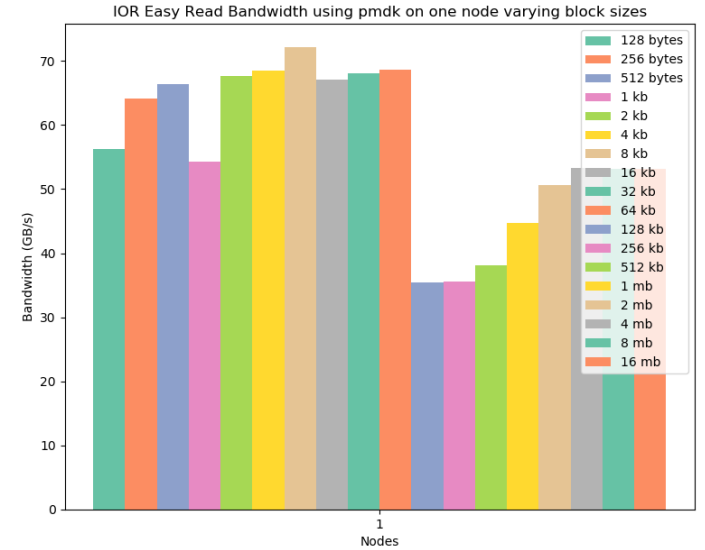
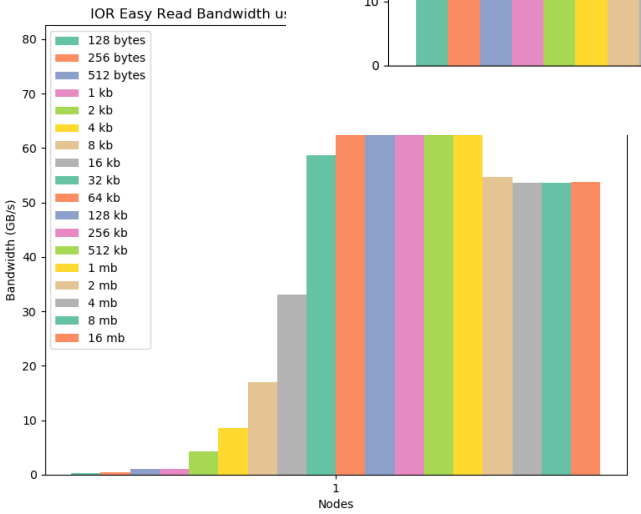
HIGH PERFORMANCE I/O

Hardware and Software interfaces

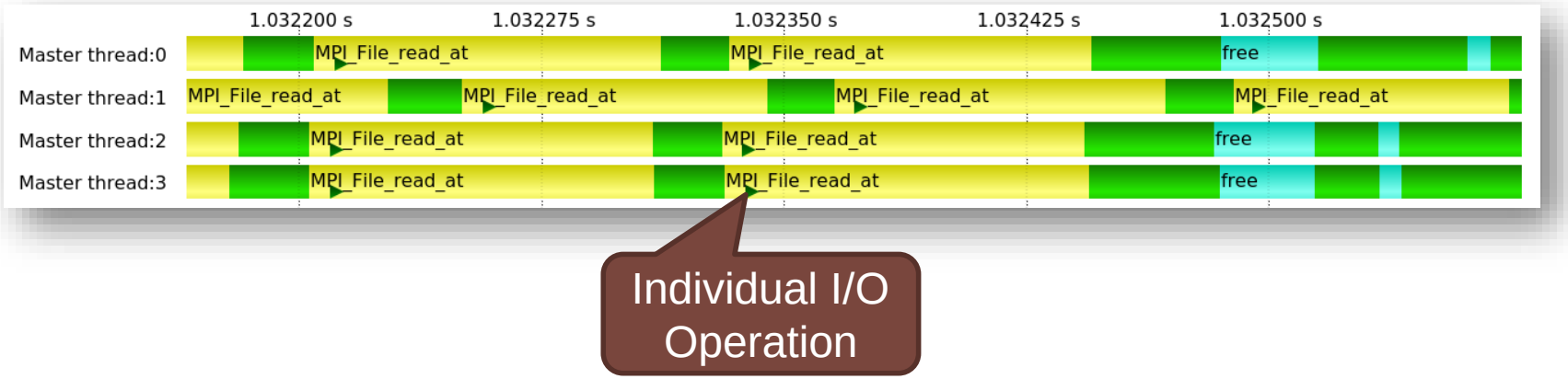
Byte Addressable/Granularity

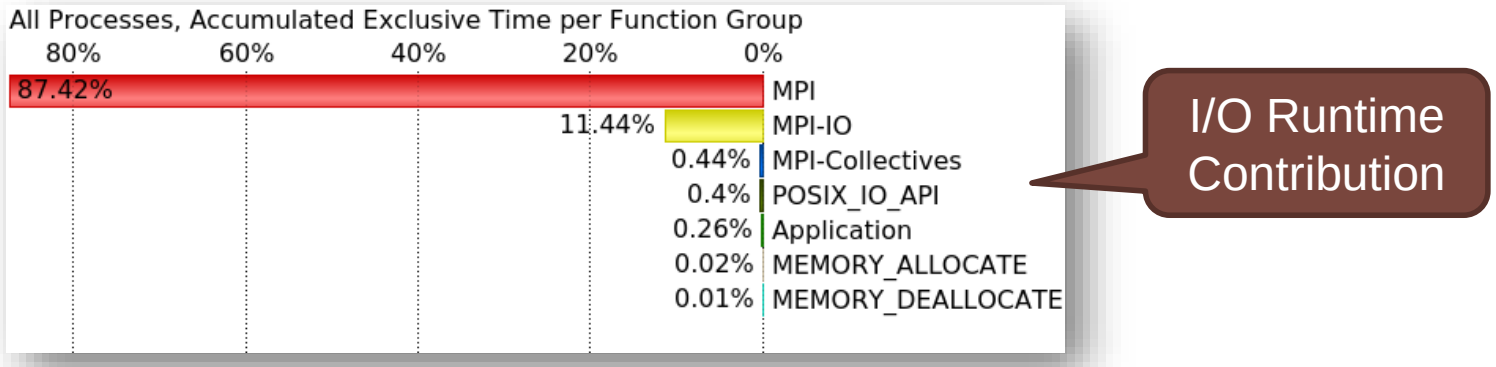
- A key feature of I/O access is the size of operations
  - Data access is the same cost for any size of operation
    - Not really, but much closer than for standard files
- Persistent memory allowed smaller accesses
  - Individual bytes accessible without large operations
  - In reality, cache-line level access
- Object storage moves away from block I/O
  - I/O and data operations can be small
  - Restructure I/O/applications





Granularity





intel

ECMWF

epcc

Standard I/O Programming

```
my_file = open("data_out.dat", O_RDWR);
read(my_file, a, bufsize);
write(my_file, b, bufsize);
close(my_file);
```

```
open(fid, "checkpoint.dat")
do n = 0,2*nharms
    write(fid)(((q(i,j,k,n),i=-1,imax1),j=-1,jmax1),k= 1,kmax1)
end do
close(fid)
```

Standard I/O Programming

- Writing to O/S buffer, operating system writes that back to the file
  - Potential for O/S caching
  - Writes data in large chunks, bad for random access
  - Requires interaction with O/S
  - I/O consistency application responsibility
    - Flush required to ensure actual persistency

- Required because of the nature of previous I/O devices
  - Asynchronous
  - I/O controller
  - Shared

Parallel I/O Programming

- Library function to write to shared or separate files
- Library collects data and orchestrates writing to the actual file
- Still block based ultimately, requires parallel filesystem to enable multiple processes writing at once to the same file
  - Or multiple processes accessing multiple files at once

```
call mpi_file_write(fid, linelength, 1, MPI_INTEGER, MPI…
```

Optimising I/O

- Optimising I/O performance can be done in a number of ways
  - Ensuring the minimal number of actual block writes is done
  - Ensure the minimal number of filesystem operations are performed
  - Use faster I/O hardware
  - Use multiple bits of I/O hardware
  - Map the file to memory for I/O operations
- Optimising hard for a range of use cases
  - Small I/O operations
  - Non-contiguous I/O operations
  - Contention on shared resources (network or filesystem)

Memory-mapped files

```
my_file = open("data_file.dat", O_RDWR);
data = mmap(NULL, filesize, PROT_READ|PROT_WRITE, MAP_SHARED, my_file, 0);
close(fd);
data[0] = 5.3;
data[1] = 75.4;
```

- Memory-mapping a file copies the file into main memory
  - Requires sufficient main memory to contain the file
- Operations can then be undertaken on that data in standard memory format
  - Load/store, cache-line level accesses
- Persistence to file requires flush
  - msync
  - Done on page level
  - Every dirty page written back to file system
- Page cache supports dirty flag
  - Only dirty pages written back
- Volatile until persisted

Higher level functionality

- Object storage has a different I/O approach
  - Data mapped to an object
  - System takes care of placement and storage details
- Similar to filesystem support for standard I/O

Object Stores

- Filesystems use Files as the container for blocks of data and the lowest level of metadata granularity

- Object stores use Objects as the container for data elements and the lowest level of metadata granularity
  - Allows individual pieces of data to be stored, indexed, and accessed separately
  - Allows independent read/write access to “blocks” of data

Object Stores

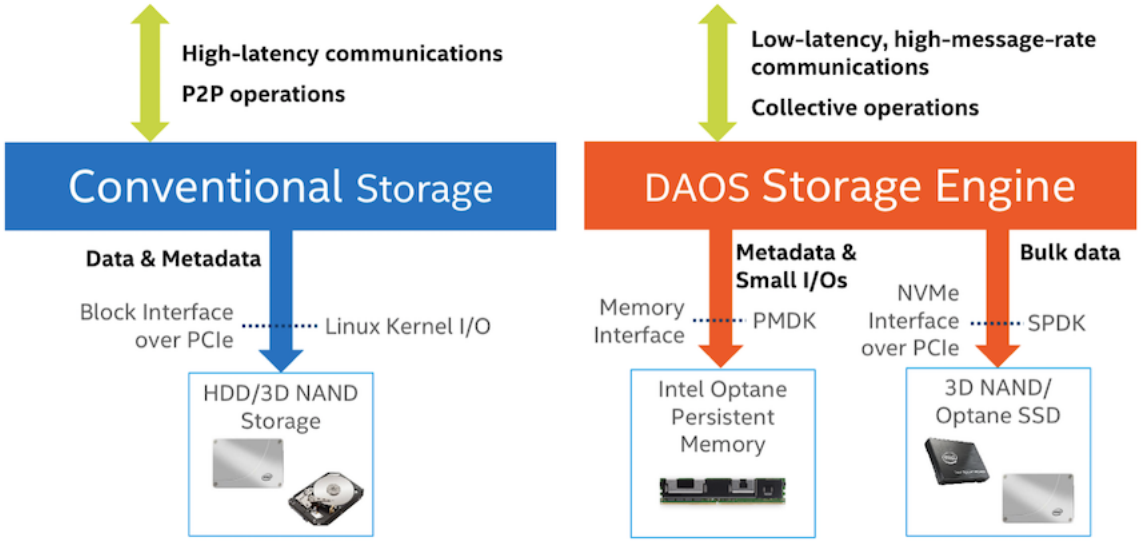
- Generally restricted interface
  - Put: Create a new object
  - Get: Retrieve the object
  - Removes the requirements for lots of functionality r.e. POSIX style I/O
- Traditionally objects are immutable
  - Once created cannot be changed
  - This removes the locking requirement seen for file writes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

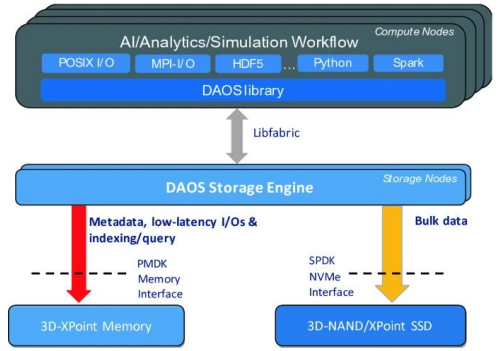
Object Stores

- Generally have 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
- Often web interfaces as part of the Put/Get interface

DAOS

- Native object store on non-volatile memory and NVMe devices
  - 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 store

- Software approach to granular data access
- Enabling generating and querying data on different “dimensions”
- Enabling data sizes to very whilst maintaining performance
- Enable “legacy” interfaces

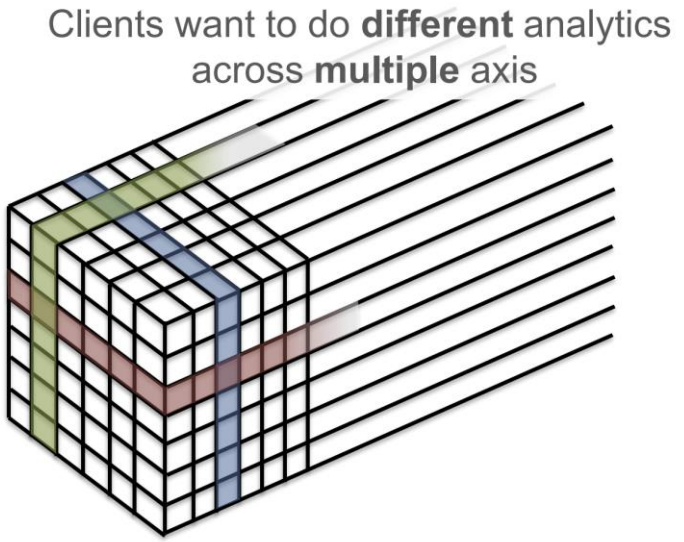


Image courteous of Tiago Quintino from ECMWF

intel ECMWF epcc

Practical Setup

- https://github.com/NGIOproject/ObjectStoreTutorial
- Take IOR and STREAMS source code
- Run on the GCP system