PANGEO

Petabyte-scale ocean data analytics on staggered grids via the grid ufunc protocol in xGCM

Or: Can we analyse the largest ocean simulation ever?

Thomas Nicholas*,
Julius Busecke, Ryan Abernathey

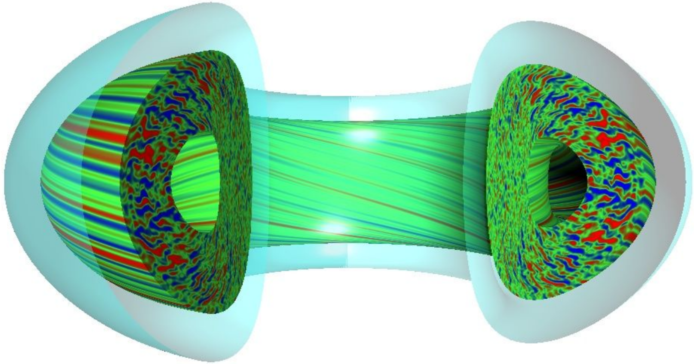
Lamont-Doherty Earth Observatory
Columbia University | Earth Institute

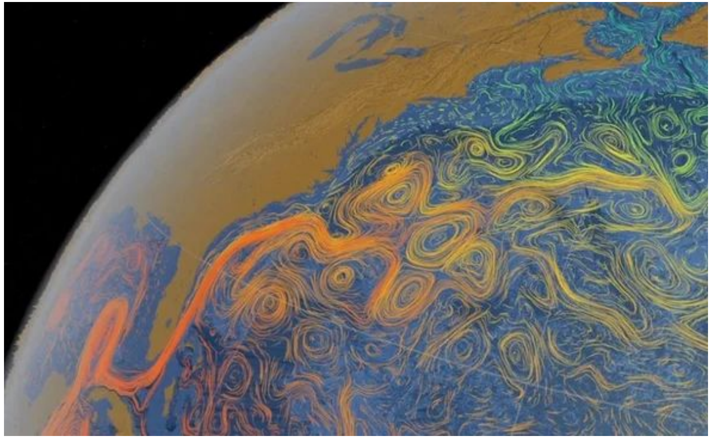
*thomas.nicholas@columbia.edu

*github.com/TomNicholas

Who am I?

- Oceanographer (ex-plasma physicist)
- Xarray core developer
- Pangeo user
- My first SciPy! 😁



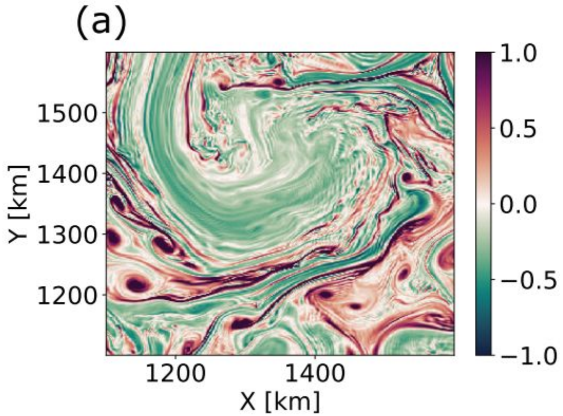


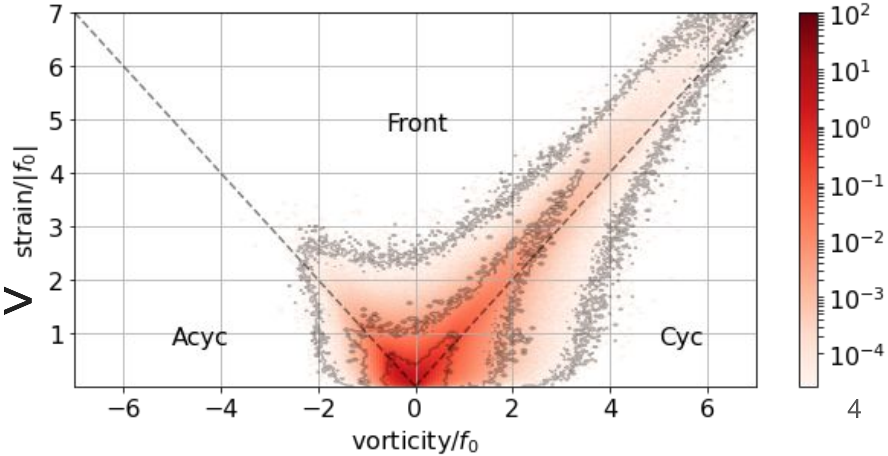
Talk overview

1. Science question 🔬
2. Scale challenges ⚖️
3. Refactoring for scalability 🚀
4. xGCM and “grid ufuncs” ⚛️
5. Dask scheduler improvements 📈

Science question: Submesoscale ocean ventilation

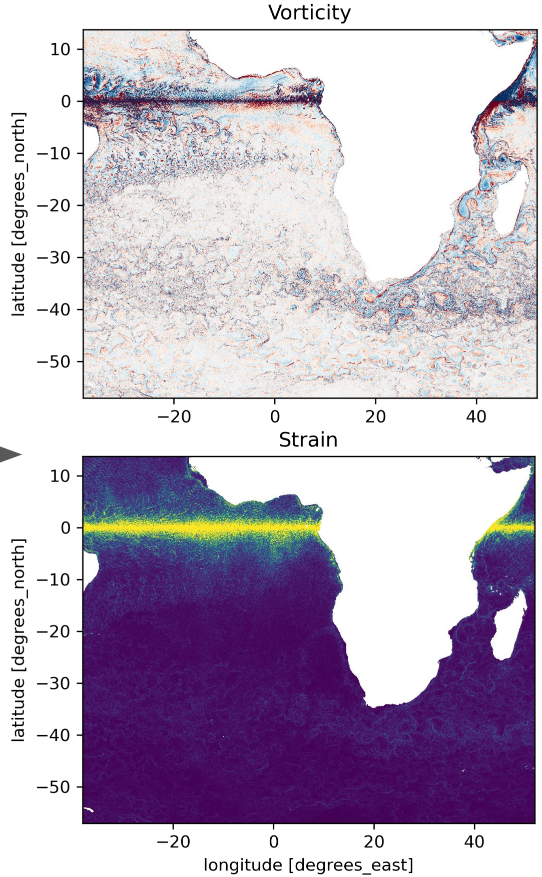
- Submesoscale ocean flows important for heat / gas transport
- Balwada (2021) analysed surface vorticity-strain correlations
- So we want to compute:
    - 1) Vorticity from flowfield
    - 2) Strain from flowfield
    - 3) Joint vorticity-strain PDF ->



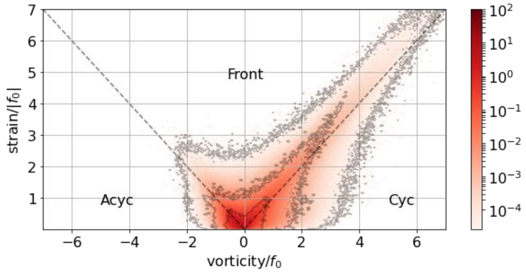


Calculation steps

Gridded velocity data
on many faces



Calculate vorticity &
strain from velocities



Bin vorticity and strain into 2D
histogram (+ time average)

Software problem #1: Scale

Aghulhas Rings

- Huge dataset to analyse
  - LLC4320 - global ocean MITgcm simulation (Menemenlis 2018)
  - 1/48° horizontal spacing
  - Hourly output

- Lives on pangeo cloud storage

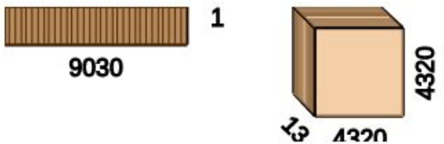
- Single variable is 8.76 TB
  - 117390 Zarr chunks

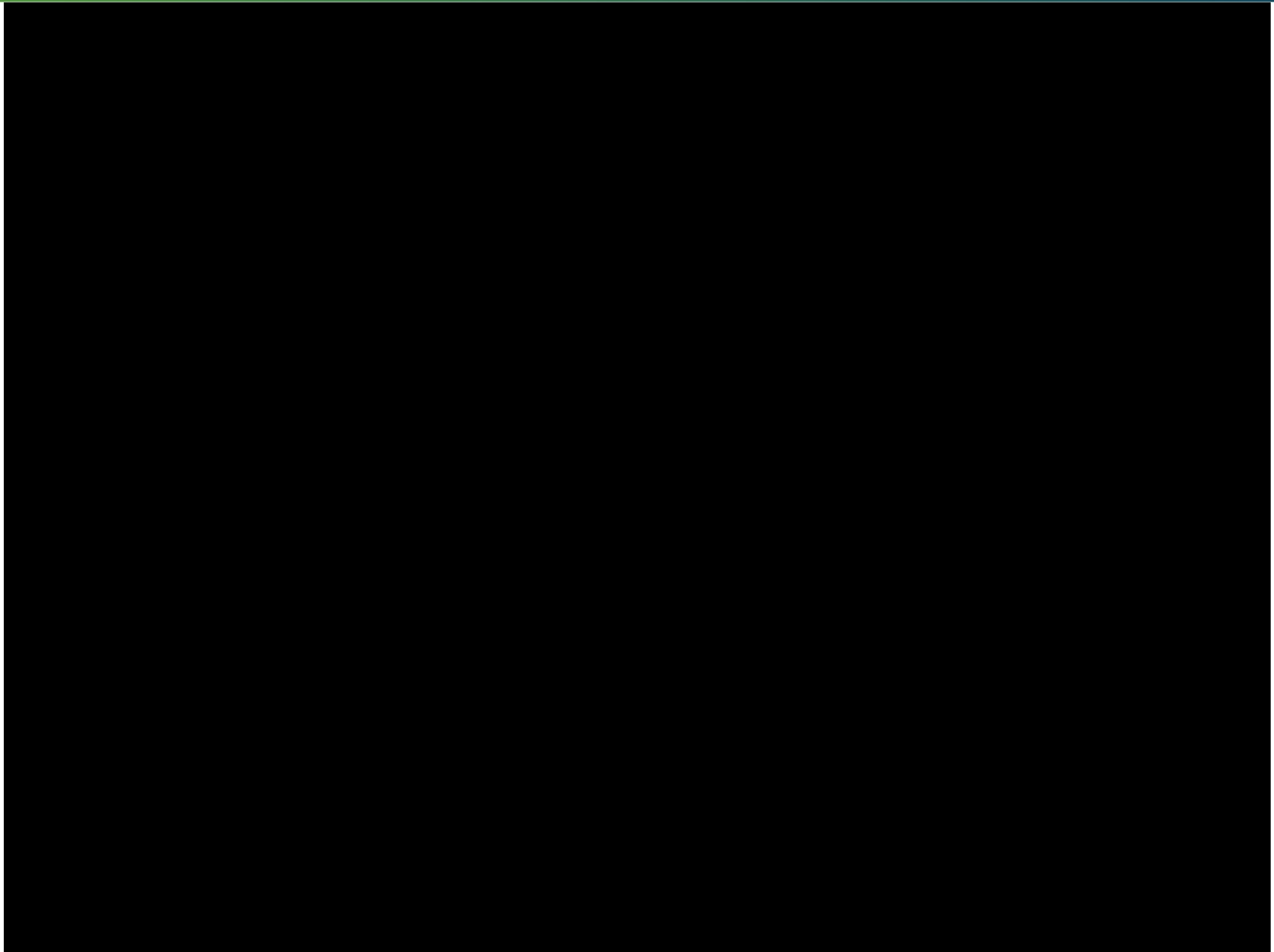
- That's just the sea surface!
  - Full dataset is 90x bigger! = ~4PB in total

▾ Data variables:

U (time, face, j, i_g) float32 dask.array<chunksize=(1, 1, 4320, 4320)>

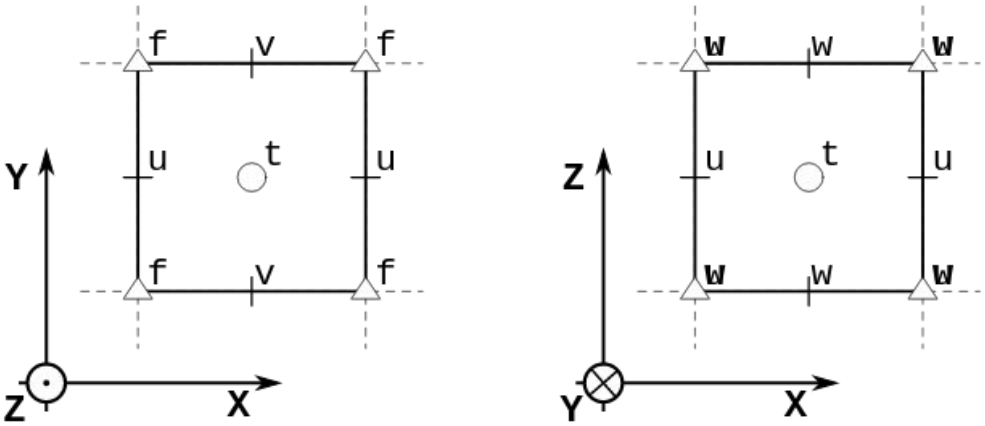
	Array	Chunk
Bytes	7.97 TiB	71.19 MiB
Shape	(9030, 13, 4320, 4320)	(1, 1, 4320, 4320)
Count	117390 Tasks	117390 Chunks
Type	float32	numpy.ndarray





Software problem #2: Staggered grids

- Fluid variables live on “Arakawa Grids”
- Variables’ positions are offset
- Finite-volume calculations must account for this to get correct results

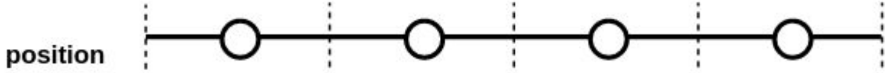


C-grid — horizontal view

C-grid — vertical view

Staggered grids with xGCM package

- xGCM handles staggered variables
- Extends xarray’s data model with **`Axes`** and **`Grid`** objects
- Variables may live on different positions along **`xgcm.Axes`**
- **`Axes`** stored in **`Grid`** object



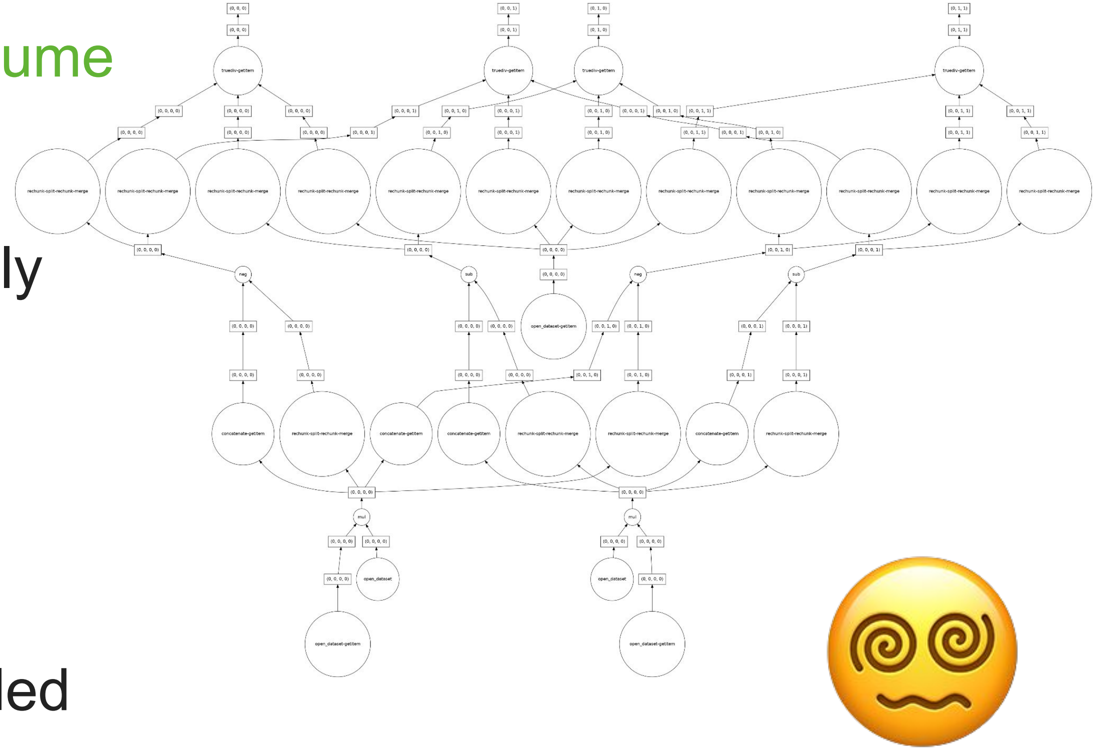
center		f[0]		f[1]		…		f[n-1]	
left	f[0]		f[1]		…		f[n-1]		
right			f[0]		f[1]		…		f[n-1]
inner			f[0]		…		f[n-2]		
outer	f[0]		f[1]		…		f[n-1]		f[n]

```
In [5]: from xgcm import Grid

In [6]: grid = Grid(ds, coords={"X": {"center": "x_c", "left": "x_g"}})

In [7]: grid
Out[7]:
<xgcm.Grid>
X Axis (periodic, boundary=None):
  * center    x_c --> left
  * left      x_g --> center
```

github.com/xgcm/xgcm

The old xGCM - Scaling bottleneck #1

- xGCM provides finite-volume functions, e.g. diff, interp
- Requires padding to apply boundary conditions
- Previously used custom reduction code
- Chaining diff, interp etc. led to explosion of dask tasks



Better idea: “grid ufuncs”

- Wrap numpy ufuncs to be grid-aware

- Positions specified through “signature”
  - `“(X:left)->(X:center)”`

- Signature is property of computational function
  - i.e. language-agnostic idea

```python
from xgcm import as_grid_ufunc


@as_grid_ufunc(signature="(ax1:center)->(ax1:left)")
def diff_center_to_left(a):
    return a - np.roll(a, -1, axis=-1)
```

`interp_left_to_center(a)`	`"(X:left)->(X:center)"`	Forward interpolation
`diff_center_to_center(a)`	`"(X:center)->(X:center)"`	Second order central difference
`mean_depth(w)`	`"(depth:center)->()"`	Reduction

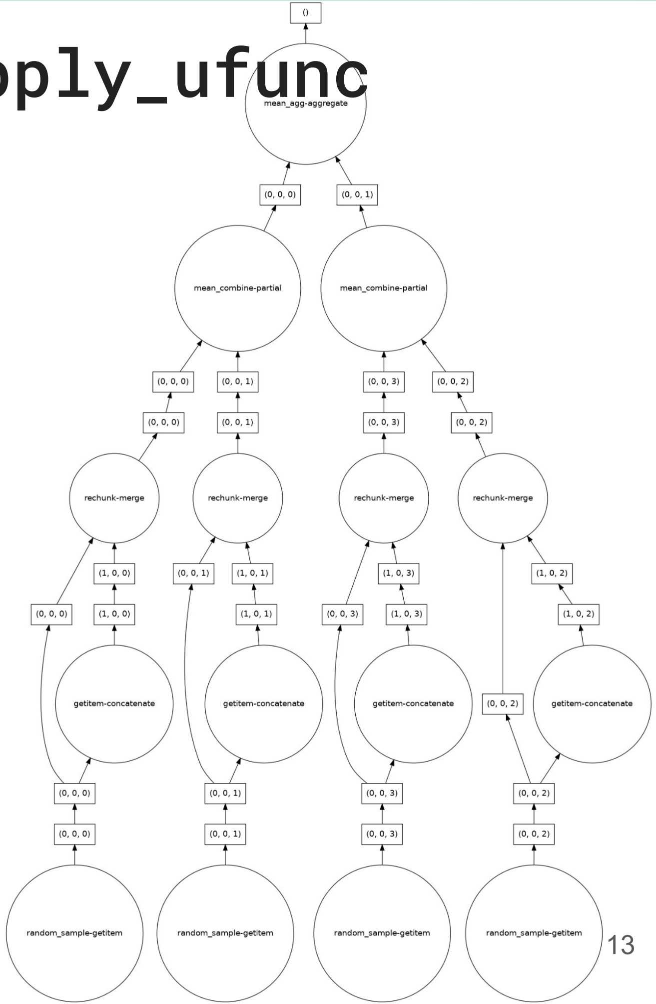
`@as_grid_ufunc` decorator

- Allows custom ufuncs
  - User-specific algorithms (e.g. from climate model)
  - Can auto-dispatch to correct ufunc for data
- Can specify grid positions via annotated type hints
- Could chain with other decorators, e.g. `numba.jit`

```python
from typing import Annotated
from xgcm import as_grid_ufunc


@as_grid_ufunc(
    boundary_width={"X": (0, 1), "Y": (0, 1)},
)
def divergence(
    u: Annotated[np.ndarray, "(X:left,Y:center)"],
    v: Annotated[np.ndarray, "(X:center,Y:left)"],
) -> Annotated[np.ndarray, "(X:center,Y:center)"]:

    du_dx = u[..., 1:, :] - u[..., :-1, :]
    dv_dy = v[..., 1:]    - v[..., :-1]

    return du_dx + dv_dy
```

Dask-optimised xGCM via `xarray.apply_ufunc`

- Apply all grid ufuncs through `xarray.apply_ufunc`
    - Common code path for all functions
- Only pad once
    - Avoids task explosion
- Creates minimal dask graph
- Ex. reduction: Almost blockwise (+ a rechunk-merge operation after padding)



xGCM: The great refactoring

- Change internals of entire package

+5,932 −340

- Without disrupting userbase (working scientists!)

- Wide test coverage was crucial

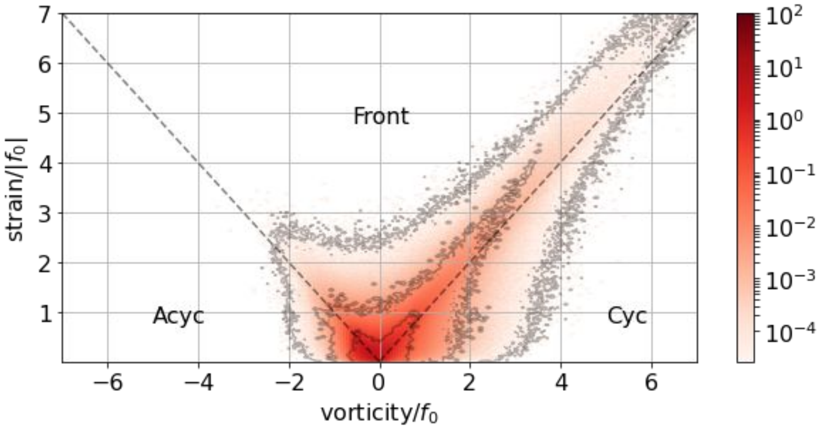
```
5358 passed, 2 skipped, 449 xfailed, 58 xpassed, 27708 warnings in 549.44s
```

- First created new code path
    - Then re-routed old functions through one-by-one
    - Avoided changing any tests until after code changes

- (Big thanks to Julius Busecke here)

Scaling bottleneck #2: Multidimensional histograms

- Use grid ufuncs for vorticity & strain
  - But what about the joint PDF?
- Need to compute histograms BUT:
  - Leave some dims unflattened
  - N-dimensional (N input arrays)
  - Ideally work with xarray objects
  - Scalable with dask
- Enter xhistogram!
  - But didn't scale well enough…



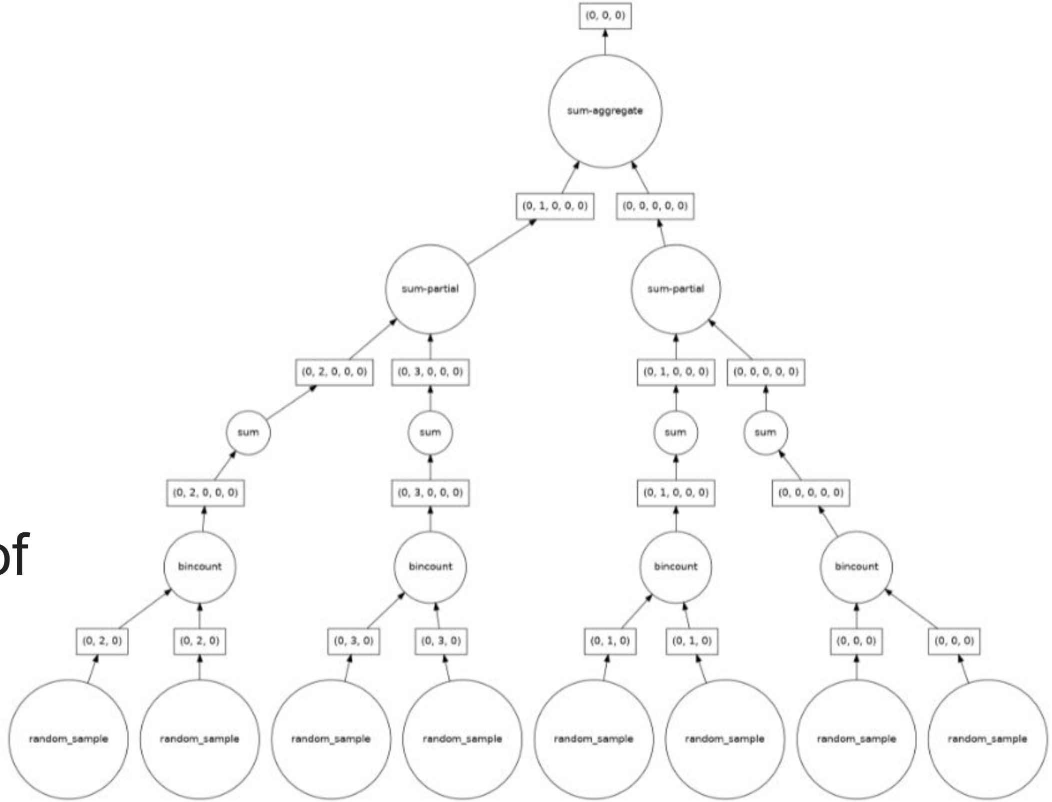
github.com/xgcm/xhistogram

Dask-optimised xhistogram with `dask.array.blockwise`

- Exploit cumulative property of histograms

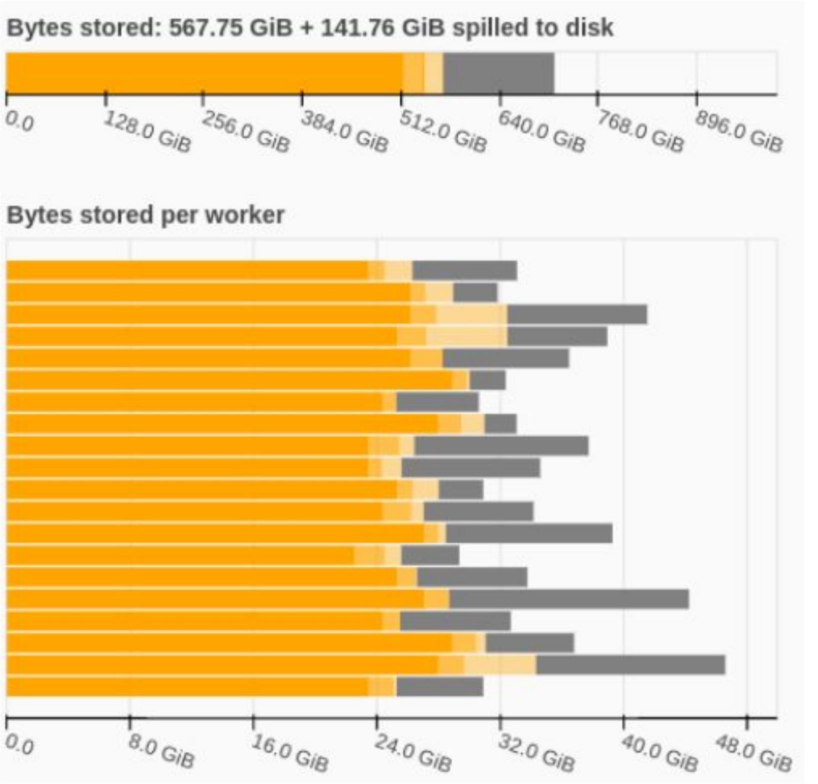
- Refactored as blockwise reduction, bincounting at each layer

- Thanks to Gabe Joseph of Coiled for the suggestion and Ryan Abernathey



So did it work?! Not exactly…

- Despite a theoretically reasonable graph…
- Would run out of memory, spilling to disk
- Even for *(some)* embarrassingly parallel graphs! 😕
- Perhaps familiar to other dask users…

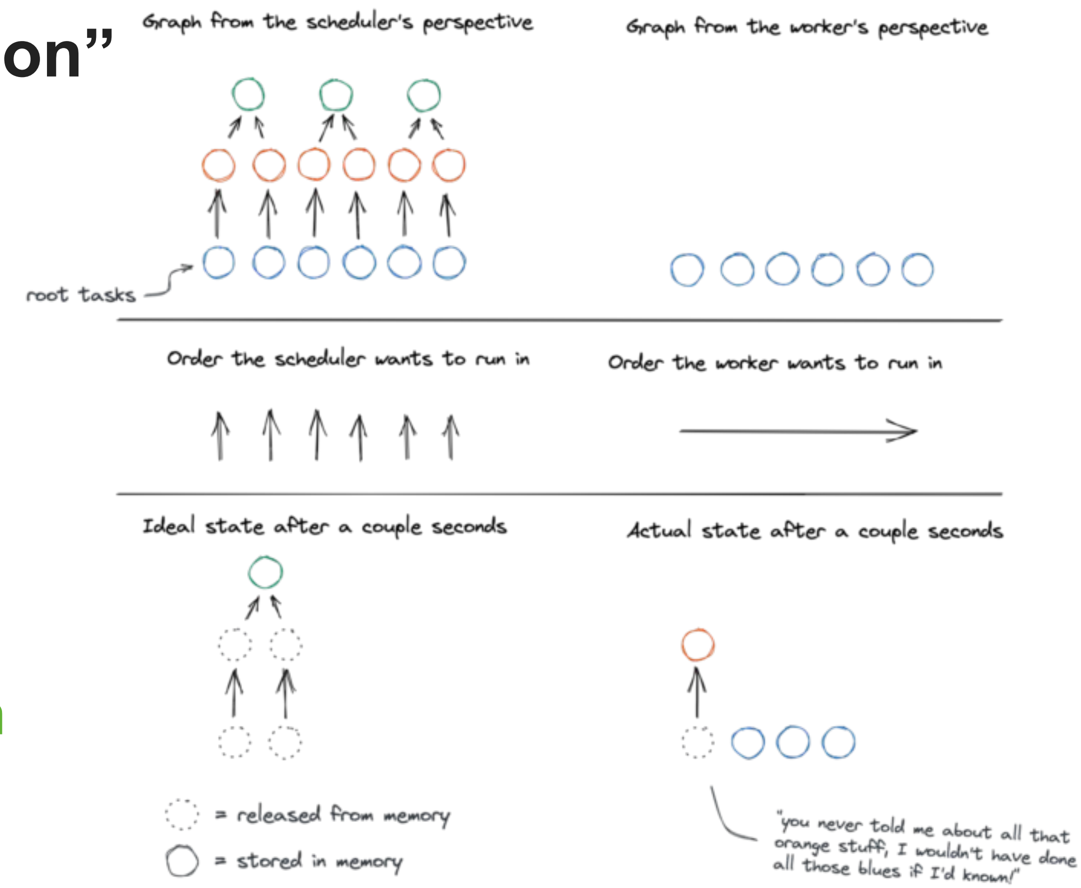


Dask scheduler issues

- So we distilled xGCM vorticity calc into minimal fail case…
- Found multiple issues with `dask.distributed` scheduler algorithm:
  - Memory problems caused by “root task overproduction”
  - Another issue with “widely-shared dependencies”
- Both likely problems in typical scientific workloads! 😬

"Root task overproduction"

- Race between data-opening and data-releasing tasks
- Embarrassingly-parallel graphs *should* work in streaming-like fashion…
- But actually race to open all data, overloading memory!



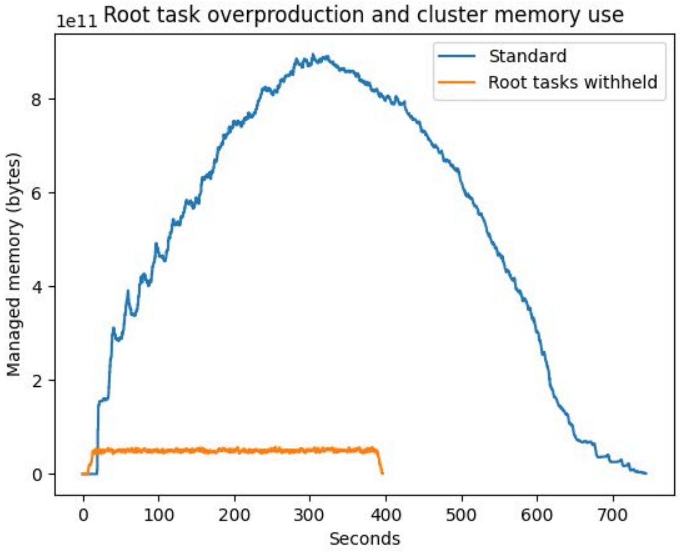
coiled.io/blog/better-shuffling-in-dask-a-proof-of-concept/

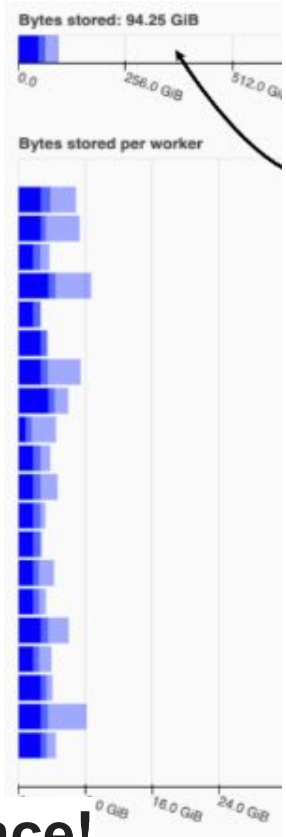
Scheduler improvements - Coiled collaboration

- Distilled xGCM vorticity calc into minimal fail case…

- Coiled team working on the issues!
    - Gabe Joseph prototyped changes to scheduler

- Amazing performance improvements on test cases





- **Exciting: Likely affects many workloads in geoscience!**

Alternative approaches

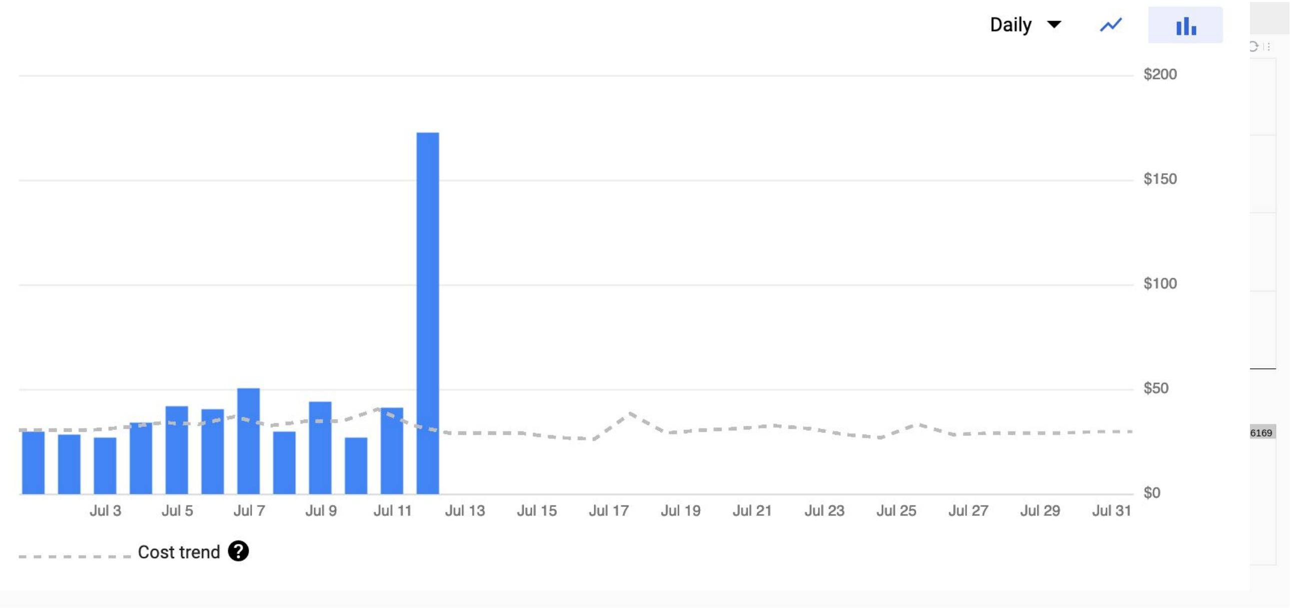
- However these improvements are works in progress

- In the meantime we looked at other approaches to scaling:
  - Dask:
    - `xarray.map_blocks` - similar scheduling issues
    - `dask.delayed` - bespoke approach…

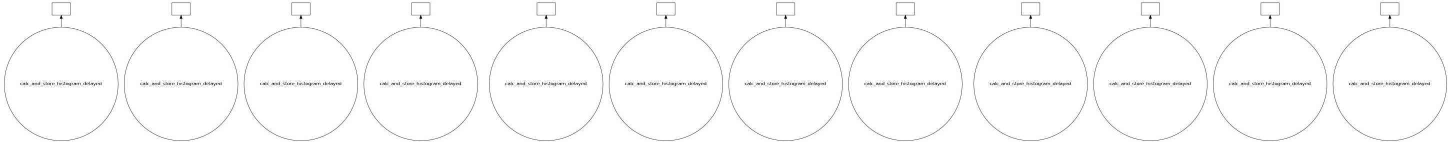
  - Other parallel execution frameworks:
    - xarray-Beam? (github.com/google/xarray-beam)
    - Cubed?? (https://github.com/tomwhite/cubed)

`dask.delayed` approach

- Rewrote computation to be less flexible but embarrassingly parallel



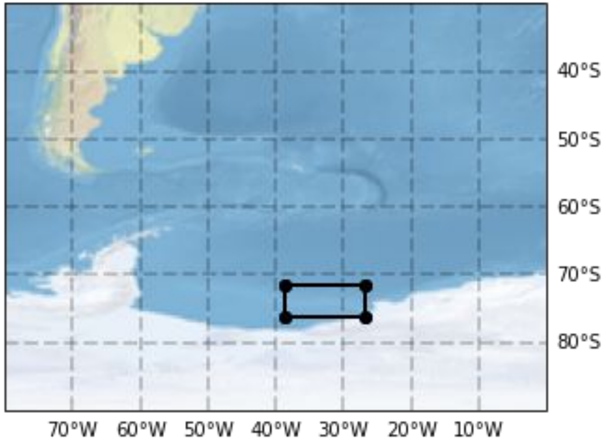
- Ran on 400 dask workers on GCP in ~2.5 hours (yesterday 😅)
- Cost ~<$130

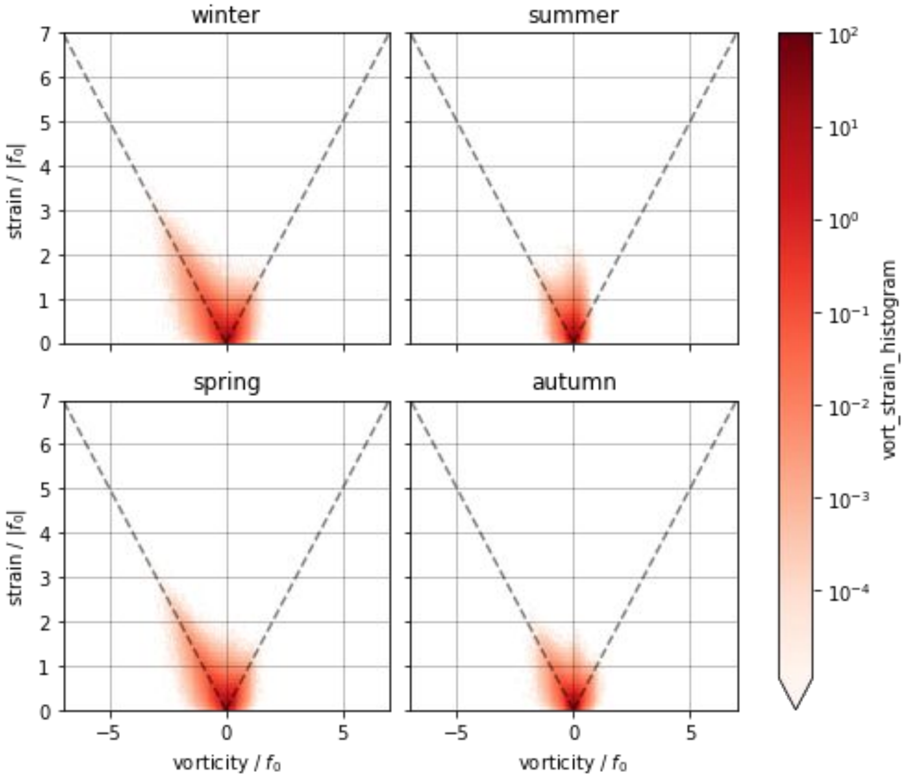


Science results

- Seasonal variation anywhere in the world's oceans

- e.g. in the Antarctic Circumpolar Current (ACC)





- More submesoscale fronts (strong vertical exchange) in winter in ACC

Takeaways

- Specific science problem at scale
- xGCM and xhistogram to now rewritten to scale better
- Plus generalised xGCM with “grid ufuncs”
- Exposed dask problems, scheduler now being improved

P.S. I am looking for my next big project 😁

github.com/xgcm/xhistogram

github.com/xgcm/xgcm

Bonus: A note on task fusion

- We tried aggressively fusing our tasks

- Doesn't help, unless you either:
    - Fuse so much that *data creation and data release are in same task*
        - (Reason why **`dask.delayed`** approach worked)
    - Fuse so much that graph becomes truly blockwise