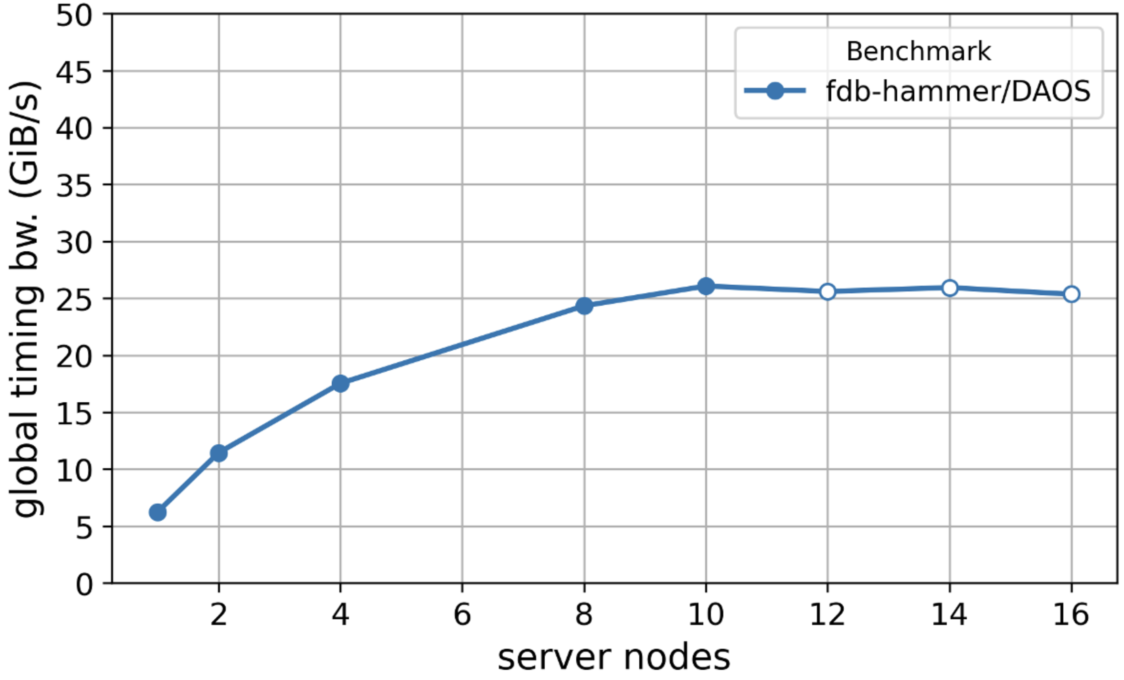
SUMMARY

Aside: DAOS Fortran interfacing

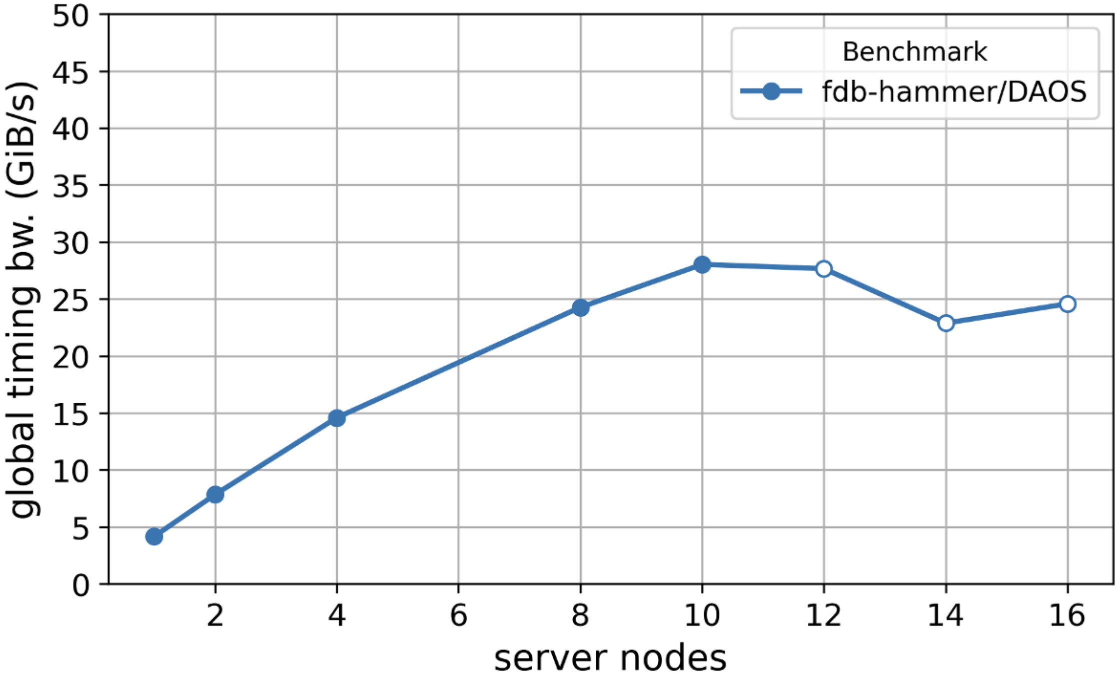
```
call daos_initialise(pool_name_c, cartcomm)
call daos_write_array(ndim, arraysize, arraygsize, arraysubsize, arraystart, out_data, object_class_c, blocksize, check_data, daosconfig, cartcomm)
call daos_write_object(ndim, arraysize, arraygsize, arraysubsize, arraystart, out_data, object_class_c, blocksize,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

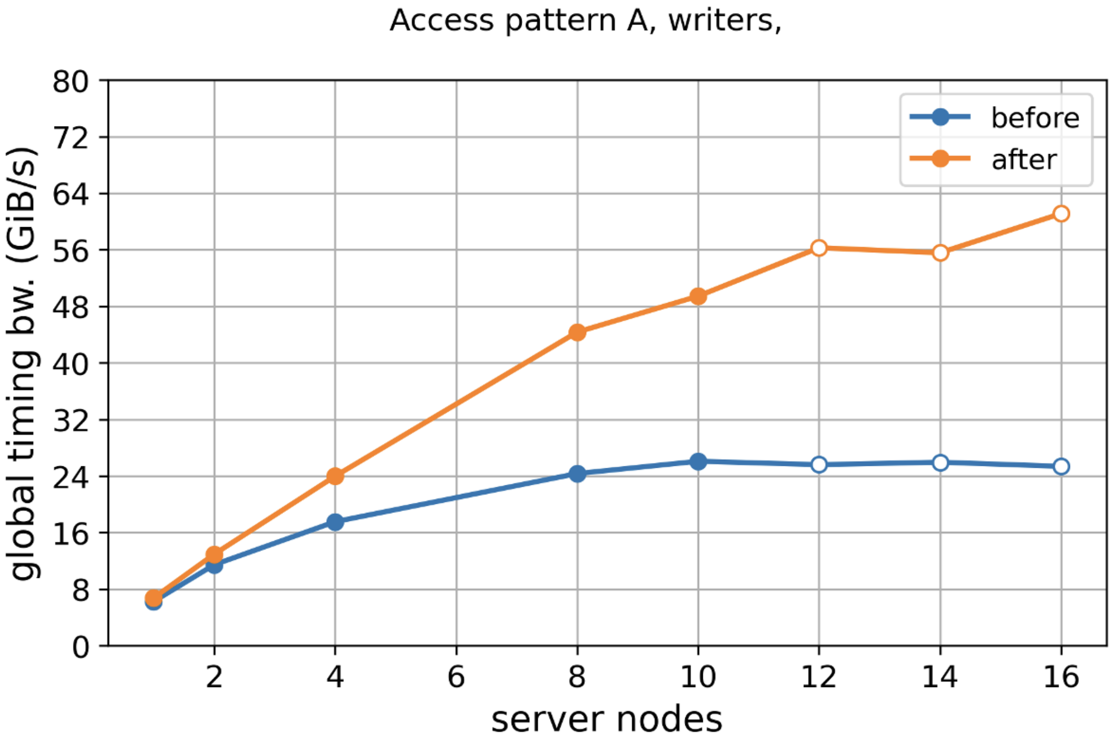
Access pattern A, writers,

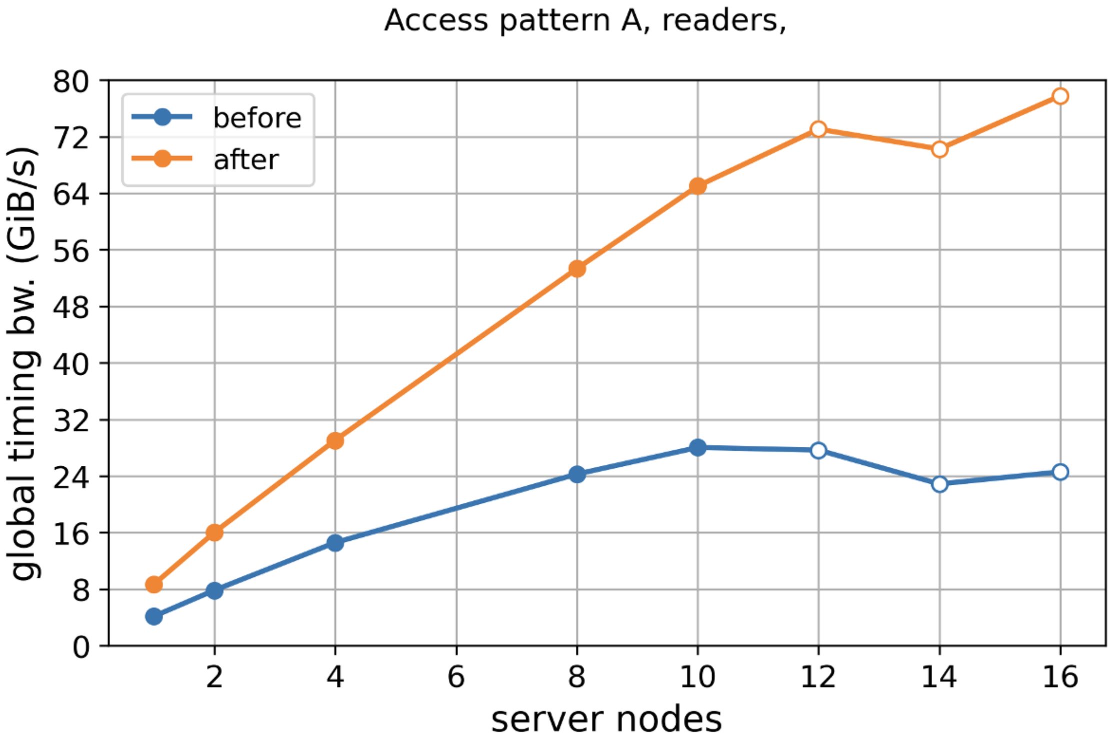


Access pattern A, readers,



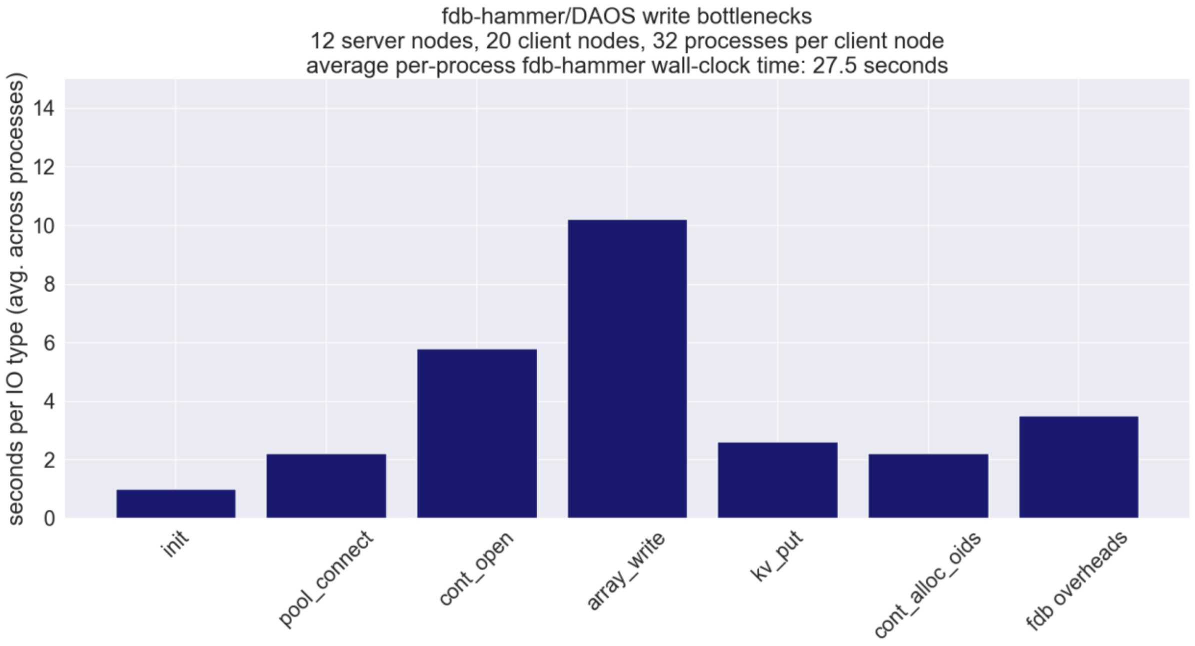
Optimised performance

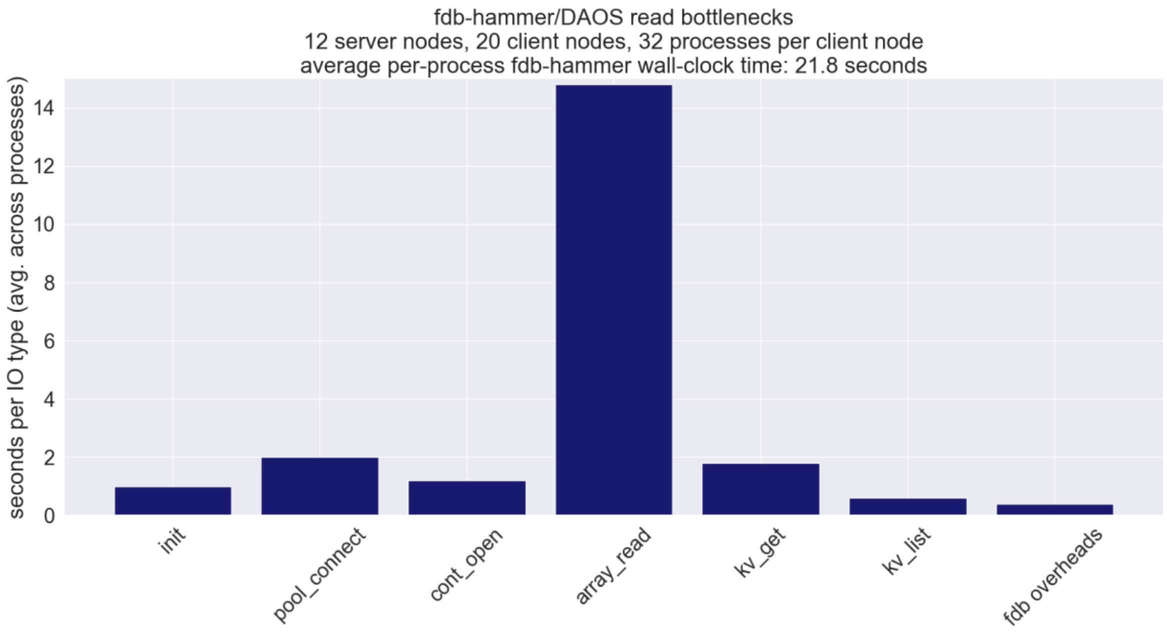




Profiling

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


fdb-hammer/DAOS write bottlenecks
12 server nodes, 20 client nodes, 32 processes per client node
average per-process fdb-hammer wall-clock time: 27.5 seconds


fdb-hammer/DAOS read bottlenecks
12 server nodes, 20 client nodes, 32 processes per client node
average per-process fdb-hammer wall-clock time: 21.8 seconds

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

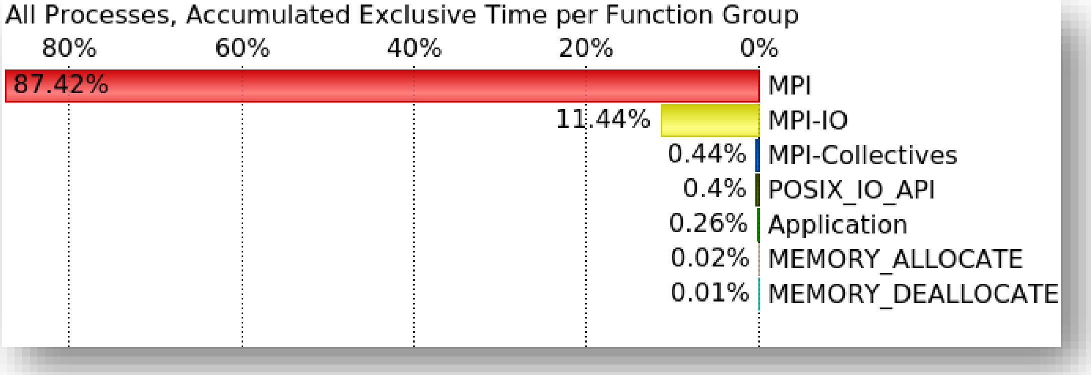
Approach/recommendations

- `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

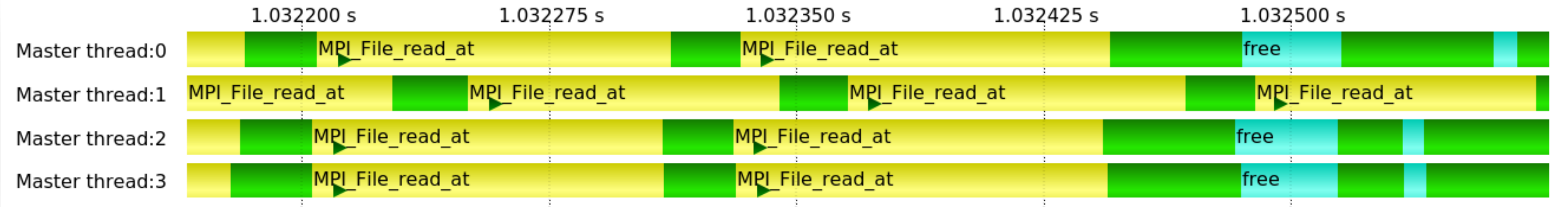
DAOS usage design

- 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



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



intel ECMWF Google Cloud epcc