An elastic framework for ensemble-based large-scale data assimilation

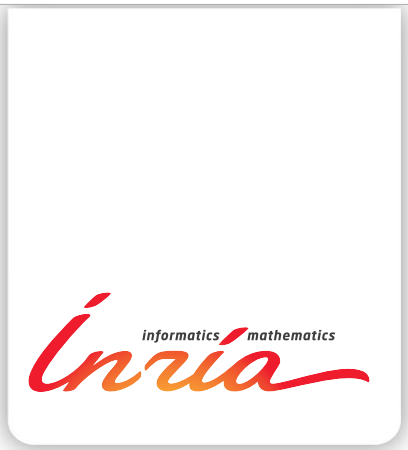
Sebastian Friedemann, Bruno Raffin

An elastic framework for ensemble-based large-scale data assimilation

Sebastian Friedemann, Bruno Raffin

RESEARCH REPORT
N° 9377
November 2020
Project-Team DataMove

ISSN 0249-6399 ISRN INRIA/RR--9377--FR+ENG



An elastic framework for ensemble-based large-scale data assimilation

Sebastian Friedemann, Bruno Raffin

Project-Team DataMove

Research Report n° 9377 — November 2020 — 37 pages

Abstract: Prediction of chaotic systems relies on a floating fusion of sensor data (observations) with a numerical model to decide on a good system trajectory and to compensate non-linear feedback effects. Ensemble-based data assimilation (DA) is a major method for this concern depending on propagating an ensemble of perturbed model realizations.
In this paper we develop an elastic, online, fault-tolerant and modular framework called Melissa-DA for large-scale ensemble-based DA. Melissa-DA allows elastic addition or removal of compute resources for state propagation at runtime. Dynamic load balancing based on list scheduling ensures efficient execution. Online processing of the data produced by ensemble members enables to avoid the I/O bottleneck of file-based approaches. Our implementation embeds the PDAF parallel DA engine, enabling the use of various DA methods. Melissa-DA can support extra ensemble-based DA methods by implementing the transformation of member background states into analysis states.
Experiments confirm the excellent scalability of Melissa-DA, propagating 16,384 members for a regional hydrological critical zone assimilation relying on the ParFlow model on a domain with about 4 M grid cells. The same use case was ported to the PDAF state-of-the-art DA framework relying on a MPI approach. A comparison with Melissa-DA at 2,500 members on 20,000 cores shows our approach is about 50% faster per assimilation cycle.

Key-words: Data Assimilation, Ensemble Kalman Filter, Ensemble, Multi Run Simulations, Elastic, Fault Tolerant, Online, In Transit Processing, Master/Worker

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

An elastic framework for ensemble-based large-scale data assimilation

1 Introduction

Numerical models of highly non-linear systems (e.g., the atmospheric, the oceanic or the ground water flow) are extremely sensitive to input variations. The goal of data assimilation (DA) is to reduce the result uncertainty by correcting the model trajectory using observation data. Observation data can be very diverse, for instance in geoscience it typically consists of a mix of values from remote sensing instruments (satellites) and in situ observations by ground based observatories, buoys, aircrafts etc. DA intends to estimate the real state of the system as accurately as possible merging model intermediate results and observations. This is critical in particular to models showing chaotic or semichaotic behavior, where small changes in the model input can lead to large changes in the model output.

Two main approaches are used for DA, variational and statistical [3, 26]. In this paper we focus on statistical ensemble-based DA, where an ensemble of several model instances (members) is executed to estimate the model error against the observation error. Various methods exist for that purpose like the classical Ensemble Kalman Filter (EnKF) we use for experiments in this paper.

Combining large numerical models and ensemble-based DA requires execution on supercomputers. Recent models have millions of degrees of freedom, but only hundreds of ensemble members are used for operational DA. Thus today's approaches suffer from undersampling: to minimize and to estimate the sampling error, much larger ensembles with thousands of members would be necessary. Current approaches rely either on files to aggregate the ensemble results and send back corrected states or on the MPI message passing programming paradigm to harness the members and the assimilation process in a large monolithic code. But these approaches are not well-suited to running ultra-large ensembles on the coming exascale computers. Efficient execution at exascale requires mechanisms to allow dynamic adaptation to the context, including load balancing for limiting idle time, elasticity to adapt to the machine availability, fault tolerance to recover from failures caused by numerical, software or hardware issues and direct communications between components instead of files to bypass the I/O performance bottleneck.

This paper addresses these issues and proposes a novel architecture, called Melissa-DA, which avoids intermediate files, supports fault tolerance, elasticity, and lead to efficient executions even at large scale. Experiments using the ParFlow parallel hydrology solver on a domain with about 4 M grid cells as numerical model were run with up to 16,384 members on 16,240 cores. Each member itself was parallelized on up to 48 MPI ranks. Melissa-DA adopts a three-tier architecture based on a launcher in charge of orchestrating the full application execution, independent parallel runners in charge of executing

members up to the next assimilation update phase, and a parallel server that gathers and updates member states to assimilate observation data into them. The benefits of this framework include:

- **Elasticity:** Melissa-DA enables the dynamic adaptation of compute resource usage according to availability. Runners are independent and connect dynamically to the parallel server when they start. They are submitted as independent jobs to the batch scheduler. Thus, the number of concurrently running runners can vary during the course of a study to adapt to the availability of compute resources.
- **Fault tolerance:** Melissa-DA's asynchronous master/worker architecture supports a simple yet robust fault tolerance mechanism. Only some lightweight book-keeping and a few heartbeats as well as checkpointing on the server side are required to detect issues and restart the server or the runners.
- **Load balancing:** The distribution of member states to runners is controlled by the server and defined dynamically according to a list scheduling algorithm, enabling to adjust the load of each runner according to the time required to propagate each member.
- **Online processing:** Exchange of state variables between the different parts of the Melissa-DA application and the different parts of an assimilation cycle happens fully online avoiding file system access and its latency.
- **Communication/computation overlap:** Communications between the server and the runners occur asynchronously, in parallel with computation, enabling an effective overlapping of computation and communication, improving the overall execution efficiency.
- **Code modularity:** Melissa-DA enforces code modularity by design, leading to a clear separation of concern between models and DA. The runners execute the members, i.e., model instances, while the server, a separate code running in a different job, is concerned with observation processing and running the core DA algorithm.

After a reminder about statistical data assimilation and especially the Ensemble Kalman Filter (Section 2), related work is presented (Section 3). The proposed architecture and the Melissa-DA framework are detailed (Section 4) before analyzing experimental results (Section 5). We end with a conclusion (Section 6).

2 Statistical data assimilation and the ensemble Kalman filter

We provide in this section the base concepts of ensemble-based DA and EnKF. This simplified view is meant to introduce the vocabulary used throughout this

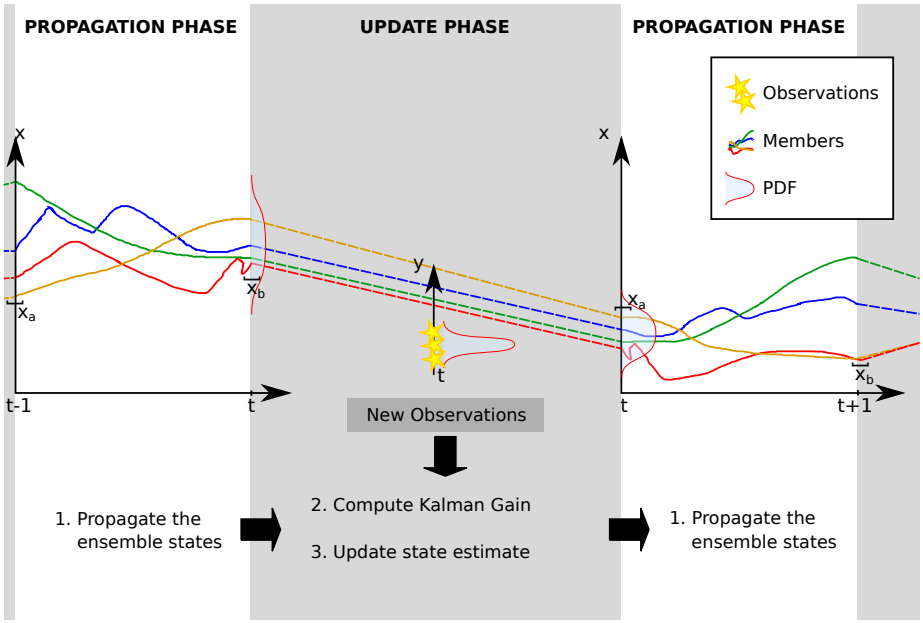


Figure 1: The ensemble Kalman filter workflow repeats assimilation cycles composed by a propagation and update phase.

paper. Refer to [3] and [15] for a more detailed introduction to DA techniques and algorithms.

The goal of DA is to estimate the real state of a physical system as accurately as possible merging model intermediate results and observations. DA acts by correcting periodically the model trajectory taking into account observations, hindering exponential growth of input errors. The model state, subject to correction, can represent the atmospheric state in Numerical Weather Prediction (NWP) or the soil moisture field when calibrating hydrological ground water models. Observations in the geoscientific domain typically contain a mix of values from remote sensing instruments (satellites) and in situ observations by ground based observatories, buoys, aircrafts etc.

Numerical models operate on the system state $x \in X \subseteq \mathbb{R}^N$ that cannot be directly observed. In the standard DA formalism, the model operator $\mathcal{M}$ fulfills the Markov property, taking the present state x_t as its *only* input to produce the next state x_{t+1}. Let us first consider the very simple case of a model $\mathcal{M}$, i.e., a function, that, from an input state x_t, computes a new state x_{t+1}, also called the *background state* in DA. The background state x_{t+1} represents a new state at a point further in time $(t+1)$:

$$x_{t+1} = \mathcal{M}(x_t). \tag{1}$$

The calculation of such background states happens during the *propagation phase*.

The output state x_{t+1} is distorted by numerical errors, initial error on x_t, intrinsic error due to the model itself, etc. Following the standard DA notations, we now remove the subscript $t+1$ and replace it with b referring to the

background state. Let us assume the sum of these errors is GAUSSIAN, unbiased (zero average), and with a variance σ_b^2, an assumption Kalman filters are based on. Let us also consider that time $t+1$ corresponds to a point in time where *observation data* is available, i.e., a state x_o provided by an other source, typically a scientific measurement instrument. The observation data also comes with some error that is also assumed GAUSSIAN, unbiased and with variance σ_o^2. We assume this variance estimate to be provided with the observation data.

The DA process consists in computing a corrected state, called the *analysis state* x_a, from the background sate x_b and the observation data x_o. The correction is obtained by finding the state x_a that minimizes the global error associated with the linear combination of x_b and x_o:

$$x_a = \alpha x_b + (1-\alpha)x_o \tag{2}$$

As both the background and observation errors are unbiased, x_a is obtained by computing the α^* value that minimizes the variance of x_a. Computing the analysis state x_a is called the *update phase*. One *propagation phase* followed by one *update (or analysis) phase* makes for one *assimilation cycle*. For the next propagation phase, the model computes the state at time $t+2$ from the analysis state x_a, and not from the background state x_b (or x_{t+1}).

To compute the analysis state, we need an estimate of the variance σ_b^2. This can be provided by the model directly, for instance if this is a solver based on a stochastic PDE. An other approach that works with any model, consists in propagating an ensemble of states to compute an estimate. A sample of M member states $(x_t^i | i \in [M])$ is propagated independently to get M background states $(\mathcal{M}(x_b^i) | i \in [M])$ used to compute σ_b^2.

Notice that here for simplicity we consider that observations are available at each timestep. If not, the model just needs to iterate up to reaching the time when the next observations are available.

As long as the state is a single scalar, this problem has a simple analytical solution. When considering multi-dimensional states the solution is more complex. First, often, the model state and the observation state are not anymore corresponding one-to-one. The observations are not necessarily of the same dimension, aligned with the model state or even of the same nature, so an observation operator $\mathcal{H}$ is required to project a model state, now a vector, into the observation space:

$$\tilde{y} = \mathcal{H}(x). \tag{3}$$

This is usually done this way as observation data are often of lower dimension than the model state. The analysis state vector becomes:

$$x_a = x_b + K(y - \mathcal{H}(x_b)), \tag{4}$$

where K is the gain matrix, somehow the multidimensional equivalent of the optimal α coefficient of Equation (2). The *Ensemble Kalman Filter* (EnKF) approach, the most studied statistical ensemble-based DA method, computes the optimal gain matrix, called the *Kalman Gain Matrix* as:

$$K = (P_b \mathcal{H}^T)[(\mathcal{H} P_b \mathcal{H}^T) + R_o]^{-1} \tag{5}$$

where P_b and R_o are the covariance matrices related to the background and observation states. As the variance before, the covariance matrix P_b is estimated from the sample of M members of the ensemble run. This is the method we rely on to validate the Melissa-DA approach. The associated workflow is common to many other ensembe-based DA filters. The EnKF inherits the Kalman Filter [21], extending it to non-linear models [3]. To summarize, the classical steps of one EnKF assimilation cycle are the following (Figure 1):

1. An ensemble of M states (*members*) $(x_a^i | i \in [M])$, statistically representing the assimilated state, is propagated by the model $\mathcal{M}$. The obtained states are the background states $(x_b^i | i \in [M])$. For the first assimilation cycle the states are initialised from an ensemble of perturbed states. Later they are the analysis states resulting from the previous assimilation cycle.

2. The Kalman gain K is calculated from the ensemble covariance P_b and the observation error R_o (Equation (5)).

3. Then for each state i, multiply the Kalman gain K with the *innovation* $\left(y - \mathcal{H}(x_b^i)\right)$ and add to the background states: $x_a^i = x_b^i + K \cdot \left(y - \mathcal{H}(x_b^i)\right)$ to obtain the new ensemble analysis states $(x_a^i | i \in [M])$.

4. Start over with the next assimilation cycle (step 1).

Notice that in the following, the background and analysis states can be named member, model or system states depending on the context.

3 Related work

DA techniques fall into two main categories, variational and statistical. Variational DA (e.g., 3D-Var and 4D-Var) relies on the minimization of a cost function evaluating the difference between the model state and the observations. Minimizing the cost function is done via gradient descent that requires adjoints for both the model and the observation operator. This approach is compute efficient, but the adjoints are not always available or may require significant efforts not always accessible. Nowadays large-scale DA applications as, e.g., used by Numerical Weather Prediction (NWP) operators typically rely on variational DA. For instance, the China Meteorological Administration uses 4D-Var to assimilate about 2.1 million daily observations into a global weather model with more than 7.7 million grid cells. As in 2019, the DA itself is parallelized on up to 1,024 processes [46].

Statistical DA takes a different approach relying on Bayesian statistics [3, 15]. An important subcategory is Ensemble based DA using an ensemble run of the model to compute an estimator of some statistical operators (co-variance matrix for EnKF, PDF for particle filters, etc.). The ensemble based approach consumes more compute power as the number of members needs to be large

enough for the estimator to be relevant, but stands for its simplicity as it only requires the model operator without adjoint as for the variational approach. But scaling the ensemble size can be challenging especially when the model is already time consuming and requires its own internal parallelization.

Two main approaches are used, identified as *file-based* (offline) and *online*. For the file-based approach, the ensemble member's simulations, i.e., the member state propagations, are executed independently after loading each member's input from the file system. The resulting background states are saved back to files. Once all members executed for that assimilation cycle, an analysis code runs to load the observations and the states from the different members to produce the analysis states, also saved to files. The members can be started again from these states for the next assimilation cycle. This file-based approach is usually simple to setup, elastic and fault tolerant. All the files act as checkpoints isolating the impact of a failing component and making a rerun easy. The number of concurrent simulations can vary from cycle to cycle depending on the supercomputer availability, providing elasticity at the granularity of a member. This approach is adopted by the EnTK (Ensemble Toolkit) framework [7], used to manage up to 4,096 members for DA on a molecular dynamics application [6] and by [40] assimilating oceanic conditions in the Red sea with O(1,000) members. A file-based approach may be the only option available when the full-supercomputer is required to propagate just a fraction of members as in [42]. The OpenDA framework also supports this file-based model, relying on NetCDF files for data exchange for the NEMO ocean model in [41]. However relying on the machine I/O capabilities using files is a growing performance bottleneck. In 10 years the compute power made a leap by a 134× factor, from 1.5 PFlop/s peak on Roadrunner (TOP500 #1 in 2008) to 201 PFlop/s peak on Summit (#1 in 2018), the I/O throughput for the same machines only increased by a 12× factor, from 204 GB/s to 2,500 GB/s. This trend is expected to continue at exascale. For instance the announced Frontier machine (2021) expected to reach 1 ExaFlop/s should offer 5 to 10 times the compute power of Summit but only 2 to 4 times its I/O throughput[1].

Existing online approaches build one single application that encompasses the different simulation instances for the propagation and update phases. In that case, states stay in memory and no intermediate file is needed. This fixes the I/O issue, but leads to a large monolithic application. If implemented with classical message passing (MPI), load balancing, elasticity and fault tolerance become challenging. They are not directly supported by MPI and need to be implemented explicitly. For instance if one process in the monolithic application fails, the full application stops. Changing the number of CPUs used during execution to enable some elasticity is not supported by MPI so far neither. The full amount of resources needed must be allocated upfront and for the full duration of the execution. Ensemble runs are particularly sensible to hardware and numerical faults. As the ensemble size increases, the probability that the model may execute in numerical domains it has not been well tested will get more im-

[1] see https://www.olcf.ornl.gov/frontier/, retrieved the 03.11.2020

portant too. The frameworks DART (Data Assimilation Research Testbed) [2] and PDAF (Parallel Data Assimilation Framework) [32] are based on this approach. PDAF for instance has been used for the regional earth system model TerrSysMP using EnKF with up to 256 members [24]. Other DA work rely on custom MPI codes. [29] ran 10,240 members with the LEnKF (localized Ensemble Kalman filter) on the K-computer in 2014. In 2018, Berndt et al. assimilated up to 4,096 members with 262,144 processors and a particle filter. They also did a production use case with 1,024 members for wind power prediction over Europe [9]. Variational and statistical DA can also be combined. NWP actors like the ECMWF (European Centre for Medium-Range Weather Forecasts), the British Met office (Meteorological Office) and the Canadian Meteorological Centre rely on such approaches using hundreds of ensemble members to improve predictions [11, 13, 18].

Looking beyond DA frameworks, various Python based frameworks are supporting the automatic distribution of tasks, enabling to manage ensemble runs. Dask [36] is for instance used for hyperparameter search with Scikit-Learn, Ray [30] for reinforcement learning [25]. But they do not support tasks that are built from legacy MPI parallel codes. Other frameworks such as Radical-Pilot (the base framework of EnTK) [34] or Parsl [5] enable such features but are using files for data exchange. Other domain specific frameworks like Dakota [14], Melissa [39] or Copernicus [35] enable more direct support of parallel simulation codes but for patterns that require less synchronizations than the ones required for DA.

To our knowledge, map-reduce tools like Spark [44] and Flink [1] have not been used for DA. Flink has been used for analyzing online the data of a parallel molecular dynamics simulations relying on its processing capabilities [45]. But performance was significantly below the one achieved with MPI based in situ processing tools. Extending such approach to DA would require to support an ensemble of simulations and efficient linear algebra operations on large matrices as needed for computing the Kalman gain in EnKF for instance.

4 Melissa-DA architecture

4.1 Overview

Melissa-DA takes a third way, compared to the online and file-based approaches presented in Section 3, keeping the flexibility of the file-based approach while avoiding intermediate files to bypass the I/O bottleneck (Figure 2). We present below a global overview motivating the Melissa-DA design choices that are then detailed in the next sections:

- **Client/Server model.** Melissa-DA relies on the standard client/server model extended to the parallel case where both the client and server are parallel codes with potentially different levels of parallelism. No intermediate files are required as all data exchanges occur through direct memory

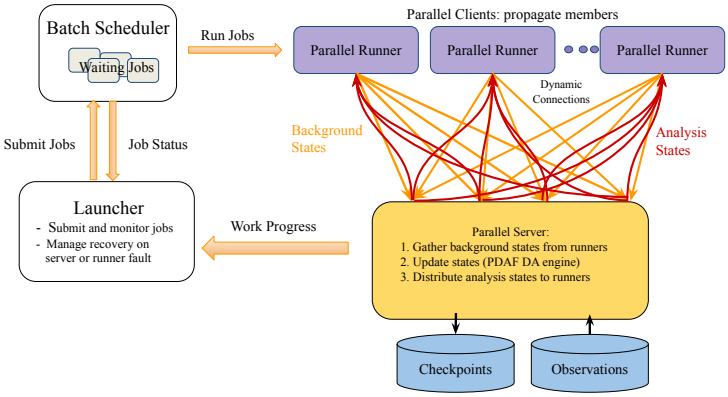


Figure 2: Melissa-DA three-tier architecture. The launcher supervises the execution in tight link with the batch scheduler. The job scheduler regulates the number of simulation jobs (runners) to run according to the machine availability, leading to an elastic resource usage. The server distributes the members to propagate to the connected runners dynamically for balancing their workload. A fault tolerance mechanism automatically restarts failing runners or a failing server.

to memory communications between the server and the clients. Additionally, the client/server pattern makes the application more modular. In Melissa-DA, a client runs the simulation code for the propagation phase and the server the code for the update phase. Because the connection between a client and the server is dynamic, a client can be stopped (voluntarily or not) and started anytime. A client failure does not lead the server to fail, thus providing a sound base to support an efficient fault tolerance protocol. The number of running clients can evolve over time depending on the resources available on the supercomputer, making the application elastic.

- **Clients are runners.** A Melissa-DA client runs one simulation instance. Once done with the propagation of a given member, it sends the full member state to the server, and so, can start running a new member. The new member state to propagate is loaded from the server. Thus a client can propagate several members per propagation phase. We call such a client *runner*. There are several benefits from enabling such a feature:
  - No need to pay the full simulation starting cost for each member. Switching from one member to an other in a runner is done by switching member states in memory, not requiring a full restart of the simulation code.
  - The time to propagate one member can vary, leading to load balancing issues, and so inefficiency, as the update phase cannot start as long as not all members have been propagated. Dynamically dis-

tributing the members to the runners according to their workload enables to balance the workload between runners, improving the execution efficiency.

- **Server.** The server collects the states propagated by runners to compute the covariance and the Kalman gain matrices, and update each state to provide the new analysis states. The server is also in charge of implementing the load balancing strategy, distributing the propagation work to runners following a list scheduling algorithm.

- **Launcher.** A Launcher executable overviews the full workflow progress. The launcher orchestrates the execution, interacting with the supercomputer's batch scheduler to request resources to start new jobs holding runners or the server, kill such jobs, monitor their status, and trigger job restarts in case of failure. It makes a single point of entry for the user to parameterize, control and monitor the application.

- **Code modularity.** The runner, server and launcher are separate codes that interact with each other over dynamic network connections. This makes the application very modular. For instance changing the code of the server for switching to a different implementation of the update phase can be done without having to recompile the runner code. Turning an existing solver simulation code into a runner requires to instrument it with the Melissa-DA API, but introduces no dependency to the server code into the solver code.

Putting all pieces together (Figure 2), the launcher starts and monitors runners and the server. During the propagation phase, the server distributes member states to the runners one-by-one. Runners propagate these member states and send them back to the server as background states. The server then performs the assimilation of observations (update phase). This generates a new ensemble of member states (analysis states) used for propagation for the next assimilation cycle. We detail this workflow and its components in the following.

4.2 Server

The server is parallel (based on MPI) and runs on several nodes. The number of nodes required for the server is primarily defined by its memory needs. The amount of memory needed is in the order of the sum of the member's state sizes. Member states contain the minimal amount of information needed to restore a given member on any runner. Each member state vector is split into roughly equally sized parts, one per server rank (spatial distribution).

The server needs to be linked against a user defined function to initialize all the member states (Figure 3 `init_ensemble`). Other functions, e.g., to load the current assimilation cycle's observation data (`init_observations`) and the observation operator $\mathcal{H}$ (`observation_operator_H`) must be provided to the server for each DA study. In the current version user functions are called sequentially

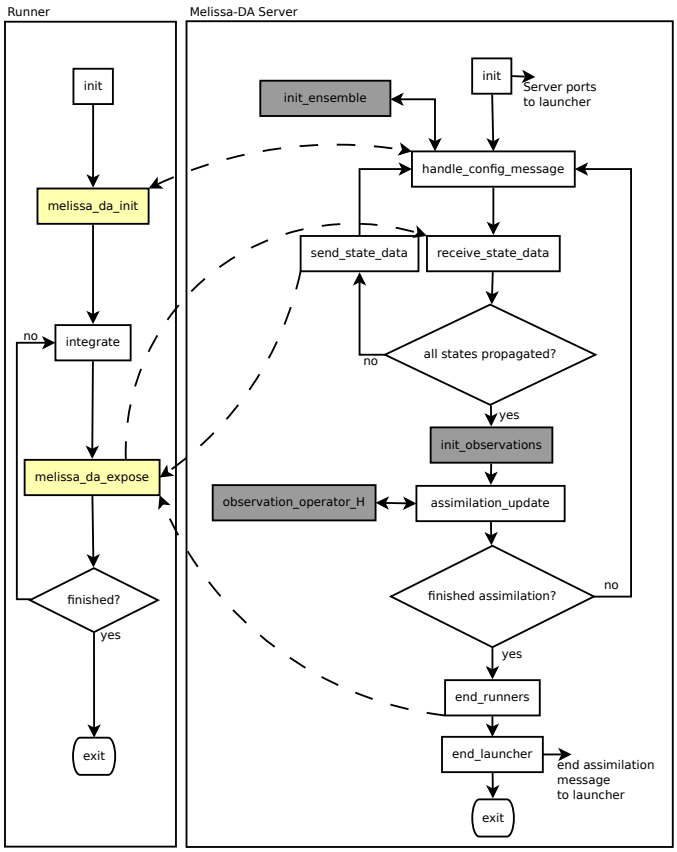


Figure 3: Melissa-DA runner and server interactions (fault tolerance part omitted for sake of clarity). Dashed arrows denote messages that are exchanged between different components. Grey boxes are methods that need to be implemented by the user. Yellow boxes are Melissa-DA API calls that need to be introduced in the simulation code to transform it into a runner.

by the server, but we expect to support concurrent calls to `init_ensemble` and `init_observations` to further improve the server performance.

The current server embeds PDAF as parallel assimilation engine. The server parallelization can be chosen independently from the runner parallelization. A N × M data redistribution takes place between each runner and the server to account for different levels of parallelism on the server and runner side. This redistribution scheme is implemented on top of ZeroMQ, an asynchronous networking library extending sockets [17]. ZeroMQ supports a server/client connection scheme allowing dynamic addition or removal of runners.

Care must be taken to coherently store each member's state vector parts. As runners are not synchronized, their state parts might not be received by all server ranks in the same order. For instance server rank 0 could receive a part of member 3's state vector while rank 1 receives a part of member 4's state (both members propagated by different runners). Even more importantly the state parts that are sent back must be synchronized so that the ranks of one runner receive the parts of the same member state vector from all the connected server ranks. For that purpose all received state parts are labeled with the member ID they belong to, enabling the server to assemble coherently distributed member states. State propagation is ensured by the server rank 0, the only one making decisions on which runner shall propagate which member state. This decision is next shared amongst all the server ranks using non blocking MPI broadcasts. This way communication between different server and runner ranks overlaps while other runner (-ranks) perform unhindered model integration.

4.3 Runners

Melissa-DA runners are based on the simulation code, instrumented using the minimalist Melissa-DA API. This API consists only of two functions: `melissa_init` and `melissa_expose` (Figure 3). `melissa_init` must be called once at the beginning to define the size of the member state per simulation rank. This information is then exchanged with the server, retrieving the server parallelization level. Next `melissa_init` opens all necessary connections to the different server ranks.

`melissa_expose` needs to be inserted into the simulation code to enable extraction of the member state held by the runner and to communicate it with the server. When called, this function is given a pointer to the runner's state data in memory that is sent to the server who saves it as background state. Next `melissa_expose` waits to receive from the server an analysis state that replaces the background state in RAM. Now the runner is ready to start propagating the next member state. The function `melissa_expose` returns the number of timesteps the received analysis state shall be propagated or a stop signal. Please note that only the part of the model state that changes from timestep to timestep needs to be exposed to Melissa-DA. Variables that are invariant between members and timesteps (such as domain decomposition or constant boundary conditions) do not need to be exposed.

4.4 Launcher

To start a Melissa-DA application, a user starts the launcher that then takes care of setting up the server and runners on the supercomputer.

The launcher typically runs on the supercomputer front node and is the only part of the Melissa-DA application that interacts with the machine batch scheduler. The launcher requests resources for starting the server job and as soon as the server is up it submits jobs for runners. Hereby it prioritizes job submission within the same job allocation (if the launcher itself was started within such an allocation). Otherwise the launcher can also submit jobs as self contained allocations (by, e.g., calling `srun` outside of any allocation on Slurm [43] based supercomputers). In the latter case it can happen that the server job and some runner jobs are not executed at the same time leading to inefficient small Melissa-DA runs. Thus ideally at least jobs for the server and some runners are launched within the same allocation ensuring the Melissa-DA application to operate efficiently even if no further runner jobs can be submitted. It is also possible to instruct the launcher to start jobs within different partitions.

If the launcher detects that too few runners are up, it requests new ones, or, once notified by the server that the assimilation finished, it deletes all pending jobs and stops the full application. The launcher also periodically checks that the server is up, restarting it from the last checkpoint if necessary. The notification system between the server and the launcher is based on ZeroMQ. There are no direct connections between runners and the launcher. The launcher only observes the batch scheduler information on runner jobs.

4.5 Fault tolerance

Melissa-DA supports detection and recovery from failures (including straggler issues) of the runners through timeouts, heartbeats and server checkpoints. Since the server stores the different members' states, no checkpointing is required on the runners. So Melissa-DA ensures fault recovery even if the model simulation code does not support checkpointing. If supported, runners can leverage simulation checkpointing to speed-up runner restart.

The server is checkpointed once during each assimilation cycle using FTI (fault tolerance interface, [8]). This enables the recovery from server crashes without user interaction. During the propagation phase checkpointing asynchronously saves received member state parts on arrival (using threaded background checkpointing). Checkpoints are finalized before each update phase begins. So checkpointing does not impair the server reactivity for its other tasks and does not consume server resources during the update phase when the server is performing work on the critical path of the assimilation cycle.

The server sends heartbeats to the launcher. If missing, the launcher kills the runner and server jobs and restarts automatically from the last server checkpoint. Thus in the case of a server crash, the application restarts from the last complete set of propagation states. In the worst case only the last update phase

and some member propagations of the current assimilation cycle that was not completely checkpointed yet need to be repeated.

The server is also in charge of tracking runner activity based on timeouts. If a runner is detected as failing, the server re-assigns the current runner's member propagation to an other active runner. More precisely, if one of the server ranks detects a timeout from a runner, it notifies the server rank 0 that reschedules this ensemble member to a different runner, informing all server ranks to discard information already received by the failing runner. Further, the server sends stop messages to all other ranks of the failing runner. The launcher, also notified of the failing runner, properly stops it and requests the batch scheduler to start a new runner that will connect to the server as soon as ready.

One difficulty are errors that cannot be solved by a restart, typically numerical errors or, e.g., a wrongly configured server job. To circumvent these cases Melissa-DA counts the number of restarts. If the maximum, a user defined value, is reached, Melissa-DA stops with an informative error message. In the case of a recurrent error on a given member state propagation, it is possible to avoid stopping the full application. One option is to automatically replace such members with new ones by calling a user defined function for generating new member states, possibly by perturbing existing ones. Alternatively when the maximum number of restarts for a member state propagation is reached, this member could simply be canceled. As the number of members is high, removing a small number of members usually does not impair the quality of the DA process. These solutions remain to be implemented in Melissa-DA.

A common fault are jobs being canceled by the batch scheduler once reaching the limit walltime. If this occurs at the server or runner level, the fault tolerance protocol operates.

The launcher is the single point of failure. Upon launcher failure the application needs to be restarted by the user.

4.6 Dynamic load balancing

As already mentioned, runners send each member state to the server. Having the full member states on the server brings an additional level of flexibility central to the Melissa-DA architecture: runners become agnostic of the members they propagate. We rely on this property for the dynamic load balancing mechanism of Melissa-DA.

Dynamic load balancing is a very desirable feature when the time to propagate the different members differ. This is typically the case with solvers relying on iterative methods, but also when runners are started on heterogeneous resources, for instance nodes with GPUs versus nodes without, or if the network topology impacts unevenly the data transfer time between the server and runners. The server has to wait for the last member to return its background state before being able to proceed with the update phase computing the analysis states. The worst case occurs when state propagation is fully parallel, i.e. when each runner is in charge of a single member. In that case runner idle time is the sum of the differences between each propagation time and the slowest

one. As we target large numbers of members, each member potentially being a large-scale parallel simulation, this can account for significant resource under utilization. To reduce this source of inefficiency Melissa-DA enables 1) to control the propagation concurrency level independently from the number of members and 2) to dynamically distribute members to runners.

The Melissa-DA load balancing strategy relies on the GRAHAM list scheduling algorithm [16]. The server distributes the members to runners on a first come first serve basis. Each time a runner becomes idle, the server provides it with the state of one member to propagate. This algorithm is simple to implement, has a very low operational cost, and does not require any information on the member propagation time. The performance of the list scheduling algorithm is guaranteed to be at worst twice the one of the optimal scheduling that requires to know the member execution time in advance (which is not the case here). More precisely the walltime T_{ls} (called makespan in the scheduling jargon) is bounded by the optimal walltime T_{opt}:

$$T_{\text{ls}} \leq T_{opt} * (2 - \frac{1}{m}) \tag{6}$$

where m is the number of machines used, in our case the number of runners [38]. This bound is tight, i.e., cannot be lowered, as there exist instances where this bound is actually met.

Static scheduling distributing evenly the members to runners at the beginning of each propagation phase, does not guarantee the same efficiency as long as we have no knowledge on the member propagation time. The worst case occurs if one runner gets the members with the longest propagation time.

Also the list scheduling algorithm is efficient independent of the number of runners, combining well with the Melissa-DA runner management strategy. While the number of expected runners can be statically defined by the user at start time, the actual number of executed runners depends on the machine availability and batch scheduler. Runners can start at different time periods, they may not all run due to resource limitations, some may crash and try to restart. With list scheduling, a runner gets from the server the next member to propagate as soon as connected and ready.

From this base algorithm several optimizations can be considered. In particular data movements could be reduced by trying to avoid centralizing all member states on the runner using decentralized extensions of list scheduling like work stealing [10]. This could be beneficial when only a small part of the member states are actually changed during the update phase. This is left as a future work.

4.7 Data Flow

Member states transit between runner and server memory directly. Transfers take place through parallel N × M communications, where each runner process exchanges data with the corresponding server processes. A single state is thus never aggregated in a single process memory at any time. The full states (differ

from member to member) are gathered on the server even though the EnKF algorithm only needs a sub part of each state for the assimilation update. Full states are transferred to the server to enable dynamic load balancing, a key component for improving execution efficiency, and to allow fault tolerance and elasticity. On the server the states are persisted asynchronously to storage using FTI (Section 4.5). The primary purpose is to enable restart from the last completed assimilation cycle on server crash. But as FTI can dump the state variables in HDF5 files, the checkpoints can also be used for post hoc analysis (often users request to keep intermediate states). Centralizing the full states to the server increases the memory needs, which can often be satisfied by increasing the number of server nodes. An other option would be to leverage FTI multi-level checkpointing capabilities to offload data not fitting in memory to the file system or some fast persistent memory like burst buffers or NVRAM when available.

4.8 Code

The code (server and API for model instrumentation) is written in C++, relying on features introduced with cpp14. This is especially handy regarding smart pointers to avoid memory leaks and having access to different containers (sets, lists, maps) used to store scheduling mappings. The assimilation update phase is contained in its own class deriving the `Assimilator`-interface, which accesses the received background member states and creates a new set of analysis member states to be propagated.

Implementing new ensemble-based DA methods within the Melissa-DA framework is straightforward, requiring to specify how to initialize the ensemble and how to transform the ensemble of background states in an ensemble of analysis states using observations.

A derived class calling the PDAF EnKF update phase methods was implemented and is linked against the user defined methods to inititalize the ensemble and observations and to apply the observation operator (Figure 3). From PDAF perspective, the Melissa-DA server acts as a parallel simulation code assembling a "flexible assimilation system" with one model instance propagating all ensemble members sequentially online [2]. By handling the server MPI communicator to PDAF, the update phase is parallelized on all server cores.

To let Melissa-DA support other assimilation algorithms implemented in PDAF (e.g., LEnKF, LETKF...[3]) only classes inheriting the `Assimilator` interface with calls to the desired PDAF filter update methods must be implemented.

The Melissa-DA launcher is written in Python. To execute a Melissa-DA study, a user typically executes a Python script for configuring the runs, importing the launcher module and launching the study.

[2] see `http://pdaf.awi.de/trac/wiki/ModifyModelforEnsembleIntegration`, retrieved the 25.08.2020

[3] for a complete list see `http://pdaf.awi.de/trac/wiki/FeaturesofPdaf`, retrieved the 25.08.2020

The Melissa-DA code base contains a test suite allowing end-to-end testing against results retrieved using PDAF only as reference implementation. The test suite also contains test cases validating recovery from induced runner and server faults.

The Melissa-DA code is available as open source at `https://gitlab.inria.fr/melissa/melissa-da`. Melissa-DA is part of the Melissa family which assembles frameworks related to large ensemble runs (`https://gitlab.inria.fr/melissa`).

5 Experimental study

Experiments in Section 5.6, Section 5.7 and Section 5.8 were performed on the Jean-Zay supercomputer on up to 500 of the 1,528 scalar compute nodes. Each node has 192 GB of memory and two Intel Cascade Lake processors with 40 cores at 2.5 GHz. The compute nodes are connected through an Omni-Path interconnection network with a bandwidth of 100 Gb/s. The other experiments ran on the JUWELS supercomputer (2 Intel Xeon processors, in total 48 cores at 2.7 GHz and 96 GB of memory per compute node, EDR-Infiniband (Connect-X4)) [20].

For all experiments we keep nearly the same problem size. Experiments assimilating ParFlow simulations leverage ≈ 92.4 MiB per member state containing spatially distributed data for the pressure, density and saturation variables (4,031,700 cells in double precision each). This represents the Neckar catchment in Germany. For our test we were provided observations from 25 ground water measuring sensors distributed over the whole catchment. Observation values were taken from a virtual reality simulation [37]. ParFlow runners are parallelized on one full node (40 processes for experiments on Jean-Zay and 48 processes for JUWELS respectively). For the experiments profiling the EnKF update phase (Section 5.4), a toy model from the PDAF examples, parallelized only on half of a node's cores, is used to save compute hours. For this experiment the member state size is smaller (4,032,000 grid cells, ≈ 30.8 MiB). Member initialization (`init_ensemble` function) uses an online approach relying on one initial system state adding some uniform random noise for each member. Loading terabytes of data for the initial ensemble states is thus avoided. To further avoid the influence from file system jitter, assimilation output to disk is deactivated if not stated differently.

For the sake of simplicity and minimal intrusion, the code was instrumented by calls to the STL-chrono library[4], allowing a precision of a nanosecond for these measurements. This minimal instrumentation was successfully validated against measurements using automatic Score-P instrumentation and Scalasca [22].

[4] see `https://en.cppreference.com/w/cpp/chrono`, retrieved the 24.06.2020

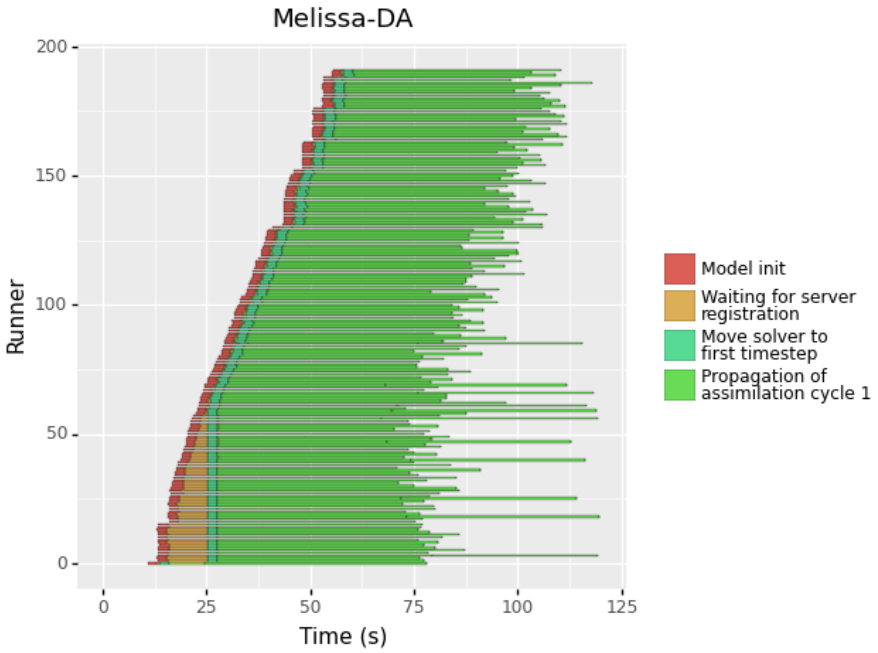


Figure 4: Traces of the startup of a Melissa-DA run with 1,000 members on 192 runners, Melissa-DA launcher starts at 0 s.

5.1 ParFlow

For first tests we assimilate ParFlow simulations with Melissa-DA. ParFlow is a physically based, fully coupled water transfer model for the critical zone that relies on an iterative Krylov-Newton solver [4, 19, 28, 23, 27]. This solver performs a changing amount of iterations until a defined convergence tolerance is reached at each timestep.

5.2 The first assimilation cycle

In the following experiments, for timing purpose we reserved all the nodes required for the run upfront (Slurm allocation), and then requested Slurm to start jobs into this allocation. The goal was to avoid introducing delays between job starts due to the machine load. For a production run the upfront resource reservation is not required, or can be done for the server and a few runners to ensure a minimal progress speed at start. However this upfront reservation is not sufficient to ensure a synchronous start (Figure 4) due to the combination of several factors:

- The launcher starts first the server and waits for a handshake before submitting runner jobs.
- The launcher submits runner jobs independently and sequentially.

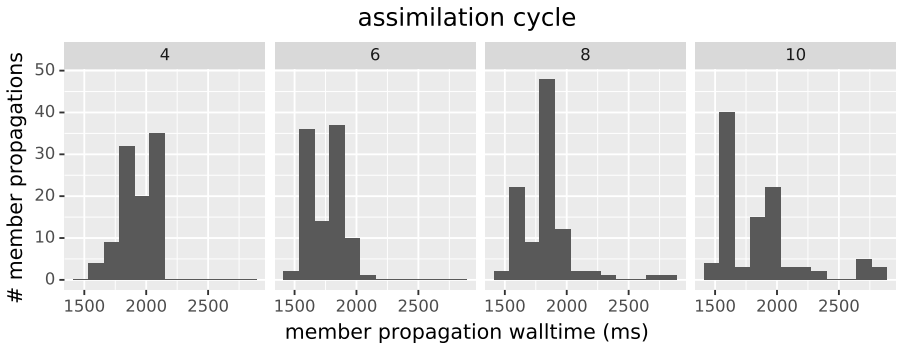


Figure 5: Histograms of propagation walltimes for 100 members during multiple assimilation cycles.

- The scheduler takes significant time - up to some seconds - to process each request and start each job.

- The server initializes its data structures holding the full state ensemble only after a first runner connection to get the full information on the N × M data redistribution scheme. Runners 0 to 60 (Figure 4) have to wait ("Waiting for server registration" in Figure 4) before they can receive propagation tasks from the server (at 25 s after launcher start).

- ParFlow takes about 14 times longer to converge for member propagations of the first assimilation cycle compared to member propagations of later assimilation cycles. This behavior of ParFlow is due to misfitting initial data challenging the solver. For the case displayed in Figure 4 each initial member propagation takes 43 s on average while later member propagations will take about 3 s on average.

These delays are amortized on long production runs, but not here as experiments ran for a few cycles only. So results presented here are based on time measures starting at the second cycle. Future work will try to improve the startup times relying on, e.g., advanced scheduler features to start multiple runners at once.

5.3 Ensemble propagation

Figure 5 shows the walltime distribution of 100 ParFlow member propagations for multiple assimilation cycles. Member propagation times starting from the second cycle compare Section 5.2 can vary significantly from about 1.5 s to 2.5 s, with an average at 1.9 s. The main cause for these fluctuations is the Krylov-Newton solver used by ParFlow that converges with a different number of iterations depending on the member state. As detailed in Section 4.6, these variations can impair the execution efficiency. Melissa-DA mitigates this effect by dynamically distributing members to runners following a list scheduling algorithm.

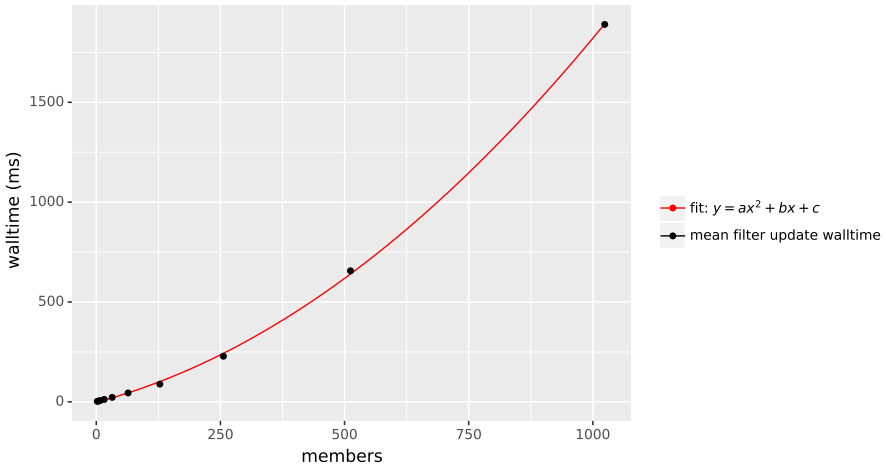


Figure 6: Assimilating 288 observations into about 4 M grid cells with up to 1,024 members on JUWELS. Mean over 25 update phase walltimes.

5.4 EnKF ensemble update

Figure 6 displays the evolution of the update phase walltime depending on the number of members, using Melissa-DA with the PDAF implementation of EnKF. The mean is computed over 25 assimilation cycles, assimilating 288 observations each time. Standard deviation is omitted as not being significant ($< 6\%$ of the update phase walltime). The EnKF update phase is executed on 3 JUWELS nodes (144 cores in total). The EnKF update phase relies on the calculation of covariance matrices over M samples resulting in a computational complexity for the EnKF update phase of $O(M^2)$, M being the number of ensemble members. This is confirmed by the experiments that fit a square function. The walltime of the update phase also depends on the number of observations. In the following experiments less observations are used, leading to an update phase of about only 1.1 seconds.

We also performed a strong scaling study (Figure 7) timing the update phase for 1,024 members and a varying number of server cores. The parallelization leads to walltime gains up to 576 cores. Computing the covariance matrix for the update phase is known to be difficult to efficiently parallelize. Techniques like localization enable to push the scalability limit. Localization is not used in this paper as we run with a limited number of observations. As Melissa-DA relies on PDAF which supports localization, localization can be easily activated by changing the API calls to PDAF in the Melissa-DA `Assimilator` interface. Refer to [32] and [31] for further EnKF/PDAF scaling experiments.

Dimensioning the Melissa-DA server optimally depends on the assimilated problem dimensions, the used assimilation algorithm and the target machine. It should be examined in a quick field study before moving to production. The results of such a field study can be seen in Figure 7, *bottom*. In the depicted case less core hours are consumed when not using a large number of server nodes

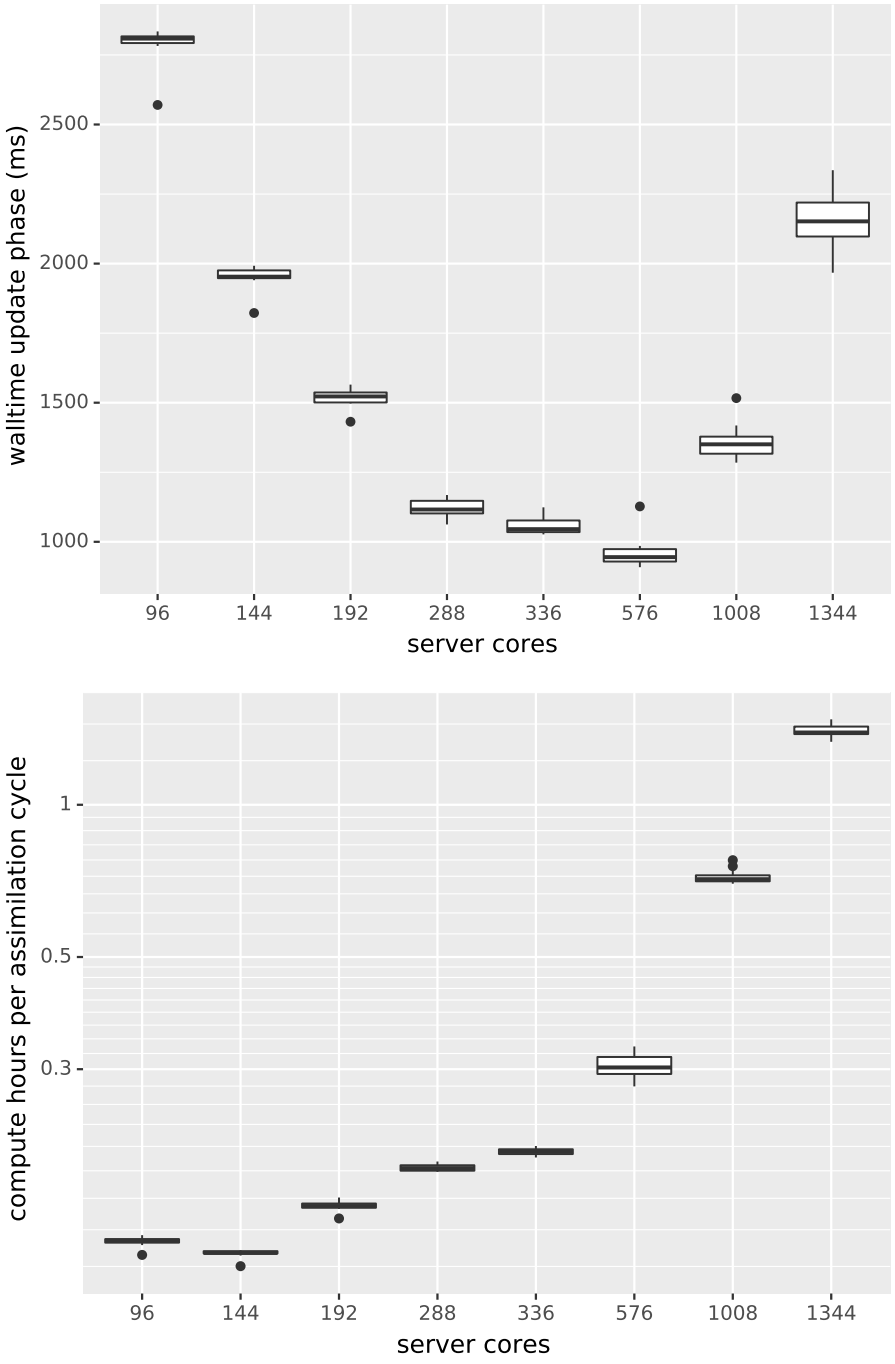


Figure 7: Boxplot of the update phase walltime (*top*) and the total compute hours per assimilation cycle (compromising update and propagation phase, *bottom*) when assimilating 288 observations into about 4 M grid cells with 1,024 members with a varying number of server cores on JUWELS. The latter graph can be used to evaluate server dimensioning.

(and cores respectively) since these idle during the whole propagation phase outplaying the walltime advantage they bring during the update phase. For the following experiments we assume that the update phase is short compared to the propagation phase leading to the policy: Use the least server nodes possible to fulfill memory requirements. If the server would be faster using a sub-part of the cores available per node, use less cores (e.g., only 24 or 12 cores per server node). Some supercomputers provide special large memory nodes that could be leveraged to run the Melissa-DA server.

5.5 Runner scaling

We now focus on the member per runner ratio. A single runner avoids idle runner time during the propagation phase, while having as many runners as members ensures the shortest propagation time but maximizes idle time. Idle time also comes from the switch between propagation and update phases: the server is mostly inactive during the propagation phase, while, in opposite runners are inactive during the update phase.

We experiment with a varying number of runners for a fixed number of members (100 and 1,024) and a given server configuration (Figure 8). Plotted values result from an average obtained from 8 executions, taking for each execution the time of the 2 last, over 3, assimilation cycles The efficiency of the propagation phase (Figure 8 top) is computed against the time obtained by running the members on a single runner. The compute hours are the total amount of consumed CPU resources (update and propagation phase, runners and server) during the assimilation cycles. For both plots, standard deviations are omitted as being small (relative standard deviations ($\frac{\text{standard deviation}}{\text{mean}}$) always smaller than 3%). The server was scaled to meet the memory needs. Each runner executed ParFlow on 48 cores (1 node).

Efficiency during the propagation phase stays beyond 90%, when each runner propagates at least 7 or 8 members, 95% for more than 10 members per runner and close to 100% for 50 or 100 members per runner. This demonstrates that Melissa-DA's load balancing algorithm maintains high efficiencies down to a relatively small number of members per runner. Obviously these levels of efficiency also depend on the distribution of propagation walltimes (see Section 5.3). Note that the update phase takes about 0.1 s and 1 s in the case of 100 and 1,024 members respectively, leading to scaling efficiencies for the full assimilation cycle decreased by at most 3%. Also the resources used for the server need to be considered. The total amount of compute hours (Figure 8 bottom) shows a U shape curve with a large flatten bottom at about 9 members per runner for the 100 members case and at about 20 for the 1,024 members case - a sufficiently large number of runners is required to amortize the server cost. Changing the runner amount around those sweet spots changes significantly the efficiency of the propagation phase but slightly impacts the total compute hours: the efficiency variation is compensated by the impact on the server idle time during the update phase that varies inversely (the server is mostly idle during this phase). This also shows that changing the number of runners in these areas

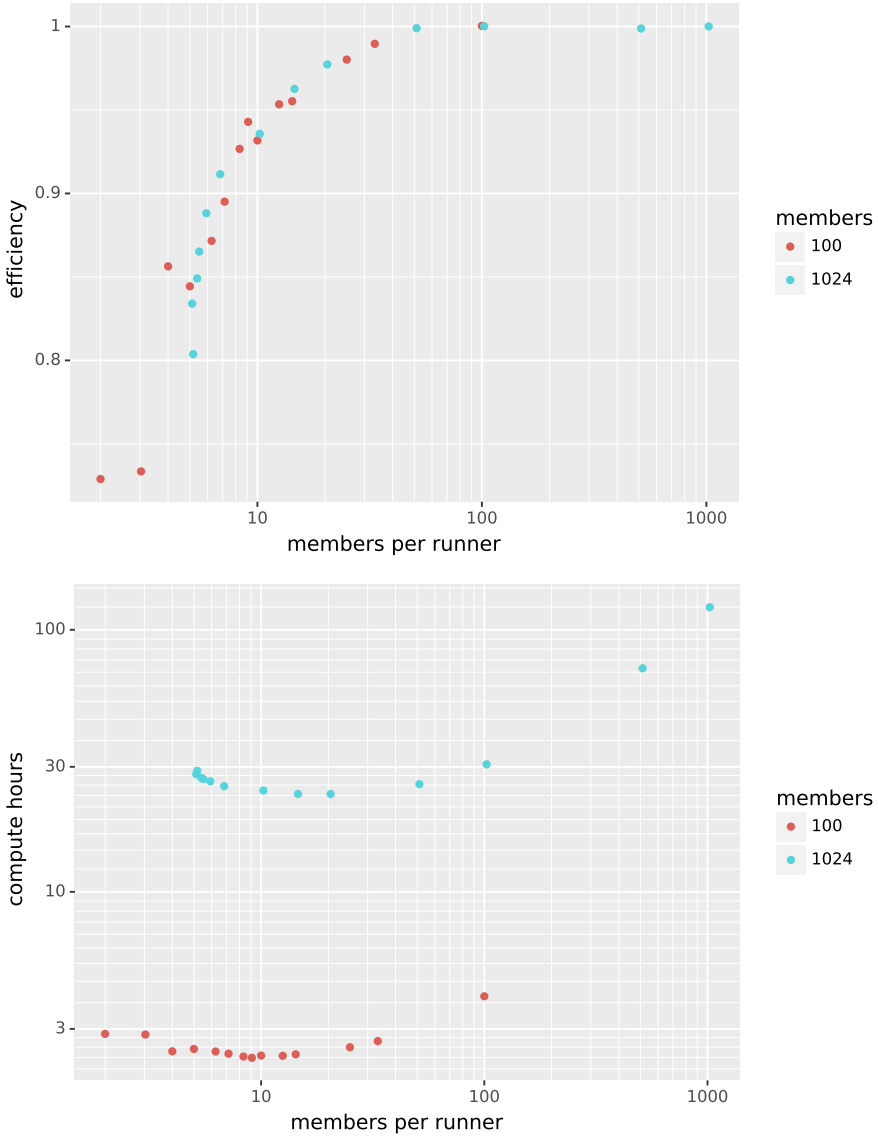


Figure 8: Efficiency of the propagation phase only (*top*) and total compute hours used per assimilation cycle (update and propagation phase) (*bottom*) for different numbers of runners while assimilating 25 observations into an about 4 M grid cell ParFlow simulation with 100 and 1,024 ensemble members.

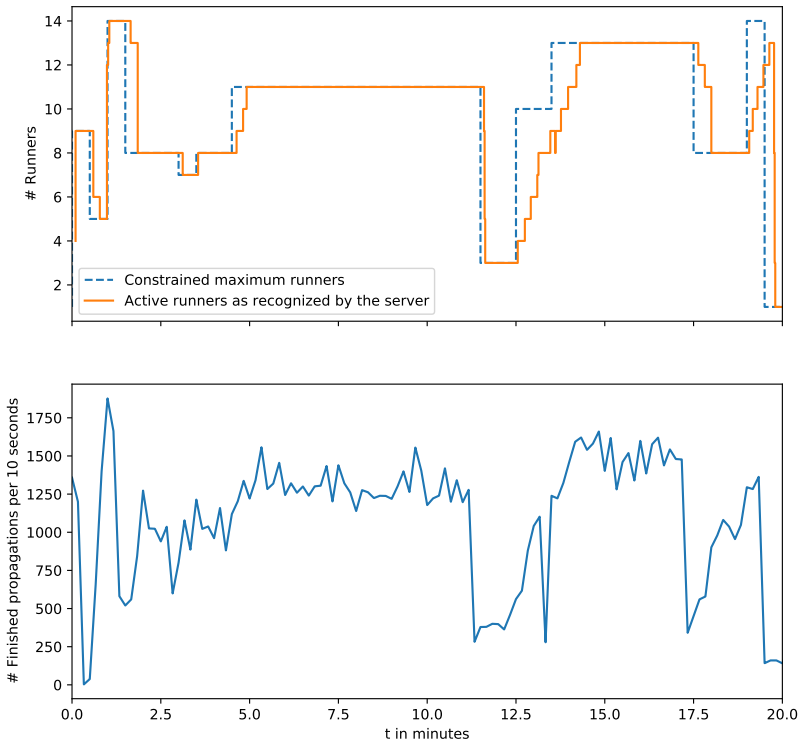


Figure 9: Elasticity with Melissa-DA: constraining the resources that may be used by runners (*top*), impacts the assimilation speed (*bottom*).

is useful, advocating for leveraging Melissa-DA's elasticity for adding/removing runners according to machine availability.

If we look at the compute hours, the 10× increase in the number of members to propagate roughly matches the increase in compute hours. Thus the resource usage is here dominated by the propagation phase and the server does not appear as a bottleneck.

The server exchanges about 92 MiB of state data per member and assimilation cycle. Up to 24 (resp. 96) states are assimilated per second for the 100 (resp. 1,024) members case (propagation *and* update phase) using about 2 (resp. 5) members per runner.

5.6 Fault tolerance and elasticity

In this experiment we demonstrate the Melissa-DA capability to have runners dynamically added or removed, giving Melissa-DA elasticity and the base of its fault tolerance protocol. The number of runners is limited by a target (dashed blue curve, Figure 9). The launcher periodically checks if more runners can be started. If so it starts new runners. If too many resources are used, some runners are stopped. Runners are killed without notice, as in the case of an error. There is delay between the launcher request to start a new runner and this runner being registered at the server (solid orange curve, Figure 9). Runner stops are also recognized delayed by the server. Only if a runner did not respond for the last 10 seconds, the server assumes it stopped. This runner timeout can be modified by the user. Obviously having more or less runners impacts the speed at which the data assimilation runs, here indicated by the state propagations that finish per interval of 10 seconds (Figure 9, solid blue curve).

Notice that here we leverage the batch scheduler capabilities (Slurm). A single allocation encompassing all necessary resources is requested at the beginning. Next jobs for the server or runners are allocated by Slurm within this envelope, ensuring a fast allocation. When the runners crash or are stopped, the restarted runners reuse the same envelope. A hybrid scheme is also possible, requesting a minimal first allocation to ensure the data assimilation to progresses fast enough, while additional runners are allocated outside the envelope, but whose availability to accept members may take longer depending on the machine load. It is planned to rely in such situations on *best effort* jobs that are supported by batch schedulers like OAR [12]. Best effort jobs can be deleted by the batch scheduler whenever resources to start other higher prioritized jobs are needed. This way Melissa-DA runners can fill up underutilized resources between larger job allocations on the cluster. All these variations on the allocation scheme require only minimal customization of the assimilation study configuration.

5.7 Ultra-large ensembles

We scale Melissa-DA to ultra-large ensembles, assimilating with up to 16,384 members. To save compute hours only a few assimilation cycles ran on up to 448 compute nodes using up to 16,240 cores of the Jean-Zay supercomputer. When doubling both, the ensemble size and the number of runners, the propagation phase's execution time stays roughly the same with an average number of about 20 members per runner up to 8,192 cores (Table 1). Dynamic load balancing leads to 96% efficiency (or 4% runner idle time) during the propagation phase at an average of 20 member propagations per runner. During this idle time the runners are either waiting for new state data to arrive or the update phase to begin.

The number of runners was not further doubled after running 16,384 members, as we were not able to get the necessary resource allocation on the machine. Thus, the walltime doubles.

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
Cores per runner	40	40	40	40
Nodes per runner	1	1	1	1
Server cores	240	240	240	240
Server nodes	6	12	24	48
Ensemble state size (sum of all member state sizes) (TiB)	0.240	0.481	0.961	1.922
RAM theoretically available on server nodes (TiB)	1.125	2.25	4.5	9
Update phase walltime (ms)	5,339	11,872	28,368	52,893
Propagation phase walltime (ms)	41,555	41,757	42,295	83,898
Mean runner idle time (propagation + update phase) (ms)	361	704.8	1,441	1,255
Median runner idle time (propagation + update phase) (ms)	19.03	12.78	15.84	19.28
Std runner idle time (propagation + update phase) (ms)	1,518	3,058	6,294	7,799
Mean runner compute time (propagation phase) (ms)	1,948	1,952	1,986	1,961
Median runner compute time (propagation phase) (ms)	1,918	1,909	1,914	1,903
Std runner compute time (propagation phase) (ms)	155.9	165.2	273.9	248.6
Runner idle time during propagation phase (%)	3.968	4.227	3.819	4.468
Scaling efficiency for the propagation phase (%)	96.03	95.77	96.2	95.76
Scaling efficiency for the assimilation cycle (%)	85.1	74.57	57.58	58.74

Table 1: Large scale Melissa-DA runs. Scaling efficiency computed against the walltime of the execution on a single runner.

The update phase takes a considerable amount of time when using EnKF on such large member counts (Section 5.4). During this time the runners are idle, giving a total scaling efficiency of only about 59 % when adding runners in the case of 16,384 members. The scalability of the update phase is a classical and well-known issue of DA. Classical solutions exist to reduce the update execution time, like relying on a localized EnKF filter (basically some terms of the covariance matrix are set to zero). Other less classical approaches to shorten the wait time during the update phase could be the iterative calculation of some parts of the EnKF update phase each time a new background state is received [33]. Investigating such algorithms for the update phase is part of future work.

The server is spread on the minimum number of nodes necessary to fulfill the memory requirements to store the ensemble and to perform the update phase computations (compare Table 1). Since Melissa-DA and the underlying EnKF update phase parallelizations are based on domain decomposition and the domain size does not change when the number of members increases, the number of allocated server processes is kept constant at 240 cores (otherwise communication cost between the server processes would overwhelm calculation).

For each assimilation cycle with 16,384 members, a total of 2.9 TiB of member state data are transferred back and forth over the network between the server and all the runners. By enabling direct data transfers, Melissa-DA avoids the performance penalty that would induce the use of files as intermediate storage.

5.8 Comparison of Melissa-DA and PDAF

We compare Melissa-DA to PDAF [32, 31][5], one of the most advanced frameworks for large scale ensemble based DA. Both are set to run the same DA use case running ParFlow with the same hydrological problem as in Section 5.1. PDAF has been configured to the best of our knowledge to get the best possible performance.

PDAF supports an offline file-based mode and an online mode. The latter is used here as it provides the best performance. In online mode PDAF runs a single large MPI executable whose processes are split into runners, called *model tasks* in PDAF jargon, where:

- Each runner is assigned a fixed static set of members to propagate in sequence at each cycle.
- Once all propagations performed, the assimilated states, i.e. the state parts necessary for the update phase, are gathered on a specific *master runner*.
- Once all data received, the master runner performs the update phase, the other runners being idle. Next the results are scattered back to their respective runners for the next cycle.

[5] http://pdaf.awi.de/trac/wiki/ModifyModelforEnsembleIntegration, retrieved the 25.11.2021

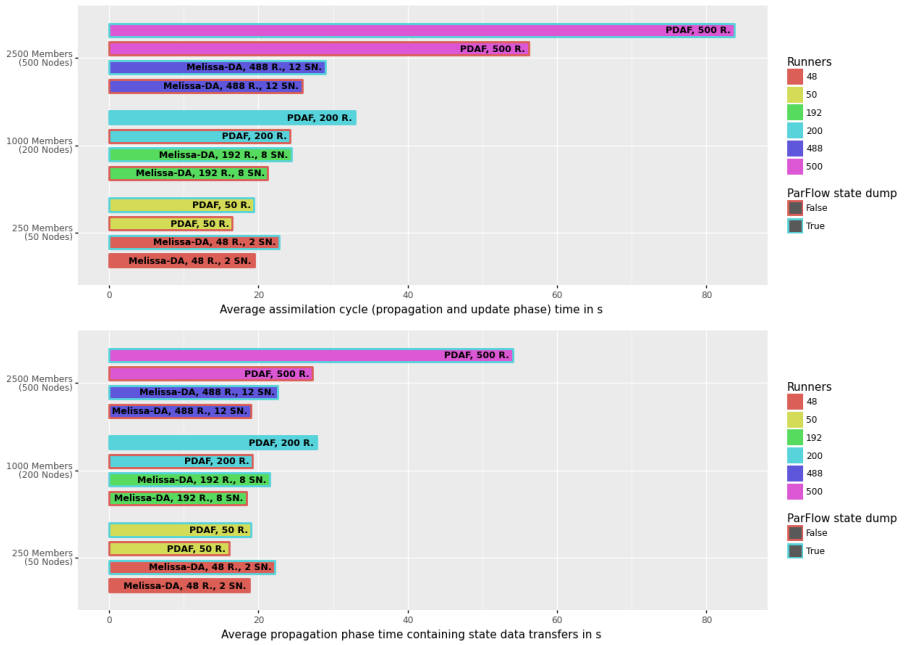


Figure 10: Comparison of the runtime of Melissa-DA and PDAF running the same DA problem on the same amount of resources. *Top*: full assimilation cycle time. *Bottom*: propagation phase time of each cycle. S.N. : number of server nodes, R. : number of runners. ParFlow state dump true if runners output each member state to disk.

PDAF transfers data between the master runner and the other runners at the beginning and end of each propagation phase, not favoring the overlap of communications with computations. In opposite Melissa-DA has a dedicated server capable of handling data transfers concurrently with member propagation. PDAF supports neither load balancing, nor fault tolerance or elasticity. A single process failure will stop the full application. These features require Melissa-DA to gather the full states on the server, leading to about three times more data transfers for this use case. PDAF performs the update phase on one runner, constraining to use the same number of processes as for one member propagation. This, however brings some simplicity regarding data transfers as it does not require a N $\times$ M data redistribution scheme. In opposite Melissa-DA can use different parallelization levels for the runner and server executables.

For both frameworks, runners execute on 40 compute cores each (1 node). To keep the runtime of the update phase reasonably fast, we ran no more than 2,500 members. As already mentioned, Melissa-DA also uses PDAF for implementing the update phase. But Melissa-DA enables to use more processes for the server, a flexibility we leveraged for the experiments. To keep the comparison fair, we always compare runs using the same global amount of resources, so with less runners for Melissa-DA than PDAF.

Figure 10 compares the assimilation cycle times. Times are measured from the second cycle (see Section 5.2). Details on the first cycle can be found in Table 2. Two types of runs were performed: runs where runners perform no output to disk, and runs where runners write the member state to disk after each propagation. This latter case introduces jitter in the propagation times, and so load unbalance, as the write time is sensitive to the file system load. This also matches a classical scenario when users require states to be saved for post hoc analysis (for that purpose Melissa-DA checkpoints can also be used, see Section 4.7). Melissa-DA outperforms PDAF except at smaller scale with 250 members, Melissa-DA being up to almost 3 times faster than PDAF at 2,500 members and state dumping.

The resources are split differently for PDAF and Melissa-DA. Melissa-DA performs the update phase on a server with 80, 320 and 480 cores for 250, 1,000 and 2,500 members, and 1,920, 7,680 and 19,520 cores are used for runners performing member propagations. PDAF uses always 40 cores for the update phase and 2,000, 8,000 and 20,000 cores for runners performing the member propagations. The update times are similar at 250 members (about 0.4 s) and reach 29 s for PDAF versus 6.5 s for Melissa-DA at 2,500 members. But Melissa-DA performance gain is not only related to the performance gain on the update phase as shown when looking at the propagation times only (Figure 10 *bottom*) or when comparing execution traces (Figure 11).

Notice that Melissa-DA is in a less favorable position than PDAF, since:

1. The number of members is chosen to be a multiple of the number of PDAF runners. Exactly 5 members are executed by every runner. That makes for an uneven number of members per runner for Melissa-DA. This is visible in Figure 11 where Melissa-DA needs to wait for the propagation of the few runners receiving 6 members before starting the update phase.

2. Even if we compare runs without failures, for all Melissa-DA runs the FTI checkpointing was active on the server, writing at least 23 GiB, 90 GiB, or 225 GiB of state data per assimilation cycle for 250, 1,000 or 2,500 members respectively.

These very likely cause most of the delay Melissa-DA experiences over PDAF for 250 members. But for larger numbers of members, the dynamic load balancing and the capability to overlap state transfer with model propagation offset this effect and permit Melissa-DA to outperform PDAF. Also notice that Melissa-DA time for the propagation phase per cycle stays nearly the same from 50 to 500 compute nodes when state dump is off, showing a strong efficiency gain compared to PDAF.

Allocating more resources to the server for Melissa-DA can be essential for performance. The trace of 500 members (Figure 12) shows that with 1 server node (40 cores, *right*) the server becomes a bottleneck impairing runner progress. At 4 server nodes (160 cores, *left*) runner idle time during the propagation phase is significantly reduced. We suspect that the ZeroMQ library used for the data transport but not designed and implemented to take full benefit of

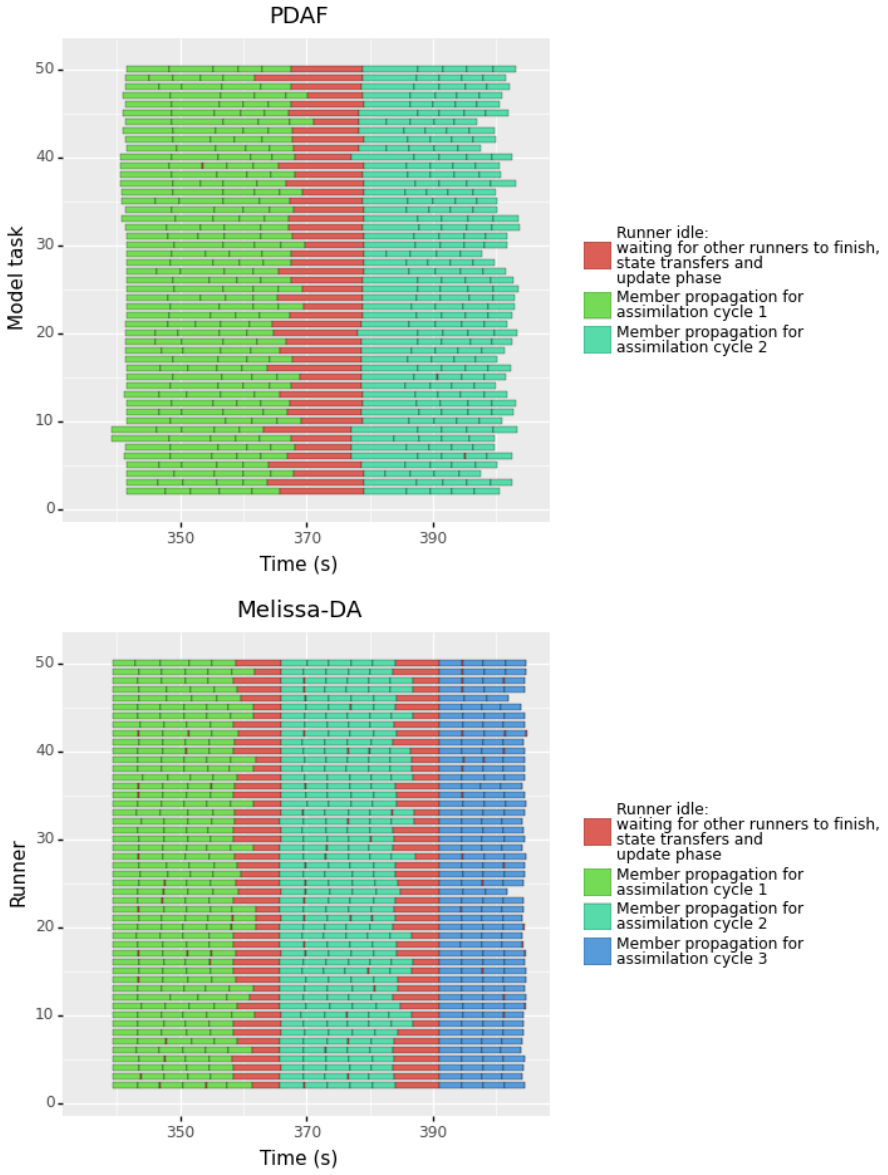


Figure 11: Traces of 50 PDAF and Melissa-DA runners. Both runs were performed on 200 nodes used as 200 runners (PDAF) or 192 runners and 8 server nodes (Melissa-DA) to propagate 1,000 members. Every runner in PDAF propagates exactly 5 members while Melissa-DA runners propagate between 5 and 6 members per cycle. Every member dumps its states during propagation.

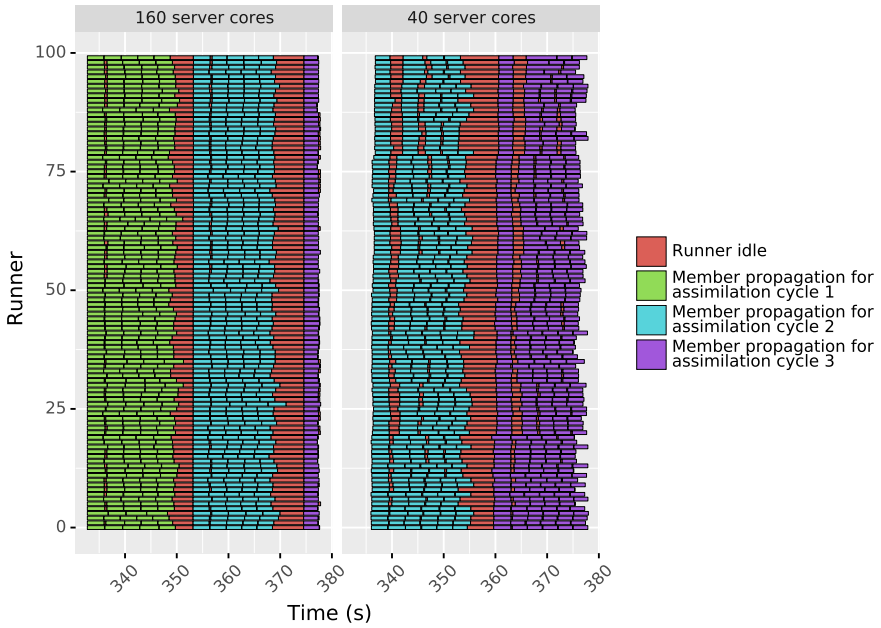


Figure 12: Traces of a Melissa-DA run with 500 members on 100 runners (4,000 cores), with either 1 (*right*) or 4 (*left*) server nodes (40 or 160 server cores respectively).

Framework	Server nodes	Nodes	Runners	Sate dump	Start 2nd cycle at time	Avg. propagation time 1st cycle	Avg. propagation time 3rd cycle
PDAF	-	500	500	off	357.25	41.36	2.58
PDAF	-	500	500	on	379.70	43.19	5.23
Melissa-DA	12	500	488	off	471.76	41.57	2.42
Melissa-DA	12	500	488	on	482.89	41.99	2.95

Table 2: Startup times (seconds) of PDAF and Melissa-DA at 2500 members.

the underlying high performance network, is partly responsible for these idle times. Future work will investigate if relying on other transport layers, like MPI dynamic communications (`MPI_Comm_connect`), may be effective to reduce it. Besides shrinking the time needed for the assimilation update phase from 3.0 s to 1.0 s, using four server nodes instead of one also decreases the propagation phase from 20.1 s to 18.8 s. In both cases we have 100 runners so the 500 members can be evenly distributed. But even in this situation, the trace shows that dynamic load balancing is critical for performance. With one server node, some runners get 6 members to propagate as some other runners experiencing significant delays only get 4 members.

We now compare startup times (Table 2, see "Start 2nd cycle"). PDAF benefits from being a single MPI executable, requiring a single request to the batch scheduler to start. Melissa-DA takes longer mainly due to the multiple requests

done to the batch scheduler (Section 5.2) as well as its dynamic architecture. Melissa-DA is designed to have runners executed in an elastic mode, started independently as resources become available, rather than waiting to have the full set of resources reserved as required for PDAF.

6 Conclusion

In this article we introduced Melissa-DA, an elastic, fault-tolerant, load balancing, online framework for ensemble-based DA. All these properties lead to an architecture allowing to run Ensemble Kalman Filters with up to 16,384 members to assimilate observations into large-scale model state vectors of more than 4 M degrees of freedom. Melissa-DA also outperforms PDAF, a MPI based approach, taking less than half the time per assimilation cycle for the same DA problem with 2,500 members on 500 compute nodes (20,000 cores).

Thanks to its master/worker architecture adding and removing resources to the Melissa-DA application is possible at runtime. In future we want to benefit from this agile adaptation to further self-optimize the compute hour consumption. We will also consider executions on heterogeneous machines (nodes with accelerators or more memory). The modular master/worker model of Melissa-DA allows for flexibility to leverage such architectures. Large memory nodes for the server can also have a very positive impact on the core hour consumption. We are also planning to integrate other assimilation methods into Melissa-DA and to run use cases with other simulation codes and at different scales.

Concerned with measuring the compute cost and environmental impact of this paper, we counted the total number of CPU hours used, including all intermediate and failed tests that do not directly appear in this paper, at about 216,000 CPU hours split between the JUWELS and the Jean-Zay supercomputers.

Acknowledgements

We would like to thank the following people: Bibi Naz, Ching Pui Hung and Harrie-Jan Hendricks Franssen from Fohrschungszentrum Juelich for the scientific exchange as well as for providing the ParFlow real world use case for data assimilation, Kai Keller and Leonardo Bautista-Gomez from Barcelona Supercomputing Center for the integration of FTI into the Melissa-DA server code as well as Lars Nerger from Alfred Wegener Institut and Wolfgang Kurtz from Leibniz Supercomputing Center for the scientific exchange on PDAF, ParFlow and TerrSysMP. We also thank the DataMove research engineers Christoph Conrads and Théophile Terraz for contributing to the Melissa-DA software stack and proof reading.

Funding

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).

References

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014.

[2] Jeffrey Anderson, 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(9):1283–1296, September 2009. Publisher: American Meteorological Society.

[3] Mark Asch, Marc Bocquet, and Maëlle Nodet. *Data assimilation: methods, algorithms, and applications*, volume 11. SIAM, 2016.

[4] Steven F. Ashby and Robert D. Falgout. A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations. *Nuclear Science and Engineering*, 124(1):145–159, September 1996.

[5] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Luksaz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive parallel programming in python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, page 25–36, 2019.

[6] Vivek Balasubramanian, Travis Jensen, Matteo Turilli, Peter Kasson, Michael Shirts, and Shantenu Jha. Adaptive ensemble biomolecular applications at scale. *SN Computer Science*, 1(2):1–15, 2020.

[7] Vivek Balasubramanian, Matteo Turilli, Weiming Hu, Matthieu Lefebvre, Wenjie Lei, Guido Cervone, Jeroen Tromp, and Shantenu Jha. Harnessing the power of many: Extensible toolkit for scalable ensemble applications. In *IPDPS 2018*, pages 536–545, 2018.

[8] Leonardo Bautista-Gomez, 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, pages 1–12, November 2011.

[9] Jonas Berndt. *On the predictability of exceptional error events in wind power forecasting —an ultra large ensemble approach—*. PhD thesis, Universität zu Köln, 2018.

[10] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[11] Massimo Bonavita, Y. Trémolet, Elias Hólm, S. T. K. Lang, Marcin Chrust, Marta Janiskova, Philippe Lopez, Patrick Laloyaux, Patricia de Rosnay, Mike Fisher, M. Hamrud, and Stephen English. A strategy for data assimilation. Technical Report 800, ECMWF, 2017.

[12] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster Computing and Grid 2005 (CCGrid05)*, pages 776–783, Cardiff, United Kingdom, 2005. IEEE.

[13] A. M. Clayton, A. C. Lorenc, and D. M. Barker. Operational implementation of a hybrid ensemble/4D-Var global data assimilation system at the Met Office. *Quarterly Journal of the Royal Meteorological Society*, 139(675):1445–1461, July 2013.

[14] Wael R Elwasif, David E Bernholdt, Sreekanth Pannala, Srikanth Allu, and Samantha S Foley. Parameter sweep and optimization of loosely coupled simulations using the dakota toolkit. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 102–110, 2012.

[15] Geir Evensen. *Data assimilation: the ensemble Kalman filter*. Springer Science & Business Media, 2009.

[16] R. L. Graham. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[17] Pieter Hintjens. *ZeroMQ, Messaging for Many Applications*. O'Reilly Media, 2013.

[18] Pieter Leopold Houtekamer, Mark Buehner, and Michèle De La Chevrotière. Using the hybrid gain algorithm to sample data assimilation uncertainty. *Quarterly Journal of the Royal Meteorological Society*, 145(S1):35–56, 2019.

[19] Jim E. Jones and Carol S. Woodward. Newton–Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems. *Advances in Water Resources*, 24(7):763–774, July 2001.

[20] Jülich Supercomputing Centre. JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre. *Journal of large-scale research facilities*, 5(A135), 2019.

[21] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[22] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg, 2012. Springer.

[23] Stefan J. Kollet and Reed M. Maxwell. Capturing the influence of groundwater dynamics on land surface processes using an integrated, distributed watershed model. *Water Resources Research*, 44(2):W02402, February 2008.

[24] W. Kurtz, G. He, S. J. Kollet, R. M. Maxwell, H. Vereecken, and H.-J. Hendricks Franssen. TerrSysMP–PDAF (version 1.0): a modular high-performance data assimilation framework for an integrated land surface–subsurface model. *Geosci. Model Dev.*, 9(4):1341–1360, April 2016.

[25] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062, 2018.

[26] Edward N. Lorenz. Deterministic Nonperiodic Flow. *Journal of the Atmospheric Sciences*, 20(2):130–141, March 1963. Publisher: American Meteorological Society.

[27] Reed M. Maxwell. A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling. *Advances in Water Resources*, 53:109–117, March 2013.

[28] Reed M. Maxwell and Norman L. Miller. Development of a Coupled Land Surface and Groundwater Model. *Journal of Hydrometeorology*, 6(3):233–247, June 2005.

[29] Takemasa Miyoshi, Keiichi Kondo, and Toshiyuki Imamura. The 10,240-member ensemble Kalman filtering with an intermediate AGCM: 10240-MEMBER ENKF WITH AN AGCM. *Geophysical Research Letters*, 41(14):5264–5271, July 2014.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577, Carlsbad, CA, USA, October 2018. USENIX Association.

[31] L. Nerger, W. Hiller, and J. Schröter. PDAF - THE PARALLEL DATA ASSIMILATION FRAMEWORK: EXPERIENCES WITH KALMAN FILTERING. In *Use of High Performance Computing in Meteorology*, pages 63–83, Reading, UK, September 2005. WORLD SCIENTIFIC.

[32] Lars Nerger and Wolfgang Hiller. Software for ensemble-based data assimilation systems—implementation strategies and scalability. *Computers & Geosciences*, 55:110–118, 2013.

[33] Elias D. Niño, Adrian Sandu, and Jeffrey L. Anderson. An Effcient Implementation of the Ensemble Kalman Filter Based on Iterative Sherman Morrison Formula. *Procedia Computer Science*, 9:1064–1072, January 2012.

[34] Ioannis Paraskevakos, Andre Luckow, Mahzad Khoshlessan, George Chantzialexiou, Thomas E Cheatham, Oliver Beckstein, Geoffrey C Fox, and Shantenu Jha. Task-parallel analysis of molecular dynamics trajectories. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

[35] S. Pronk, G. R. Bowman, B. Hess, P. Larsson, I. S. Haque, V. S. Pande, I. Pouya, K. Beauchamp, P. M. Kasson, and E. Lindahl. Copernicus: A new paradigm for parallel adaptive molecular dynamics. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, Nov 2011.

[36] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[37] Bernd Schalge, Gabriele Baroni, Barbara Haese, Daniel Erdal, Gernot Geppert, Pablo Saavedra, Vincent Haefliger, Harry Vereecken, Sabine Attinger, Harald Kunstmann, Olaf A. Cirpka, Felix Ament, Stefan Kollet, Insa Neuweiler, Harrie-Jan Hendricks Franssen, and Clemens Simmer. Presentation and discussion of the high resolution atmosphere-land surface subsurface simulation dataset of the virtual Neckar catchment for the period 2007-2015. *Earth System Science Data Discussions*, pages 1–40, March 2020. Publisher: Copernicus GmbH.

[38] D.B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 131–140, San Juan, Puerto Rico, 1991. IEEE Comput. Soc. Press.

[39] Théophile Terraz, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin. Melissa: Large scale in transit sensitivity analysis avoiding intermediate files. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*, pages 1–14, Denver, 2017.

[40] Habib Toye, Samuel Kortas, Peng Zhan, and Ibrahim Hoteit. A fault-tolerant hpc scheduler extension for large and operational ensemble data assimilation: Application to the red sea. *Journal of Computational Science*, 27:46 – 56, 2018.

[41] Nils van Velzen, Muhammad Umer Altaf, and Martin Verlaan. OpenDA-NEMO framework for ocean data assimilation. *Ocean Dynamics*, 66(5):691–702, May 2016.

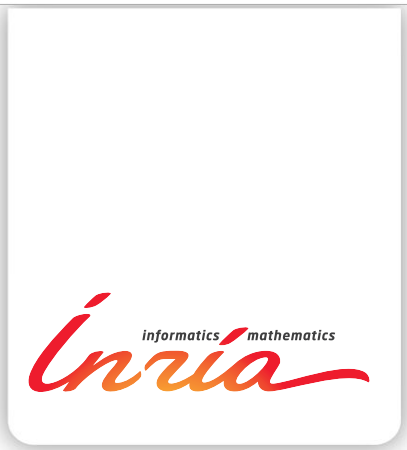
[42] H. Yashiro, K. Terasaki, Y. Kawai, S. Kudo, T. Miyoshi, T. Imamura, K. Minami, H. Inoue, T. Nishiki, T. Saji, M. Satoh, and H. Tomita. A 1024-member ensemble data assimilation with 3.5-km mesh global weather simulations. In *Supercomputing 2020: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, Los Alamitos, CA, USA, nov 2020. IEEE Computer Society.

[43] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[44] Matei Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. PhD thesis, EECS Department, University of California, Berkeley, Feb 2014.

[45] Henrique C. Zanúz, Bruno Raffin, Omar A. Mures, and Emilio J. Padrón. In-transit molecular dynamics analysis with Apache flink. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 25–32, Dallas Texas USA, November 2018. ACM.

[46] Lin Zhang, Yongzhu Liu, Yan Liu, Jiandong Gong, Huijuan Lu, Zhiyan Jin, Weihong Tian, Guiqing Liu, Bin Zhou, and Bin Zhao. The operational global four-dimensional variational data assimilation system at the China Meteorological Administration. *Quarterly Journal of the Royal Meteorological Society*, 145(722):1882–1896, 2019.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399