DataMove

Data Aware Large Scale Computing

MELISSA-DA

A new framework for elastic ensemble-based data assimilation at large-scale

Sebastian Friedemann, Inria

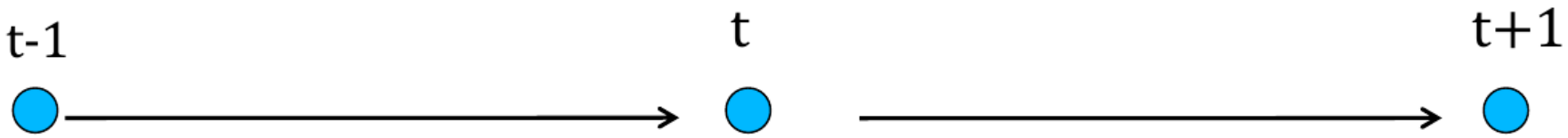
sebastian.friedemann@inria.fr, bruno.raffin@inria.fr

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 824158 (EoCoE-2). This work was granted access to the HPC resources of IDRIS under the allocation 2020-A8 A0080610366 attributed by GENCI(Grand Equipement National de Calcul Intensif).

Motivation

- Data Assimilation (**DA**) for more and more complex models
  - Higher resolution (curse of dimensionality)
  - Account for very *chaotic* situations
  - Some filters need much more members (e.g., Particle Filters)
- Traditional approaches rely on file system or large monolithic jobs that are hard to govern (long wait time, large lost on crash of a single component)
- This motivates a framework that is
  - Fast avoiding startup costs and data transport through the file system
  - Resilient, recovering from faults...
  - Modular
  - Easy to deploy
    - Independence of allocation size from member amount
    - Different degrees of parallelism
- We will assimilate the hydrological model code ParFlow:
  - a physically based, fully coupled water transfer model for the critical zone
  - O(4M) degrees of freedom, 16 000 members, using EnKF

DA workflow at the example of Ensemble Kalman Filter (EnKF)



New Observations

Given parallel model code

Given assimilation scheme

1. Predict Ensemble

$x^b_{t,m} = M(x^a_{t-1,m})$

2. Compute Kalman Gain

$$P^bH^T = X^b(X^b)^TH^T = X^b(HX^b)^T = \frac{1}{M-1}\Sigma_m(x^b_m - \langle x^b_m \rangle)\,(Hx^b_m - \langle H(x^b_m)\rangle)^T$$

$$HP^bH^T = HX^b(HX^b)^T = \frac{1}{M-1}\Sigma_m(Hx^b_m - \langle H(x^b_m)\rangle)\,(Hx^b_m - \langle H(x^b_m)\rangle)^T$$

$$K = P^bH^T(HP^bH^T + R)^{-1}$$

3. Update state estimate

$$x^a = x^b + K(y - H(x^b)) = (I-KH)\,x^b + Ky$$

Update

1. Predict Ensemble

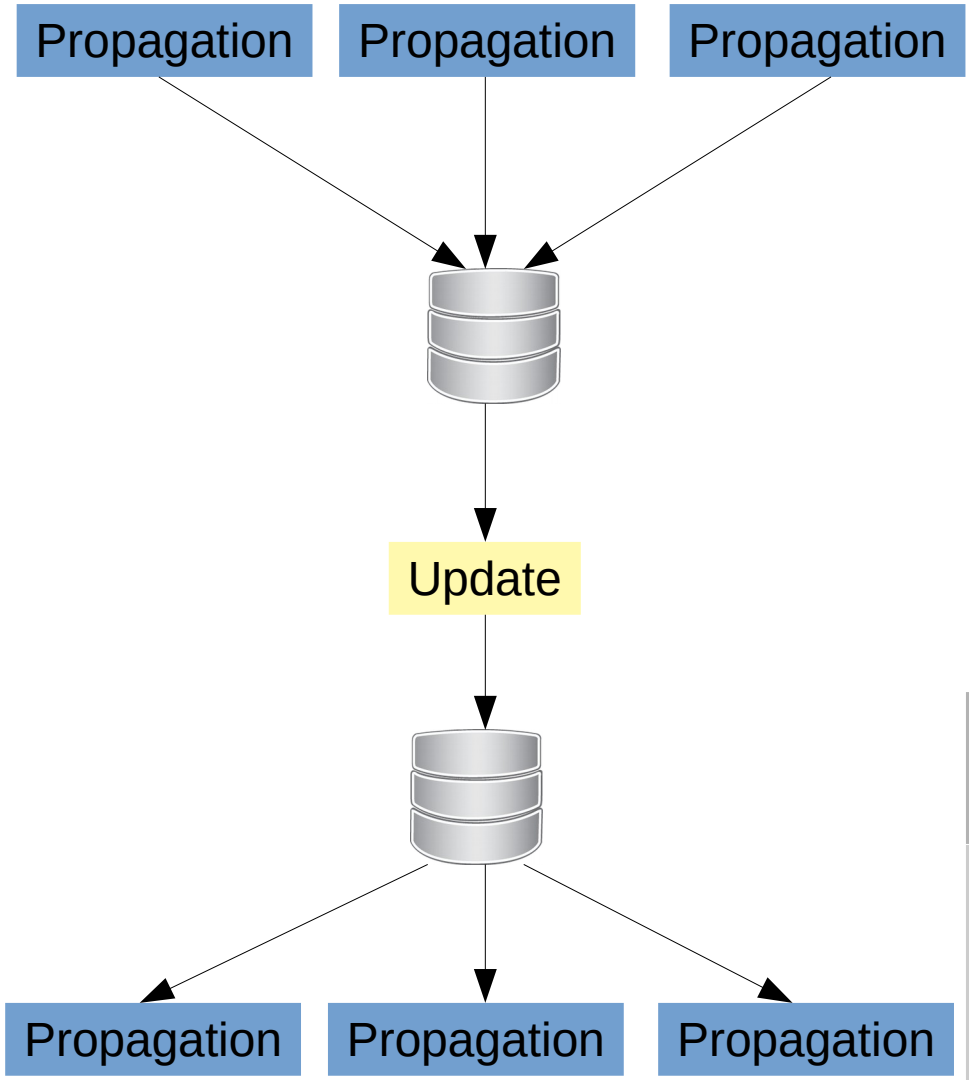
$x^b_{t,m} = M(x^a_{t-1,m})$

Propagation

Propagation

We will design a framework to map the workflow parts on a HPC platform

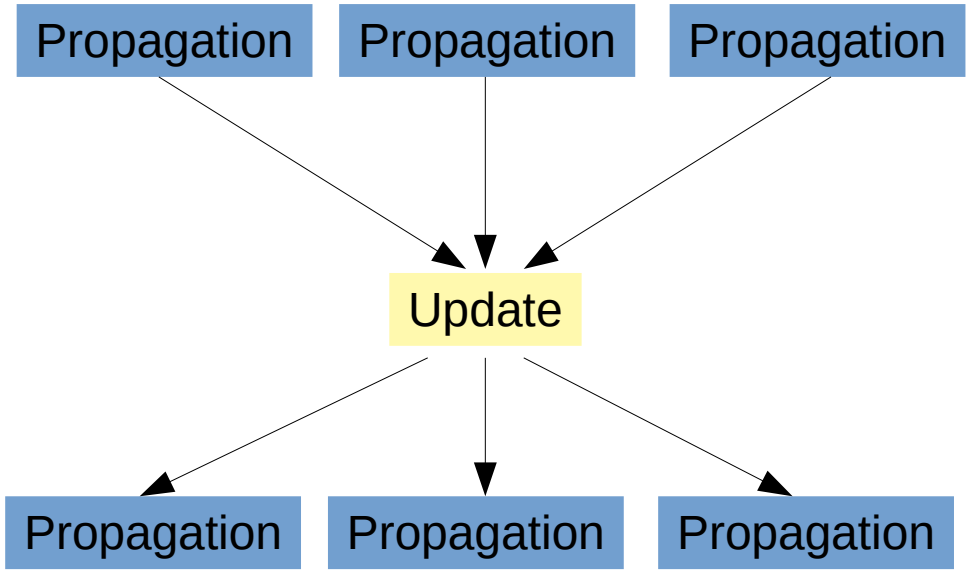
Off-line workflow



- Members are propagated
- Output is written to disk
- Output is read in
- Filter update is performed
- New model input is written out
- New model input is loaded → next cycle
- EnTK, DART, PDAF, … support this

+	-
• Easily allows different parallelisms • Fault tolerant (restart problematic component)	• Enormous pressure on the file system when going large scale • Model startup overhead many times

On-line workflow

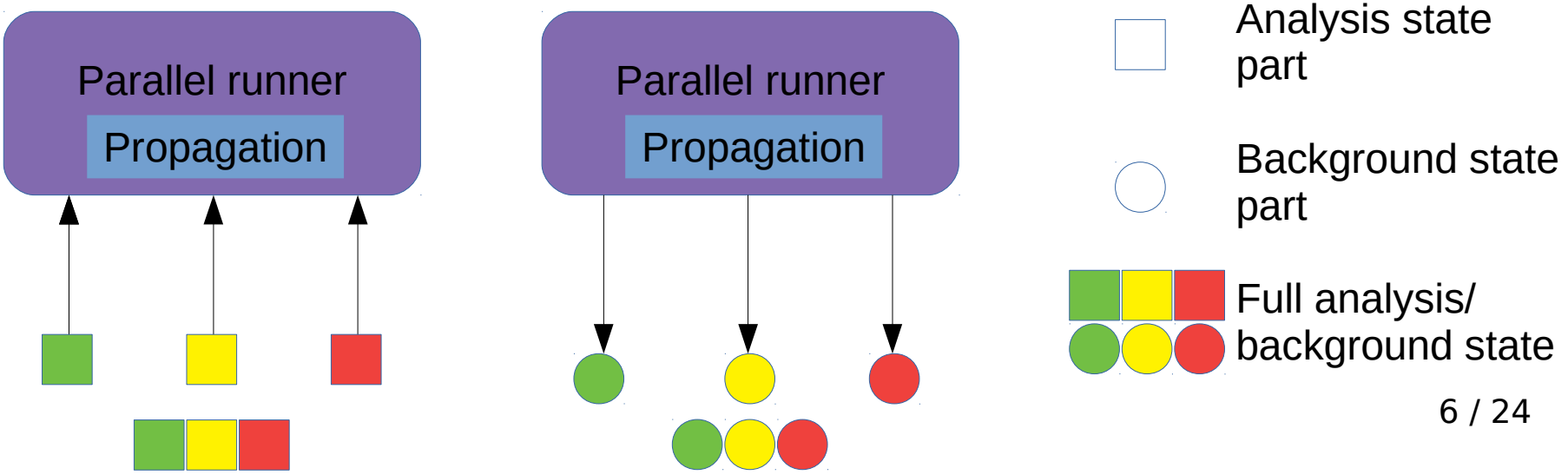


+	-
• Avoid file system bottleneck • Avoid model startup cost • Multiple propagations by the same resource to decouple resource usage and members	• Not fault tolerant (one member failing, will crash the whole run) • Big monolithic jobs are necessary • Idle resources during update phase

- Members are propagated
- Output is gathered on some compute resources (only RAM to RAM copy)
- Filter update is performed on those compute resources
- New model input is scattered back (only RAM to RAM copy)
- Often implemented using MPI (e.g., DART, PDAF)

Towards a new framework: parallel runners

- Fast avoiding startup costs and data transport through the file system:
  - Transform simulation codes into parallel runners
  - Parallel runners open connections to a central point where they ask for an analysis state to propagate
  - This is parallelized (there is actually one connection per rank transferring state **parts** that need to be assembled to build the full state)
  - Then runners propagate this state and send back the resulting background state



In the parallel runner

```
x = Model_Init()

for t < t_end:
    Integrate(x)
    Write_Output(x)



Model_Finalize(x)
```

In the parallel runner

Parallel runner

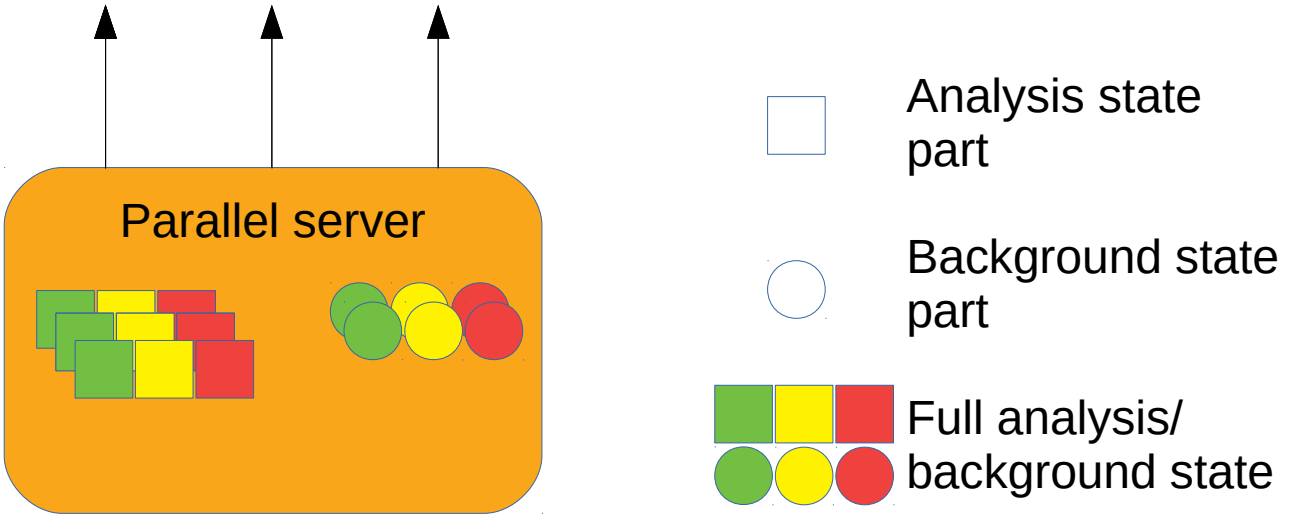
```
x = Model_Init()

melissa_da_init(…)

while melissa_da_expose(x) != 0:
    Integrate(x)
    # optional: Write_Output(x)

Model_Finalize(x)
```

Init data assimilation

Change simulation's time stepping loop to propagate state coming from the Melissa-Da server

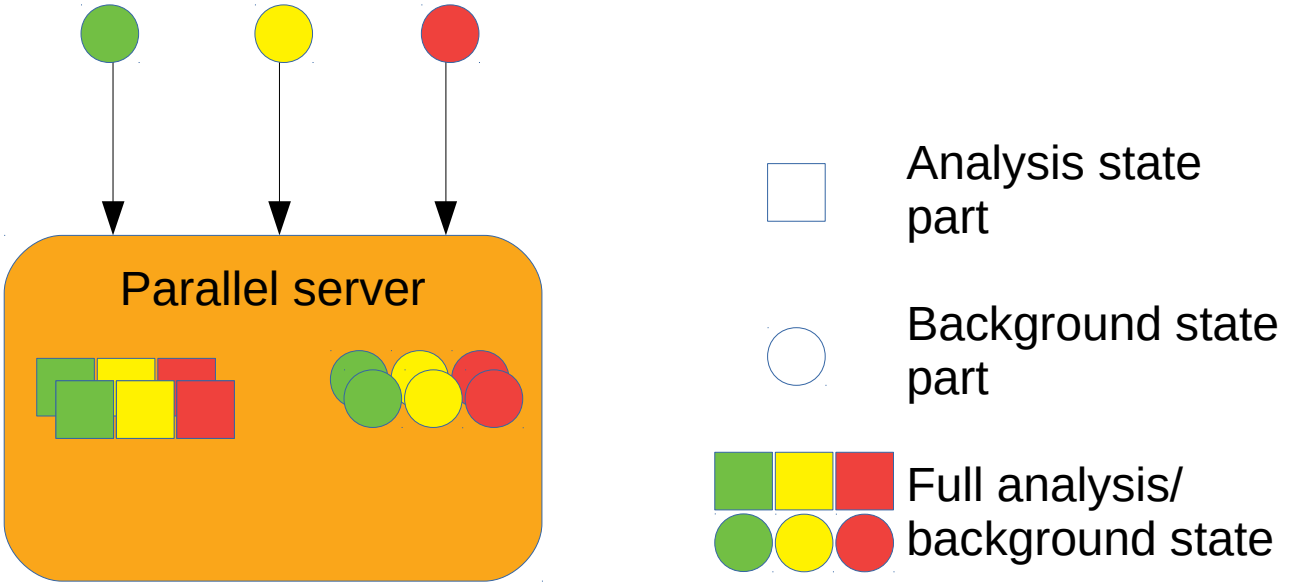
The parallel server

- The Parallel server distributes analysis states and receives background states



The parallel server

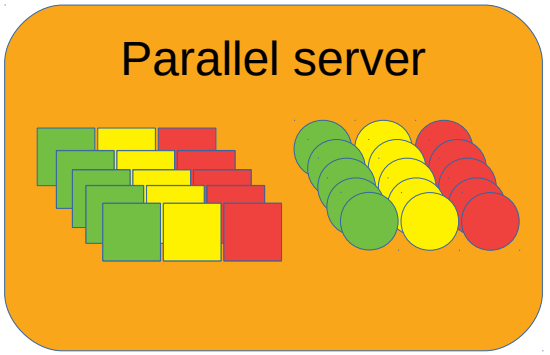
- The Parallel server distributes analysis states and receives background states

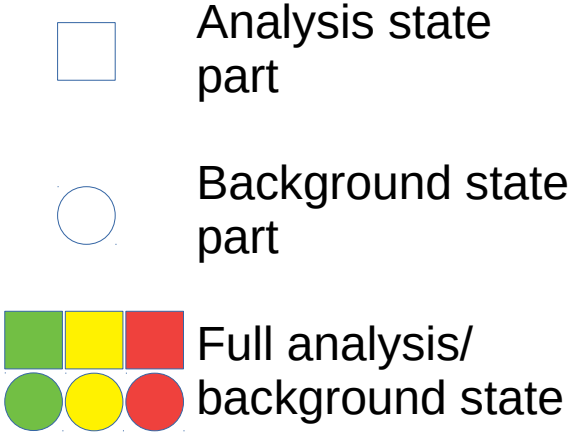


Update Phase

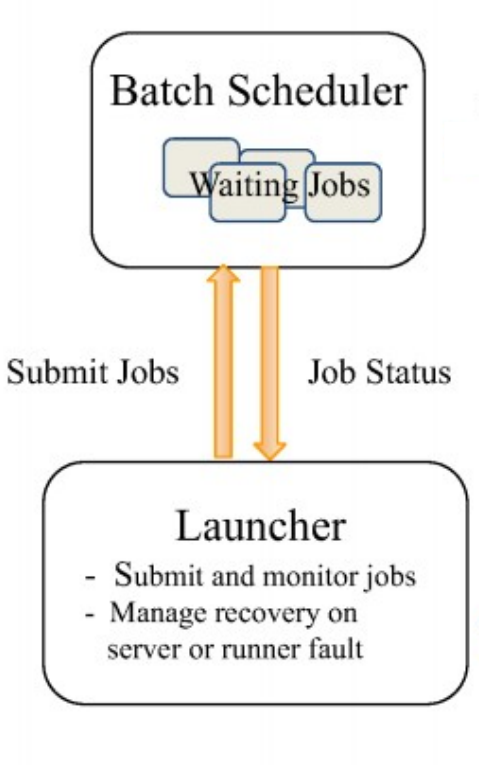
- The parallel server performs the filter update generating the new set of analysis states
  - It must be provided a function transforming background states into analysis states

PDAF Parallel Data Assimilation Framework



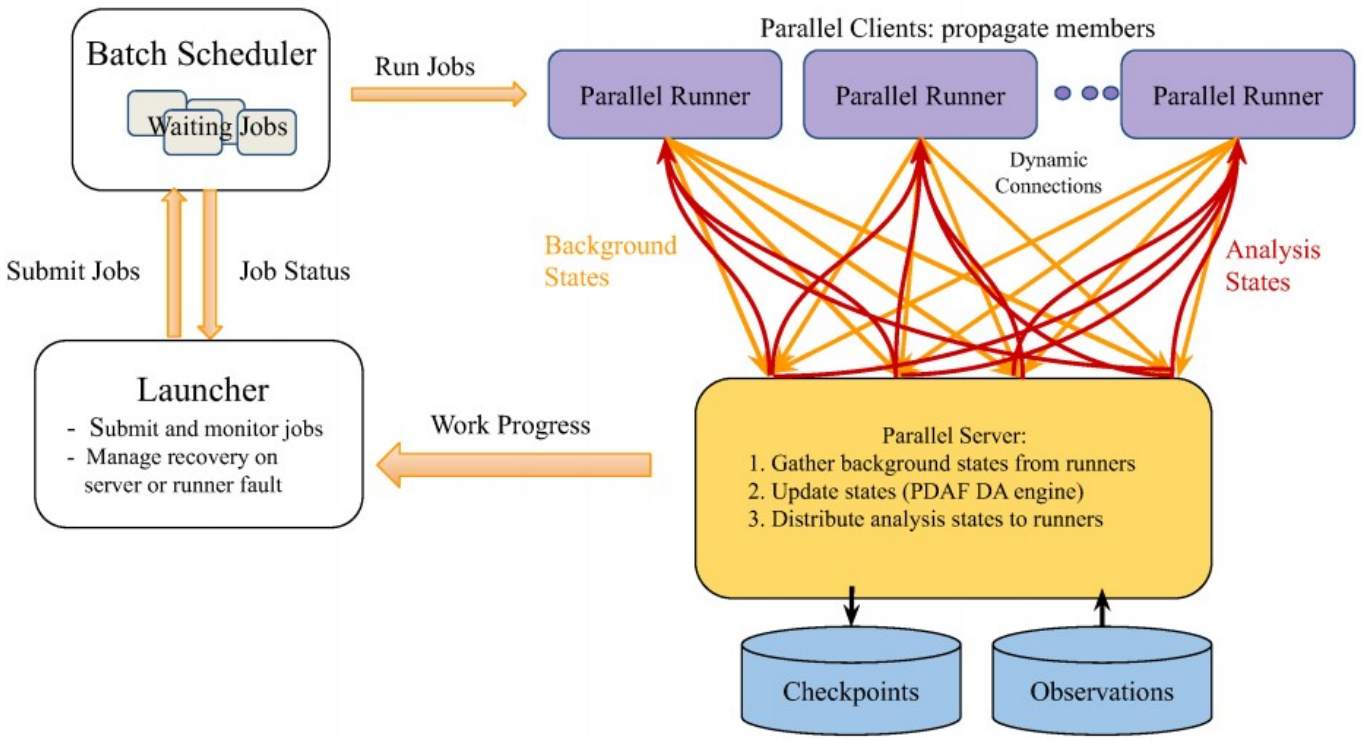


Deployment using the launcher



- Launcher starts up components (runner, server) one by one interacting with the batch scheduler
  - Each in its own job allocation
- Launcher also monitors the components restarts on error

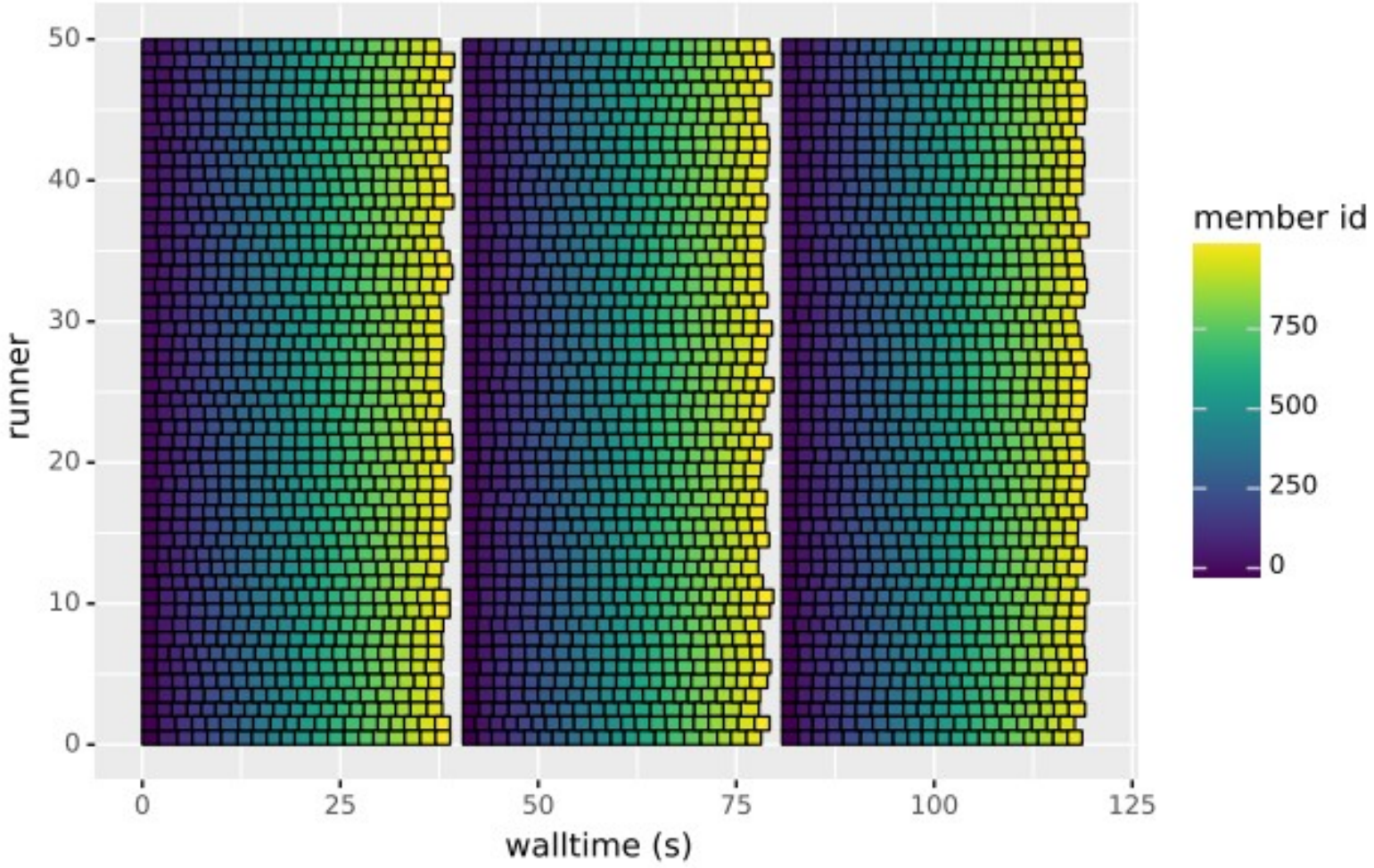
Our Approach: MELISSA-DA



- All connections between components are dynamic (relying on ZeroMQ)
  - Components may join / leave the distributed application at any time
- The launcher takes care to restart components that crashed
- If the server does not receive a full background state in time it will send the according analysis state for propagation to another runner

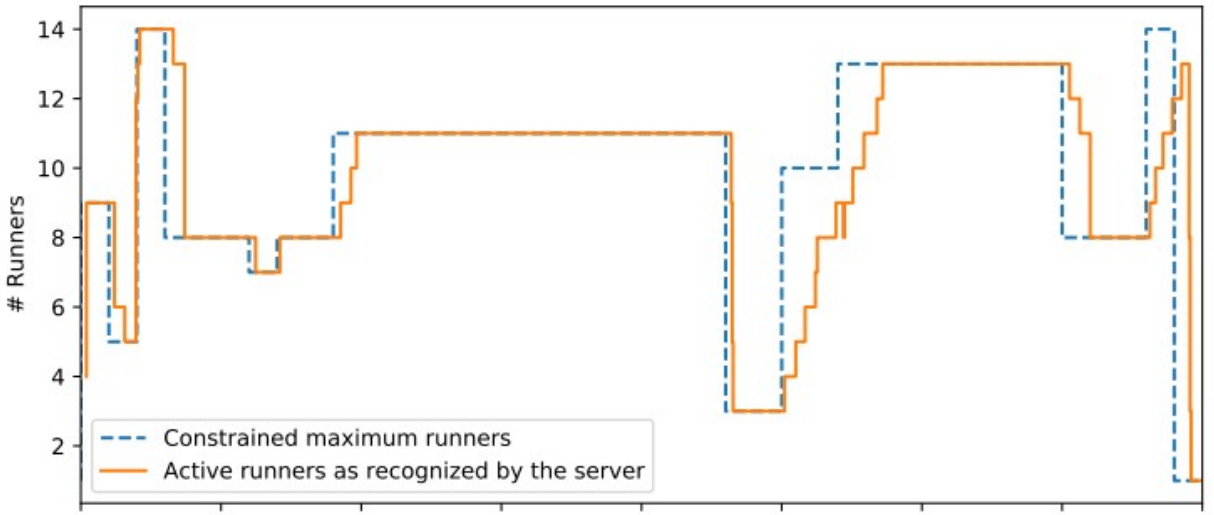
A typical run of Melissa-DA

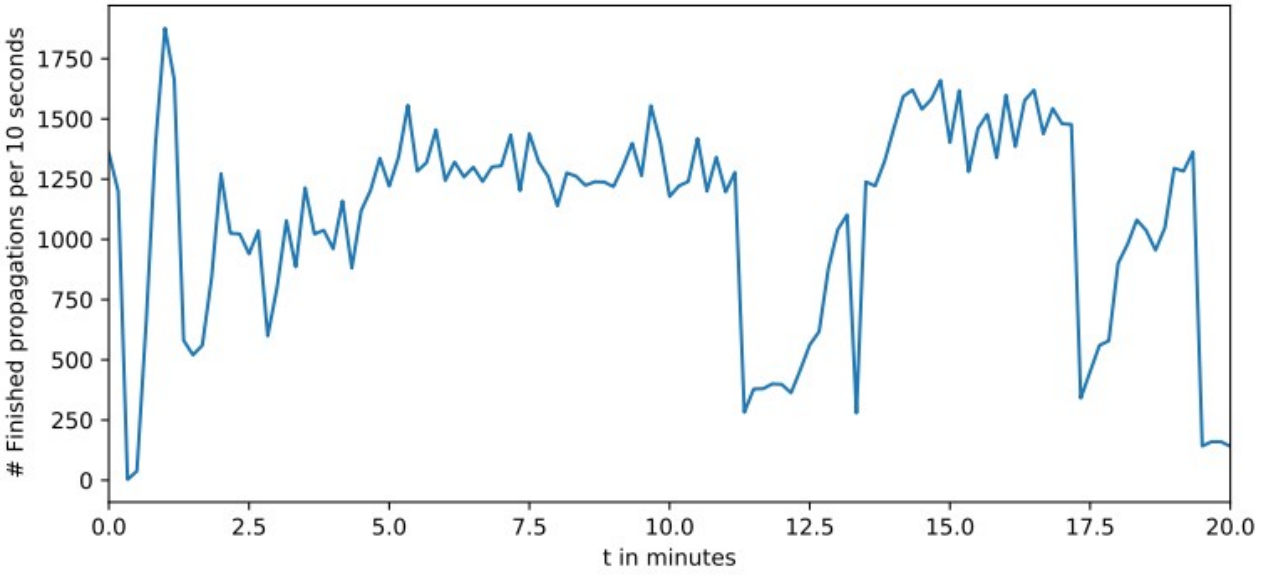
Member propagations per runner



→ dynamic load balancing
→ run on heterogeneous architecture

Elasticity, Fault tolerance





Runner amount constrained, varies
→ The execution never stops, only its pace changes

Modularity

- Modules for different batch scheduler systems (local, Slurm (JUWELS, Jean-Zay, Marenostrum…))
  - can be extended for personal needs
- Any (instrumented) simulation code (***M***) can be coupled with any Assimilation method
  - Necessitates user to provide functions converting data, ***H*** operator, reading of observations
  - C/C++, Fortran or Python API available
- Assimilation methods are modules too
  - Use existing code (e.g., PDAF) or prototype your own (Python possible)

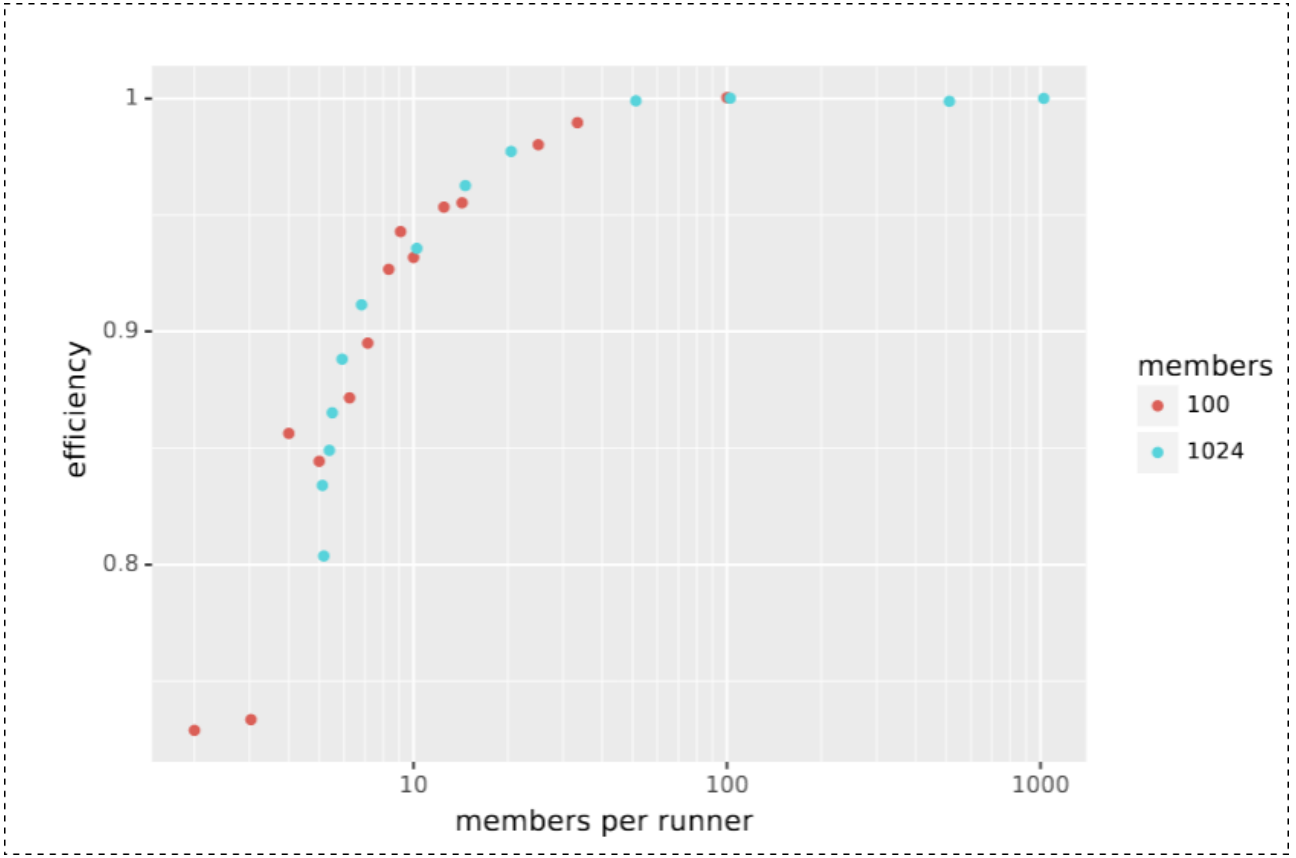
Deployment

TLDR - How to run a DA study:

1. Install Melissa-DA & dependencies
2. Instrument and link your model against Melissa-DA (or use one of the example models to start)
3. Configure your assimilator by writing a new assimilator or writing a new pdaf-wrapper library to be preloaded at runtime (or use one of the existing assimilators for the beginning)
4. Launch your simulation from within a simple python script:

```
from melissa_da_study import *

run_melissa_da_study(
        runner_cmd='simulation1',               # which model code to use
        total_steps=3,                          # how many assimilation cycles to run
        ensemble_size=3,                        # ensemble size
        assimilator_type=ASSIMILATOR_DUMMY,     # which assimilator to chose during DA update phase.
                                                # Often further options must be specified using environment
                                                # variables passed to the server to configure the assimilator
                                                # further (see additional_server_env parameter)

        cluster=LocalCluster(),                 # on which cluster to execute, LocalClsuter will run on localhos
                                                # default: empty. it will try to select the cluster automatically

        procs_server=2,                         # server parallelism
        procs_runner=3,                         # model parallelism
        n_runners=2)                            # how many runners
```

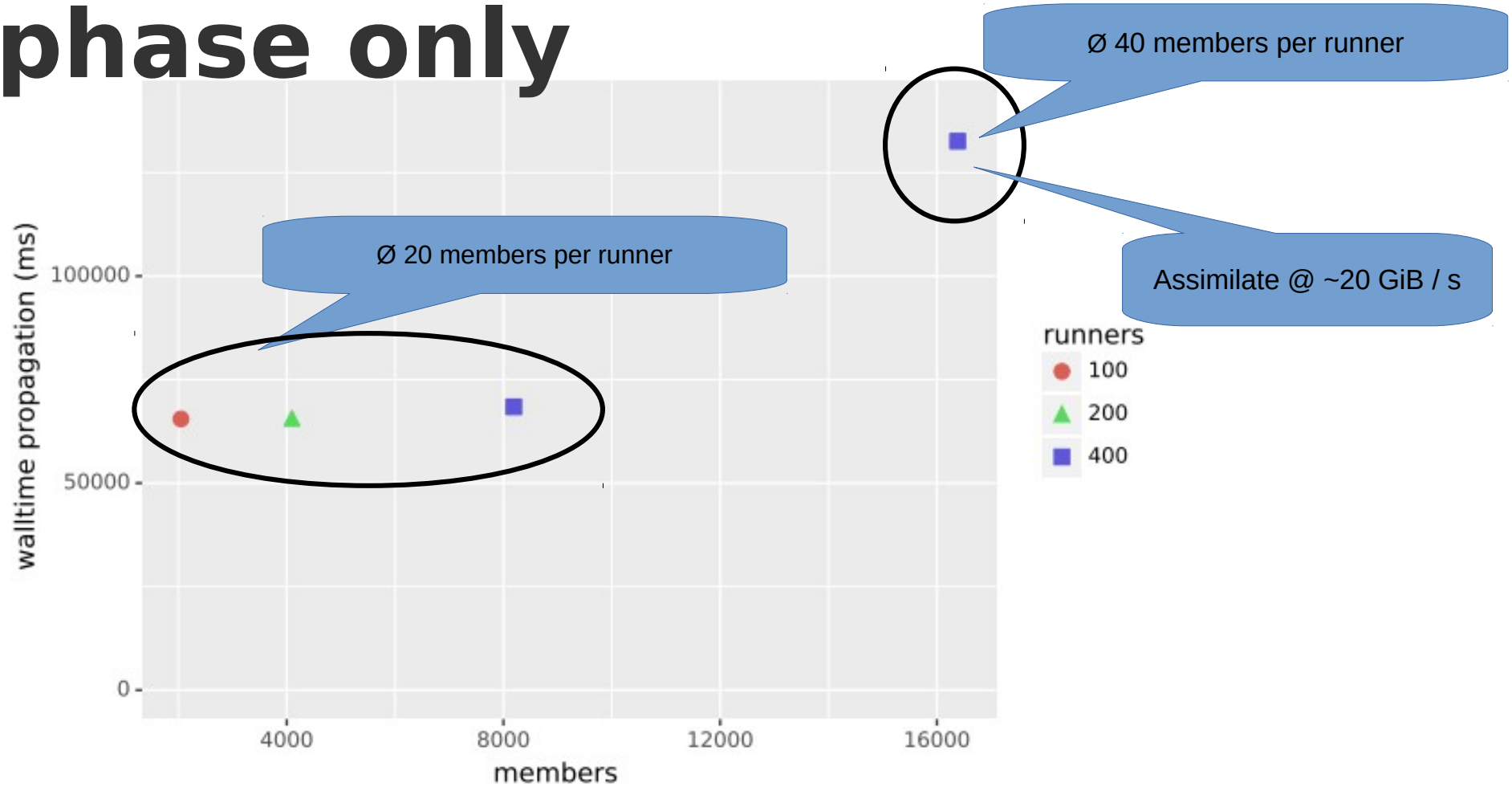
Scaling – Propagation phase only



ParFlow: Neckar catchment (**800 m horizontal resolution**, changing vertical resolution, **242 km * 214 km * 100 m, 15 min time steps**, **assimilation every 12 hours**) → Assimilating **25 obs** into **4 M grid cells**, model parallelized on **40-48 cores**,

Note: scaling efficiency of the **full cycle is 0-3% worse** since update step is fast (0.1 s / 1 s)

Scaling – Propagation phase only



16384 members: 2.9 TiB of Data transfer between runners and server per assimilation cycle

Open Challenges:

- EnKF does not scale

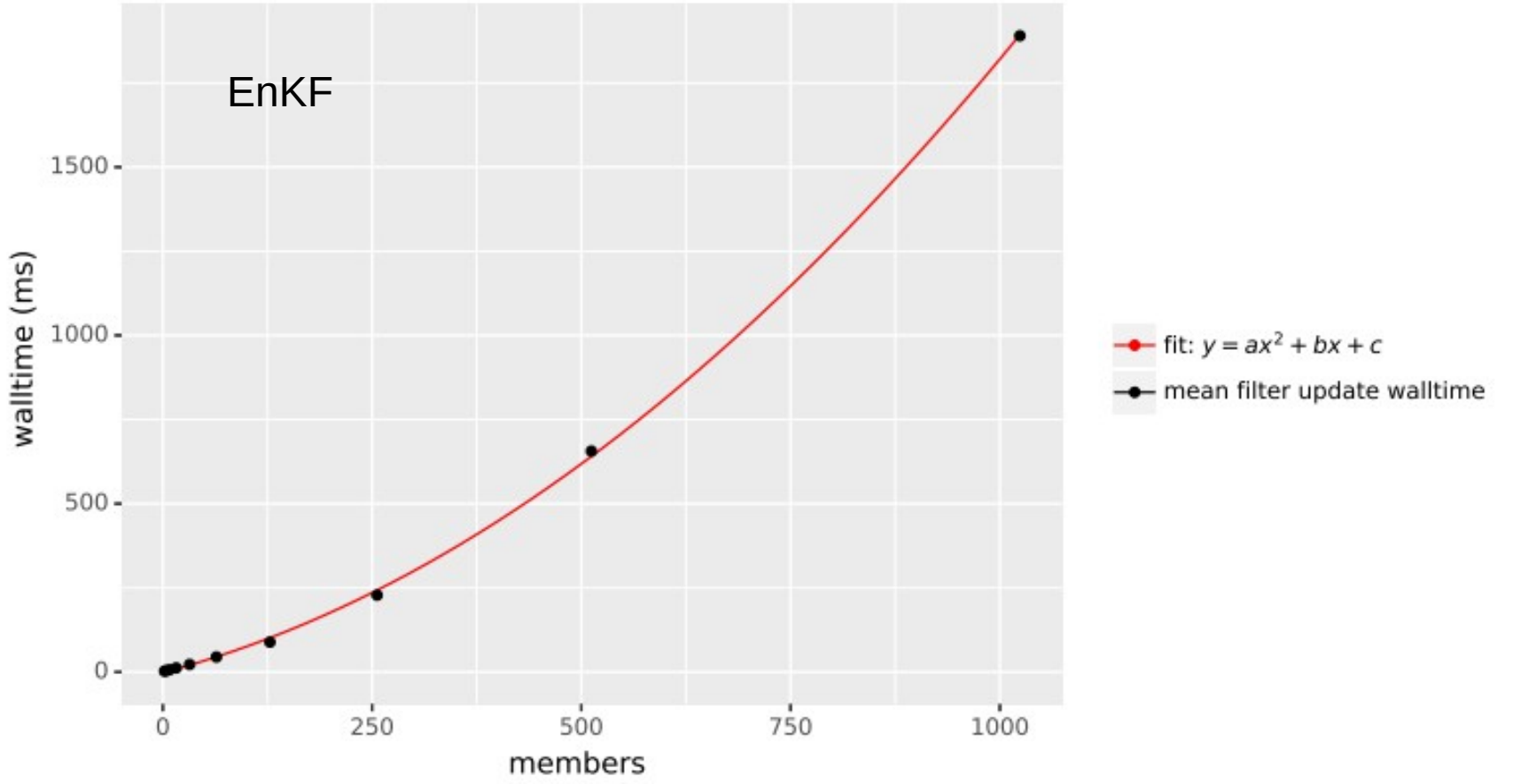
  for large ensembles

  - use spatially distributed EnKF here
  - 16k case scaling efficiency counting full assimilation cycle: 58 % only
- High RAM consumption

Members	2,048	4,096	8,192	16,384
Amount of runners	100	200	400	400
Average members per runner	20.48	20.48	20.48	40.96
Server cores	240	240	240	240
Server nodes	6	12	24	48
Update phase walltime (ms)	5,339	11,872	28,368	52,893
Propagation phase walltime (ms)	41,555	41,757	42,295	83,898
Scaling efficiency during propagation phase (reference: walltime 1 runner propagation phase, not counting server nodes) (%)	96.03	95.77	96.2	95.76
Scaling efficiency during assimilation cycle (reference: walltime 1 runner, not counting server nodes) (%)	85.1	74.57	57.58	58.74

Open Challenges:

- EnKF



16,384
400
40.96
240
48
52,893
83,898
95.76
58.74

- 288 Observations assimilated into 4M degrees of freedom on 144 JUWELS cores

Mitigation and future work

- Use faster (e.g., LEnKF) and less RAM hungry filter update method
- Iterative EnKF calculation starting during propagation phase already?
- Revisit EnKF parallelization

- Store member states differently (runner local RAM, NVRAM…)
- Run the server on runner nodes. Load balancing will help.

- Iterative surrogate learning, adaptive ensemble size...

Conclusion

- Framework where member propagation can be on any runner
- Based on multiple components
- Communication between components uses dynamic connections
- This allows:
  - Avoid file I/O
  - Fast (avoid startup of new models per propagation)
  - Load balance different propagation tasks
  - Resilient
  - Easy to deploy (different degrees of parallelism, instrument any model, interchangeable Assimilation Update phase backends
- Paper: “An elastic framework for ensemble-based large-scale data assimilation“ (hal: https://hal.archives-ouvertes.fr/hal-03017033v2)
- Current work: Melissa-DA tuned for particle filters with light weight server

References for the Slides

- Fowler, Alison. “The Ensemble Kalman Filter,” Lecture notes, 20.
- Dorier, Matthieu, Matthieu Dreher, Tom Peterka, Gabriel Antoniu, Bruno Raffin, and Justin M. Wozniak. “Lessons Learned from Building In Situ Coupling Frameworks,” 2015. https://doi.org/10.1145/2828612.2828622.
- Nerger, L., W. Hiller, and J. Schröter. “PDAF - THE PARALLEL DATA ASSIMILATION FRAMEWORK: EXPERIENCES WITH KALMAN FILTERING.” In Use of High Performance Computing in Meteorology, 63–83. Reading, UK: WORLD SCIENTIFIC, 2005. https://doi.org/10.1142/9789812701831_0006.
- Bautista-Gomez, Leonardo, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. “FTI: High Performance Fault Tolerance Interface for Hybrid Systems.” In SC ’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 1–12, 2011. https://doi.org/10.1145/2063384.2063427.
- Théophile Terraz, Personal Communication.
- Bruno Raffin, Personal Communication.
- Terraz, Théophile, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin. “Melissa: Large Scale In Transit Sensitivity Analysis Avoiding Intermediate Files,” 1–14, 2017. https://hal.inria.fr/hal-01607479/document.
- Petrie, Ruth Elizabeth. “Localization in the Ensemble Kalman Filter,” n.d., 80.
- https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/JUWELS_node.html, retrieved the 03.09.20.
- http://www.idris.fr/jean-zay/jean-zay-presentation.html, retrieved the 03.09.20.
- Image by University of the Fraser Valley - https://www.flickr.com/photos/ufv/14698165796/, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=54110312
- Anderson, Jeffrey, Tim Hoar, Kevin Raeder, Hui Liu, Nancy Collins, Ryan Torn, and Avelino Avellano. “The Data Assimilation Research Testbed: A Community Facility.” Bulletin of the American Meteorological Society 90, no. 9 (September 1, 2009): 1283–96. https://doi.org/10.1175/2009BAMS2618.1.