The ExcaliData Implementation of Active Storage.

ExcaliData is an Excalibur Cross Cutting Project
Excalibur is a UK exascale readiness programme
https://excalibur.ac.uk

Grenville Lister
(and many more, see next slide)

National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL

University of Reading

EXCALIBUR 10

ExcaliData — Active Storage Authors and Partners

ExcaliData is a big project, active storage is one part of it, where the key participants are:

- Bryan Lawrence (UoR & NCAS)
- Jean-Thomas Acquaviva (DDN)
- Konstantinos Chasapis (DDN)
- Scott Davidson (StackHPC)
- Mark Goddard (StackHPC)
- David Hassell (UoR & NCAS)
- Grenville Lister (UoR & NCAS)
- Valeriu Predoi (UoR & NCAS)
- Matt Pryor (StackHPC)
- Stig Telfer (StackHPC)

University of Reading

National Centre for Atmospheric Science
NATURAL ENVIRONMENT RESEARCH COUNCIL

ddn

StackHPC

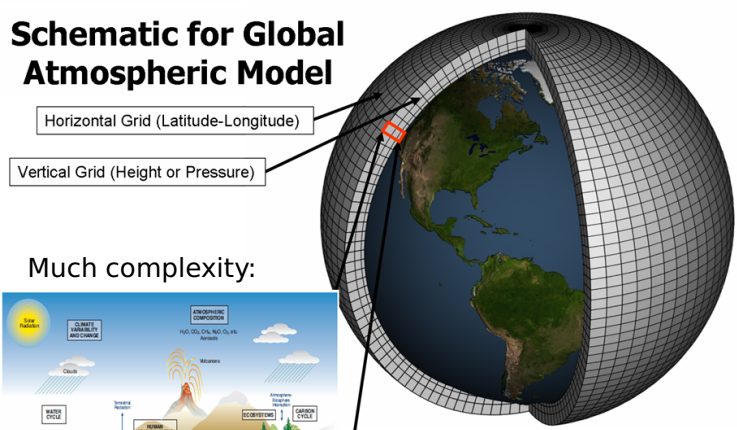
Funding via

Met Office

Outline

Co-Design: Not just about HPC Simulation, about analysis too!

Everything is solved on a grid

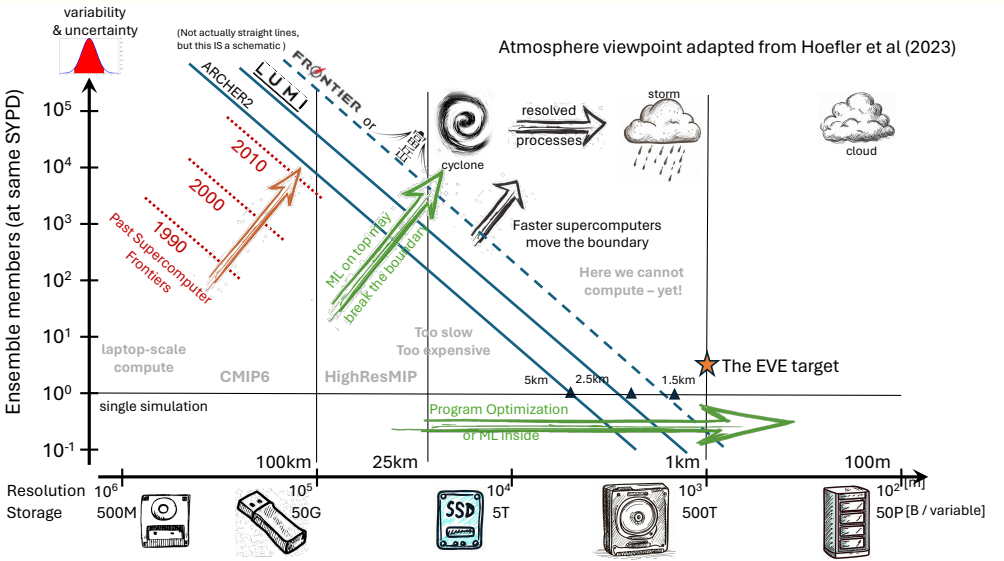


Much complexity:

Given knowledge of state of
- o(100) variables
at every grid point
for time t,
calculate
at every grid point
for every variable,
state at $t + \Delta t$.

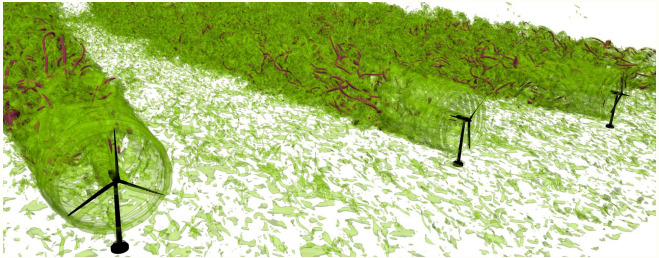
Many points, integrated for years with timestep of *o(minutes)*!

Climate modelling: an exaflop and an exabyte challenge



Excalidata and Active Storage are cross-cutting

Other big data challenges which should benefit from active storage:



Square Kilometre Array (SKA), Simulating turbulence, and much more.

Picture Credits:

- SKA Project
- Imperial College

Problem Statement

- Exascale science requires access to increasingly large amounts of data
- These data might not be located near to the compute resources
- Moving data over the network, from storage to compute nodes, is costly
- The cost can be thought of as
  - Time or CPU cycles taken to move the data (particularly for read operations, which are always blocking, we can buffer or offload writes)
  - Network bandwidth used (and cost of ensuring the necessary fabric exists)
  - Energy consumed (to move the data)

- Can we avoid some of that movement?
  - We could, if we could do some computation in storage!

What is Active (Computational) storage?

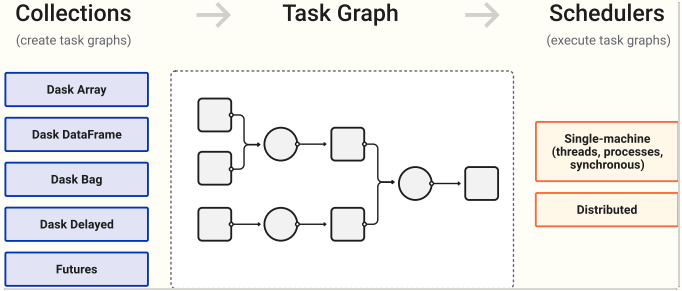
- Nearly all storage systems now have lots of compute, it's necessary for error correction, and managing which bytes go where, but it's basically underused
  - JASMIN Quobyte storage system has (in 2021) 40PB usable storage powered by 131 compute nodes (3460 cores)
- For a long time now, people have been talking about utilising some of that underused compute, by doing "some compute tasks in storage".
- Previous ideas have included shipping functions, VMs or containers
  - If I ship a VM/container, how do I know it is not going to do bad things to the data and/or system?
  - How do I include complex systems in a workflow? How does this VM/container/function fit into my greater workflow?

What is Active (Computational) storage?

- Nearly all storage systems now have lots of compute, it's necessary for error correction, and managing which bytes go where, but it's basically underused
  - JASMIN Quobyte storage system has (in 2021) 40PB usable storage powered by 131 compute nodes (3460 cores)
- For a long time now, people have been talking about utilising some of that underused compute, by doing "some compute tasks in storage".
- Previous ideas have included shipping functions, VMs or containers
  - If I ship a VM/container, how do I know it is not going to do bad things to the data and/or system?
  - How do I include complex systems in a workflow? How does this VM/container/function fit into my greater workflow?
- We limit ourselves to reductions (a la MPI), and we let the Dask workflow tool handle the workflow (including identifying when we can use reductions).

Dask

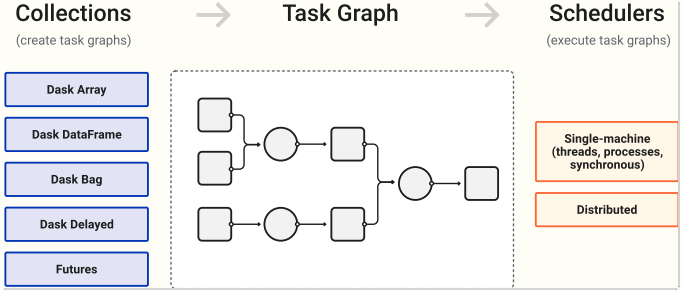
- A task-based parallel computing library for Python.
- Dask partitions work into *computational chunks*.



- Each *computational chunk* may map to multiple *storage chunks*, each of which will consist of a bunch of blocks in storage.
- Scheduler analyses workflow to create a task graph and allocates task to the available computing elements

Dask

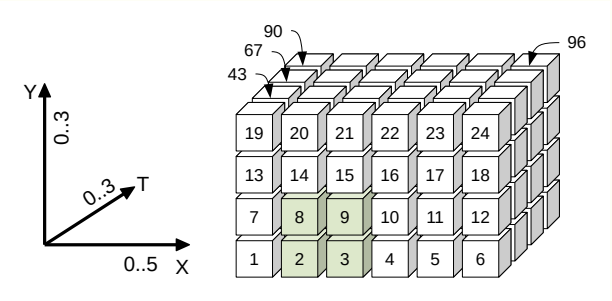
- A task-based parallel computing library for Python.
- Dask partitions work into *computational chunks*.



- Each *computational chunk* may map to multiple *storage chunks*, each of which will consist of a bunch of blocks in storage.
- Scheduler analyses workflow to create a task graph and allocates task to the available computing elements
- Objective: Manipulate the task graph to push some tasks into the storage!

Storage Chunks

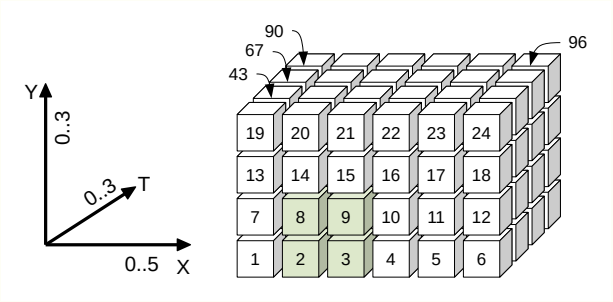
Consider a simple $6 \times 4 \times 4$ grid:



- Data is written one chunk at a time (the size of a chunk is under user control). Deeper down the stack one chunk might be multiple blocks.
- For HDF/NetCDF/Zarr, data is also read into memory one chunk at a time.

Storage Chunks

Consider a simple $6 \times 4 \times 4$ grid:



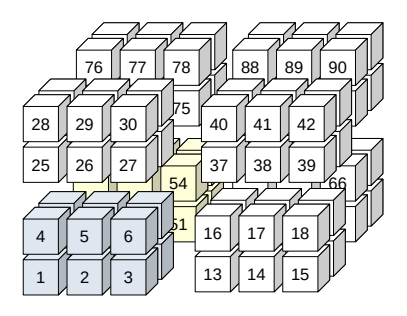
- Data is written one chunk at a time (the size of a chunk is under user control). Deeper down the stack one chunk might be multiple blocks.
- For HDF/NetCDF/Zarr, data is also read into memory one chunk at a time.
- Consider a chunk size of 6 and what happens if we we want to get a map (XY data) at a specific time (T) which corresponds to elements $(2, 3, 8, 9)$
  – we need to read two chunks, and extract what we need.

Storage Chunks (cont)

- The default chunking has a preferred access. If you are reading in the direction you wrote, it will be efficient.
- But much of the time we don’t do that.
- What about alternatives?

Storage Chunks (cont)

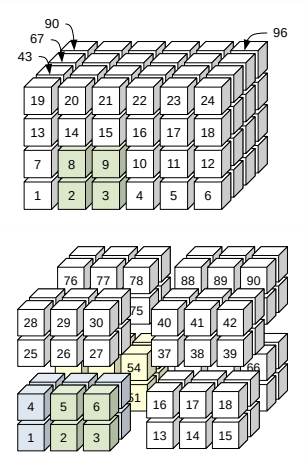
- The default chunking has a preferred access. If you are reading in the direction you wrote, it will be efficient.
- But much of the time we don't do that.
- What about alternatives?



- Now nearly all ways of sampling into the cube along different routes than the original will be approximately equally efficient (but chunk size and dimensioning matter . . . a lot)!

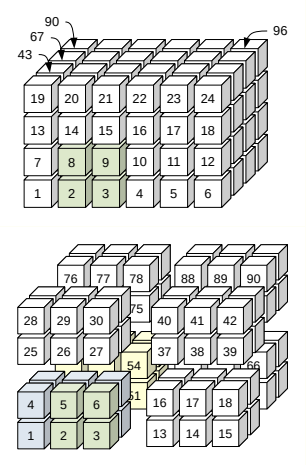
Why do we care about chunks?

- When we read data, we read data one chunk at a time
- If compressed, the chunk is uncompressed, then
- We extract the slice we want from the chunk.
- In the (green) map example, with the two toy chunking strategies shown, EITHER
  - read two chunks, decompress two chunks, extract two slices, build map, OR
  - read one chunk, decompress one chunk, extract one slice build map
- We might have other interesting chunk properties too: missing data, filters etc.

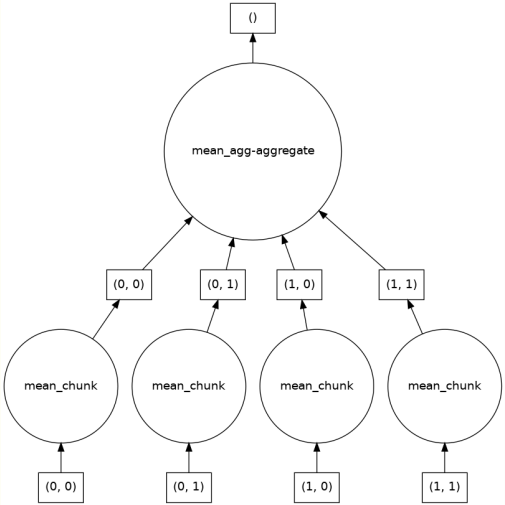


Why do we care about chunks?

- When we read data, we read data one chunk at a time
- If compressed, the chunk is uncompressed, then
- We extract the slice we want from the chunk.
- In the (green) map example, with the two toy chunking strategies shown, EITHER
  - read two chunks, decompress two chunks, extract two slices, build map, OR
  - read one chunk, decompress one chunk, extract one slice build map
- We might have other interesting chunk properties too: missing data, filters etc.
- Now extend our thinking from “extracting data for a map” to “doing some calculation” . . .



A simple reduction workflow: mean



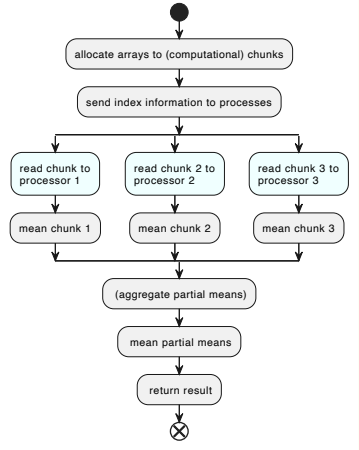
Taking a mean using four computational chunks:

- four lots of data re read from storage
- four means are taken
- means are aggregated result is calculated from the partial means
- requires reading all the data from the storage system, moving it into the compute node(s).

(see also Blelloch algorithm.)

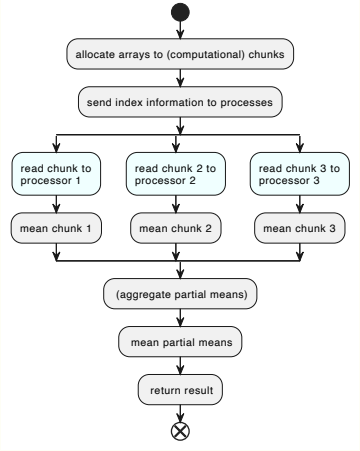
An active storage reduction workflow: mean

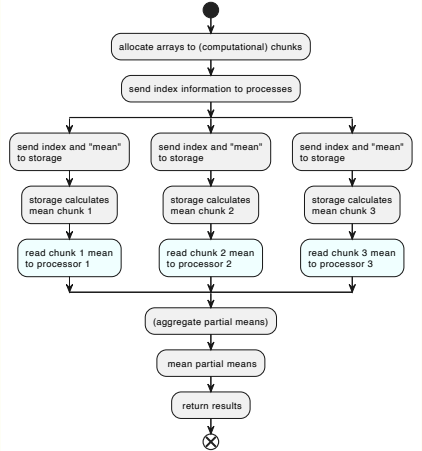
(This time with three, and note direction of flow has changed)



An active storage reduction workflow: mean

(This time with three, and note direction of flow has changed)

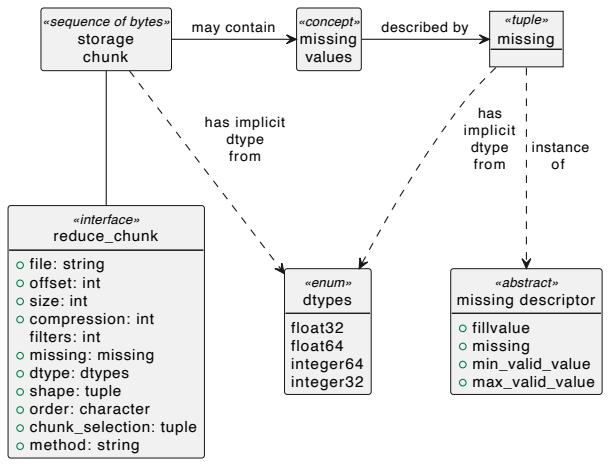




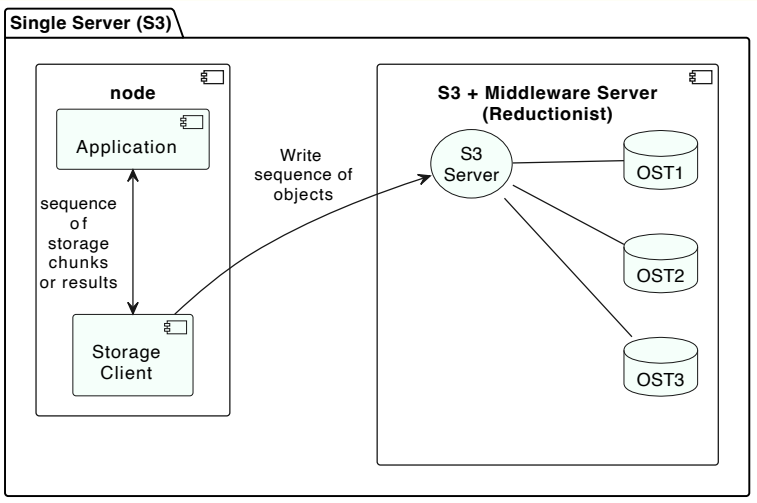
For, say a 3GB array, instead of reading 1 GB to each processor, we are reading a few bytes to each processor — much less data movement!

Some gory details: the interface

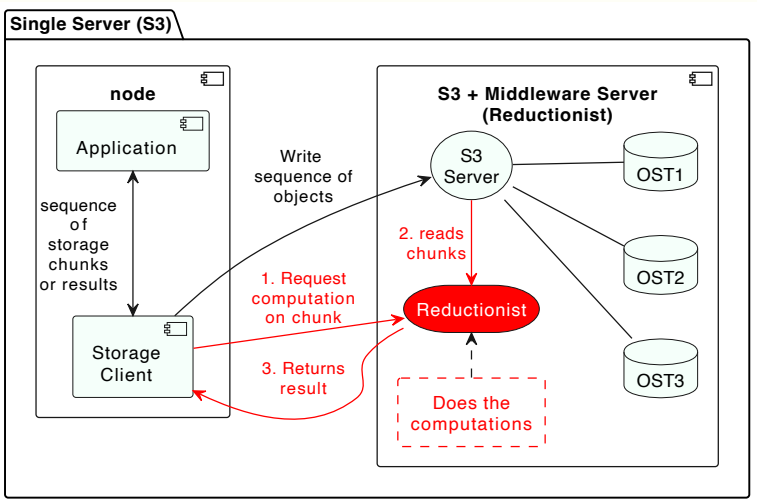
Each computational chunk is operating on many storage chunks, but we need to know quite a lot about each storage chunk to do the calculations - knowledge held by the application, but not the storage, unless we tell it.



Implementing Active Storage — S3



Implementing Active Storage — S3



Reductionist: https://github.com/stackhpc/reductionist-rs

Implementing Active Storage — Reductionist HTTP API

Request:

```
POST /operations/{operation}
Headers:
  Authorization: Basic =auth_token=
  Content-Type: application/json
Body:
{
"source": "https://s3.server.address/",
"bucket": "my-bucket",
"object": "path/to/object",
"dtype": "int32",
"byte_order": "little",
"offset": 0,
"size": 128,
"shape": [20, 5],
"order": "C",
"selection": [[0, 19, 2], [1, 3, 1]],
"compression": {"id": "zlib"},
"filters": [{"id": "shuffle", "element_size"
    : 4}],
"missing": {"missing_value": 42}
}
```

Response:

```
HTTP/1.1 200 OK
Headers:
  Content-Length: 4
  Content-Type: application/octet-stream
  x-activestorage-dtype: int32
  x-activestorage-byte-order: little
  x-activestorage-shape: []
  x-activestorage-count: 1
Body:
42
```

Implementing Active Storage — Reductionist HTTP API

Request:

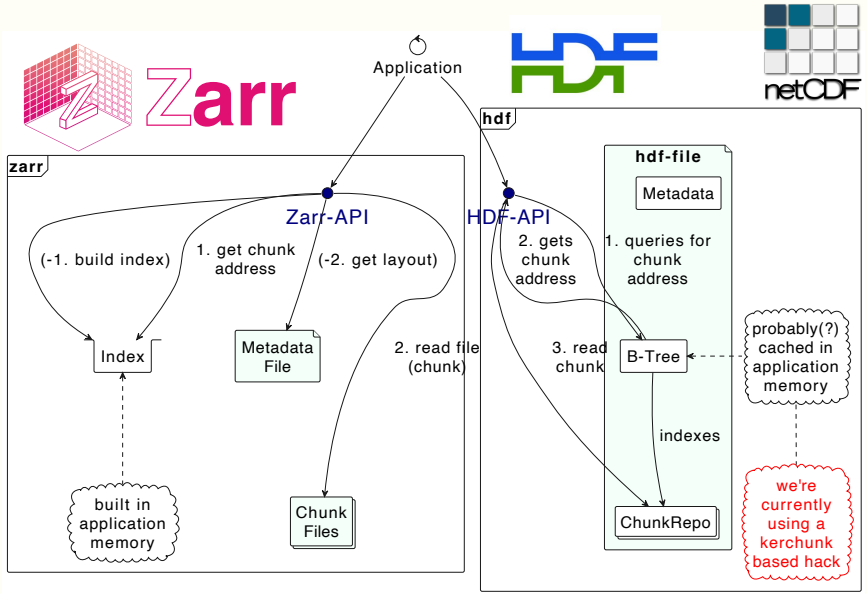
```
POST /operations/{operation}
Headers:
  Authorization: Basic =auth_token=
  Content-Type: application/json
Body:
{
"source": "https://s3.server.address/",
"bucket": "my-bucket",
"object": "path/to/object",
"dtype": "int32",
"byte_order": "little",
"offset": 0,
"size": 128,
"shape": [20, 5],
"order": "C",
"selection": [[0, 19, 2], [1, 3, 1]],
"compression": {"id": "zlib"},
"filters": [{"id": "shuffle", "element_size"
    : 4}],
"missing": {"missing_value": 42}
}
```

Response:

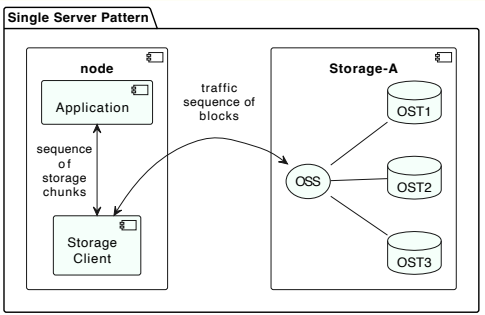
```
HTTP/1.1 200 OK
Headers:
  Content-Length: 4
  Content-Type: application/octet-stream
  x-activestorage-dtype: int32
  x-activestorage-byte-order: little
  x-activestorage-shape: []
  x-activestorage-count: 1
Body:
42
```

How do we find the offset for the chunk operation?

File Formats

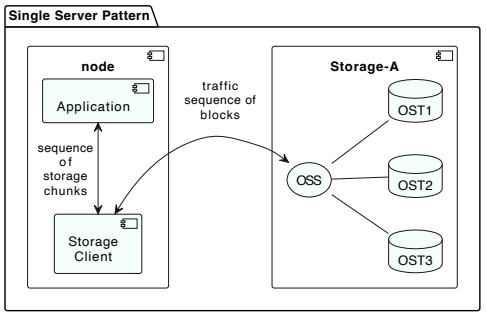


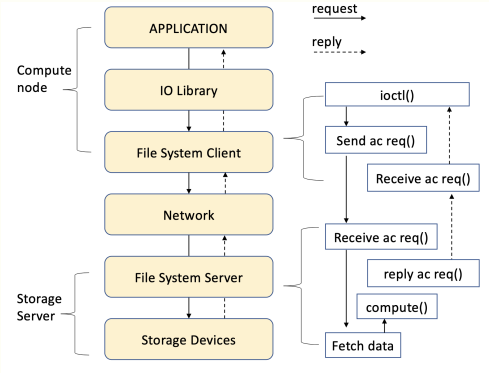
POSIX



- The big question is how to get the request through the storage stack?

POSIX





- The big question is how to get the request through the storage stack?
- Solution: Hack `ioctl`.
- (Implemented first in Fuse for proof of concept, then in DDN Infinia)

Active Storage Server – Compute Requirements

- Can address a byte range within the file (a chunk) and fill a buffer with those bytes
  - (return error if buffer can't support decompressed chunk.)
  - (respect float description and return error if unrecognised.)
- Support for specific filter and decompression algorithms (those supported by Zarr and netcdf).
  - (Return error for unrecognised filter and/or decompression.)
- Support for the defined operations: mean, sum, max, min, count (more could be added provided they were reductions); applied to a slice within the buffered array and respecting missing data.
  - (Return error for unrecognised operation.)
  - (Return error if slice operation is out of bounds or misunderstood.)
  - (Return error for unsupported missing description.)

(We have discussed a streaming version of this as well, but this is the simplest version.)

Active Storage - Client Side

- Science Application (untouched!): any script written in Python, utilising
- CF-Python (climate forecasting domain specific analysis library), which itself uses
- Dask (provides flexible task-based parallelism from threads to nodes).

In the current implementation we have modified CF-Python to patch the Dask graph when it recognises active storage to use the

- PyActiveStorage middleware (handles computational offload to storage and returns results).

to return the first computational reduction.
(Rather than what would otherwise be at the bottom of the graph: a read and then the computation).

Active Storage - Client Side

- Science Application (untouched!): any script written in Python, utilising
- CF-Python (climate forec… specific
- …allelism from

In th… …ified CF-P… …ecognises activ…

- …omputational …

to ret… …ion.
(Rath… …what would otherwise be at the bottom of the graph: a read and then the computation).

It is possible to handle it all in Dask without modification further up the stack.
We have demonstrated that in a fork of Dask, but we wont try to push our patches upstream until there are some active storage servers deployed for real.

Summary

- We've built and tested code to push filters, decompression, and reductions into storage.
  - S3 implementation is feature complete. Performance testing planned for early next year.
  - POSIX implementation in DDN Infinia shows promise.
  - Interested in encouraging other POSIX implementations.
- Existing version is domain specific (utilising CF-Python) but methodology to make discipline independent (via minor modification to Dask) exists.
- First science deployment likely in 2025 as part of European Horizon Europe project — EXPECT.