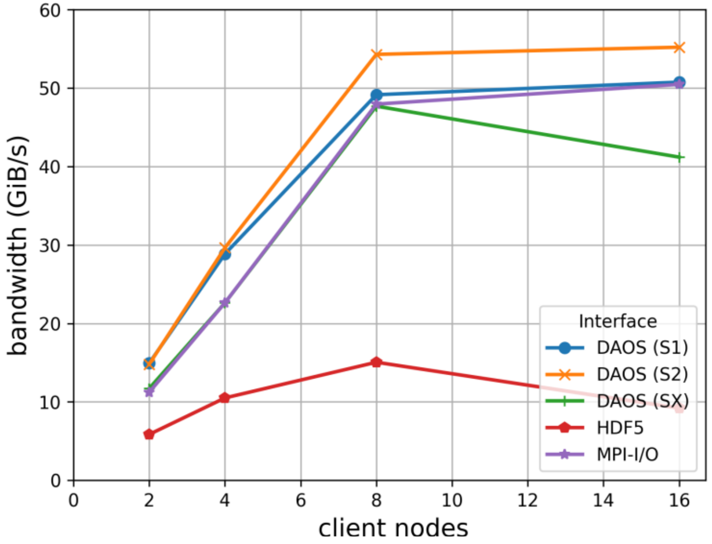
- Benchmarking interface
  - DAOS api
  - Fuse for HDF5/MPI-I/O
- Wide range of configs

Hard

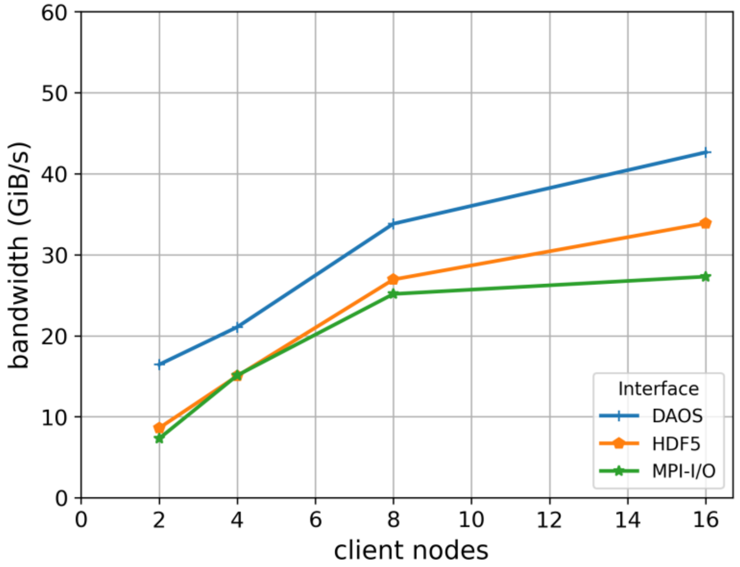
Easy

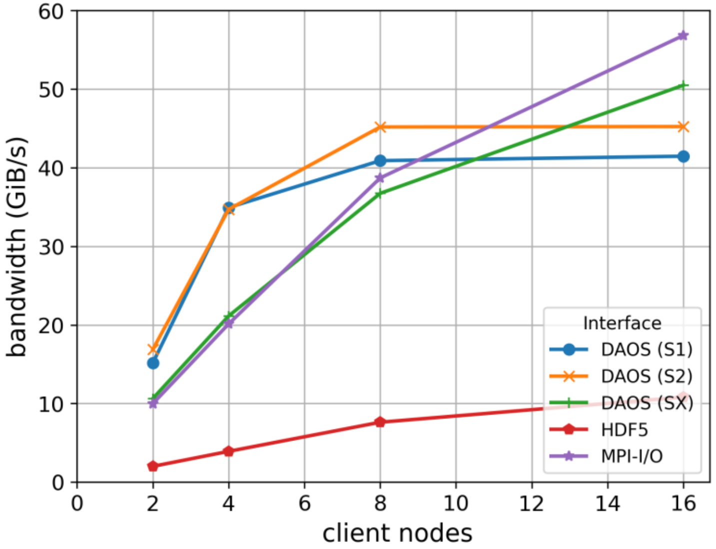
Read



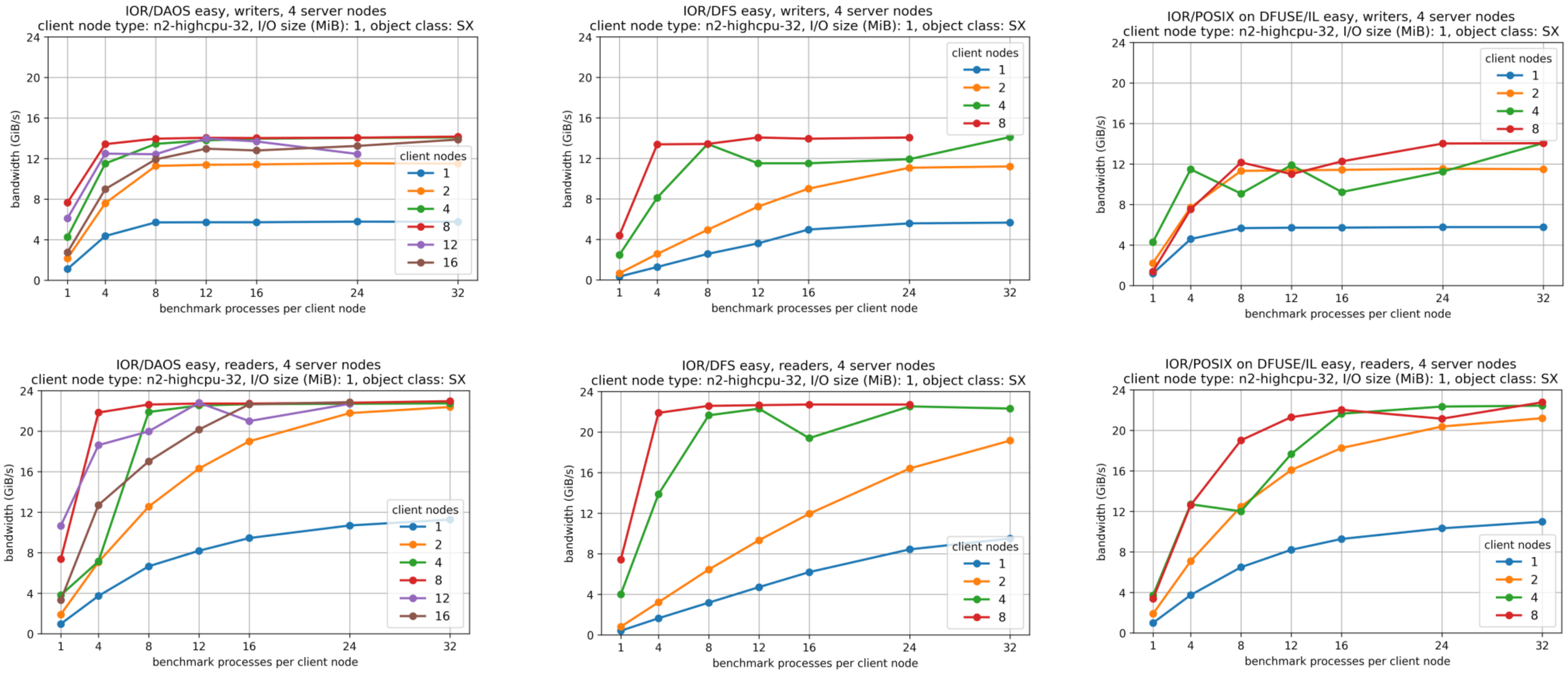


Write





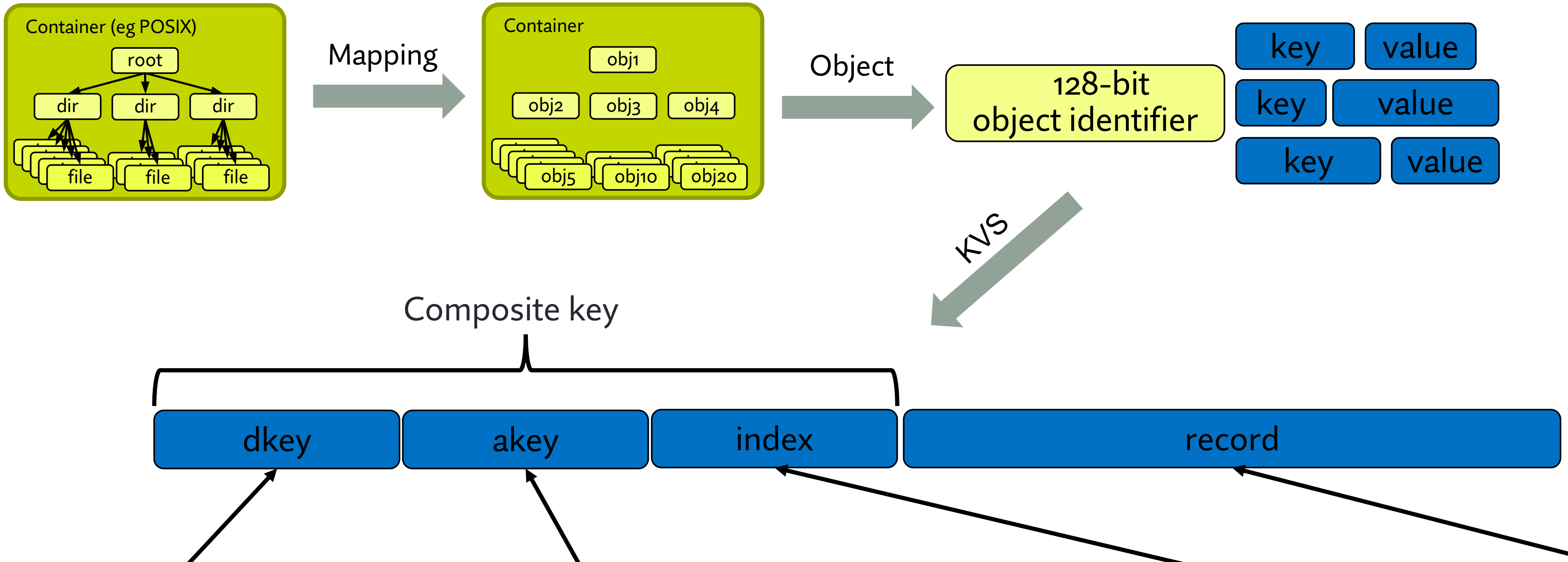
FUSE vs DAOS


IOR/DAOS easy, writers, 4 server nodes
client node type: n2-highcpu-32, I/O size (MiB): 1, object class: SX
IOR/DFS easy, writers, 4 server nodes
client node type: n2-highcpu-32, I/O size (MiB): 1, object class: SX
IOR/POSIX on DFUSE/IL easy, writers, 4 server nodes
client node type: n2-highcpu-32, I/O size (MiB): 1, object class: SX
IOR/DAOS easy, readers, 4 server nodes
client node type: n2-highcpu-32, I/O size (MiB): 1, object class: SX
IOR/DFS easy, readers, 4 server nodes
client node type: n2-highcpu-32, I/O size (MiB): 1, object class: SX
IOR/POSIX on DFUSE/IL easy, readers, 4 server nodes
client node type: n2-highcpu-32, I/O size (MiB): 1, object class: SX
bandwidth (GiB/s)
benchmark processes per client node
client nodes
1
2
4
8
12
16

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

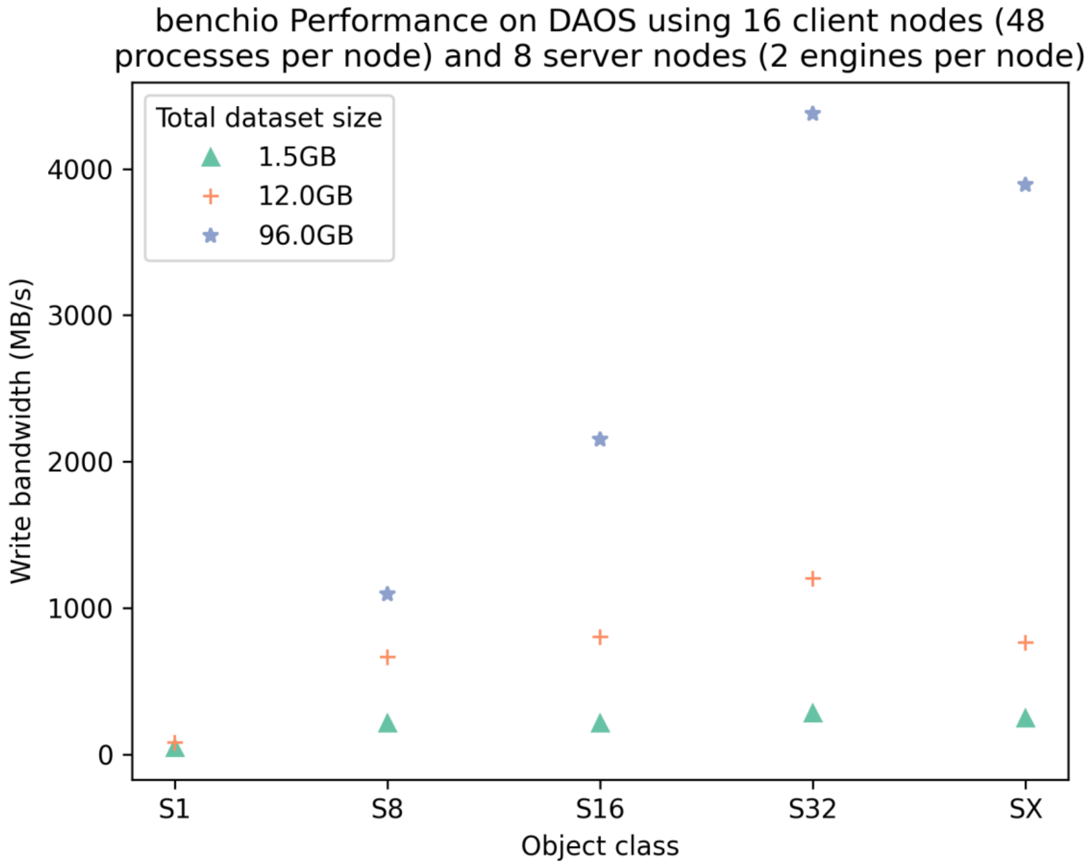
Index
64-bit Integer
Address array of value
Support operation over range of indices (= extent)

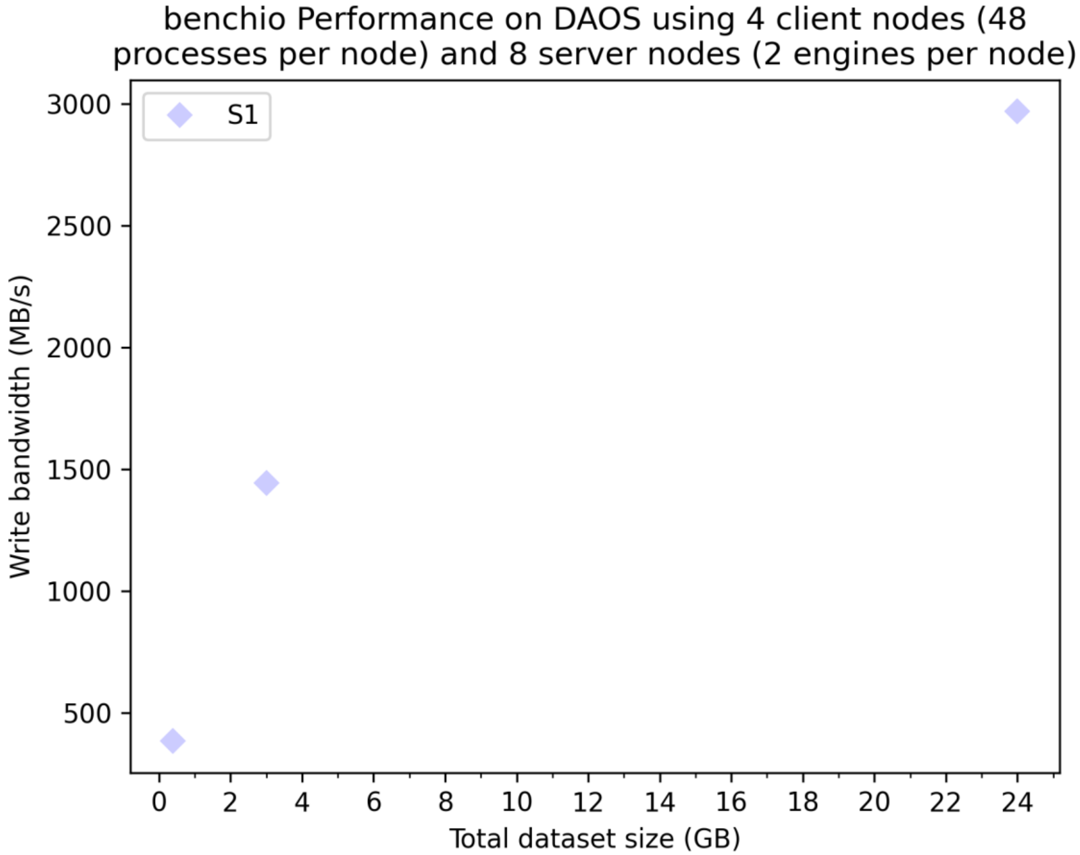
Record
Variable length value
From 1 byte to GB's

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

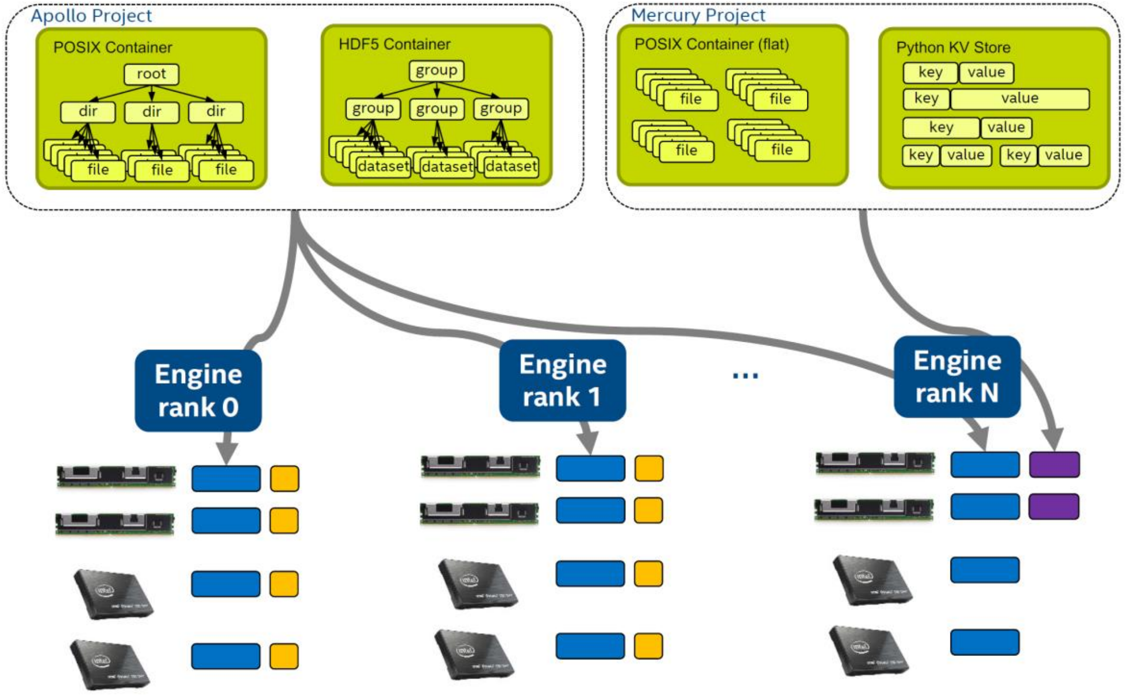
Object class





intel ECMWF epcc

System Hierarchy

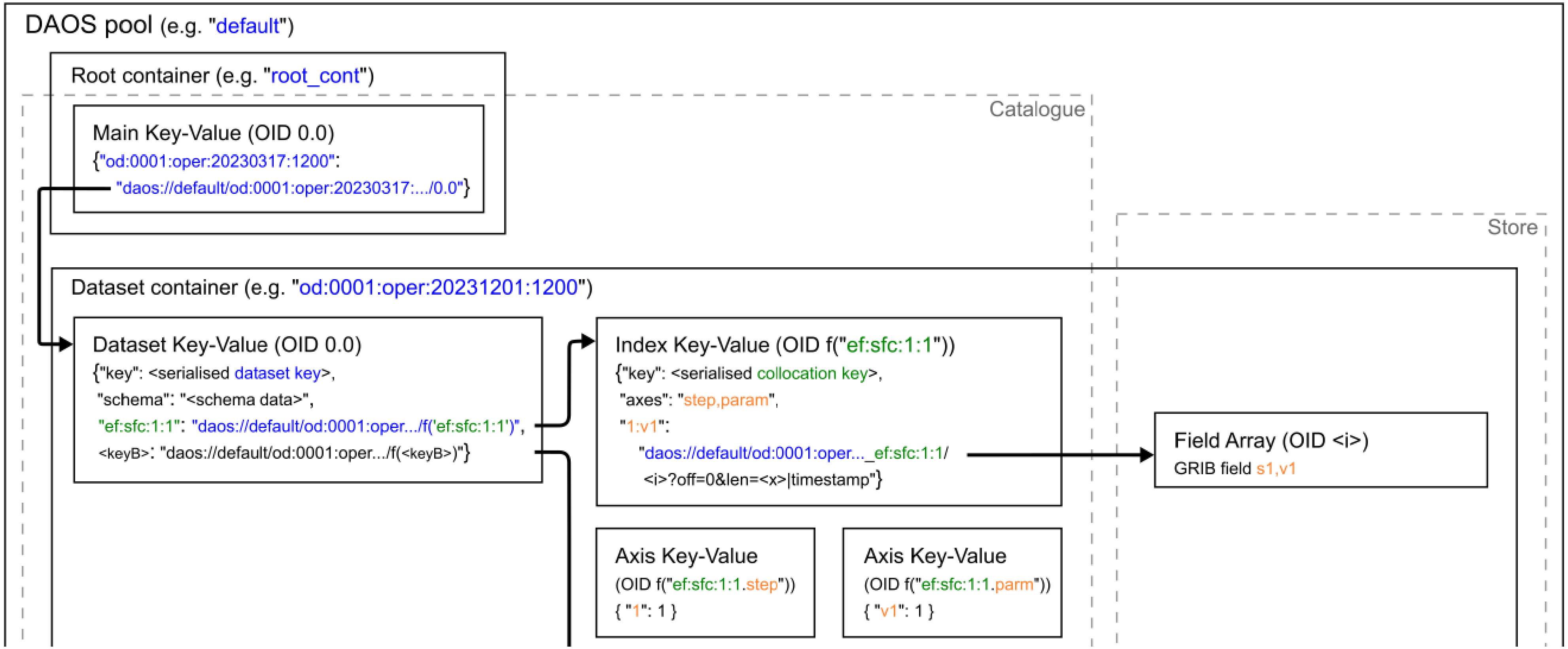


- Pools
    - Highest level construct in the system
    - Possibly analogous to a filesystem
    - Maybe more like a root of a tree
    - Maps to servers/storage hardware
    - Orders of ones per system
- Containers
    - Namespace/logical space inside a pool
    - Group together objects
    - Orders 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

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)
```

- Dataset key
- Collocation key
- Element key



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
```

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

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);
```

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

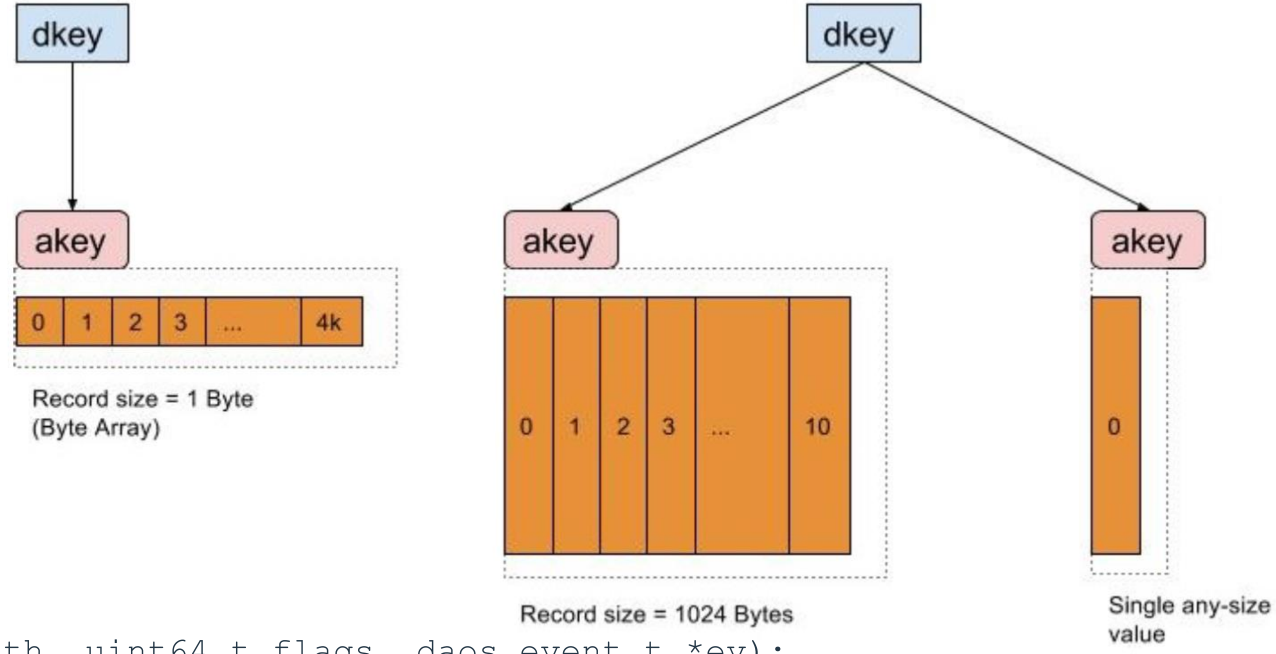
```
int daos_kv_open(daos_handle_t coh, daos_obj_id_t oid, unsigned int mode, daos_handle_t *oh,
daos_event_t *ev);

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`)

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);
```

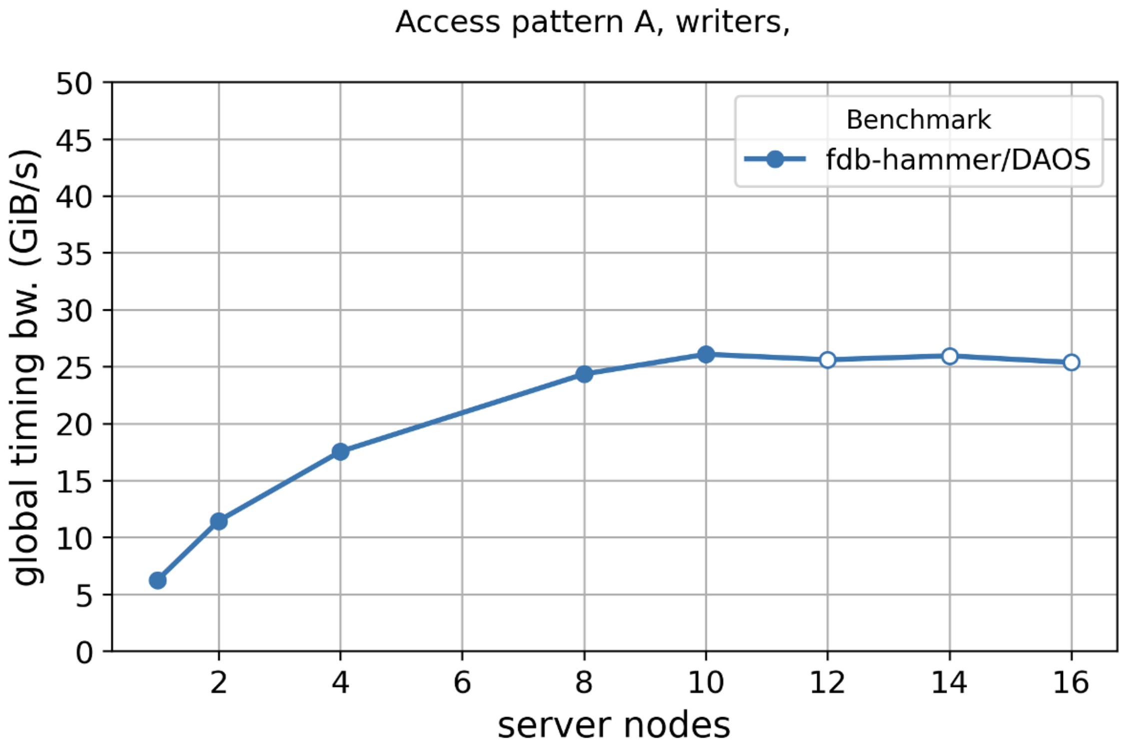
Multi-level KV example

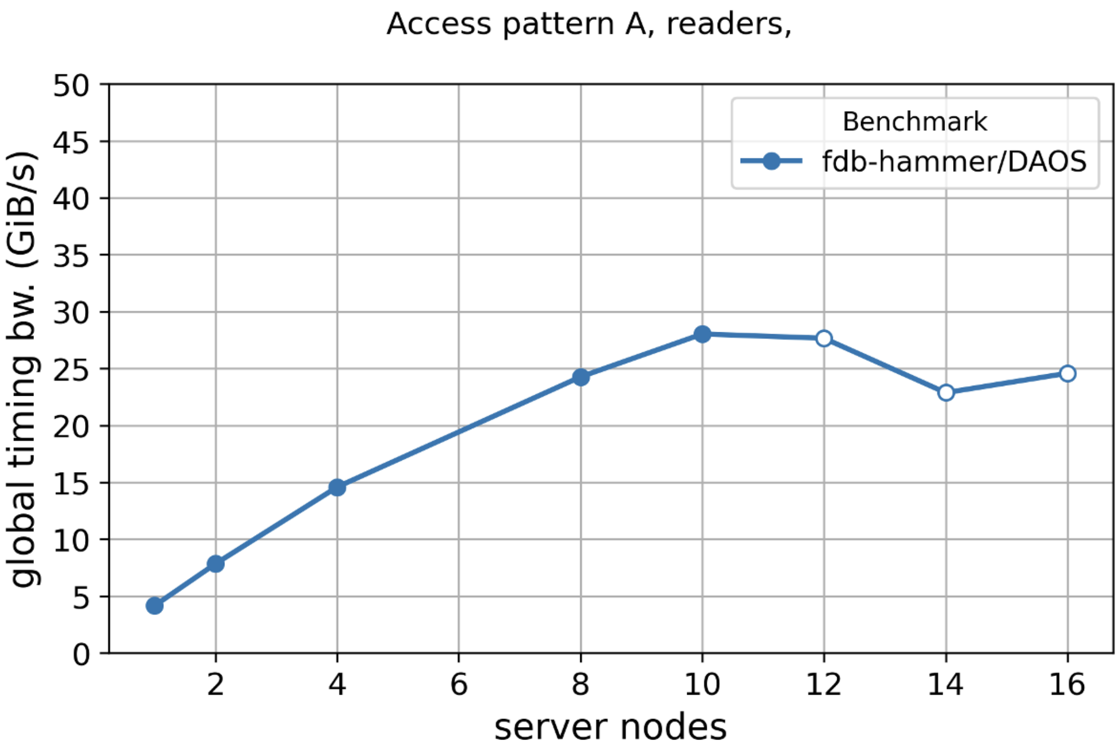
```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;
d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2",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

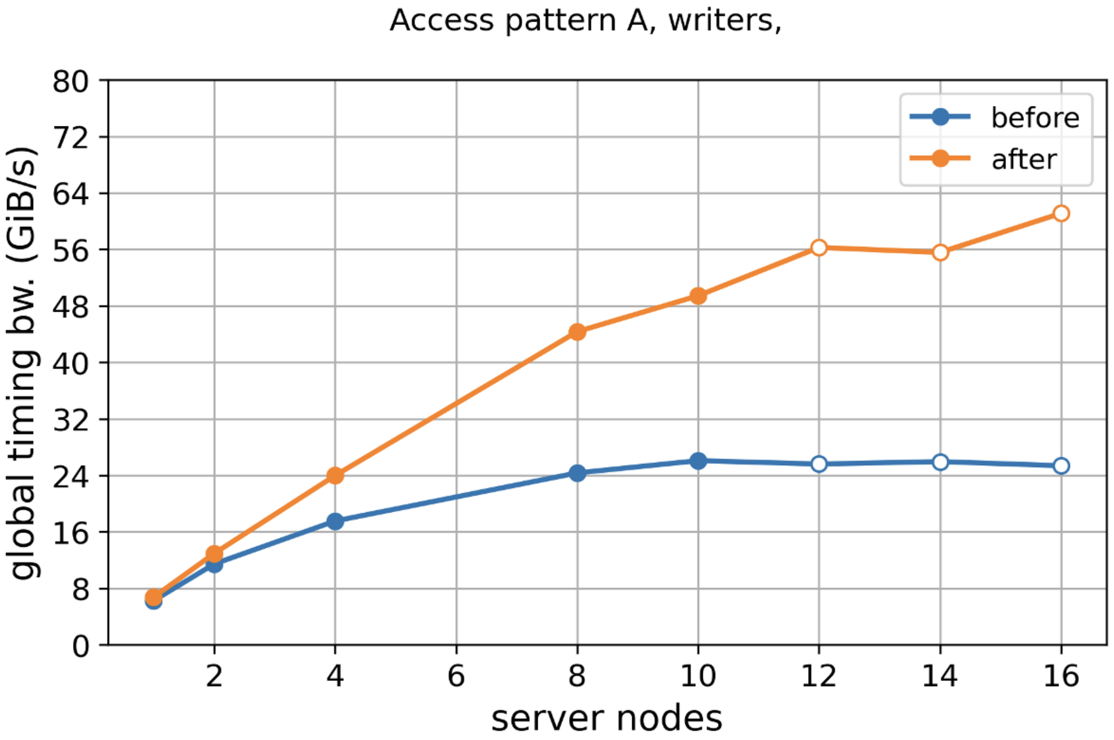
```
call daos_initialise(pool_name_c, cartcomm)
call daos_write_array(ndim, arraysize, arraygsize, arraysubsize, arraystart, out_data, object_class_c, blocksize, check_data, daosconfig, cartcomm)
call daos_write_object(ndim, arraysize, arraygsize, arraysubsize, arraystart, out_data, object_class_c, blocksize, 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);
```

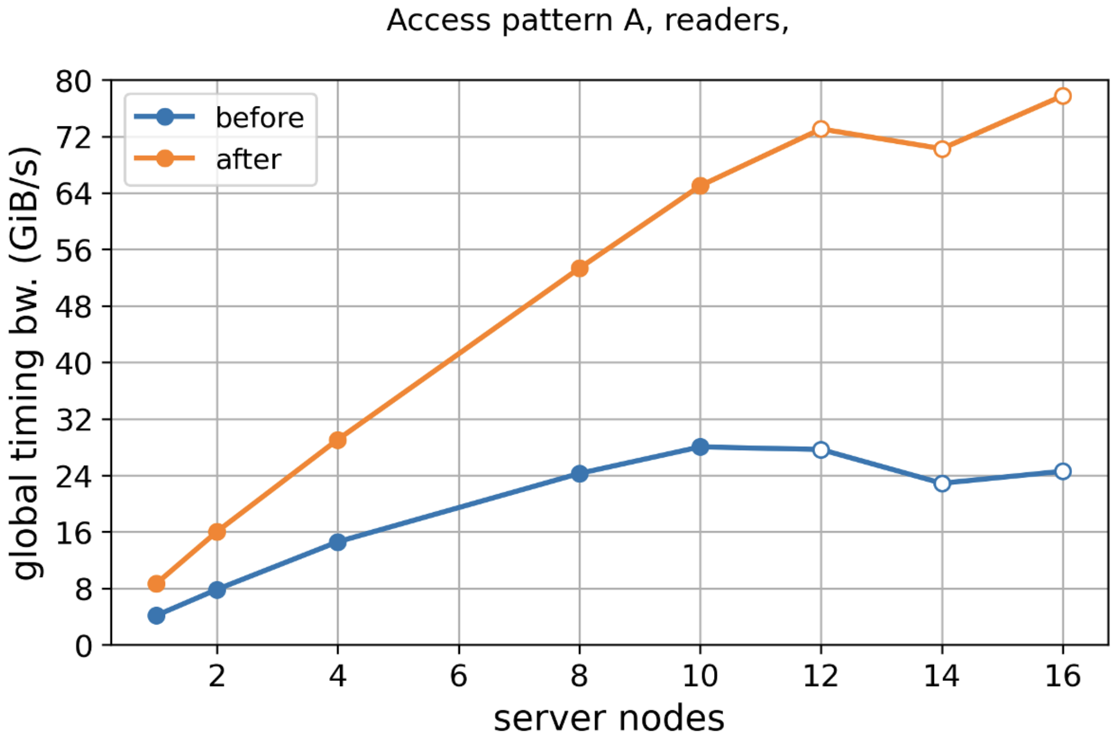
Evaluate performance/approach





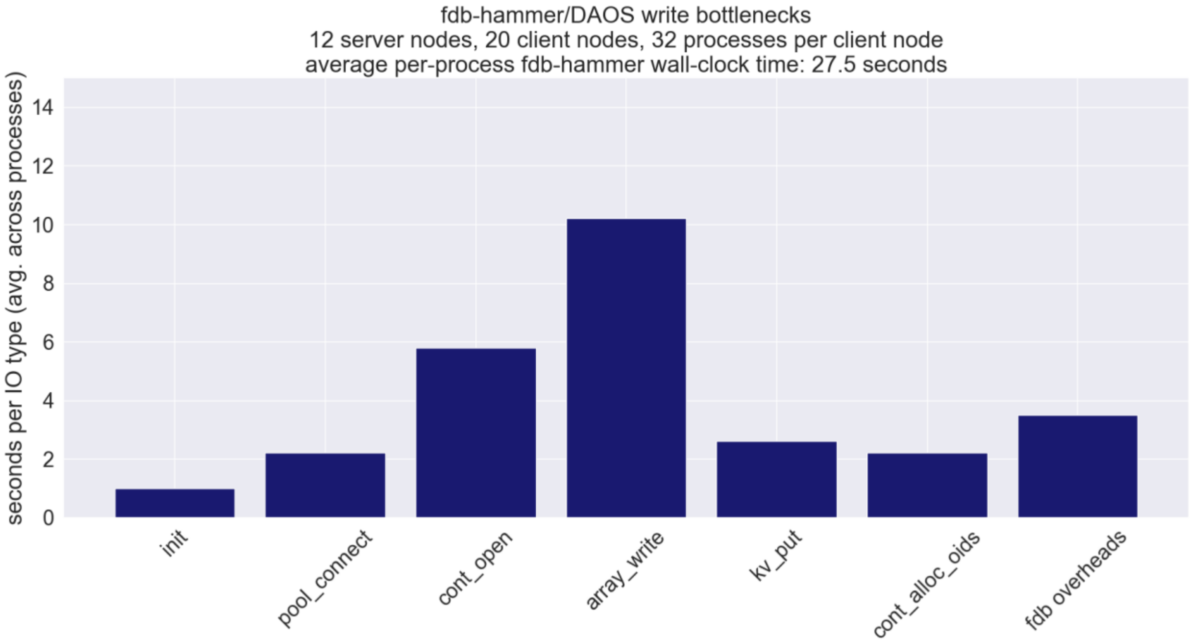
Optimised performance

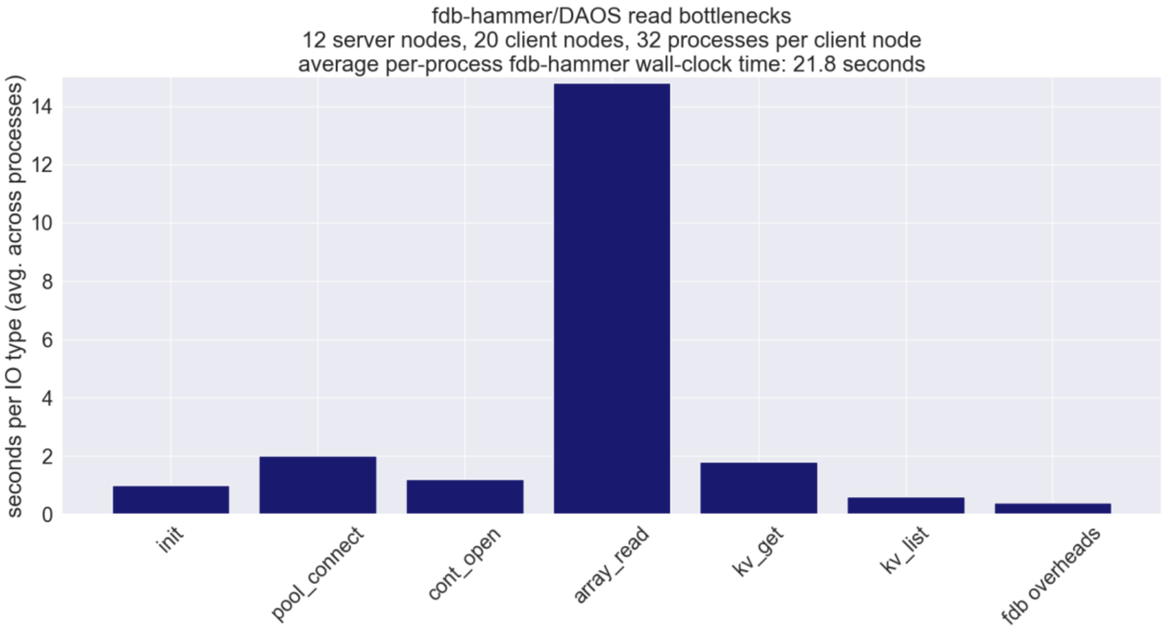




Profiling

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





intel ECMWF 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

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

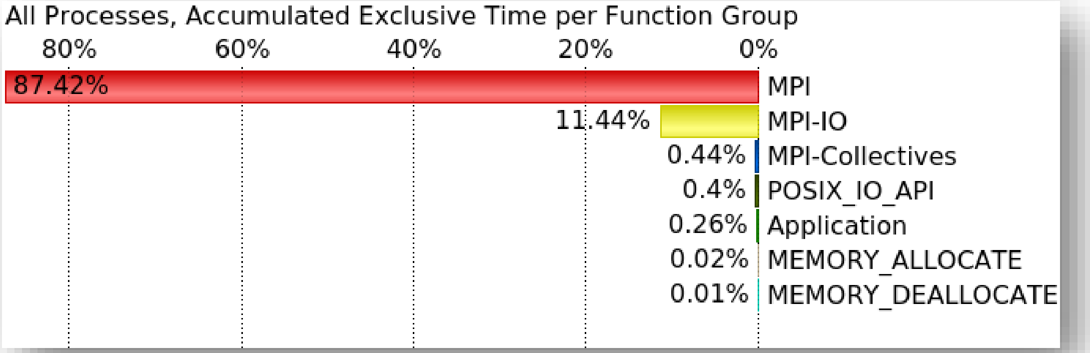
Approach/recommendations

- `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

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