Veros

High-performance earth system modelling in pure Python

Dion Häfner[*]**,** Markus Jochum[*], Steffen Randrup[*], Roman Nuterman[*], Andreas Schmittner[†], Mads Kristensen[*], and Brian Vinter[*]

[*] Niels Bohr Institute, Denmark
[†] Oregon State University, USA

EGU General Assembly
11-04-2019

Photography encouraged

What is Veros?

The versatile ocean simulator.

- Translation of PyOM2 (Fortan GCM) & MOBI (biogeochemistry) to Python
- Full 3D primitive equations
- Finite difference discretization on Arakawa C grid
- Runs on your laptop, gaming PC (GPU), or cluster
- Idealized and realistic setups
- Accessible, verifiable, adaptable

Why Python?

- Your time is more valuable than your computer’s
  - Code is part of the user experience
  - Better abstraction, thus higher signal-to-noise ratio
- Interfaceability with scientific Python ecosystem:
  - Internal: Linear algebra, I/O
  - External: Machine learning, orchestration, hybrid models, data pipelines
- People seem to enjoy it

Fortran to Python

xkcd.com/208/


```
do i=js_pe-1,je_pe
```

Search:

```
do (\w)=((\w|[\+\-])+,(\w|[\+\-])+)/
```

Replace:

```
for \1 in range(\2):
```

```
for i in range(is_pe-1, ie_pe):
```

Python to NumPy

Straightforward

```
do j=js_pe,je_pe
   do i=is_pe-1,ie_pe
      flux_east(i,j,:) = &
         0.25*(u(i,j,:,tau)+u(i+1,j,:,tau)) &
              *(utr(i+1,j,:)+utr(i,j,:))
   enddo
enddo
```

↓

```
vs.flux_east[1:-2, 2:-2,:] = \
   0.25 * (vs.u[1:-2, 2:-2, :, vs.tau] + vs.u[2:-1, 2:-2, :, vs.tau]) \
        * (vs.utr[1:-2, 2:-2, :] + vs.utr[1:-2, 2:-2, :])
```

(~90% of code)

Python to NumPy

Intermediate

```
yt(1)=yu(1)-dyt(1)*0.5
do i=2,n
   yt(i) = 2*yu(i-1) - yt(i-1)
enddo
```

↓

```
yt[0] = yu[0] - dyt[0] * 0.5
yt[1:] = 2 * yu[:-1]

alternating_pattern = np.ones_like(yt)
alternating_pattern[::2] = -1
yt[...] = alternating_pattern * np.cumsum(alternating_pattern * yt)
```

(~10% of code)

Python to NumPy

Hard

```
do j=js_pe,je_pe
 do i=is_pe,ie_pe
   k=kbot(i,j)
   if (k>0.and.k<nz) then
     eke_lee_flux(i,j)=c_lee(i,j)*eke(i,j,k,taup1)*dzw(k)
   endif
 enddo
enddo
```

```
ks = vs.kbot[2:-2, 2:-2] - 1
ki = np.arange(vs.nz)[np.newaxis, np.newaxis, :]
boundary_mask = (ks >= 0) & (ks < vs.nz - 1)
full_mask = boundary_mask[:, :, np.newaxis] & (ki == ks[:, :, np.newaxis])

vs.eke_lee_flux[2:-2, 2:-2] = np.where(
    boundary_mask,
    np.sum(vs.c_lee[2:-2, 2:-2, np.newaxis] * vs.eke[2:-2, 2:-2, :, vs.taup1]
           * vs.dzw[np.newaxis, np.newaxis, :] * full_mask, axis=-1),
    vs.eke_lee_flux[2:-2, 2:-2]
)
```

(~0.1% of code)

Demo

A simple 4x4 degree setup

In [3]:

```
from veros.setup.global_flexible import GlobalFlexibleResolutionSetup

class EGUSetup(GlobalFlexibleResolutionSetup):
    min_depth = 50

    def set_parameter(self, vs):
        super().set_parameter(vs)
        vs.nx = 90
        vs.ny = 40
        vs.nz = 15

        vs.dt_tracer = 86400
        vs.dt_mom = 1800

        vs.diskless_mode = True

    def set_diagnostics(self, vs):
        pass
```

In [4]:

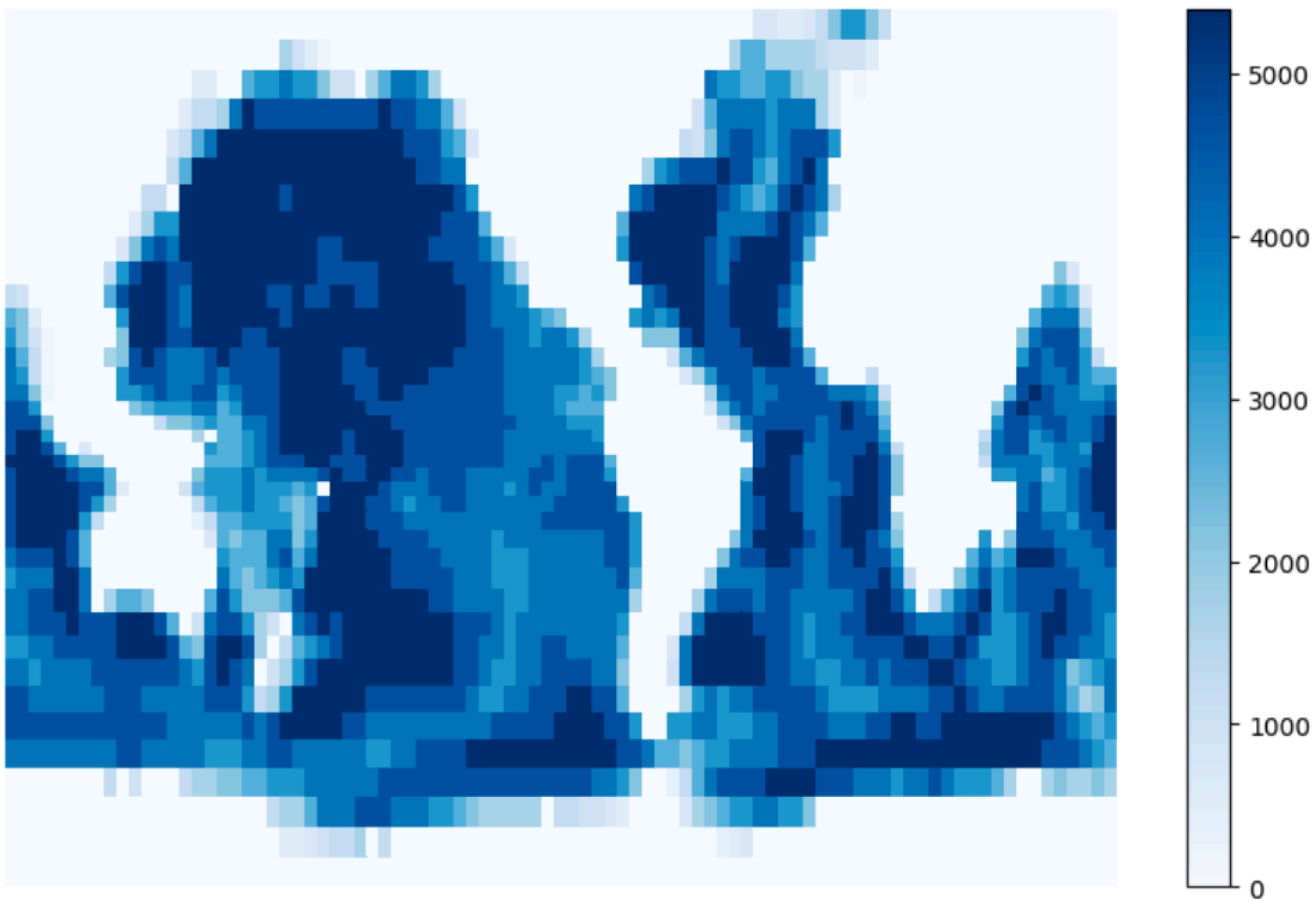
```python
sim = EGUSetup()
sim.setup()
```

```
Setting up everything
Initializing streamfunction method
 determining number of land masses

                                        Land mass and perimeter

    0    5   10   15   20   25   30   35   40   45   50   55   60   65   70   75   80   85   90
 43 1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
 42 1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
 41 11111111111111111111111111111111111111111111111111111111111111111*******1111111111111111111111
 40 11111111111111111111111111111111111111111111111111111111111111****00000**111111111111111111111
 39 111111111111111111111111****11111111111111111111111111111111***00000****1111111111111111111111
 38 11111111111111111**11****00***1******111111111111111111111***0000000*1*11111111111111111111111
 37 1111111111111111******0000000***0000***111111111111111111**000000000*1111111111111111111111111
 36 111111111111111**000000000000000000000**1111111111111111**000000000**1111111111111111111111111
 35 111111111111111**000000000000000000000*11111111111111****0000000000*11111111111111111111111111
 34 1111111111111**1*000000000000000000000**111111111111**0000000000000**1111111111111111111111111
 33 111111111111*****0000000000000000000000**1111111111**0000000000000**11111111111111111111111111
 32 11111111111**000000000000000000000000000***11111111**000000000000**111111111111111111111111111
 31 1111111111**000000000000000000000000000000**11111111**00000000000*11111111111111111111**111111
 30 11**111111**0000000000000000000000000000000*111111111*00000000000*1111111111111111111****111**
 29 1****111111*0000000000000000000000000000000****11111**00000000000*111111111111111111**00*11***
 28 1*00**11111*0000000000000000000000000000000000***1111***000000000*11111111111111111**000*11*00
 27 1*000*1111**000000000000000000000000000000000000**11111**00000000**1111111111111111*0000**1*00
 26 **000*11111*0000000000000000000000000000000000000**11111***0000000**111*11111111111*00000***00
 25 0000**11111**000000000000000000000000000000000000**1111111*00000000******111111111**0000000000
 24 0000*1111111***0000000000000000000000000000000000*11111111**000000000000*11111111**00000000000
 23 0000**11111111******00000000000000000000000000000*111111111***0000000000*11111111*000000000000
 22 00000*11111111111*1*00000000000000000000000000000*11111111111**000000000*1111111**000000000000
 21 00000**1*111111111**00000000000000000000000000000*111111111111*000000000**111111*0000000000000
 20 000000****11111111*000000000000000000000000000000**11111111111*0000000000*111111**000000000000
 19 000000000**111111**0000000***000000000000000000000*1111111111**0000000000*1111111***0000000000
 18 0000000000**1111**00000000*2*000000000000000000000*1111111111*0000000000**111111111*0000000000
 17 000000000**111111*00000000***000000000000000000000**111111111*0000000000*1111111111*0000000000
 16 00000000**1111111**00000000000000000000000000000000**11111111*0000000000**111111111*0000000000
 15 00000000*111111111**00000000000000000000000000000000*1111111**00000000000*111111*1**0000000000
 14 00000000*1111111111*00000000000000000000000000000000*111111**000000000000*11111****00000000000
 13 00000000*1111111111*00000000000000000000000000000000*11111**0000000000000**111**00000000000000
 12 00000000*1****1111**000***0000000000000000000000000**1111**000000000000000*1***000000000000000
 11 00000000***00***1**000**3*0000000000000000000000000*1111**0000000000000000***00000000000000000
 10 000000000000000***000**3**0000000000000000000000000*111**0000000000000000000000000000000000000
  9 000000000000000000000*3**00000000000000000000000000*111**0000000000000000000000000000000000000
  8 000000000000000000000***000000000000000000000000000*1111*0000000000000000000000000000000000000
  7 000000000000000000000000000000000000000000000000000**11**0000000000000000000000000000000000000
  6 0****************00000000000000000000000000000000000****00000000000000000000000000****00000***
  5 **44444444*4*444********00000000000000000000***0000**44**000000000*****************44*******44
```

In [5]:
```
vs = sim.state
```

In [6]:
```
coords = np.meshgrid(vs.xt[2:-2], vs.yt[2:-2], indexing='ij')
```

In [7]:
```
plt.pcolormesh(*coords, vs.ht[2:-2, 2:-2], cmap='Blues')
plt.axis('off')
plt.colorbar();
```



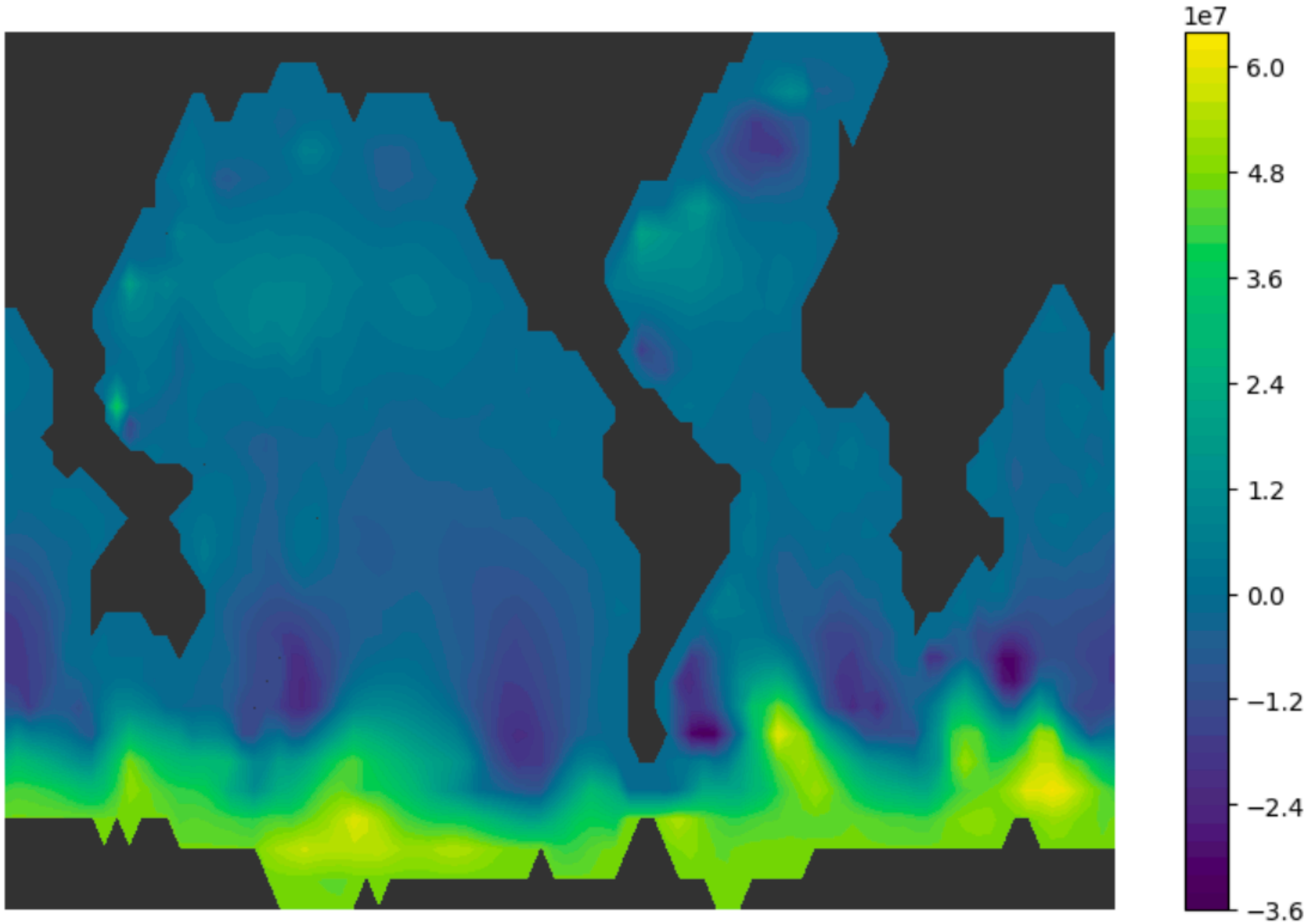
In [8]:
```
vs.runlen = 86400 * 20
```

In [9]:
```
sim.run()
```

```
Starting integration for 20.0 days
Current iteration: 1
Current iteration: 2
Current iteration: 3
Current iteration: 4
Current iteration: 5
Current iteration: 6
Current iteration: 7
Current iteration: 8
Current iteration: 9
Current iteration: 10
Current iteration: 11
Current iteration: 12
Current iteration: 13
Current iteration: 14
Current iteration: 15
Current iteration: 16
Current iteration: 17
Current iteration: 18
Current iteration: 19
Current iteration: 20
```

In [10]:
```
cs = plt.contourf(
    *coords,
    vs.psi[2:-2, 2:-2, vs.tau],
    50
)
plt.contourf(*coords, vs.maskT[2:-2, 2:-2, -1], [-1e-3, 1e-3],
             colors='0.2')
plt.axis('off')
plt.colorbar(cs)
```

Out[10]: <matplotlib.colorbar.Colorbar at 0x132700d30>

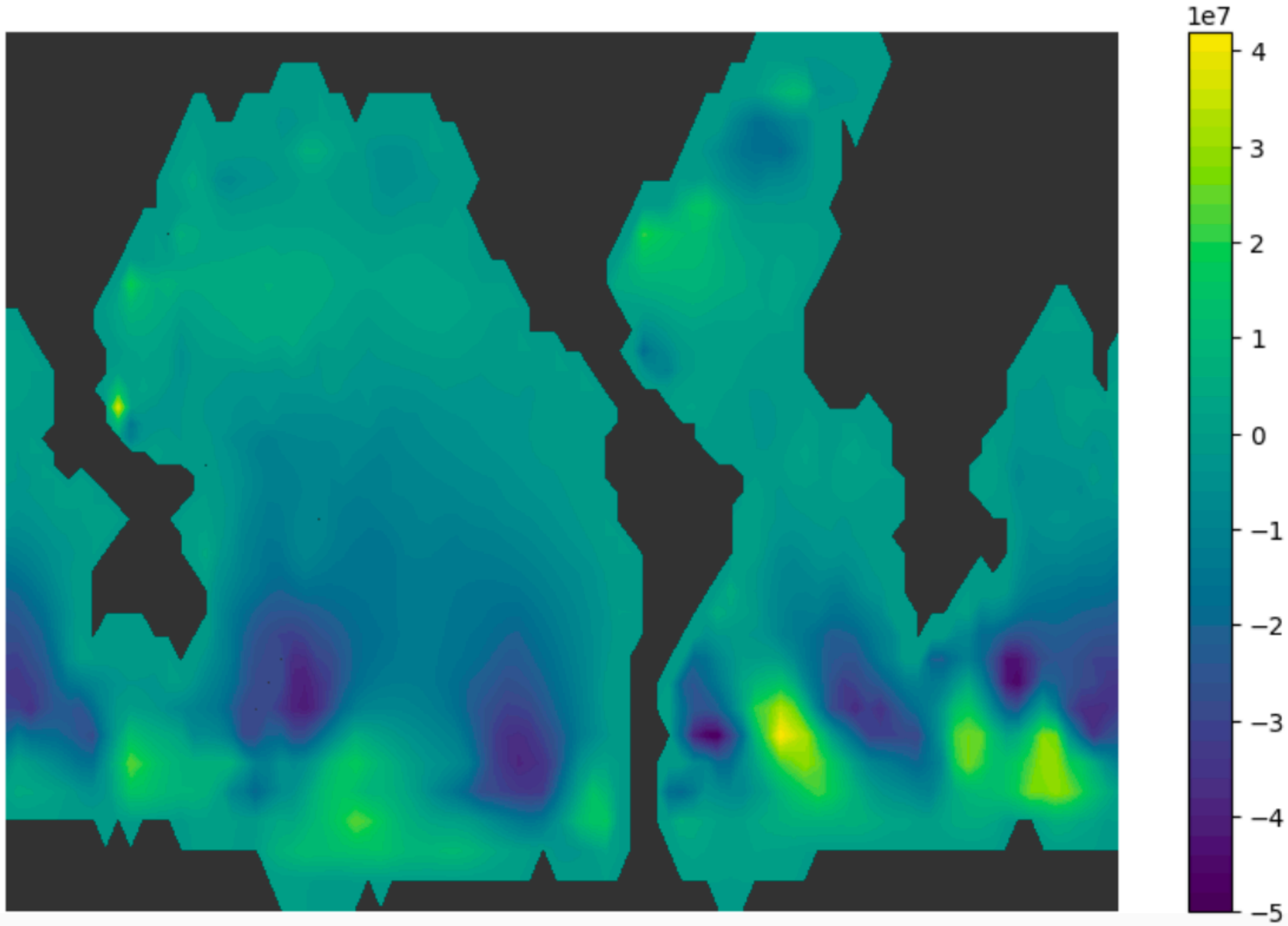


Let's modify that!

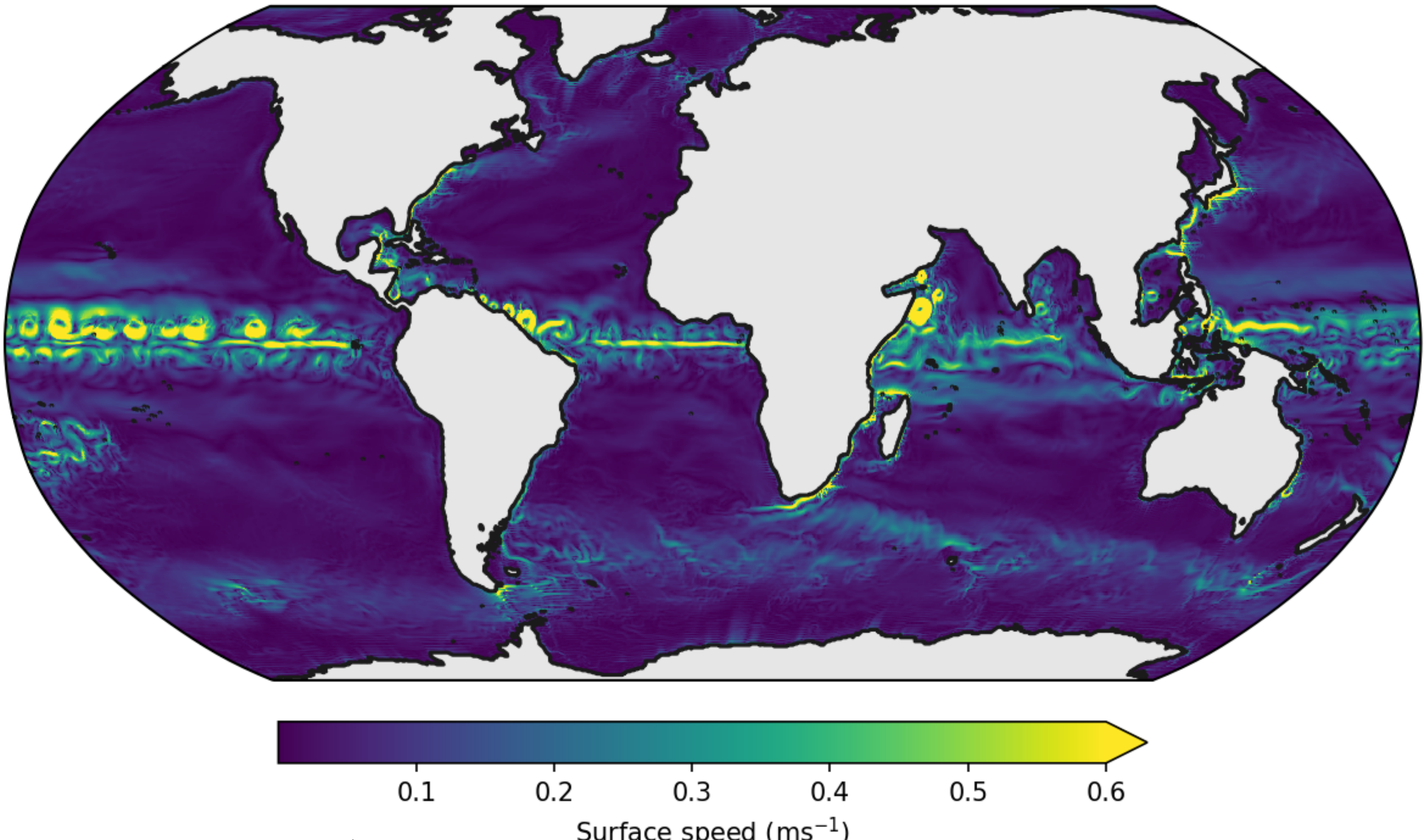
In [11]:
```
class ModifiedEGUSetup(GlobalFlexibleResolutionSetup):
    min_depth = 50

    def set_parameter(self, vs):
        super().set_parameter(vs)
        vs.nx = 90
        vs.ny = 40
        vs.nz = 15

        vs.dt_tracer = 86400
        vs.dt_mom = 1800
        vs.runlen = 86400 * 20

        vs.diskless_mode = True

    def set_topography(self, vs):
        super().set_topography(vs)
        vs.kbot[52:55, 3:8] = 0

    def set_diagnostics(self, vs):
        pass
```

close Drake passage

In [15]:
```
vs = sim2.state

cs = plt.contourf(
    *coords,
    vs.psi[2:-2, 2:-2, vs.tau],
    50
)
plt.contourf(
    *coords,
    vs.maskT[2:-2, 2:-2, -1],
    [-1e-3, 1e-3],
    colors='0.2'
)
plt.axis('off')
plt.colorbar(cs);
```



High-resolution setups



(0.25° x 0.25° after ~1 year)

Bohrium

Provides a JIT compiler for NumPy code

```
import bohrium as np

a = np.ones((100, 100))
a.sum()
```

Bohrium

Provides a JIT compiler for NumPy code

```
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
#include <complex.h>
#include <tgmath.h>
#include <math.h>

void execute(double* __restrict__ a0, uint64_t vo0, uint64_t vs0_0, uint64_t vs0_1,
             uint64_t vo1, uint64_t vs1_0, const double c1){

    double t2;
    t2 = 0;
    #pragma omp parallel for reduction(+:t2)
    for(uint64_t i0 = 0; i0 < 100; ++i0) {
        double t1;
        t1 = 0;
        #pragma omp simd reduction(+:t1)
        for(uint64_t i1 = 0; i1 < 100; ++i1) {
            const uint64_t idx0= (vo0 +i0*vs0_0 +i1*vs0_1);
            a0[idx0] = c1;
            t1 += a0[idx0];
        }
        t2 += t1;
    }

}

void launcher(void* data_list[], uint64_t offset_strides[], union dtype constants[]) {
    double *a0 = data_list[0];
    execute(a0, offset_strides[0], offset_strides[1], offset_strides[2],
            offset_strides[3], offset_strides[4], constants[0].BH_FLOAT64);
}
```

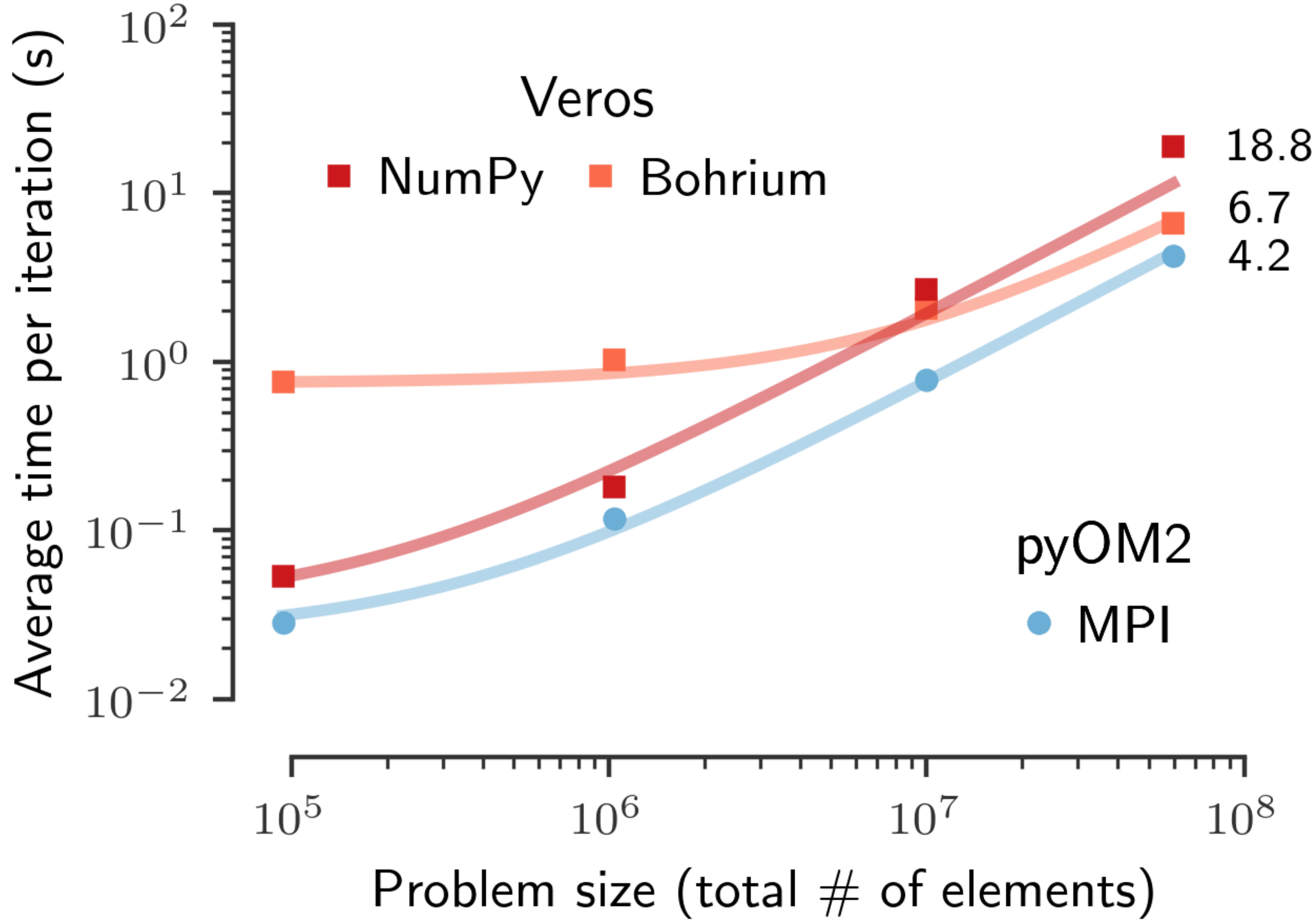
Bohrium

Provides a JIT compiler for NumPy code

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#include <kernel_dependencies/complex_opencl.h>
#include <kernel_dependencies/integer_operations.h>

__kernel void execute(__global double* __restrict__ a0, __global double* __restrict__ a1,
                      ulong vo0, ulong vs0_0, ulong vs0_1, ulong vo1, ulong vs1_0,
                      const double c1) {
    // The IDs of the threaded blocks:
    const uint g0 = get_global_id(0); if (g0 >= 100) { return; } // Prevent overflow

    {const ulong i0 = g0;
        double s1;
        s1 = 0;
        for (ulong i1 = 0; i1 < 100; ++i1) {
            const ulong idx0= (vo0 +i0*vs0_0 +i1*vs0_1);
            a0[idx0] = c1;
            s1 += a0[idx0];
        }
        a1[vo1 +i0*vs1_0] = s1;
    }
}
```

Benchmarks

4 nodes (64 CPU cores)



(lower is better)

Our vision I

Modern HPC

- Scale horizontally and vertically
- Native GPU and FPGA support
- Exploit new HPC research as it happens

Our vision II

Leverage the Python ecosystem

- Let’s unify pre-processing, modelling, post-processing:

API Draft

```
>>> data = sim.state.to_xarray()
<xarray.Dataset>
Dimensions:             (Time: 2, xt: 90, xu: 90, yt: 40, yu: 40, zt: 15, zw: 15)
Coordinates:
  * Time                (Time) float64 15.0 30.0
  * xt                  (xt) float64 2.0 6.0 10.0 14.0 ... 350.0 354.0 358.0
  * xu                  (xu) float64 4.0 8.0 12.0 16.0 ... 352.0 356.0 360.0
  * yt                  (yt) float64 -78.0 -74.0 -70.0 -66.0 ... 70.0 74.0 78.0
  * yu                  (yu) float64 -76.0 -72.0 -68.0 -64.0 ... 72.0 76.0 80.0
  * zt                  (zt) float64 -4.855e+03 -4.165e+03 ... -65.0 -35.0
  * zw                  (zw) float64 -4.51e+03 -3.87e+03 -3.28e+03 ... -50.0 0.0
Data variables:
    E_iw                (Time, zw, yt, xt) float64 ...
    Hd                  (Time, zt, yt, xt) float64 ...
    Nsqr                (Time, zw, yt, xt) float64 ...
    area_t              (yt, xt) float64 ...
...

>>> data['psi'].sel(Time=30).plot()
```

Our vision II

Leverage the Python ecosystem

- Dynamic model execution:
    - Hybrid models
    - Ensembles / sensitivity studies via machine learning (e.g. Bayesian optimization)

Our vision III

Cut out the boring parts

```
vs.flux_east[1:-2, 2:-2,:] = \
   0.25 * (vs.u[1:-2, 2:-2, :, vs.tau] + vs.u[2:-1, 2:-2, :, vs.tau]) \
        * (vs.utr[1:-2, 2:-2, :] + vs.utr[1:-2, 2:-2, :])
```

↓ Abstraction

API Draft

```
vs.flux_east.update(
    on('t', vs.u[..., vs.tau]) * on('t', vs.utr)
)
```

Our vision:

A high-level, high-performance earth system model

Thank you for listening

Find these slides:

github.com/dionhaefner/veros-egu-2019

Learn more:

veros.readthedocs.org

Contribute:

github.com/dionhaefner/veros

Give love to:

github.com/bh107/bohrium

References

(1) Häfner, Dion, et al. "Veros v0. 1–a fast and versatile ocean simulator in pure Python." *Geoscientific Model Development* 11.8 (2018): 3299-3312.

(2) Muglia, Juan, et al. "Combined effects of atmospheric and seafloor iron fluxes to the glacial ocean." *Paleoceanography and Paleoclimatology* 32.11 (2017): 1204-1218.

(3) Eden, Carsten (2016). “Closing the energy cycle in an ocean model”. In: *Ocean Modelling 101*

(4) Kristensen, Mads RB, et al. (2013). “Bohrium: un-modified NumPy code on CPU, GPU, and cluster”. In: *Python for High Performance and Scientific Computing (PyHPC 2013)*.

(5) Larsen, Mads Ohm et al. (2016). “Current Status and Directions for the Bohrium Runtime System”. In: *Compilers for Parallel Computing 2016.*

Bonus slides

Bohrium

A simple example

In [1]:
```
import time
import numpy as np
```

In [2]:
```
a = np.random.rand(10000, 10000)

def bench():
    b = a ** 2
    c = a * b
    d = c * np.sum(a, axis=1)[:, None] - a ** 3 + 17.2
    e = np.sum(a + b + c + d)
    return e
```

In [3]:
```
while True:
    start = time.time()
    res = bench()
    end = time.time()
    print("result: {:.2e}; time: {:.2f}s".format(float(res), end-sta
```

```
result: 1.27e+11; time: 9.02s
result: 1.27e+11; time: 9.01s
result: 1.27e+11; time: 9.00s
result: 1.27e+11; time: 9.01s
result: 1.27e+11; time: 9.00s
result: 1.27e+11; time: 9.00s
```

Bohrium

A simple example

In [6]:
```
import bohrium as np
```

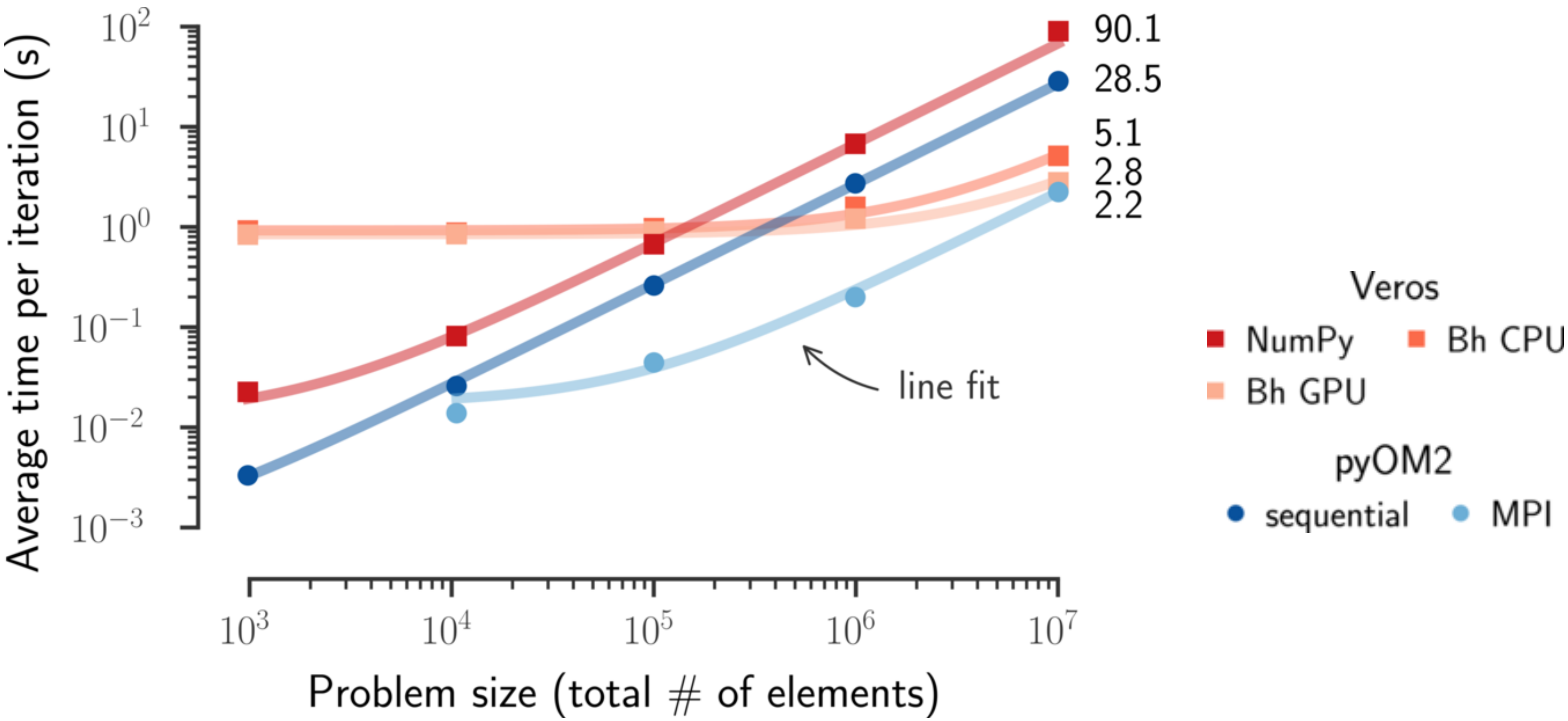
In [7]:
```
a = np.random.rand(10000, 10000)

def bench():
    b = a ** 2
    c = a * b
    d = c * np.sum(a, axis=1)[:, None] - a ** 3 + 17.2
    e = np.sum(a + b + c + d)
    return e
```

In [8]:
```
while True:
    start = time.time()
    res = bench()
    try:
        np.flush()
    except AttributeError:
        pass
    end = time.time()
    print("result: {:.2e}; time: {:.2f}s".format(float(res), end-sta
```

```
result: 1.27e+11; time: 1.05s
result: 1.27e+11; time: 0.55s
result: 1.27e+11; time: 0.54s
result: 1.27e+11; time: 0.55s
result: 1.27e+11; time: 0.54s
```

Benchmarks

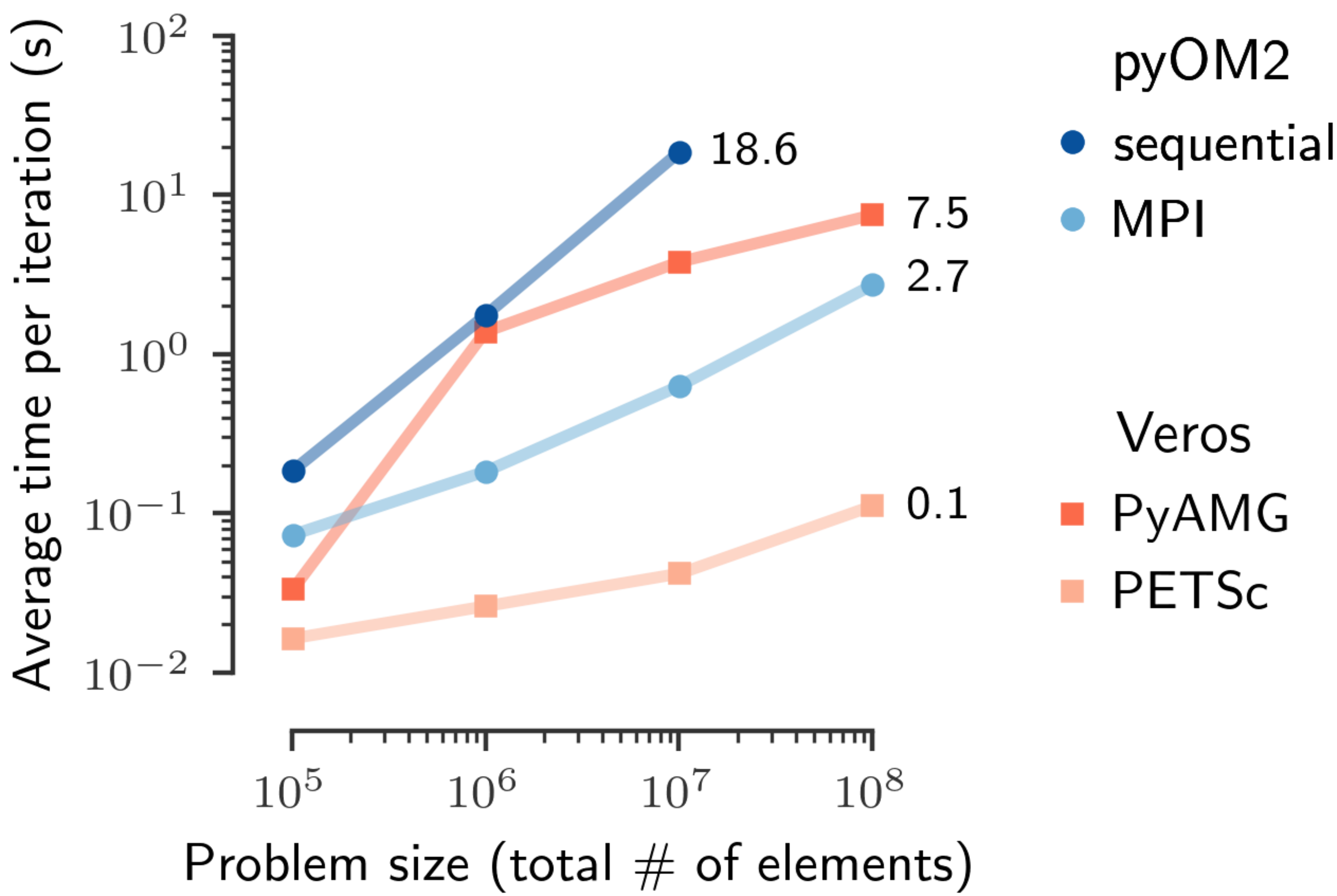
Cluster node with GPU (24 cores + NVIDIA Tesla P100)



(lower is better)

Benchmarks

Streamfunction solver



(lower is better)

Biogeochemistry