Introducing NEDAS:

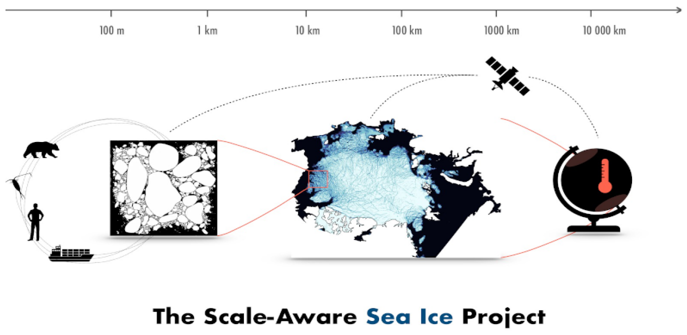
Next-generation Ensemble DA System

Yue (Michael) Ying

NERSC

Supported by: *SASIP, ACCIBERG*

NERSC



ACCIBERG

Ensemble Data Assimilation

Model state $\boldsymbol{\psi}$ updated by observation $\boldsymbol{\varphi}$ to find best estimate

$$p(\boldsymbol{\psi}|\boldsymbol{\varphi}) = \frac{p(\boldsymbol{\varphi}|\boldsymbol{\psi})p(\boldsymbol{\psi})}{p(\boldsymbol{\varphi})}$$

Use an ensemble of state $\boldsymbol{\Psi} = \left(\boldsymbol{\psi}_1, \dots, \boldsymbol{\psi}_{N_e}\right) \in \mathbb{R}^{N_{state} \times N_e}$
as samples of $p(\boldsymbol{\psi})$

and "observation priors" $\boldsymbol{\Phi} = \left(\boldsymbol{\varphi}_1, \dots, \boldsymbol{\varphi}_{N_e}\right) \in \mathbb{R}^{N_{obs} \times N_e}$
comparing with actual observation $\boldsymbol{\varphi}^o$ to give the likelihood $p(\boldsymbol{\varphi}|\boldsymbol{\psi})$

Goal: update $\boldsymbol{\Psi}$ so that it characterizes $p(\boldsymbol{\psi}|\boldsymbol{\varphi})$

Algorithm: $\boldsymbol{\Psi} \leftarrow \mathcal{A}(\boldsymbol{\Psi}, \boldsymbol{\Phi}, \boldsymbol{\varphi}^o)$

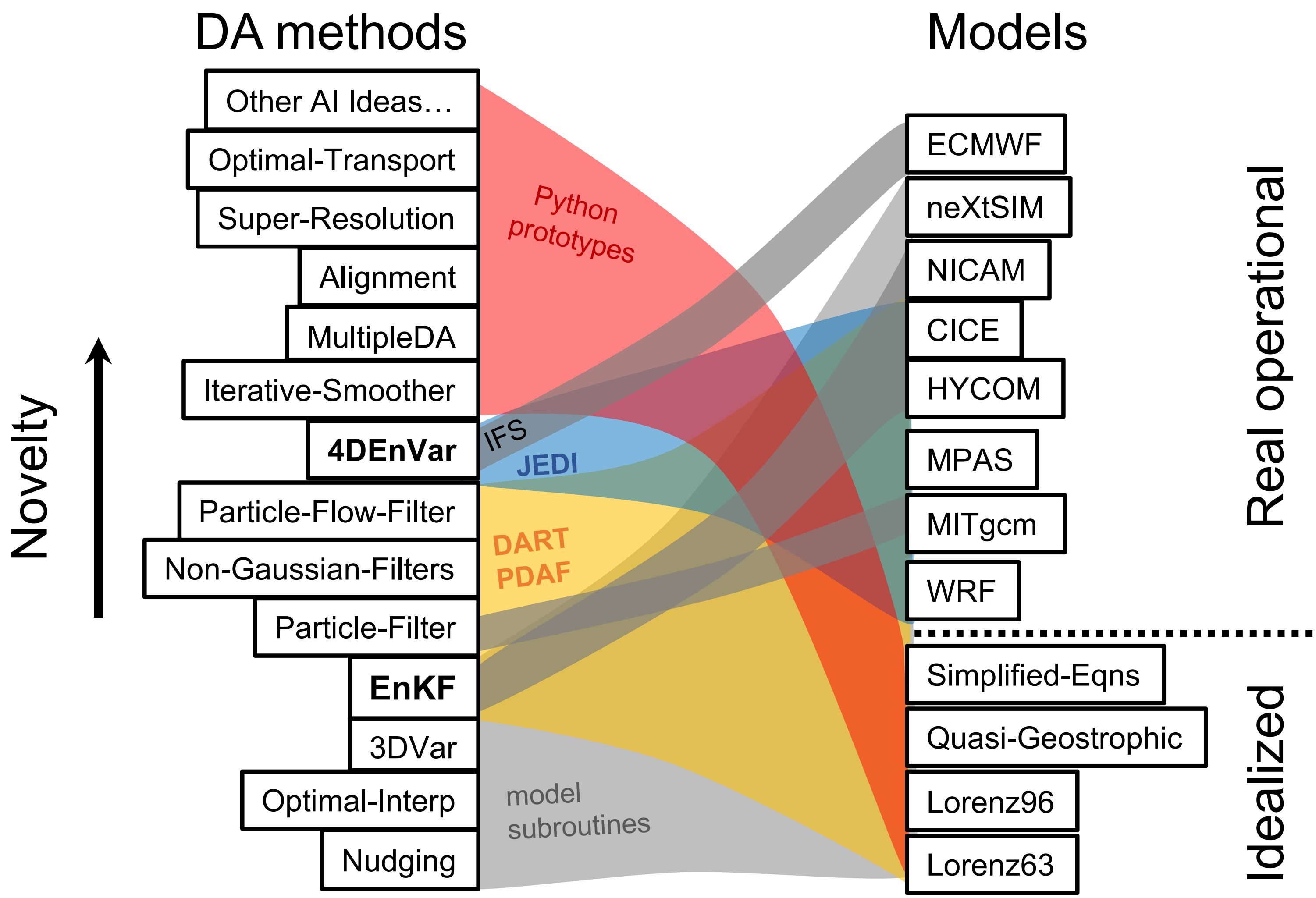
How much effort is needed for testing novel algorithms in real models?

Simple method: just implement in the model code (WRF nudging/fdda)

Complex method: dedicated DA software:

Data Assimilation Research Testbed (DART; Anderson et al. 2009)
Parallel Data Assimilation Framework (PDAF; Nerger & Hiller 2013)
Joint Effort in DA Integration (JEDI; JCSDA)

Conception → Python prototype → implement in *DART / PDAF / JEDI*
→ test in real model → operational use



New ideas for nonlinear filtering for large dimensional systems, but a lot of them stuck at Python prototype phase…

NEDAS enters the market

Conception → Python prototype → test in real models → implement in DART / PDAF / JEDI → operational use

Python code is light-weight and easier to maintain:

DAPPER to benchmark a collection of DA methods and use in teaching

- add `mpi4py`, `numpy`, `numba.jit` allow scalability and efficiency
- flexibility: use functions in workflow

Does the software run efficiently for DA problems?

NEDAS implementation

Sequential DA with pause-restart strategy

1: **for** $n = 1, \ldots, N_t$ **do**

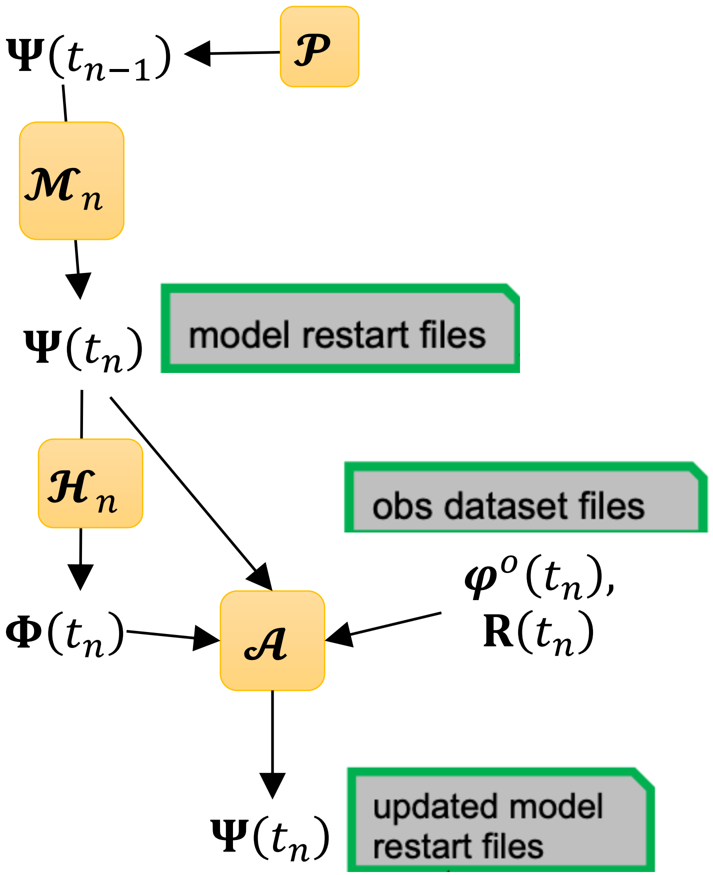
2: $\quad \mathbf{\Psi}(t_{n-1}) \leftarrow \mathcal{P}[\mathbf{\Psi}(t_{n-1})]$

3: $\quad \mathbf{\Psi}(t_n) = \mathcal{M}_n\,[\mathbf{\Psi}(t_{n-1})]$

4: $\quad \mathbf{\Phi}(t_n) = \mathcal{H}_n\,[\mathbf{\Psi}(t_n)]$

5: $\quad \mathbf{\Psi}(t_n) \leftarrow \mathcal{A}[\mathbf{\Psi}(t_n), \mathbf{\Phi}(t_n), \boldsymbol{\varphi}^o(t_n), \mathbf{R}(t_n)]$

6: **end for**

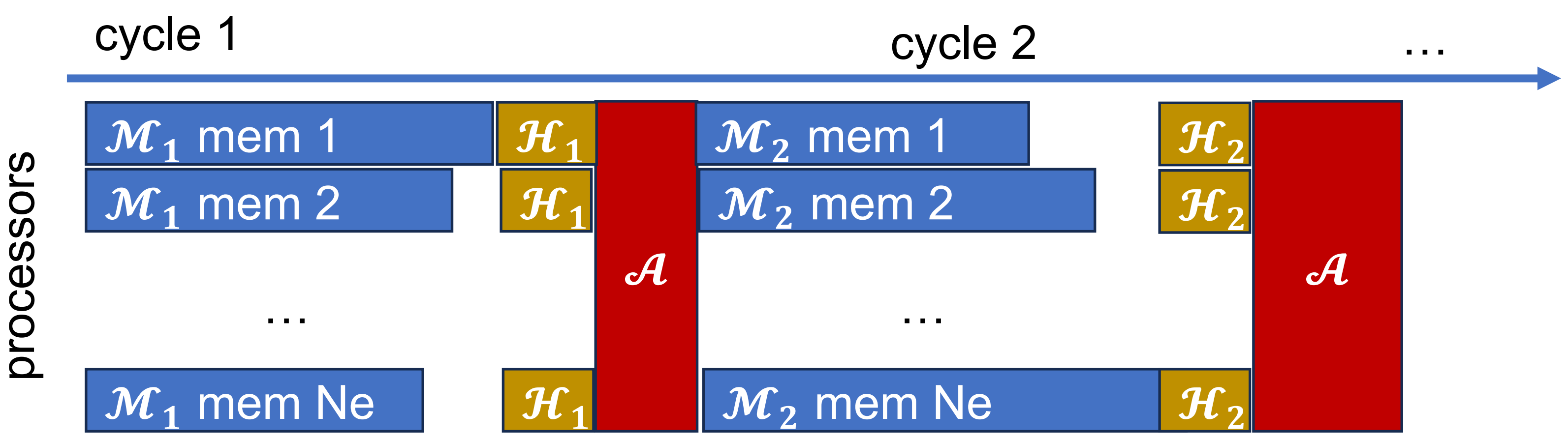


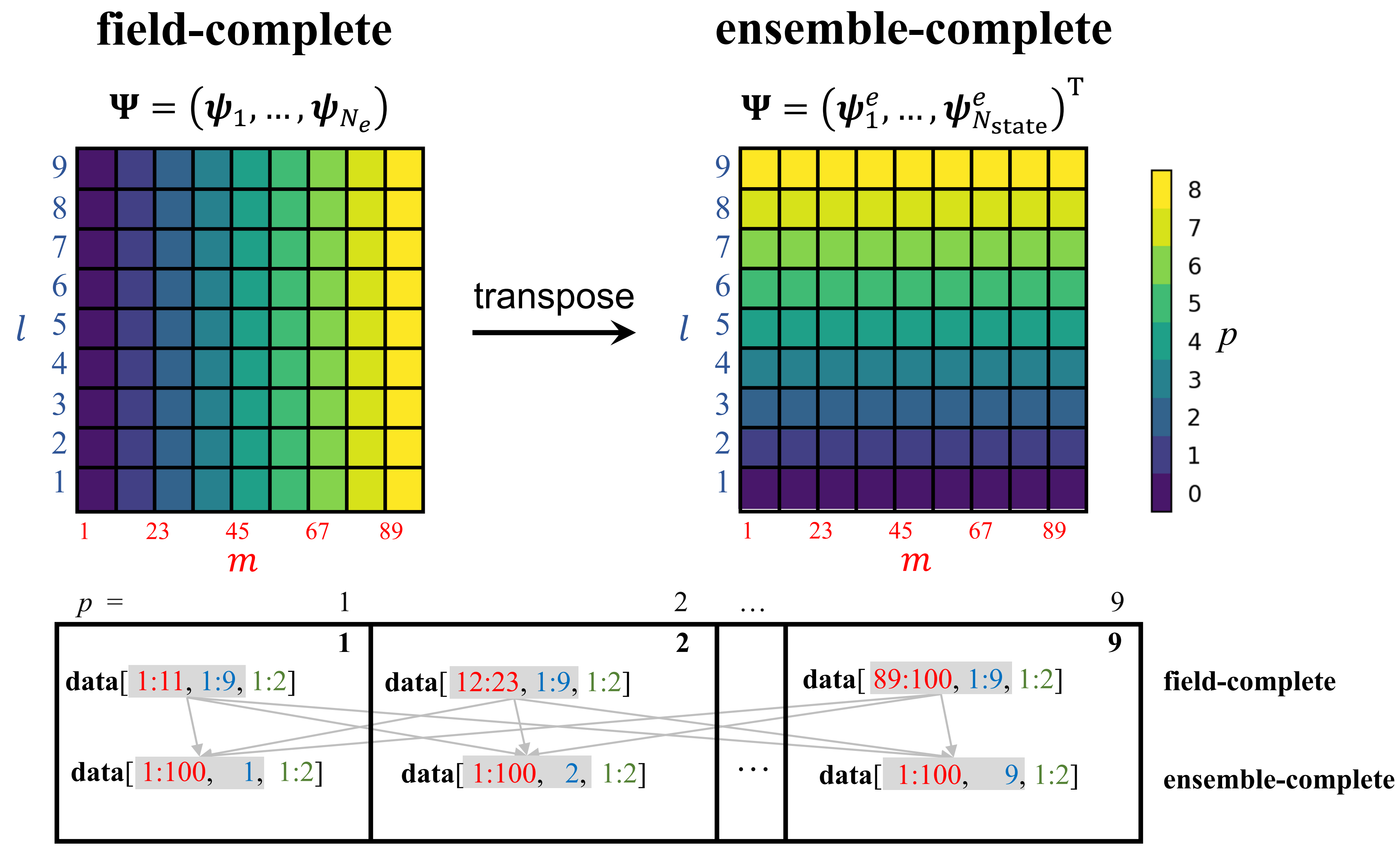
NEDAS implementation

Sequential DA with pause-restart strategy

Bottleneck for large-dimensional DA problems:
- slow file I/O -- keep states in memory
- large data volume -- local analysis

Scalability: domain decomposition and use MPI





field-complete
$\mathbf{\Psi} = (\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{N_e})$
ensemble-complete
$\mathbf{\Psi} = (\boldsymbol{\psi}^e_1, \ldots, \boldsymbol{\psi}^e_{N_{\text{state}}})^{\mathrm{T}}$
transpose
l
m
1 23 45 67 89
1 2 3 4 5 6 7 8 9
p
0 1 2 3 4 5 6 7 8
p = 1 2 … 9
1
data[1:11, 1:9, 1:2]
data[1:100, 1, 1:2]
2
data[12:23, 1:9, 1:2]
data[1:100, 2, 1:2]
9
data[89:100, 1:9, 1:2]
data[1:100, 9, 1:2]
field-complete
ensemble-complete

Parallelization strategy

Batch assimilation (e.g. PDAF)

for $i = 1, \ldots, N_{\text{lstate}}$:

$$\mathbf{S} = \mathbf{R}^{-1/2}\left(\boldsymbol{\Phi} - \overline{\boldsymbol{\varphi}}\mathbf{1}^{\mathrm{T}}\right) \circ (\boldsymbol{\rho}\mathbf{1}^{\mathrm{T}})/\sqrt{N_e - 1}$$

$$\boldsymbol{d} = \mathbf{R}^{-1/2}(\boldsymbol{\varphi}^o - \overline{\boldsymbol{\varphi}}) \circ \boldsymbol{\rho}/\sqrt{N_e - 1}$$

$$\boldsymbol{\Xi} = \left(\mathbf{I} + \mathbf{S}^{\mathrm{T}}\mathbf{S}\right)^{-1}$$

$$\mathbf{T} = \boldsymbol{\Xi}\mathbf{S}^{\mathrm{T}}\boldsymbol{d}\mathbf{1}^{\mathrm{T}} + \boldsymbol{\Xi}^{1/2}$$

$$\boldsymbol{\psi}_i^{eT} \leftarrow \boldsymbol{\psi}_i^{eT}\mathbf{T}$$

cost:

$\mathcal{O}(N_{\text{lobs}}N_e^2 + N_e^3)\ \times N_{\text{lstate}}$

"local analysis"

Serial assimilation (e.g. DART)

for $j = 1, \ldots, N_{\text{obs}}$:

$$\xi = \sigma_{o,j}^2/(\sigma_j^2 + \sigma_{o,j}^2)$$

$$\boldsymbol{\delta}_j^e = \xi\overline{\varphi}_j + (1-\xi)\varphi_j^o + \sqrt{\xi}\left(\boldsymbol{\varphi}_j^e - \overline{\varphi}_j\right) - \boldsymbol{\varphi}_j^e$$

broadcast $\boldsymbol{\delta}_j^e$

$$\boldsymbol{\Psi} \leftarrow \boldsymbol{\Psi} + \left(\boldsymbol{\rho}^{\psi} \circ \boldsymbol{c}_{\psi,\varphi_j}/\sigma_j^2\right)\boldsymbol{\delta}_j^{e\,\mathrm{T}}$$

$$\boldsymbol{\Phi} \leftarrow \boldsymbol{\Phi} + \left(\boldsymbol{\rho}^{\varphi} \circ \boldsymbol{c}_{\varphi,\varphi_j}/\sigma_j^2\right)\boldsymbol{\delta}_j^{e\,\mathrm{T}}$$

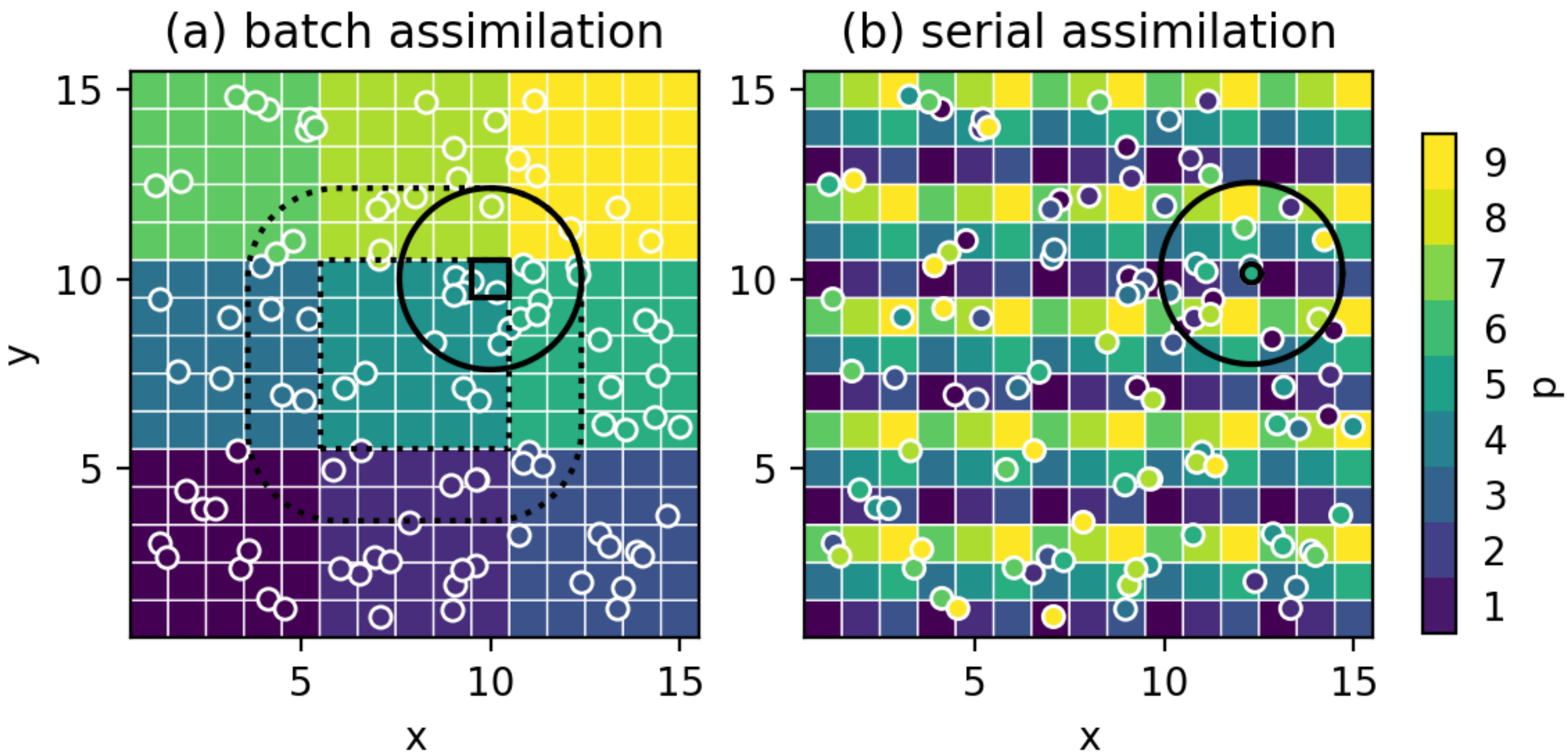
cost:

$\mathcal{O}\left(N_e \log N_p + N_e N_{\text{lstate}} + N_e N_{\text{lobs}}\right)\ \times N_{\text{obs}}$

"obs_incr": nonlinear filters possible

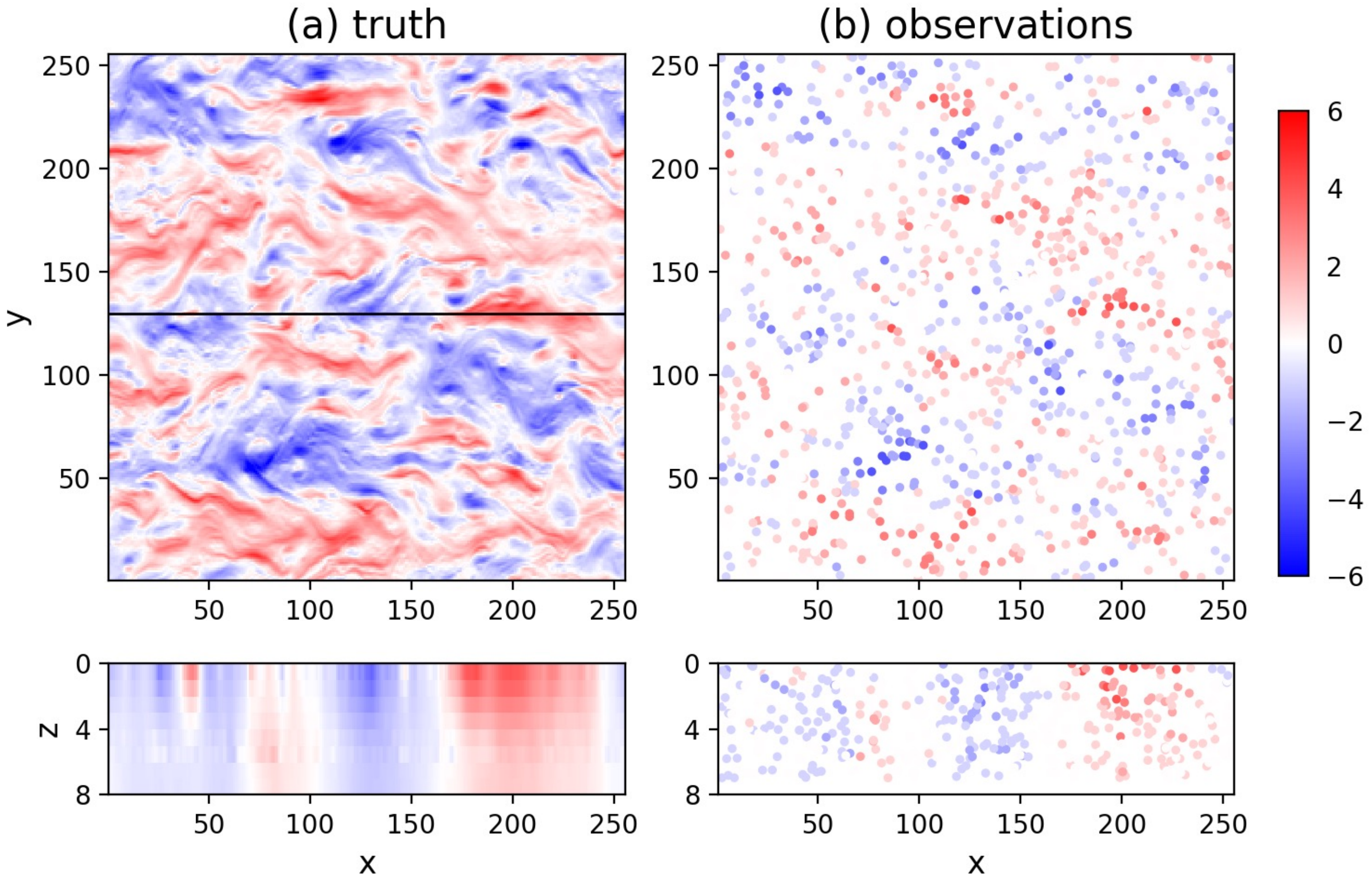
"obs_updates_ens": linear, probit-space

Memory layout for state/obs

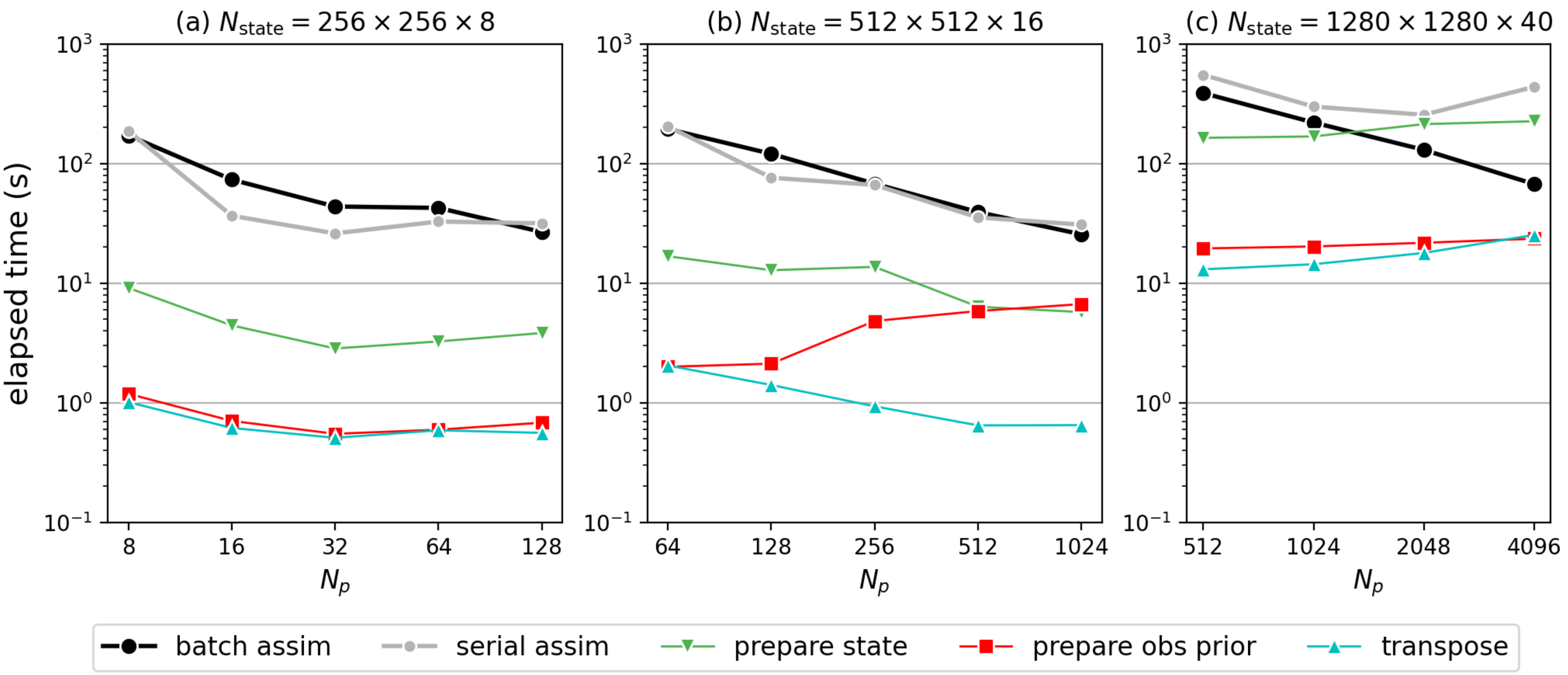


Benchmarking: QG model example

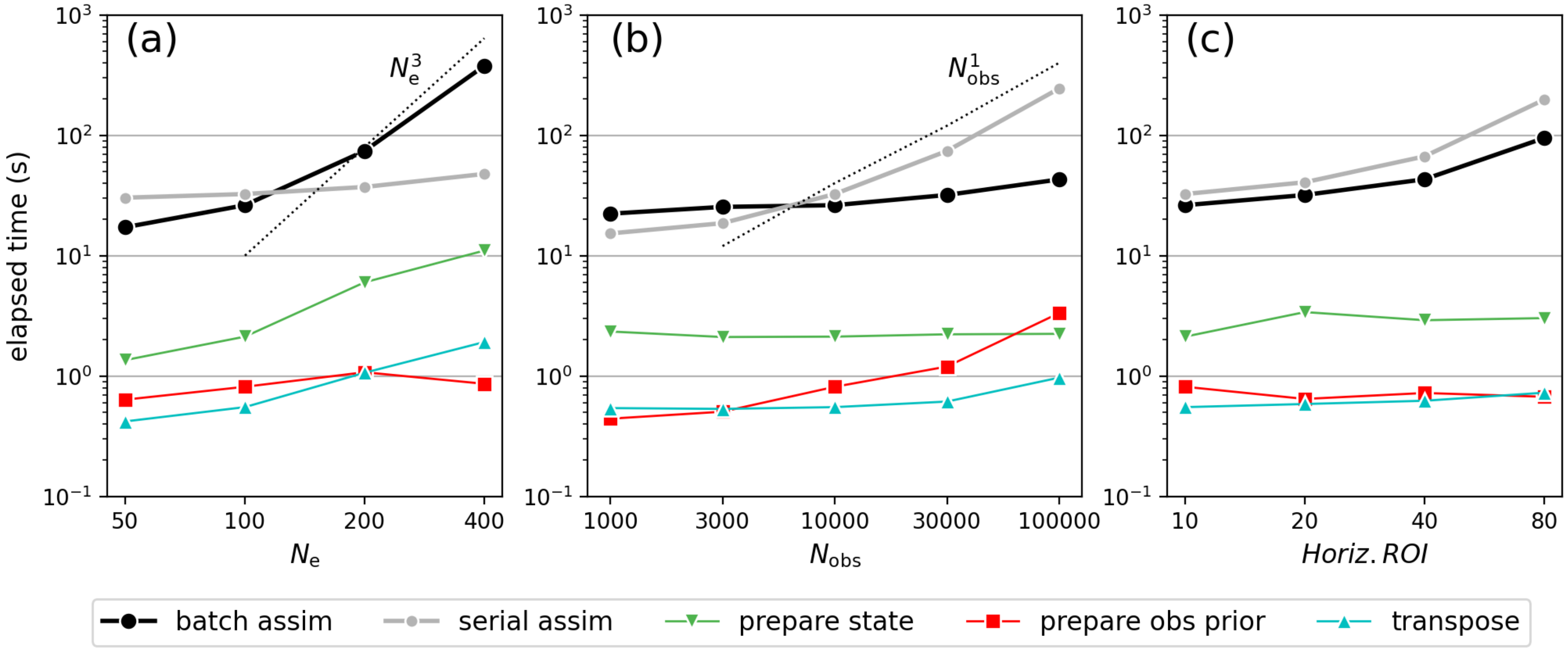
state and observations: velocity fields
Nstate = 256x256x8, Nobs = 10000, obs_err = 0.5, Ne = 100, hroi=10, vroi=5



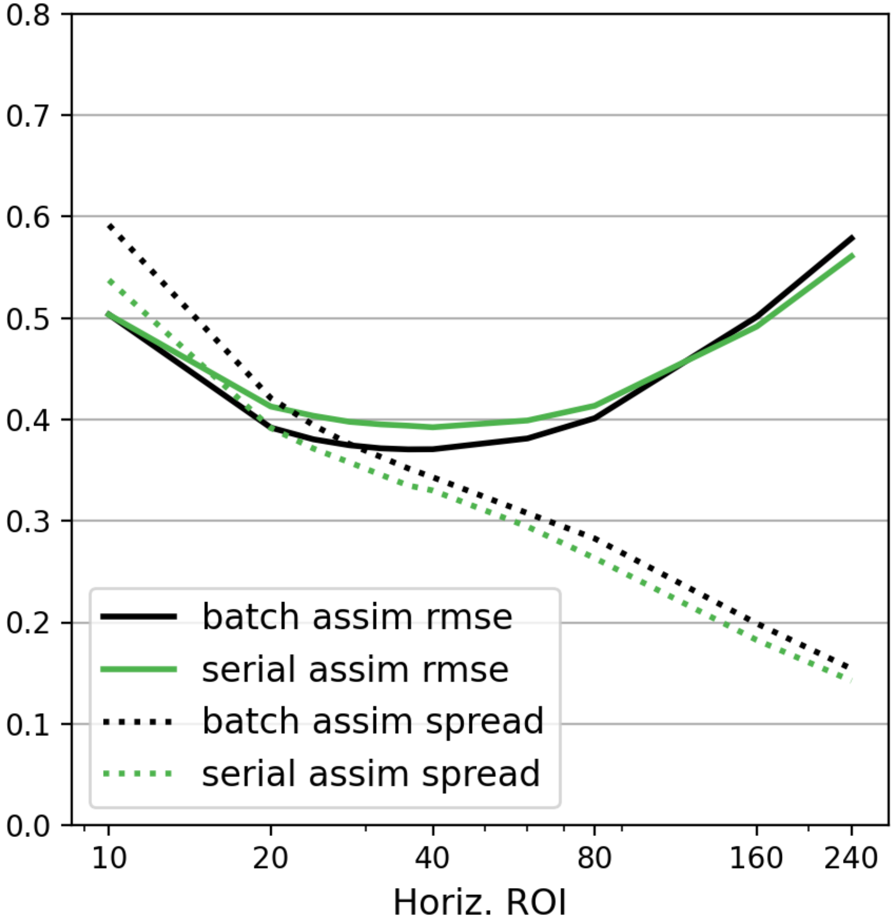
Scalability of $\mathcal{A}$ as N_p increases



How $\mathscr{A}$ scales as dimensionality increases



Analysis error/spread comparison



Both strategies produced comparable results

Serial assimilation fits more to observations (lower posterior spread); slightly less accurate (higher rmse)

Consistent with previous studies (Holland & Wang 2012; Nerger 2015).

Does the software run efficiently for DA problems?
Yes

Is it easy to change the DA workflow for new methods?

Different core algorithms in $\mathcal{A}$
Miscellaneous transform functions in $\mathcal{A}$

for $s = 1, \dots, N_s$ **do**

$\widetilde{\varphi}^o = \mathcal{T}^o_s(\varphi^o)$

for $m = 1, \dots, N_e$ **do**

$\widetilde{\varphi}_m = \mathcal{T}^o_s(\varphi_m)$

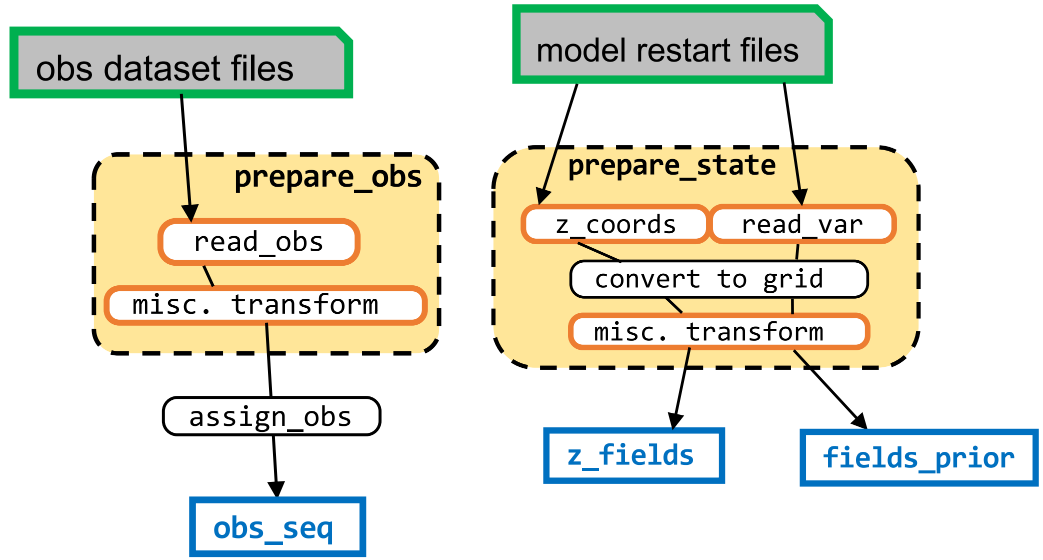
$\widetilde{\psi}_m = \mathcal{T}_s(\psi_m)$

end for

$\widetilde{\boldsymbol{\Psi}} = (\widetilde{\psi}_1, \dots, \widetilde{\psi}_{N_e})$

$\widetilde{\boldsymbol{\Phi}} = (\widetilde{\varphi}_1, \dots, \widetilde{\varphi}_{N_e})$

$(\widetilde{\psi}'_1, \dots, \widetilde{\psi}'_{N_e}) = \widetilde{\boldsymbol{\Psi}}' = \widetilde{\mathcal{A}}(\widetilde{\boldsymbol{\Psi}}, \widetilde{\boldsymbol{\Phi}}, \widetilde{\varphi}^o, \widetilde{\mathbf{R}}_s, \boldsymbol{r}^{\psi}_s, \boldsymbol{r}^{\varphi}_s, \boldsymbol{L}_s)$

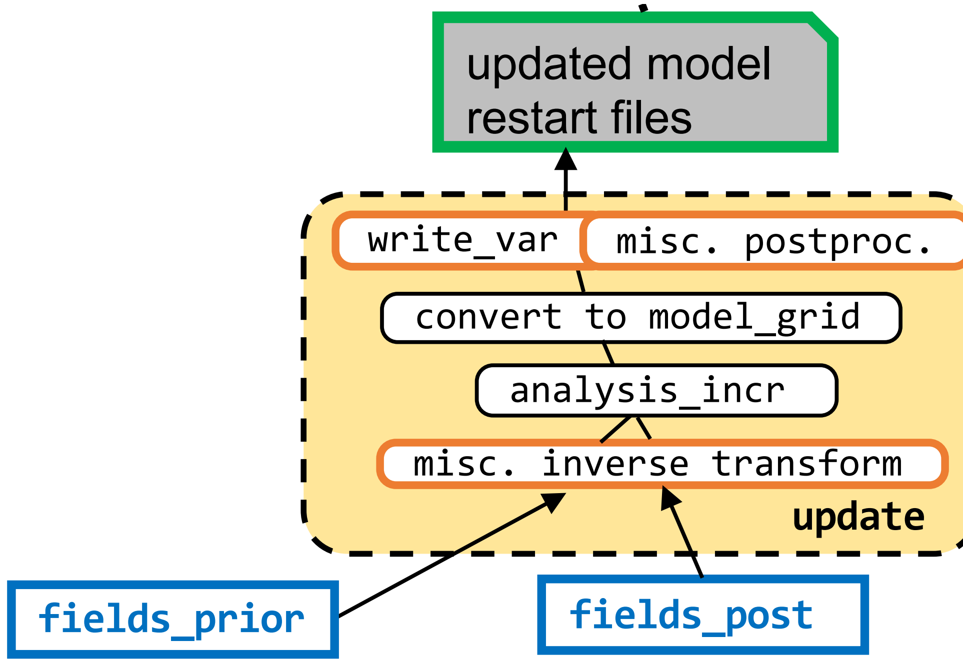


- Gaussian anamorphosis (Simon & Bertino 2009)
- Multiscale decomposition (Ying 2019, 2020), gcm-filters (Grooms et al. 2021)
- Super-resolution (Barthelemy et al 2022)
- Whitening (Snyder’s talk); Mapping to latent space (Chipilski 2023)

…

In update functions $\mathcal{U}$:

for $s = 1, \ldots, N_s$ **do**
    $\widetilde{\varphi}^o = \mathcal{T}_s^o(\varphi^o)$
    for $m = 1, \ldots, N_e$ **do**
        $\widetilde{\varphi}_m = \mathcal{T}_s^o(\varphi_m)$
        $\widetilde{\psi}_m = \mathcal{T}_s(\psi_m)$
    end for
    $\widetilde{\boldsymbol{\Psi}} = (\widetilde{\psi}_1, \ldots, \widetilde{\psi}_{N_e})$
    $\widetilde{\boldsymbol{\Phi}} = (\widetilde{\varphi}_1, \ldots, \widetilde{\varphi}_{N_e})$
    $(\widetilde{\psi}'_1, \ldots, \widetilde{\psi}'_{N_e}) = \widetilde{\boldsymbol{\Psi}}' = \widetilde{\mathcal{A}}(\widetilde{\boldsymbol{\Psi}}, \widetilde{\boldsymbol{\Phi}}, \widetilde{\varphi}^o, \widetilde{\mathbf{R}}_s, \boldsymbol{r}_s^{\psi}, \boldsymbol{r}_s^{\varphi}, \boldsymbol{L}_s)$
    for $m = 1, \ldots, N_e$ **do**
        $\psi_m \leftarrow \mathcal{U}_s(\psi_m, \widetilde{\psi}_m, \widetilde{\psi}'_m)$
    end for
end for



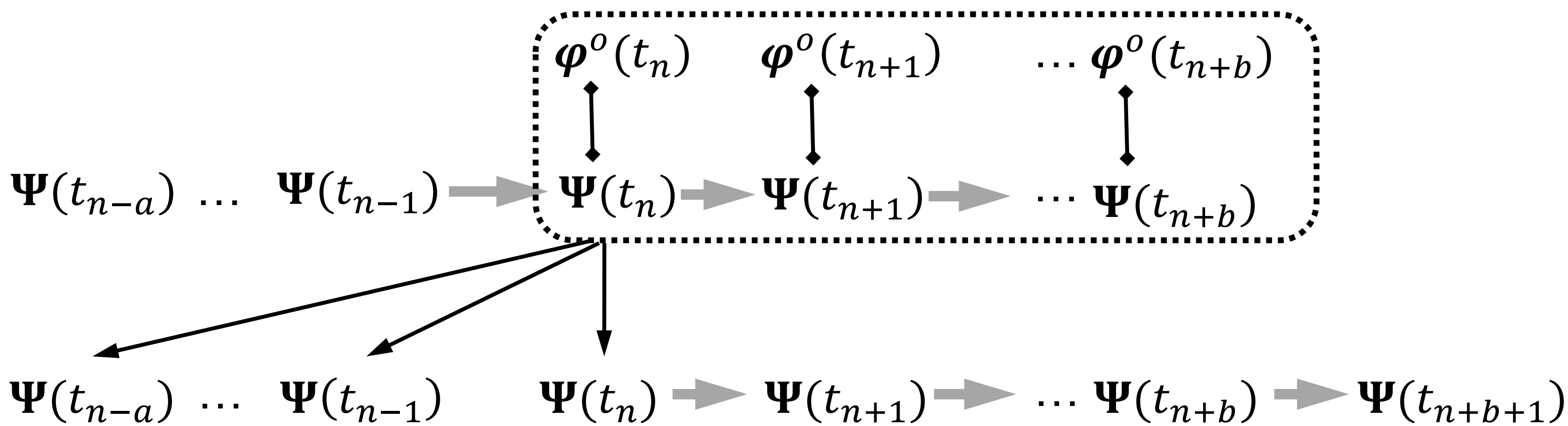
- Inverse transform, add increments
- Alignment techniques (Ying 2019)
- Adaptive inflation

...

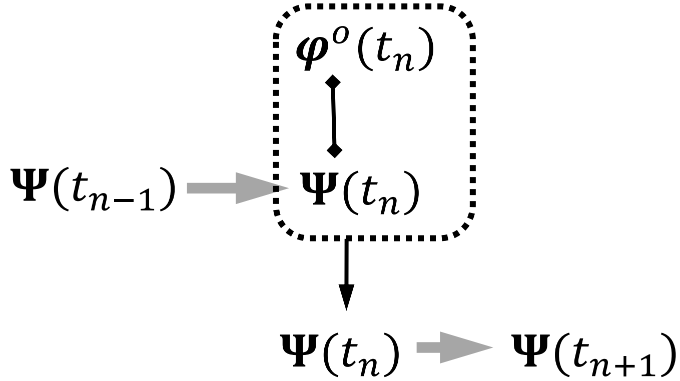
Extend to 4D analysis: smoothers

for $n = 1, \dots, N_t$ **do**
$\quad \boldsymbol{\Psi}(t_{n-1}) \leftarrow \mathcal{P}[\boldsymbol{\Psi}(t_{n-1})]$
$\quad \boldsymbol{\Psi}(t_n) = \mathcal{M}_n\left[\boldsymbol{\Psi}(t_{n-1})\right]$
$\quad \boldsymbol{\Phi}(t_n) = \mathcal{H}_n\left[\boldsymbol{\Psi}(t_n)\right]$
$\quad \boldsymbol{\Psi}(t_n) \leftarrow \mathcal{A}[\boldsymbol{\Psi}(t_n), \boldsymbol{\Phi}(t_n), \boldsymbol{\varphi}^o(t_n$
end for

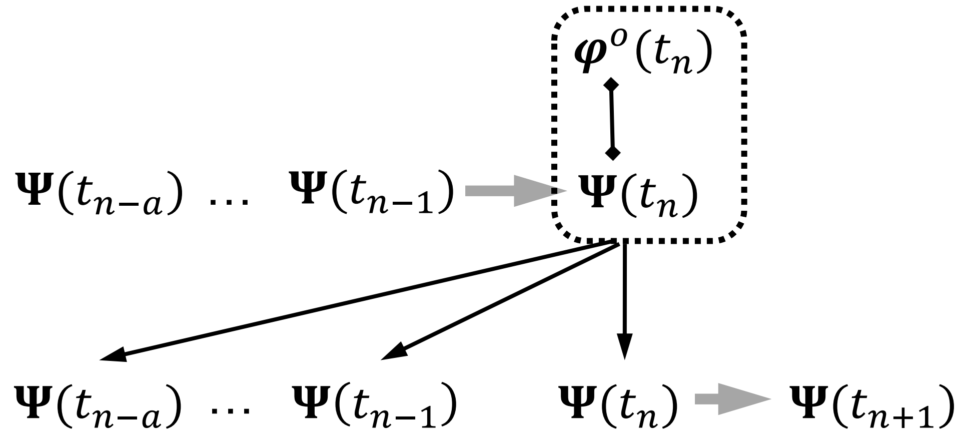
for $n = 1, \dots, N_t$ **do**
$\quad \boldsymbol{\Psi}(t_{n-1}) \leftarrow \mathcal{P}[\boldsymbol{\Psi}(t_{n-1})]$
$\quad$ **for** $k = 0, \dots, b$ **do**
$\quad\quad \boldsymbol{\Psi}(t_{n+k}) = \mathcal{M}_{n+k}\left[\boldsymbol{\Psi}(t_{n+k-1})\right]$
$\quad\quad \boldsymbol{\Phi}(t_{n+k}) = \mathcal{H}_{n+k}\left[\boldsymbol{\Psi}(t_{n+k})\right]$
$\quad$ **end for**
$\quad \boldsymbol{\Psi}(t_{n-a:n}) \leftarrow \mathcal{A}[\boldsymbol{\Psi}(t_{n-a:n}), \boldsymbol{\Phi}(t_{n:n+b}), \boldsymbol{\varphi}^o(t_{n:n+b}), (b+1)\mathbf{R}(t_{n:n+b})]$
end for



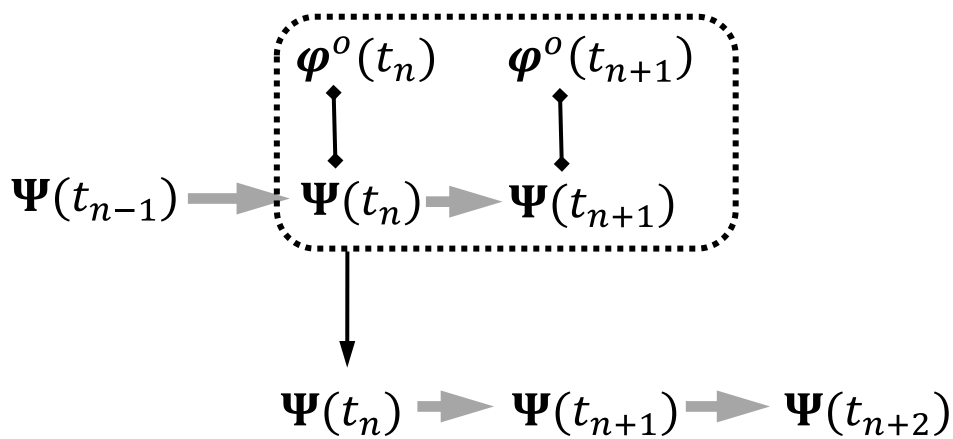
$a = b = 0$: filter



$b = 1, a > 0$:
recursive smoother



$b = 1, a = 0$:
one-step-ahead
smoother



Does the software run efficiently for DA problems?

Yes

Is it easy to change the DA workflow for new methods?

Yes

Is it easy to add a new model?

Support various model grid geometries and map projections

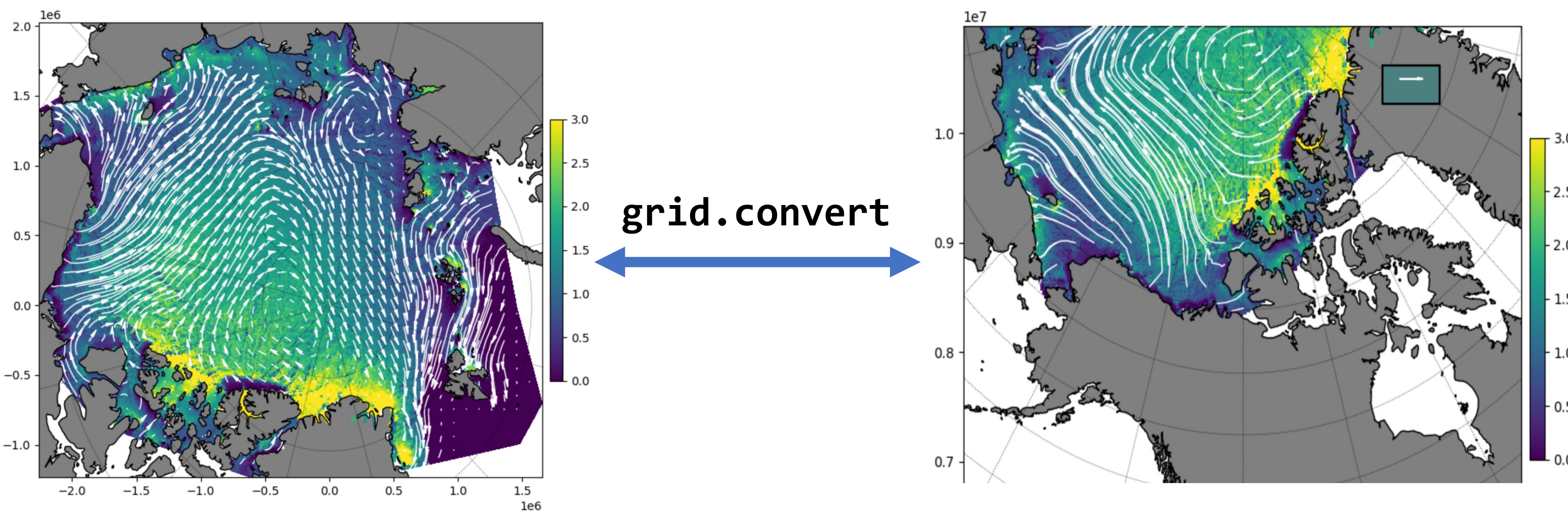
Efficient conversion between analysis grid and model native grid
- cached self.interp_weights for repeated calls to self.convert()

neXtSIM:

```
self.grid = Grid(proj, x, y, regular=False, triangles=triangles)
```

QG:

```
self.grid = Grid(proj, x, y, regular=True, cyclic_dim='xy')
```



```
read_var(self, path, name='ocean_temp', member=m, k=k):
    ##user provided code to look for
    ##variable[name] in the model restart file
    ##for ensemble member m
    ##at model level k
    return field

write_var(self, field, **kwargs):
    ##write the field (analysis) back to restart file

z_coords(self, **kwargs):
    ##vertical coordinate for a given field
    return z


run(self, task_id=0, task_nproc=16, **kwargs):
    ##user provided procedure to
    ##run the forecast model
```

File Edit View Insert Cell Kernel Widgets Help Not Truste

```
output_ens_mean(c, state_info, mem_list, rec_list, fields_prior, mean_file)

##z coordinates
z_file = path+'/z_coords.bin'
output_ens_mean(c, state_info, mem_list, rec_list, z_fields, z_file)
```

```
number of unique field records, nrec=1
prepare state by reading fields from model restart
................................| 100% done.
save state to /Users/yueng/scratch/vort2d_testcase/cycle/200101010300/analysis//prior_state.bin
................................| 100% done.
compute ensemble mean, save to /Users/yueng/scratch/vort2d_testcase/cycle/200101010300/analysis//
in
................................| 100% done.
compute ensemble mean, save to /Users/yueng/scratch/vort2d_testcase/cycle/200101010300/analysis//
................................| 100% done.
```

In [11]:

```
##get obs info
obs_info = parse_obs_info(c)

##distribute obs_rec_id among processors
obs_rec_list = build_obs_tasks(c, obs_info)

##read dataset and prepare the obs sequence
obs_seq = prepare_obs(c, state_info, obs_info, obs_rec_list)

##partition the analysis grid
partitions = partition_grid(c)

##assign obs to each partition
obs_inds = assign_obs(c, state_info, obs_info, partitions, obs_rec_list, obs_seq)

##figure out the workload on each partition
##and distribute the par_id among processors
par_list = build_par_tasks(c, partitions, obs_info, obs_inds)

##prepare the obs priors, call state_to_obs
##to apply the forward obs operators
obs_prior_seq = prepare_obs_from_state(c, state_info, mem_list, rec_list, obs_info,
                                       obs_rec_list, obs_seq, fields_prior, z_fields)
```

```
reading obs sequences from dataset
number of velocity obs from vort2d: 150
compute obs priors
................................| 100% done.
```

In [12]:

```
##transpose from field-complete to ensemble-complete
state_prior = transpose_field_to_state(c, state_info, mem_list, rec_list,
                                       partitions, par_list, fields_prior)

z_state = transpose_field_to_state(c, state_info, mem_list, rec_list,
                                   partitions, par_list, z_fields)

lobs = transpose_obs_to_lobs(c, mem_list, rec_list, obs_rec_list,
                             par_list, obs_inds, obs_seq)

lobs_prior = transpose_obs_to_lobs(c, mem_list, rec_list, obs_rec_list,
                                   par_list, obs_inds, obs_prior_seq,
                                   ensemble=True)
```

```
transpose field to state
................................| 100% done.
transpose field to state
................................| 100% done.
obs sequences: transpose obs to local obs
................................| 100% done.
obs prior sequences: transpose obs to local obs
................................| 100% done.
```

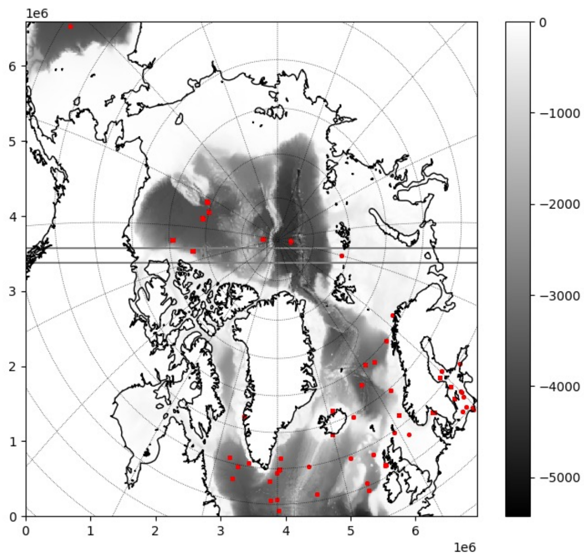
In [13]:

```
##the core assimilation algorithm

if c.assim_mode == 'batch':
    assimilate = batch_assim
elif c.assim_mode == 'serial':
    assimilate = serial_assim

state_post = assimilate(c, state_info, obs_info, obs_inds,
                        partitions, par_list, rec_list,
                        state_prior, z_state,
                        lobs, lobs_prior)
```

```
assimilate in batch mode:
................................| 100% done.
```

```
grid.plot_land(ax)

ax.scatter(obs['x'], obs['y'], 20, color='r', marker='.')

ax.plot(grid.x[j,:], grid.y[j,:]+dy, color='gray')
ax.plot(grid.x[j,:], grid.y[j,:]-dy, color='gray')
```

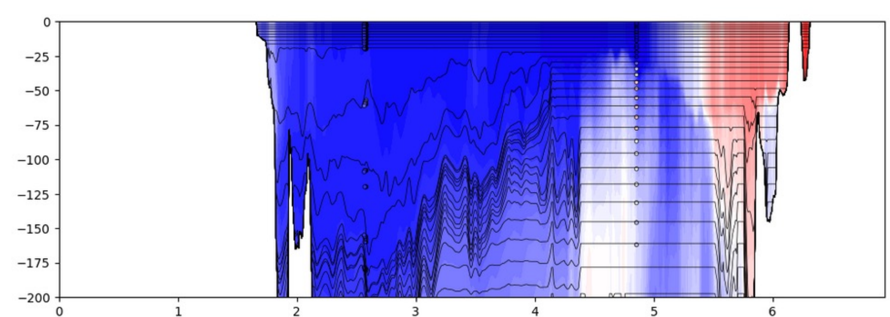
Out[10]: [<matplotlib.lines.Line2D at 0x7f5bbff85010>]



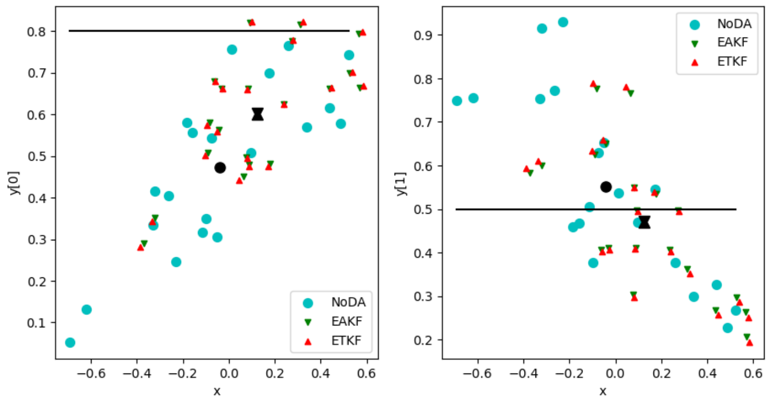
In [11]:

```
fig, ax = plt.subplots(1, 1, figsize=(12,4))

x, k = np.meshgrid(grid.x[j, :], np.arange(nz))

vmin = -2
vmax = 8
cmap = [plt.cm.bwr(x) for x in np.linspace(0, 1, round(vmax-vmin)+1)]
ax.contourf(x, z[:, j, :], var[:, j, :], np.arange(vmin, vmax, (vmax-vmin)/40), cmap=plt.cm.bwr)

##show isopycnal grid
ax.contour(x, z[:, j, :], k, np.arange(0,nz+1,1)-1e-10, colors='k', linewidths=0.5)

#ax.set_xlim(3e6, 5e6)
ax.set_ylim(-200, 0)

##show obs value
vout = np.array(obs['obs'])[inds]
cind = np.maximum(np.minimum(np.round(vout-vmin), int(np.round(vmax-vmin))), 0).astype(int)
ax.scatter(np.array(obs['x'])[inds], np.array(obs['z'])[inds], 10, color=np.array(cmap)[cind, 0:3], edgecolor='k', l
```

Out[11]: <matplotlib.collections.PathCollection at 0x7f5bc0fcaa90>

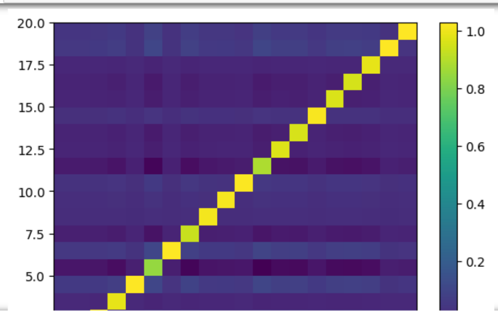


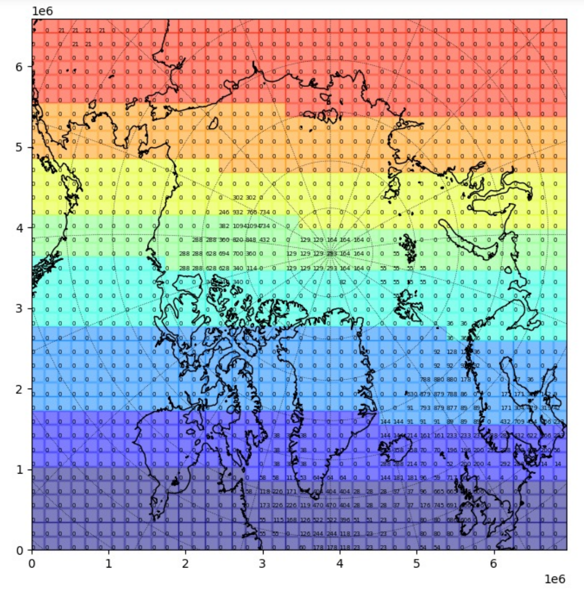
```
In [44]: fig, ax = plt.subplots(1, nlobs, figsize=(10,5))

         for i in range(nlobs):
             ax[i].scatter(xb, yb[i,:], 50, color='c', marker='o', label='NoDA')
             ax[i].plot([xb.min(), xb.max()], [yo[i], yo[i]], 'k')
             ax[i].scatter(xa1, ya1[i,:], 20, color='g', marker='v', label='EAKF')
             ax[i].scatter(xa2, ya2[i,:], 20, color='r', marker='^', label='ETKF')
             ax[i].plot(np.mean(xb), np.mean(yb[i,:]), 'ko', markersize=8)
             ax[i].plot(np.mean(xa1), np.mean(ya1[i,:]), 'kv', markersize=8)
             ax[i].plot(np.mean(xa2), np.mean(ya2[i,:]), 'k^', markersize=8)
             ax[i].set_xlabel('x')
             ax[i].set_ylabel('y[{}]'.format(i))
             ax[i].legend()
```



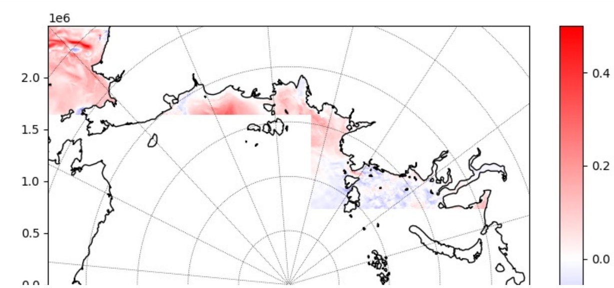

```
In [45]: xa1, xa2, np.mean(xa1), np.mean(xa2)

Out[45]: (array([ 0.57072927, -0.06367865, -0.03026054, -0.0810362 ,  0.31261495,
                 -0.37024458,  0.56507758,  0.06408989,  0.2384875 ,  0.08337977,
                  0.52690252, -0.31991302,  0.07762127,  0.08918161,  0.43755631,
                 -0.09118427,  0.1786681 , -0.04217762,  0.09376354,  0.27385974]),
          array([ 0.58394821, -0.05906138, -0.02642342, -0.09664717,  0.32381735,
                 -0.38667353,  0.58000381,  0.04468252,  0.24127983,  0.0808685 ,
                  0.5371702 , -0.33505647,  0.0808253 ,  0.08696181,  0.4479457 ,
                 -0.10231946,  0.17080215, -0.05264181,  0.09885362,  0.27752865]),
          0.12567185880638684,
          0.12479322098338604)
```

```
In [32]: im = plt.pcolor(W.real)
         plt.colorbar(im)
         for m in range(nens):
             print(np.sum(W[:, m]))
```






```
In [23]: fld_id = [i for i,r in field_info['fields'].items() if r['name']=='ocean_velocity' and r['k']==
         mem_id = 0

In [24]: mem_id, fld_id

Out[24]: (0, 0)

In [ ]: proc_id = 8
        state = np.load('/cluster/work/users/yingyue/dat.{:04d}.npz'.format(proc_id), allow_pickle=True

In [33]: fig, ax = plt.subplots(1, 1, figsize=(9,7))

         out = np.full((2, ny, nx), np.nan)

         for tile_id in state[mem_id, fld_id]:
             istart,iend,jstart,jend = tile_list_proc[proc_id][tile_id]
             tile_j, tile_i = np.where(~c.mask[jstart:jend, istart:iend])
             out[..., jstart:jend, istart:iend][..., tile_j, tile_i] = state[mem_id, fld_id][tile_id]

         im = c.grid.plot_field(ax, out[0, ...], vmin=-0.5, vmax=0.5, cmap='bwr')
         plt.colorbar(im)
         c.grid.plot_land(ax)
```



NEDAS

NERSC

Does the software run efficiently for DA problems?
Yes

Is it easy to change the DA workflow for new methods?
Yes

Is it easy to add a new model?
Yes

Code publicly available:
https://github.com/nansencenter/NEDAS (new develop branch)

Manuscript in review on JAMES:
Ying: “Introducing NEDAS: a light-weight and scalable Python solution for ensemble data assimilation”

Contact me: yue.ying@nersc.no

More detailed workflow for $\mathcal{A}$