

# Dynamic Load/Propagate/Store for Data Assimilation with Particle Filters on Supercomputers

Sebastian Friedemann, Kai Keller, Yen-Sen Lu, Bruno Raffin, Leonardo Bautista Gomez

▶ **To cite this version:**

Sebastian Friedemann, Kai Keller, Yen-Sen Lu, Bruno Raffin, Leonardo Bautista Gomez. Dynamic Load/Propagate/Store for Data Assimilation with Particle Filters on Supercomputers. 2024. hal-03927612v1

**HAL Id: hal-03927612**

**https://hal.science/hal-03927612v1**

Preprint submitted on 6 Jan 2023 (v1), last revised 14 Feb 2024 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# A Framework for Large Scale Particle Filters Validated with Data Assimilation for Weather Simulation

Sebastian Friedemann[a], Kai Keller[b], Yen-Sen Lu[c], Bruno Raffin[a,*], Leonardo Bautista-Gomez[b]

[a] *Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000, Grenoble, France*
[b] *Barcelona Supercomputing Center, Barcelona, 08034, Spain*
[c] *Forschungzentrum Juelich, Juelich, 52428, Germany*

**Abstract**

Particle filters are a group of algorithms to solve inverse problems through statistical Bayesian methods when the model does not comply with the linear and Gaussian hypothesis. Particle filters are used in domains like data assimilation, probabilistic programming, neural network optimization, localization and navigation. Particle filters estimate the probability distribution of model states by running a large number of model instances, the so called particles. The ability to handle a very large number of particles is critical for high dimensional models. This paper proposes a novel paradigm to run very large ensembles of parallel model instances on supercomputers. The approach combines an elastic and fault tolerant runner/server model minimizing data movements while enabling dynamic load balancing. Particle weights are computed locally on each runner and transmitted when available to a server that normalizes them, resamples new particles based on their weight, and redistributes dynamically the work to runners to react to load imbalance. Our approach relies on a an asynchronously managed distributed particle cache permitting particles to move from one runner to another in the background while particle propagation goes on. This also enables the number of runners to vary during the execution either in reaction to failures and

*Corresponding author

*Email addresses:* sebastian.friedemann@inria.fr (Sebastian Friedemann), kai.keller@bsc.es (Kai Keller), ye.lu@fz-juelich.de (Yen-Sen Lu), bruno.raffin@inria.fr (Bruno Raffin), leonardo.bautista@bsc.es (Leonardo Bautista-Gomez)

restarts, or to adapt to changing resource availability dictated by external decision processes. The approach is experimented with the Weather Research and Forecasting (WRF) model, to assess its performance for probabilistic weather forecasting. Up to 2,555 particles on 20,442 compute cores are used to assimilate cloud cover observations into short–range weather forecasts over Europe.

*Keywords:* ls Data Assimilation, Particle Filter, Ensemble Run, Resilience, Elasticity

---

## 1. introduction

Given an output and a transformation function, finding the input states represents a so called **inverse problem**. A wide range of approaches to address this central problem exist. Statistical Bayesian methods stand out as they provide uncertainty measures of the proposed input in form of probability density functions. In this paper, we consider **particle filters**, a statistical Bayesian method combining uncertainties of both the dynamical system and observations, to estimate the system **state**. Several realizations of the dynamical system, called **particles**, with differently perturbed internal states, are **propagated** up to a time where **observation data** are available. These particles are then **weighted** corresponding to their distance to the observations. The weights are used to generate a new sample of particles that better matches the observations. This process repeats while observations are available.

Particle filters are used for several purposes, like *Data Assimilation (DA)* [1], probabilistic programming [2, 3, 4], neural network optimization [5], localization and navigation[6]. Particle filters stands by their ability to work with nonlinear and/or non-Gaussian state space models in opposition to technics like Ensemble Kalman Filtering (EnKF). But this ability comes with a need to run larger number of particles. If the dynamical system is an advanced parallel high-dimensional numerical model solver, as for geoscience applications, thousands of particles may be necessary to avoid undersampling and degeneracy. While high-dimensional large-scale solvers are compute intense already, the execution of several thousands of instances adds orders of magnitude of calculations. Large scale DA with particle filters is for instance used for geoscience applications such as weather forecasting [7]. Supercomputers, reaching today Exascale, have the compute power to support very large scale

particle filters. But using such resources efficiently, time and energy wise, is challenging. Applications need to limit the use of the Parallel File System (PFS), a classical supercomputer bottleneck, and favor instead in situ data processing as well as local data storage to reduce data movements, asynchronism to overlap tasks whenever possible. Applications also need to be flexible to adapt to changes during execution, requiring support for resilience, elasticity and dynamics load balancing.

Existing large scale approaches can be divided into two types: **online** and **offline** approaches. Offline approaches use temporary files to exchange data. To propagate one particle, one model instance starts, loads the particle from a file, propagates it up to a given time, stores the resulting particle back to a file and shuts down. This approach is flexible, fault tolerance is easy to support, but performance, especially at scale is impaired by the heavy use of the file system and the cost of starting a new model instance for each propagation. Online approaches bypass the file system by running a large MPI application that encompasses the full workflow, where the particles are distributed to the different model instances through the network via MPI communications. While saving I/O overheads, this approach loses flexibility. For instance, a fault during a single particle propagation stops the entire application. Thus existing online approaches, as will be detailled in the related work section (Section 6), usually do not support fault tolerance or dynamic load balancing.

In this paper we develop an alternate approach that leads to a high efficient yet flexible framework. The key to achieve this goal is the **virtualization** of particle propagations. We turn a numerical model solver instance into a **runner** capable of propagating several particles one after the other with low overheads and idle times. Each runner is coupled with a node-local distributed state cache enabling fast loads and stores of particles. The caches are asynchronously persisted to the file system for checkpointing and load balancing between runners. Asynchronous prefetching of particles into the cache enables overlapping particle loads with the particle propagation. A **server** organizes the work distribution to the runners and performs the centralized tasks of the particle filter update and (re-)sampling. Runners and server are each executed as independent executables to support elasticity and facilitate fault tolerance. The association of these different features complemented with a fault tolerance protocol, leads to an elastic and resilient framework, minimizing data movements while enabling dynamic load balancing. Particle virtualization enables to decouple resource allocation from the number

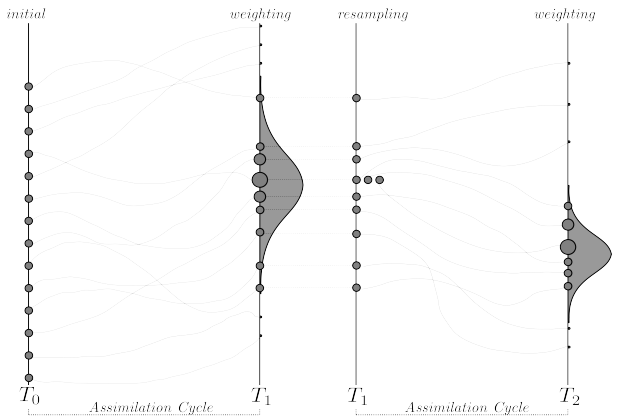



**Figure 1:** Initially particles are uniformly sampled. They are propagated to $T_1$ where they are weighted taking into account observation data. Resampling leads to discard some particles with low weights (top and bottom), while others with high weights become parent of several ones (3 here).

of particles. The number of runners can vary during the execution either in reaction to failures and restarts, or to adapt to changing resource availability dictated by external decision processes. The proposed architecture is designed for running at extreme scale, leveraging deep storage hierarchies and heterogeneous cluster designs of current and future supercomputers.

We strain our proposed particle filter framework with a realistic use-case, interfacing with the Weather Research and Forecasting (WRF, version 3.7.1) model [8]. WRF is a widely used weather model for operational forecasting and research. Using our particle filter, we are able to run 2,555 particles on 20,442 compute cores for WRF simulations on a European domain with 87 % efficiency.

The rest of the paper is structured as follows: Section 2 reviews the principles of particle filters and the associated workflow. Section 3 presents the architecture of our proposed approach, while Section 4 is dedicated to experiments and Section 5 to discussion. The papers ends with related work in Section 6 and a conclusion in Section 7.

## 2. Particle Filters

In this section, we give a brief introduction on the particle filter formalism, focusing on properties that we exploit in our proposal. For a comprehensive

introduction, we refer to [1, 9]. Let $\mathcal{M}$ be a numerical model, that propagates a particle $p$ from state $\mathbf{x}_{p,t-1}$ at time $t-1$ to state $\mathbf{x}_{p,t}$ at time $t$:

$$\mathbf{x}_{p,t} = \mathcal{M}(\mathbf{x}_{p,t-1}) + \boldsymbol{\beta}^t \tag{1}$$

Where $\boldsymbol{\beta}$ is a random forcing representing errors in the model. Let $\mathcal{H}$ be the projection operator from the state space to the observation space:

$$\mathbf{y} = \mathcal{H}(\mathbf{x}) + \boldsymbol{\epsilon}^t \tag{2}$$

Where $\boldsymbol{\epsilon}$ is a random vector, representing the measurement errors.

The bootstrap particle filter formalism can be derived using Bayes' theorem:

$$p(\mathbf{x}_t|\mathbf{y}_t) = \frac{p(\mathbf{y}_t|\mathbf{x}_t)\, p(\mathbf{x}_t)}{p(\mathbf{y}_t)} \tag{3}$$

Where $p(\mathbf{x}_t|\mathbf{y}_t)$ is the posterior Probability Density Function (PDF), $p(\mathbf{x}_t)$ is the prior PDF, $p(\mathbf{y}_t|\mathbf{x}_t)$ is the likelihood of observing $\mathbf{y}_t$ if $\mathbf{x}_t$ would represent the true state, and $p(\mathbf{y}_t)$ is the evidence available. The goal of the filtering formalism is to derive the posterior $p(\mathbf{x}_t|\mathbf{y}_t)$, which describes the PDF of the state $\mathbf{x}_t$ taking into account the evidence $\mathbf{y}_t$.

In the bootstrap particle filter, the prior $p(\mathbf{x}_t)$ is estimated via sampling an ensemble of $P$ particles $\mathbf{x}_{p,t}$ representing different model states

$$p(\mathbf{x}_t) = \frac{1}{P} \sum_{p=0}^{P-1} \delta(\mathbf{x}_t - \mathbf{x}_{p,t}), \tag{4}$$

The likelihood $p(\mathbf{y}_t|\mathbf{x}_t)$ is assumed to be known, estimated when calibrating the sensor. It is derived from the PDF of $\boldsymbol{\epsilon}$ applied to the distance between state and observation $\mathbf{y}_t - \mathcal{H}(\mathbf{x}_t)$:

$$p(\mathbf{y}_t|\mathbf{x}_t) = p_\epsilon(\mathbf{y}_t - \mathcal{H}(\mathbf{x}_t)) \tag{5}$$

The evidence $p(\mathbf{y}_t)$ can be computed by:

$$p(\mathbf{y}_t) = \int p(\mathbf{y}_t|\mathbf{x}_t) p(\mathbf{x}_t)\, d\mathbf{x}_t \tag{6}$$

$$= \sum_{p=0}^{P-1} \frac{1}{P}\, p(\mathbf{y}_t|\mathbf{x}_{p,t}) \tag{7}$$

$$\tag{8}$$

Putting all together and replacing the expressions in Bayes' theorem (Equation 3) we arrive to the expression for the posterior [1]:

$$p(\mathbf{x}_t|\mathbf{y}_t) \approx \sum_{p=0}^{P-1} \hat{w}_{p,t}\, \delta(\mathbf{x}_t - \mathbf{x}_{p,t}) \tag{9}$$

With $\hat{w}_{p,t}$ being the *normalized* particle weights:

$$\hat{w}_{p,t} = \frac{p(\mathbf{y}_t|\mathbf{x}_{p,t})}{\sum_{q=0}^{P-1}\ p(\mathbf{y}_t|\mathbf{x}_{q,t})} = \frac{w_{p,t}}{\sum_{q=0}^{P-1}\ w_{q,t}} \tag{10}$$

and $w_{p,t}$ being the *unnormalized* particle weights:

$$w_{p,t} = p(\mathbf{y}_t|\mathbf{x}_{p,t})\, w_{p,t-1} \tag{11}$$

Note that the initial weights are set equal to $w_{p,0} = 1/P$.

Especially for high dimensional models, particle filters tend to suffer from weight degeneration, i.e., one normalized weight is close to one and all the others are close to zero. A classical approach against ensemble degeneration is Sequential Importance Resampling (SIR) [10, 11]. The procedure of SIR consists in resampling particles from the posterior (Equation 9) at the end of the propagation step; $P$ particles are randomly drawn, *resampled*, from the existing particles, each with a probability $w_{p,t}$. Low weighted particles become discarded, while high weighted particles can become the starting point of multiple particle propagations (Figure 1). More precisely, the resampling leads to the multiset $P$ defined by the ordered pair $(Q, \boldsymbol{\alpha})$. Where $Q$ is the set of unique particles $q$ in P, and $\alpha_q$ the number of the occurrences of $q$ in $P$. The particles $q$ are hereinafter called **parent particles**.

The resampled particles are all assigned the same weight of $w_{p,t} = 1/P$ again. Particles departing from the same parent may need to become stochastically perturbed if the model does not contain a stochastic component itself. Otherwise, the trajectories of those particles would be identical.

Different flavors of SIR and resampling algorithms, like Residual Resampling, exist [12]. Some perform a resampling step after each propagation phase, while others make this dependent on criteria like the variance of the weights. In this paper we rely on SIR with resampling after each propagation phase.

**Box 1**

(a) The propagation of particle $p$ depends only on the associated state $\mathbf{x}_{p,t}$ and can be performed independently of other particles.

(b) Weights $w_{p,t}$ depend only on the associated particle $p$ and observation vector $y_t$, and can be computed independently of other weights and particles.

(c) The filter update only depends on the weights $w_{p,t}$, and not on the particles and associated states.

(d) The states $\mathbf{x}_{p,t}$ associated to the particles $p$ remain unchanged during the filter update.

Box 1 lists the properties of particle filters that are the basis for our implementation.

We exploit property (d): In contrast to other DA techniques, such as EnKF, particle states remain unchanged during the filter update. Particles that have departed too much from the observations (low weights) are discarded, and the sample set is narrowed around the best particles (high weights). The associated states, however, are not changed. Property (a) follows directly from Equation 11. Property (b) results from decoupling the weight calculation from the filter update (decentralization). The update itself, only consist of the weight normalization and particle resampling. Finally, property (c) is an intrinsic property of the bootstrap particle filter, since particles are either withdrawn or selected, but not changed. In the following sections, we will show how we can exploit those properties to improve efficiency of and resilience particle filter implementations.

## 3. Architecture

In this section we detail the proposed architecture to run a large number of particles with parallel numerical models. The algorithm, as presented in Section 2, is a sequence of two main steps:

1. A first compute intensive massively parallel step where particles can be processed concurrently to:
   (a) propagate each sate: $\mathbf{x}_{p,t} = \mathcal{M}(\mathbf{x}_{p,t-1})$,

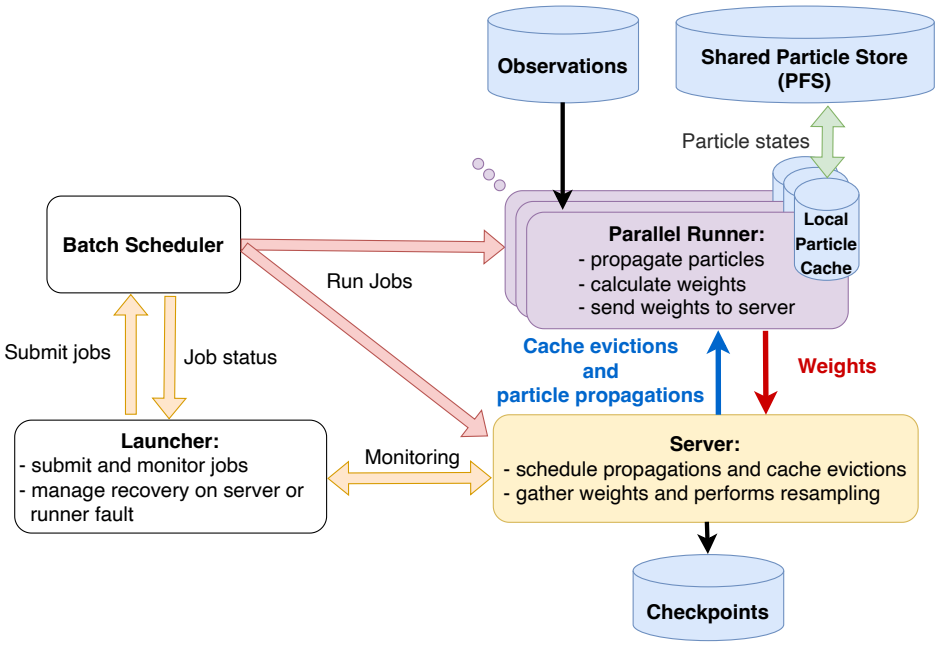



**Figure 2:** Architecture overview.

(b) compute each unormalized weight from each state and observation data:

$$w_{p,t} = p_\epsilon(\mathbf{y}_t - \mathcal{H}(\mathbf{x}_{p,t})) \tag{12}$$

2. A second lightweight step that requires to gather all unormalized weights $w_{p,t}$, usually one double per weight, for normalization and resampling.

We attribute the first step work to runners and the second step to a **server**. A **runner** is designed to propagate several particles one after the other with low overheads and idle times (Figure 2). Each one is coupled with a node-local distributed cache enabling fast loads and stores of particles. The caches are asynchronously persisted to the global file system for checkpointing and dynamic load balancing (i.e., ensure global availability of the particles). Because resampling can lead to discard some particles, or duplicate others originating from the same parent (with a local perturbation if needed), states need to be dynamically redistributed to runners to keep them evenly busy. The server drives the dynamic distribution of particle propagation tasks to runners. Runners use the distributed cache to load from the file system the

missing states. This design ensures low communications between the server and runners, and reduced state movements. The runners and the server run as independent executables, enabling to have a dynamically changing number of runners. This is a key feature used for fault tolerance and elasticity. Elasticity (sometimes also called maleability) is the ability to run under changing resource availability, here varying number of runners.

In the following we detail this design: the runners (Section 3.1), the server (Section 3.2), the distributed cache (Section 3.3), the workflow between these components (Section 3.4), the particle propagation scheduling (Section 3.5), the jobs monitoring (Section 3.6), and the fault tolerance protocol (Section 3.7) before ending with additional implementation details (Section 3.8).

### *3.1. Runners*

Runners are built from the simulation code, often an advanced parallel code or even a coupling of several parallel codes, with significant start-up times to load and build the different internal data structures. To avoid paying the cost of a restart for each particle propagation, we augment the simulation code with a mechanism to store and load particle states. This is the base of **particle virtualization**: a runner can load a particle, propagate it, store the result, and repeat this as many times as necessary. Runners are associated with a distributed cache to accelerate state loads and stores as detailled in Section 3.3. Runners also compute the associated weights $w_{p,t}$. Hence, each runner also needs to load the observations $\mathbf{y}_t$ once per cycle. Notice that the size of the observations is typically much smaller than the size of the states $\mathbf{x}_{p,t}$.

### *3.2. Server*

The server is entrusted with multiple tasks. First, it is responsible for scheduling the particle propagations to the runners (Section 3.5). Second, it gathers the weights from the runners and performs the resampling at the end of each assimilation cycle. Third, it controls the content of a distributed particle cache (Section 3.3). To collect the weights $w_{p,t}$, the server is messaged from the runners after each propagation. If there are still particles to propagate in the current cycle, the server responds to the message with an id uniquely defining a particle (hereinafter called particle-id) for the next propagation. If not, the server performs the resampling and starts the new cycle by scheduling the sampled particles to the runners. Very little data is

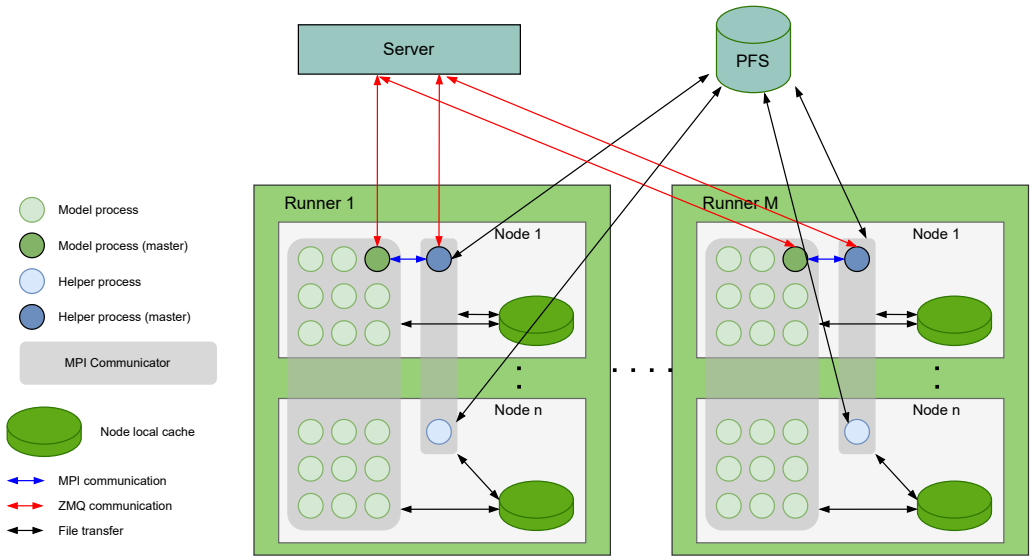



**Figure 3:** Internal runner architecture and interactions with the server and global storage (PFS). Communications with the server combine MPI and ZMQ data exchanges.

exchanged between a runner and server. The runners send the particle-id (a single int) and the corresponding weight (a single float), and the server responds with the particle-id next to propagate.

### *3.3. Distributed Particle Cache*

To allow multiple propagations of one particle on different runners, it is necessary to make them globally available. A straight forward approach is to store particles on global storage. However, on supercomputers global storage is subject to large throughput variability due to the high workload and the limited bandwidth. Node-local storage, on the other hand, is only used by the processes that run on the nodes, and the bandwidth can be stacked. Storing the particles locally results in scalable I/O performance, scaling linearly with the number of nodes.

To leverage node-local storage while still providing the particles globally, runners rely on a **distributed particle cache**. Each runner executes **helper processes** (one per node) in addition to the **model processes**, where both groups of processes are associated with its own MPI communicator (Figure 3). The model processes propagate the particles and store the associated states locally on the nodes allocated to the runner (RAM disk or other node-local storage when available). The helper processes then stage the states from local

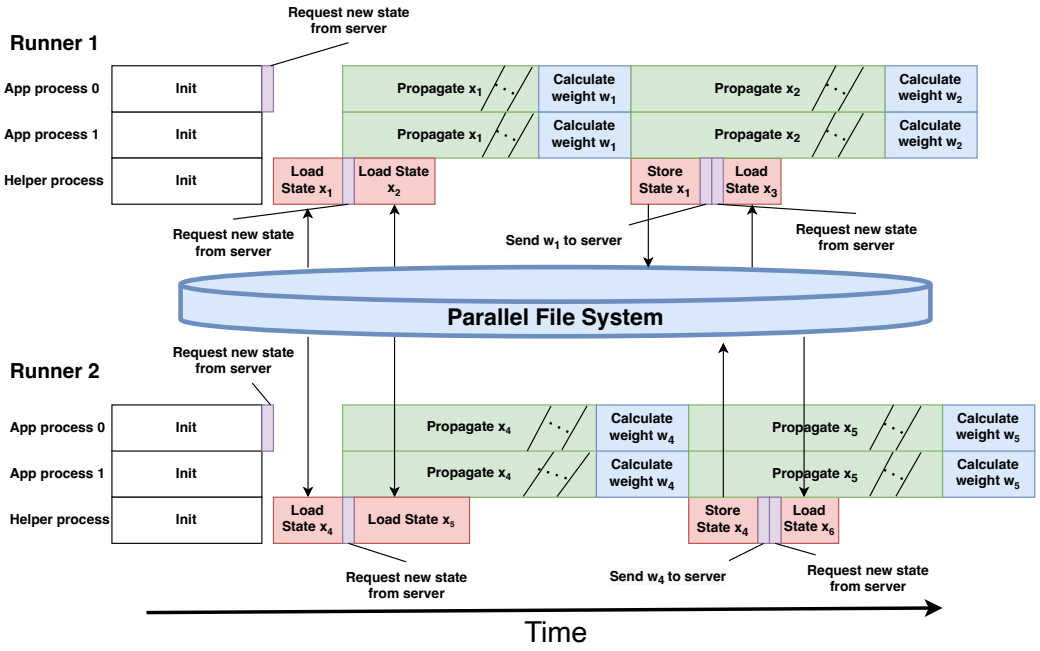



**Figure 4:** Schematic Gantt diagram showing the activity of two runners (initialization followed by two assimilation cycles). Focus on how the helper process asynchronous loads and stores enables to shadow the parallel file system accesses. For sake of simplicity no cache is used here.

to global storage asynchronously, enabling to overlap the associated I/O costs (Figure 4). Also notice that persisting particles to global storage acts as a particle checkpoint used by the fault tolerance protocol (Section 3.7).

We allow keeping a number of particles in each of the runner caches to exploit property (d) from Box 1: resampling does not change the particle states. Hence, keeping propagated particles in the cache, increases the probability to find a particle locally for future propagations (i.e., during the next cycle). If available in its local cache, a runner can propagate a particle without loading it from global storage. To further minimize cache loads, runners implement an optimized cache eviction strategy. The eviction strategy becomes especially important if the cache capacity is exceeded by the accumulated size of the particles propagated during one cycle. Because the runners have no knowledge about the status of the particle filtering (propagations, resampling), the evictions are controlled by the server and directed to the runners.

As explained in Section 3.4, each time a particle has been stored in the cache by the model processes upon successful propagation, the helper processes copy it in the background to global storage. Hence, all propagated particle states can be selected for eviction, since they are safely stored on global storage. When an eviction is required, the server selects a particle from the cache in the following order:

1. A particle from the previous cycle discarded by resampling;
2. A parent particle from the current cycle for which all associated propagations have been performed, and all weights received;
3. The particle with the lowest weight propagated during the current cycle;
4. A randomly selected particle.

The particle states for cases 1 and 2 can safely be removed from cache, since those particle are not needed anymore for future propagations. In case 3, we select the particle state with the lowest weight, as it has the lowest probability of being selected for future cycles during the resampling.

### *3.4. Runners/Server Workflow*

Once a runner job has started, it dynamically connects to the server and requests a particle to propagate from it. The server selects the particle fol-

lowing a scheduling policy described in Section 3.5. The model checks the location of the particle. If already located inside the local cache, the propagation starts. Otherwise, the model processes request the helper processes to load the state into the cache. The model processes block until the helper processes fetched the particle into the cache, and afterwards start the propagation.

Once a particle propagation finishes, the model computes the associated weight $w_p$ and stores the particle into the cache. Further, the weight and particle-id are sent to the helper processes and a new particle is requested for propagation. The helper processes, after receiving the weight from the model processes, stage the particle from the cache to global storage and afterwards sends the weight and particle-id to the server. This order ensures that the server receives a weight only after the corresponding particle is propagated and successfully stored on global storage.

The helper processes further prefetch particles in parallel to the propagations (Figure 4). The goal is to avoid blocking the model processes while waiting for a particle load from global storage (cache miss). Each time helper processes send a weight to the server, they also request the next-to-next particle-id to propagate. This particle is prefetched into the cache to become locally available for the next to follow propagation. Prefetching is suspended at the end of each propagation cycle, as propagation work for the next cycle becomes only known after the server has performed the resampling of all particles. Notice that a helper may need to cancel prefetching if the prefetched particle was in the meantime assigned to another runner, making idle the model process while waiting for the next particle to propagate. When the server makes such a decision to better balance the work load, it also takes care of ensuring coherency between runners. Globally prefetching proved to be very efficient for overlapping particle state loads with propagation (Section 4.2).

### *3.5. Particle Propagation Scheduling*

In this section, we present the scheduling algorithm implemented on the server to distribute the particle propagations to the runners. The algorithm aims to ensure an even load balancing between runners and minimizing the global particle loads, i.e. transfers of particles from global to local storage.

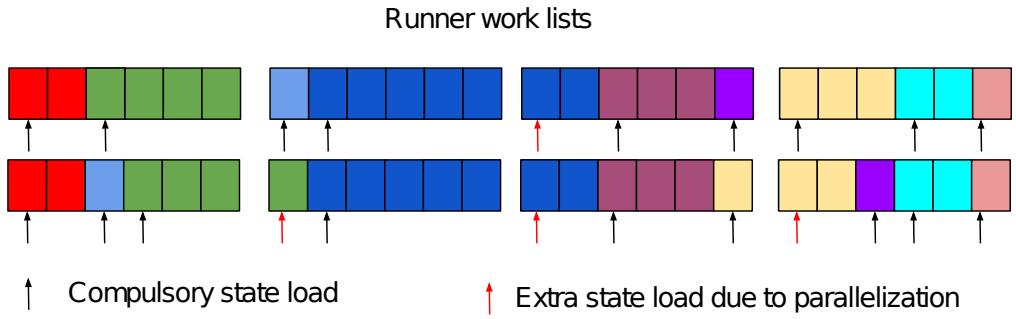



**Figure 5:** Two possible schedules of 24 propagation tasks of equal duration on 4 runners. All particles propagated from the same parent particle state have the same color (9 parents here). Top schedule is optimal with 9 compulsory loads (one per parent), and one for the dark blue parent that cannot fit in one runner. The bottom schedule, with 2 more sate loads, is a possible one that our on-line scheduling algorithm can produce. This is not optimal but still below the general $Q+R-1$ bound as the algorithm ensures that no more than $R-1$ "color cuts" occur and avoids the same runner loads more than once a given parent particle state.

### *3.5.1. Static Scheduling*

Let $R$ be the number of runners. Let $P$ be the number of particles to propagate. Resampling may lead to have some parent particles drawn to be propagated several times. Let $Q$ the number of parent particles $q$, and $\alpha_q$ the number of times the particle $q$ needs to be propagated. The total number of particles to propagate is:

$$P = \sum_{0 \leq q < Q} \alpha_q \tag{13}$$

To assess the performance of our scheduling algorithm, we first derive a lower and upper bound of the minimum number of particle loads $c^*$ for the static case, where: (i) runners do not cache states, (ii) the number of runners is constant, and (iii) all particle propagations take the same amount of time. Under these conditions, each runner propagates $\frac{P}{R}$ particles. Without local cache, each parent particle $q$ needs to be loaded at least once. Therefore, the number of compulsory particle loads is $Q$. If $\alpha_q = 1$ for all $q$, that is, every particle is drawn only once, then $c^* = P$. Otherwise, parallelizing the propagation on $R$ runners may require some particles to be loaded by more than one runner, accounting for extra particle loads beyond the compulsory ones. Indeed, each particle $q$ needs to be provided at least on $s_q$ runners,

where

$$s_q = \left\lceil \frac{\alpha_q}{\frac{P}{R}} \right\rceil. \quad 14$$

Distributed to $R$ runners, the list of $P$ particles is cut $R-1$ times. Consequently, the extra particle loads are at most $R-1$. This is visualized in Figure 5. This upper bound occurs if all particles are propagated from a single parent($Q=1$). Thus, the minimum number of particle loads is tightly bound by

$$Q \leq c^* \leq Q + R - 1. \quad 15$$

We can apply a static schedule that respects the upper bound: distribute $\frac{P}{R}$ particles per runner, where each parent particle $q$ is given to no more than $s_q$ runners, and by imposing that runners do not switch to the next particle before completing all propagations associated to the current one.

*3.5.2. Dynamics List Scheduling*

However, we target a more general case. We soften the initial assumptions now considering that the number of runners can vary, and the time to propagate particles may vary significantly and is not known beforehand (but we still have no cache). In this context we propose to rely on the classical dynamic list scheduling algorithm: when idle, a runner requests work from the server that returns a particle-id to propagate. In the general case the list scheduling algorithm guarantees to be at worst twice as long as the optimal schedule that requires to know the particle propagation time in advance [13, 14]. Instead of blindly selecting the next particle to propagate, we adapt the static scheduling strategy for particle selection with the goal of limiting the number of particle loads. The scheduling is based on the split factor $s_q$ (Equation 14). However, we adapt the static scheduling to the dynamic case by recomputing $s_q$ each time with the updated values of $\alpha_q$, $P$, and $R$. To implement this algorithm on the server, we need a bookkeeping of the number of runners $R_q$ currently propagating particle $q$, and the number $\alpha_q$ of remaining propagations for particle $q$. Let $r$ be the runner requesting a particle for propagation, the particle distribution algorithm works as follows:

1: If $\alpha_q > 0$ for particle $q$ last propagated by $r$, decrement $\alpha_q$ and assign $q$ again. If $\alpha_q = 0$ continue with (2);

2: Select a different particle $q'$ with $\alpha_{q'} > 0$;

3: Compute split factor $s_{q'}$. If $R_{q'} < s_{q'}$ assign $q'$, increment $R_{q'}$, and decrement $\alpha_{q'}$. If $R_{q'} = s_{q'}$ continue with (2).

Notice that when the server recognizes the loss of one runner, it needs to update the bookkeeping to reintegrate the particle that this runner was propagating.

In conditions of even propagation time and a static number of runners, this algorithm leads to the same distribution as for the static schedule and respects the upper bound of Equation 15.

*3.5.3. Cache Aware Scheduling*

We now remove the last assumption to propose a scheduling strategy that takes into consideration the particle cache. This is a heuristic build upon the previous strategy and validated though several experiments. The particle selection strategy is:

1. Select a parent particle $p_i$ already loaded in the runner cache (cache hit);
2. Select a parent particle $p_i$ that is in no runner cache (cache miss);
3. Select a particle $p_i$ fulfilling the split factor criterion (cache miss);
4. Select a parent particle $p_i$ with maximal split factor $s_i$ (even if voids the split factor) (cache miss).

The three first items comply with the scheduling proposed in Section 3.5.2. The first item gives priority to particles already in the cache, before they may be evicted to provide space for a particle load. The next two items pursue with the strategy of Section 3.5.2, favoring particles with no previous propagation. The rational is to start as soon as possible with new parent particles and, once in a cache, propagate them has often as required, and intend to reduce the need for splitting. The last item departs from the strategy of Section 3.5.2, but its addition proved efficient by our experiments. This case occurs when reaching the end of a cycle. It proved to be an efficient strategy to keep runners busy, even at the cost of extra loads, to improve load balancing and so completion time.

*3.6. Job Submission and Monitoring*

The workflow is controlled by the **launcher**. The launcher is the user entry point to configure and start the application. The launcher starts first and is responsible to start and monitor the runner and server instances, that all run in separate executables/jobs. The launcher is also in charge of killing and restarting the runners or server as requested by the fault tolerance protocol (Section 3.7), or for elasticity purpose.

The launcher tightly interacts with the job scheduler (Slurm or OAR for instance) of the machine. The launcher can be configured to submit one job per runner and server to the batch scheduler. This strategy offers the maximum flexibility for the batch scheduler to optimize the machine ressource allocation, but the execution progress becomes very dependent on the machine availability. The user may need more control on the number of concurrently running runners. In that case the launcher can be set to request to the batch scheduler one or several large resource allocations and fit several runner instances in each one. To support this feature the launcher relies on a combination of Slurm salloc/srun [15], or OAR containers[16]. For even more flexible schemes, we plan to support workflow pilot-based schedulers like Radical-Pilot [17] or QCG-PilotJob [18].

*3.7. Fault Tolerance*

The fault tolerance relies on the fast identification of failures from runner and server instances. Runner failures are detected in two different ways. Runner crashes are recognized by the launcher, which is monitoring their execution using the cluster scheduler. Unresponsive runners are identified by the server relying on timeouts for the particle propagations. If propagations exceed the timeout, the server requests the launcher to terminate the respective runner. In both cases, the launcher eventually starts a new runner instance. The new runner connects to the server and requests work. If a runner fails, the server cancels the on-going propagation, and the time spent in the propagation plus the time to recognize the runner failure is lost.

Server failures are detected similarly, either directly if the server crashes, or if the server exceeds a timeout. The timeout is mediated by a periodical exchange of signals between launcher and server (heartbeats). If the server fails, the launcher terminates all runner instances and restarts the framework as a whole. The server frequently stores the status of the propagations in checkpoints, and in case of failures, the framework can recover to the point of the last successful propagations.

Finally, a launcher failure is detected by the server monitoring the heartbeat connection between launcher and server. In case of a missing heartbeat, the server checkpoints the current particle state and shuts down. In parallel, the runners detect the server crash and shut down, again using timeouts. While runner or server failures lead to an automatic restart, the framework needs to be restarted manually if the launcher fails.

### *3.8. Implementation Details*

The launcher and server are developed in Python. The runner relies on the simulation code instrumented with our framework API, supporting C/C++, Fortran and Python. The implementation reuses some software components, like the launcher, from the framework developed for EnKF DA [19]. The distributed cache implementation relies on the Fault Tolerance Interface (FTI) [20]. FTI is a multilevel checkpoint-restart library supporting asynchronous checkpointing to global storage. One of the main modification performed to FTI is related to its event loop. Events are triggered in form of MPI communication between the application and FTI processes. The events are identified by tags. To extend this mechanism, we enabled to register a callback function. This callback function is called inside the event loop and can trigger user defined events using unique tags. With this, it becomes possible to use the application checkpointing into all available levels of reliability FTI provides, and to implement the cache mangement on the dedicated FTI processes.

The communication between helper and model processes relies on asynchronous MPI messages. Communications with the server are implemented in two steps for efficiency purpose. Only rank 0 (master) of the application (i.e., model) communicator and the rank 0 (master) of the helper process communicator communicate with the server. As a dynamic connection is needed, each master connects to the server using a socket through the ZMQ library. Information that needs to be propagated between helper or model processes relies on MPI collective communications in the associated communicator (Figure 3).

The framework code is available at `https://gitlab.inria.fr/melissa/melissa-da`.

## 4. Experiments

*4.1. WRF Use Case*

Experiments rely on an established Numerical Weather Prediction (NWP) system; the Weather Research and Forecasting Model (WRF) (V3.7.1)[8]. The core of WRF is based on solving fully compressible non-hydrostatic equations with complete Coriolis and curvature terms, and a large set of physics options. The simulation domain covers most of Europe (See Figure 6) and is composed of 220 by 220 grid cells with a horizontal resolution of 15 km and 49 vertical levels with uneven thickness. We perform one day-ahead weather forecasting (24 hours of initial time plus 48 hours of production time) of an arbitrary date (2018-10-12) with 24-seconds or 100-seconds time steps. The model employs the WSM6 microphysics, MYNN2 boundary layer physics, Grell-3 cumulus parameterization, Eta Monin-Obukhov similarity surface layer processes, and RUC land surface model. Also non-hydrostatics are activated to provide more details in simulated clouds and precipitation. The input, initial, and boundary conditions are based on the reanalyzed ERA5 dataset from the European Center for Medium-Range Weather Forecasts (ECMWF), which is updated every 3 hours. Data assimilation is performed using the cloud fraction (CFRACT). The particle weights are determined by comparison against the observed cloud mask obtained from the EUMETSAT Level-2 satellite data of the cloud mask. The simulated cloud fraction is converted into cloud mask and the observed cloud mask data is upscaling to the size of the model gridcells for the further applications. The data is hourly available, thus, one assimilation cycle comprises 150 (36) model time steps ($150 \times 24\,\mathrm{s} \mathrel{\widehat{=}} 1\,\mathrm{h}$ or $36 \times 100\,\mathrm{s} \mathrel{\widehat{=}} 1\,\mathrm{h}$).

The experiments presented in this article leverage our proposed Particle Filtering (PF) implementation with a sample size of up to 2,555 particles on the European domain. In that case, we utilize 20,442 compute cores on 512 Nodes of the Jean-Zay supercomputer. The compute nodes are equipped with 2 Intel Cascade Lake 6,248 processors, summing up to 40 cores with 2.5 GHz and 192 GiB RAM per node. Intel Omni-Path (100 GB/s) connects the compute nodes, and the global file system is an IBM Spectrum Scale (ex-GPFS) parallel file system with SSD disks (GridScaler GS18K SSD). For all experiments the node-local caches were stored on RAM disk. In Table 1 we list the parameters of our main experiments.

The meteorological state of the European domain associated to one particle comprises 2.5 GiB of data. Hence, the data from 2,555 particles for the full simulation period of 48h (48 time steps) correspond to an aggregated size of about 300 TiB. The experiments performed for this article, including

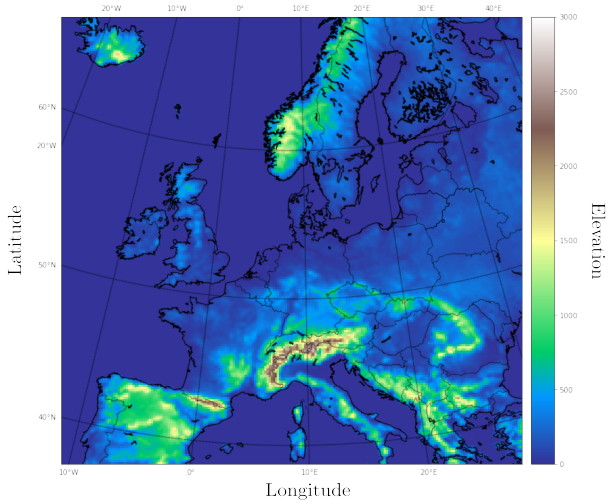



**Figure 6:** The topography of the target domain of Europe for the simulation.

small beta-stage experiments, account for about 900,000 CPU hours split between the JUWELS, Jean-Zay and Marenostrum supercomputers.

### *4.2. Runner Activity*

The benefit of the local cache in combination with the cache-aware scheduling leads to a drastic reduction in transfers from global to local file system layers. The cache hit ratio, i.e., the ratio of particles found inside the cache to the total number of particle loads, depends on the cache size and the ratio of particles per runner. Figure 7 shows how the cache misses develop for different cache sizes. Our experiments demonstrate a cache hit ratio of 88 % for 128 particles, 32 runners, and a cache size of 9 particles. This translates to a saving of 88 % in transfers from global to local storage. The pattern of cache hits and misses is visualized in Figure 8. The initial phase is dominated by starting up the runners, and all the particles are fetched from the global storage (cache warm up). But beginning with the next assimilation cycle, the low transfer ratio from global to local storage starts to establish.

Runners are designed to separate I/O operations to the PFS from other tasks: model processes only perform local I/O operations. We observe in our experiments that this leads to a high computational efficiency. The local I/O accesses are negligible compared to the computational tasks ($< 0.1$ s compared to up to 6 s). Some general idle periods can be observed between assimilation cycles when runners are waiting for the last propagations of one

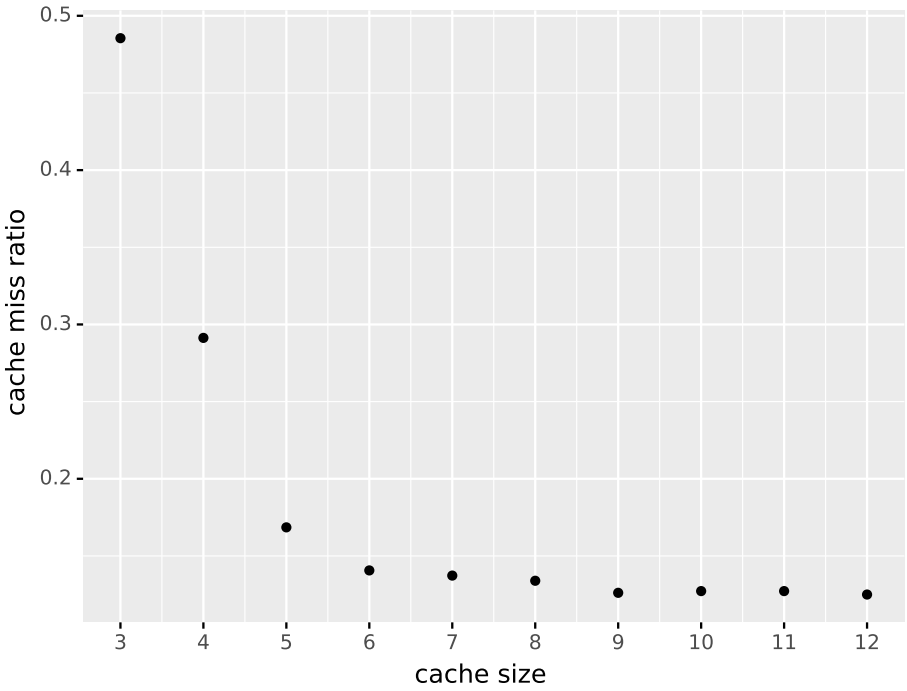



**Figure 7:** Cache miss ratio for different cache sizes on each runner. In total 128 particles run on 32 runners. First and last assimilation cycles were disregarded to remove warm up effects and not fully recorded cycles.

| Experimental Setup | | | | |
|---|---|---|---|---|
| Particles | 315 | 635 | 1,275 | 2,555 |
| Number of runners | 63 | 127 | 255 | 511 |
| Number of nodes | 64 | 128 | 256 | 512 |
| Model processes | 2,457 | 4,953 | 9,945 | 19,929 |
| Particles per runner (avg.) | 5 | 5 | 5 | 5 |
| Particle state size (GiB) | 2.5 | 2.5 | 2.5 | 2.5 |
| **Performance Data** | | | | |
| Scaling efficiency | 92% | 91% | 92% | 87% |
| Resampling (ms) | 2.21 | 4.06 | 8.16 | 16.37 |
| Assimilation cycle (s) | 136 | 138 | 139 | 146 |
| Propagation (s) | 25.1 | 25.2 | 25.1 | 25.0 |
| Load particle state from PFS to cache (s) | 2.1 | 2.1 | 2.4 | 4.1 |
| Write particle state from cache to PFS (s) | 1.4 | 1.6 | 1.8 | 2.3 |
| Writes to PFS per cycle (TiB) | 0.77 | 1.55 | 3.11 | 6.24 |
| Reads from PFS per cycle (TiB) | 0.30-0.40 | 0.64-0.79 | 1.27-1.79 | 2.54-3.82 |

**Table 1:** Experimental setting and performance overview at 4 different scales. The times are given as average in all cases. Model time steps were set to 100 seconds.

cycle to finish so that the server can normalize weights, resample and start to distribute work again. This is illustrated in Figure 9 where we show a trace recorded from the execution of an arbitrary runner. The trace illustrates the efficiency of the runners in performing the actual tasks of the simulation, particle propagation and weight calculation, while the I/O tasks are moved to the background.

A global parallelization of the computational tasks is achieved by dynamically distributing the particle propagations to the available runners. The fully parallelized case corresponds to $R = P$, i.e., the number of runners matches the number of particles. The sequential case corresponds to $R = 1$, i.e., all propagations are performed by only one runner. However, The best parallel efficiency is achieved at values between those limits. Because WRF propagates particles with very low time variability (maximum variation of 10%), we observe an even distribution of propagations to runners when $R$ divides $P$ (Figure 8). A single-particle propagation takes between 24 and 26.5 seconds, globally making from 87% to 92% of an average assimilation cycle. Calculating weights takes 1% of the time and communicating with the server from

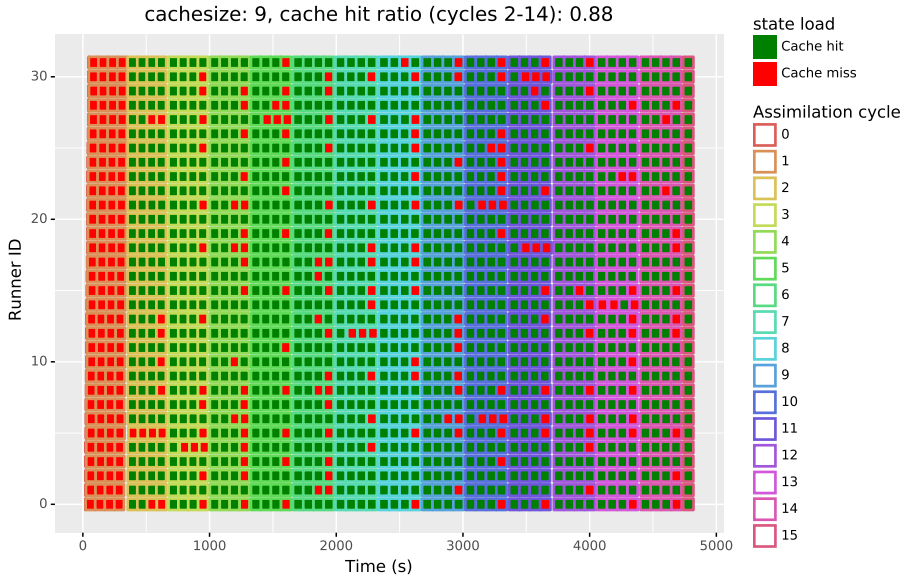



**Figure 8:** Gantt chart of particle propagations executed by the 32 runners over 15 assimilation cycles. Tasks are green if the associated parent particle state was already present in the runner cache and did not require a load from the PFS (red otherwise).

7%to 12% including the idle time at the end of each cycle (Table 1 – *Performance Data*). The extra resources for helper processes, one core per runner node, and the server, 1 node, comprise only 2.7% for our largest experiment at 512 nodes. On the other hand, leveraging the runner's particle cache, and the cache aware dynamic scheduling on the server, move $> 97\%$ of the state loads completely into the background. Loading and writing particle states synchronously would otherwise add about 6.4 seconds to each single-particle propagation corresponding to 14% of the average propagation time (Table 1 – *2,555 particles*).

Note that in contrast to the traditional offline approach, we start-up the numerical simulation code only once for several particle propagations. The setup involves the request and allocation of the runner job and initializing the simulation code. On the other hand, the traditional offline approach associates each particle propagation with a different job, and the start-up has to be performed anew for each particle. Starting up the WRF model on the European domain on one node until the first model propagation begins takes 3-4 s, excluding the provisioning of the job allocation via the cluster

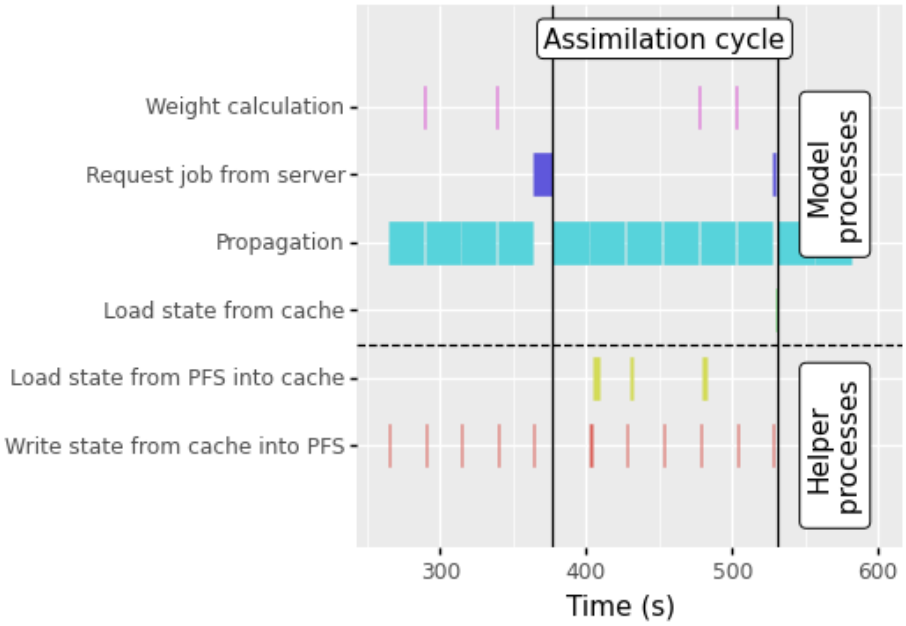



**Figure 9:** Trace detailing the activity of a runner over the course of an assimilation cycle. Helper processes enable to keep model processes busy with particle propagation, except at the end of assimilation cycles when they wait for the server to finish particle resampling (dark blue). Some activities are so thin that they are not visible here (state copies from cache to model).

scheduler.

### *4.3. Server Activity*

We further measured the server responsivity to runner requests. The response time is always in the order of a few hundred microseconds, except for some job requests that take up to a few seconds (Figure 10). However, these are outliers at the end of the assimilation cycle, resulting from idle times due to the load balancing and the particle filter update. During our largest experiments with 511 runners, the server processes 676 requests per second at maximum load. This shows that the server is fast enough to support this scale, even though it is a sequential python code. Simple optimizations are at reach if the server needs to be accelerated (e.g., adding parallelization).

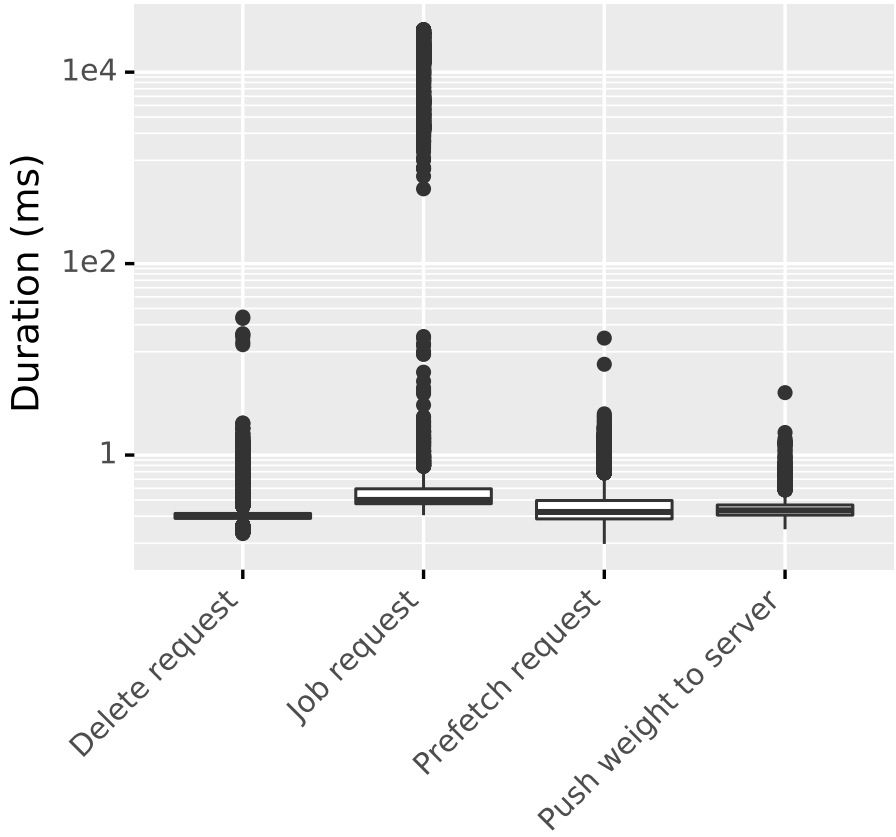



**Figure 10:** Server response times on runner requests.

### *4.4. State Transfers To/From PFS*

Next, we take a closer look at the particle loads from the PFS (Figure 11). With a sample size of 1024 particles, leveraging 256 runners, and a local cache size of 9 particles, between 121 and 248 particles are loaded to the cache during each cycle. The number of distinct parent particles $Q$ propagated per cycle lies between 813 and 889. Each one is propagated at most 5 times to sum to a total of 1024 particles. The cache enables to achieve significantly less loads than the $Q + R - 1$ upper bound expected with static scheduling and no cache (Equation 15).

The access times to the Parallel File System (PFS) (load/store) vary significantly and increase with the number of runners (Figure 12), showing that

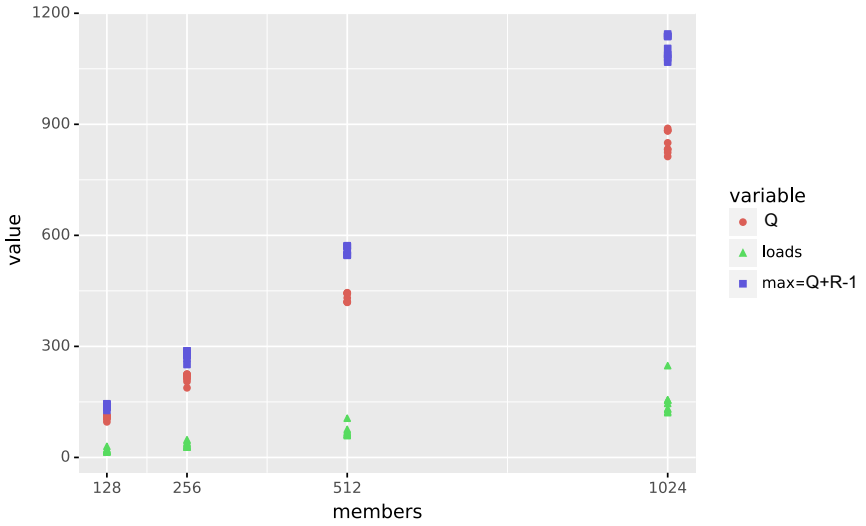



**Figure 11:** Number of parent particles $Q$, particles loads from the PFS to the cache, and $Q + R - 1$ upper bound from Equation 15 for different ensemble sizes, a cache size of 9 particles with 4 particles per runner.

our application alone can stress the PFS [1]. Each particle is associated with 2.5 GiB of data. During each assimilation cycle, all the propagated particles are written to the PFS for supporting fault-tolerance and dynamic load balancing. For our experiments at 512 nodes with 2,555 particles, this accumulates to about 6.2 TiB of data each cycle (compare Table 1). However, our experiments on the Jean-Zay and JUWELS supercomputer demonstrate that our framework performs most of those transfers asynchronously (Section 4.2). In less than 2% of the cases, the model processes wait more than 0.1 seconds for a particle to be loaded corresponding to cases where helper processes do not (entirely) finish prefetching. Time to perform the local loads and stores from the cache shows a constant average independently on the number of runners (Figure 13).

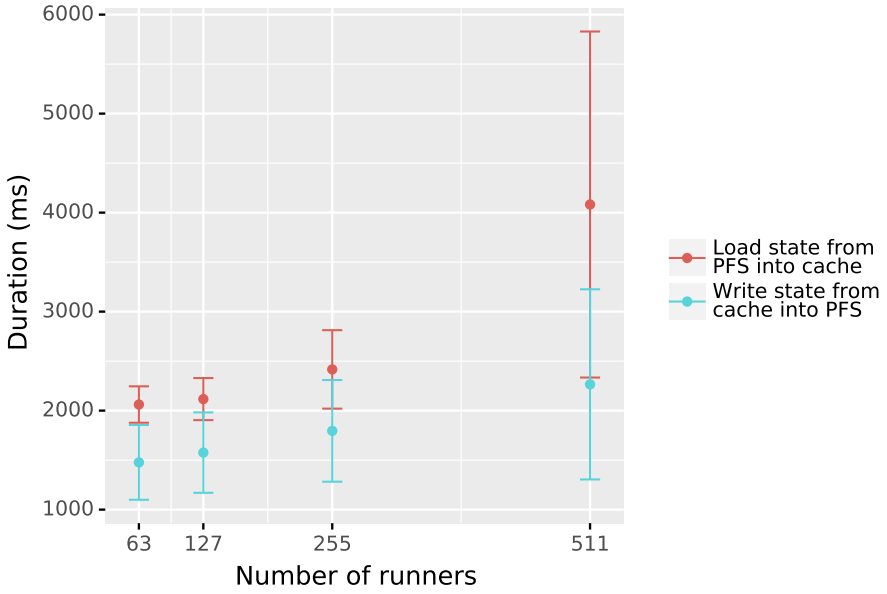



**Figure 12:** Mean time to load or store particle states of 2.5 GiB from / to the PFS with different numbers of runners.

### *4.5. Fault Tolerance, Elasticity and Load Balancing*

Fault tolerance relies on 1) persisting the particle to the PFS 2) the framework elasticity enabling to adjust dynamically the number of runners.

[1] these numbers may also be impacted by other jobs on the cluster

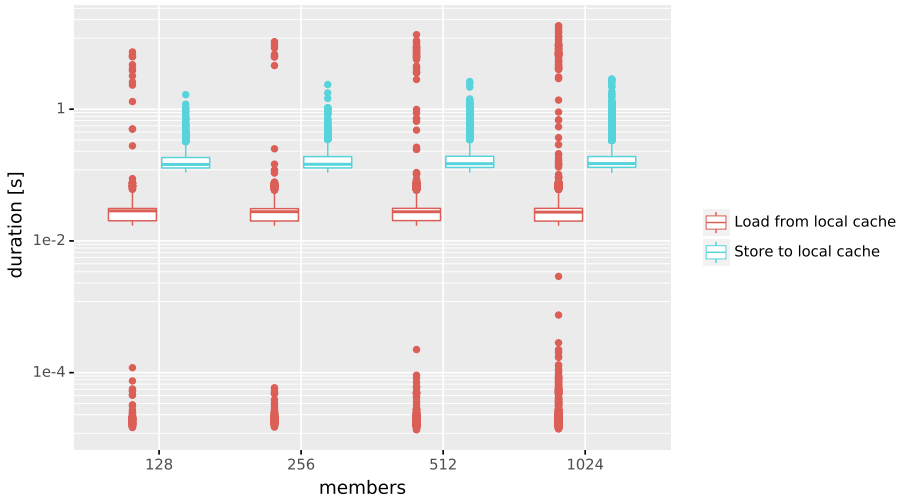



**Figure 13:** Box plot of the time spent for loads and stores from/to the local cache with different numbers of particles.

If a runner fails, the launcher requests the execution of a new one, so as to maintain a constant number of runners. Once this new runner connects to the server, it asks for a particle to propagate to the server, assigned according to the load balancing algorithm.

We tested the fault tolerance and elasticity on an experiment with 63 runners provoking the crash of 2 runners (Figure 14). First, notice that the fault tolerance algorithm reacts appropriately as it restarts a new runner after each crash. The first crash (runner #53) occurs in the worst-case situation: just when propagating the last particle of the current cycle, leading to a significant idle period. The idle period is caused first, because the server needs to wait for the timeout (set to 60 s) to acknowledge that runner #53 is unresponsive and second, there is no work left except the particle that runner #53 was propagating, which is re-assigned to runner #44. Meanwhile all other runners stay idle until the beginning of the next cycle. If the crash happens earlier during a cycle, smaller idle periods appear. This can be observed during the second crash (runner #48), as the other runners are kept busy with propagation work. We generally observe an efficient load balancing, as the work load is kept well distributed amongst runners, even when their number varies.

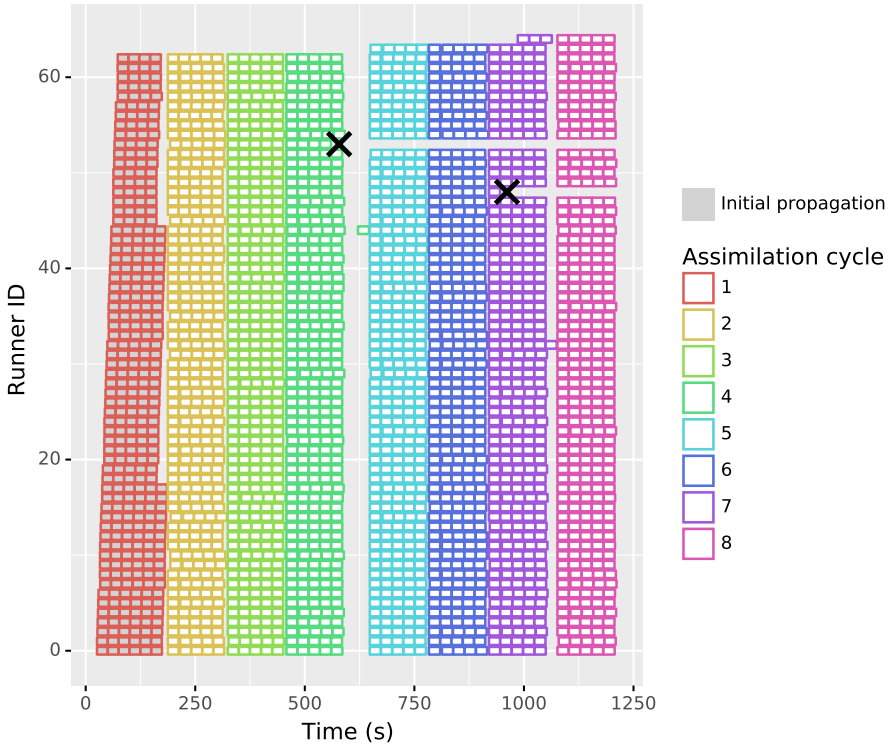



**Figure 14:** Gantt chart of particle propagations executed by the 63 runners over 8 assimilation cycles. After runners #48 and #53 crashed (black cross), two new ones restarted (top 2 runners #63 and #64).

### *4.6. Scaling*

We evaluated the performance of the particle filter in a strong scaling scenario, constant number of runners while increasing the number of particles, and a weak scaling scenario, constant particle-to-runner ratio while increasing the number of runners. In the strong scaling scenario we observe that the runner utilization shows an upwards trend when increasing the number of particles per runner, with a plateau at about 96% (Figure 15). As global I/O operations are almost completely shadowed, thanks to the asynchronous prefetching, increasing the number of particles per runner mainly enables to better amortize the cost of the synchronization associated with resampling. We observe an almost constant time for the assimilation cycle, demonstrating a desirable weak scaling behavior. The time for the cycles increase only by 8% from 63 to 511 runners, indicating an efficient scaling behavior of the

framework up to production scale (Figure 16). Particle filtering with WRF on a European domain for short-range weather prediction at this scale is an important advancement of the previous work done by Berndt et. al. [21]. Moreover, besides assimilating at a higher frequency, our proposal offers fault tolerance, automatic load balancing and elasticity while minimizing the I/O cost and time to calculate weights.

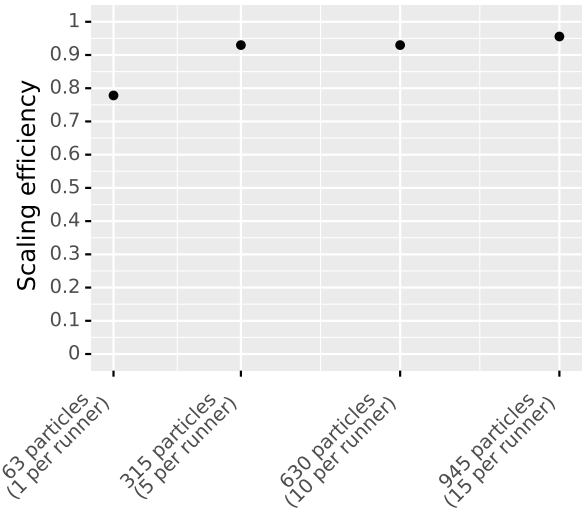



**Figure 15:** Scaling efficiency using different numbers of particles with 63 runners. One runner sets the reference case.

### *4.7. Comparison to a File-based Approach*

| | Melissa | | | ESIAS-met | | | ESIAS-met/Melissa |
|---|---|---|---|---|---|---|---|
| part. | cores | time (s) | core.s/part. | cores | time(s) | core.s/part. | resource usage ratio |
| 128 | 384 | 1062 | 3186 | 1536 | 267 | 3204 | 1.01 |
| 256 | 768 | 1062 | 3186 | 3072 | 317 | 3804 | 1.19 |
| 512 | 1536 | 1068 | 3204 | 6144 | 422 | 5064 | 1.58 |
| 1024 | 3072 | 1071 | 3213 | 12288 | 761 | 9132 | 2.84 |

**Table 2:** Comparing the resource usage (core.second/particle) per cycle for Melissa and ESIAS-met (file-based) runs.

We compare Melissa with the file-based approach ESIAS-met [22] using the same simulation code WRF (V3.7.1) and the same data set. For the same number of particle, both approaches use a very different amount of

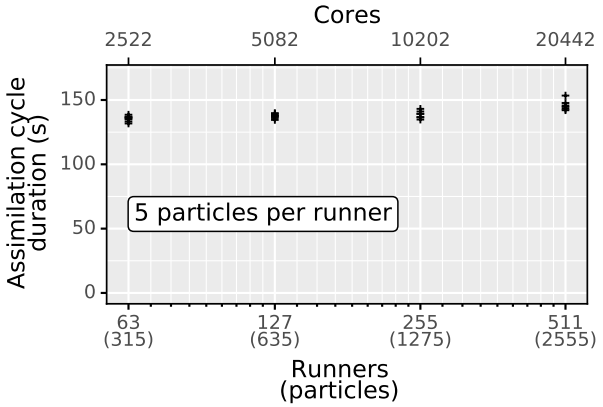



**Figure 16:** Weak scaling performance test: assimilation cycle duration for different numbers of runners, but always 5 particles per runner.

cores (Table 2). With ESIAS-met each particle propagation requires to start a dedicated instance of WRF. Each time it includes the cost from loading and storing the particle state from/to a file. At 1024 particles ESIAS-met uses 12288 cores while Melissa just needs 3072 cores as runners propagate several particles each. ESIAS-met execution time is thus shorter as highly parallelized, but the resource usage (core.second/particle/cycle) is 2.84 times improved for Melissa due to the combined strategies developed to improve efficiency. The gain increases with the number of particles, showing that the Melissa approach is particularly beneficial when targeting the large ensemble size.

## 5. Discussion

in Section 4.1 we derived that the total amount of data resulting from 48 time-steps of particle filtering on the european domain with 2,555 particles accumulates to about 300 TiB. Post processing this amount of data is challenging. Our framework could be extended using in situ data processing techniques as presented in Terraz et. al. [23].

We only considered the case where the propagation time is longer than the time for loading states from the PFS; while applications where propagations are shorter than loading the states would limit our proposal efficiency, as we cannot further hide the I/O cost in that case. On the other hand, we already have short propagation times in the WRF context as we per-

form hourly resampling. We chose this frequency primarily to stress our proposal. Production runs usually do not require such a high frequency, and rather have even longer propagation times as in our experiments. However, to minimize transfer times further, we are evaluating approaches leveraging node-local persistent storage as globally shared storage layer. Solutions for this are readily available in form of distributed ad hoc file-systems [24] such as BeeOND, GekkoFS, and BurstFS. We have also experimented with connecting the runners, establishing a peer to peer network, where runners can exchange directly the required states between each other.

The particle propagation time in our experiments with WRF is relatively even, showing at most a 10% variability. Situations with more variability are possible using different physics in WRF, with other simulation codes, or, if runners execute on heterogeneous resources, some runners propagating faster than others by leveraging GPUs for instance. Also use cases from other contexts such as Simulation Based Inference (SIB) and ensemble classification, which can be performed using our framework, might lead to vastly different propagation times. Therefore, testing our framework under such conditions is an important future work.

Our proposal currently relies on filters that do not compute internal member state corrections. Extending our approach to such particle filters [1] would possibly require aggregating more than just the particle weights to the server. Exploring the requirements to align our framework to such cases is the goal of a future implementation of the particle filter that we propose. We validated our proposal with the SIR particle filter, but many variations exist and are active research topics [25, 26]. One challenge is the exponentially growing required particle number with the dimension of the problem [27, 28]. This is particularly acute for geoscience use cases that, as in this paper, work in high dimensional spaces. The survey [1] gives an extensive overlook of DA by particle filters for geoscience and ways to cope with dimensionality issues.

Particle filters, as used here, require a synchronization point at the end of each assimilation cycle. For our framework, this is the major remaining source of inefficiency. Loosening this requirement needs revisiting the particle filtering algorithm, which constitutes an active topic of research [29, 30, 31].

## 6. Related Work

The DA domain encompasses a large variety of techniques and algorithms, like nudging [32], kriging [33], ensemble Kalman Filter [34], ensemble max-

imum likelihood filter [35], or particle filter [36]. For an overview, we refer to [37, 38]. We focus here on statistical DA relying on an ensemble run of the model to compute a statistical estimator (co-variance matrix for EnKF, PDF for particle filters).

To aggregate the data produced by all members (i.e., particles) two main groups of approaches are used. Either the data is stored to files and then processed in a second step (off-line mode), or the data is processed on-line usually within a large MPI code in charge of running the members and data processing. Frameworks relying on the off-line mode include EnTK [39], with the largest published DA use cases reaching 4,096 members for a molecular dynamics application with an EnKF filter [40]. OpenDA also follows this model, using NetCDF for data exchange with the NEMO code [41]. DART supports both [42], with reports of large scale DA in off-line mode in [43] (about 1,000 members with an oceanic code), or [44, 45] (1,024 member, LETKF filter, 6 M Fugaku cores). File based approaches have the benefit of their simplicity, providing fault tolerance and elasticity. But these solutions do not support member virtualization, state caching and prefetching. So starting or restarting a member requires to request a new resource allocation launching a new instance of the model code with all the associated start-up costs. Node-local persistent storage capabilities, for instance with SSDs, can store intermediate files, avoiding the PFS to loosen the I/O bottleneck. They are used for member state storage in [44], but without specific fault tolerance mechanism. So if a node fails and the node-local storage becomes unavailable, the lost member states need to be recomputed. Besides leveraging the node storage for the distributed cache, using node-storage rather than the parallel file system as a globally shared file system layer is one of our future goals.

The on-line mode avoids the I/O bottleneck. PDAF [46], which supports both modes, has for instance been used on-line for the assimilation of observations into the regional earth system model TerrSysMP. DA was based on EnKF with up to 256 members [47]. ESIAS uses on-line DA via particle filters with up to 4,096 particles on a wind power simulation on Europe [21]. Notice that we work with the same WRF component of ESIAS in this paper, using a configuration on a similar domain but at higher spatial resolution and with more advanced and more time consuming physics. We also find ad hoc MPI codes for on-line DA as in [48] (atmospheric model, 10,240 members, Local ENKF filter, 4,608 compute nodes). But all these MPI approaches lead to monolithic code without support for fault tolerance, elasticity or load balancing. In [49], the authors analyze various particle propagation schedul-

ing but at limited scale (6 compute nodes and 300 particles). We performed experiments on a similar architecture as our proposed one, but for EnKF instead of PF [19]. We demonstrated fault-tolerance, elasticity and scalability for experiments using up to 16 k members, 16 k cores for DA with EnKF for the hydrology code Parflow. In contrast to our novel proposal, EnKF requires a centralized filter update, gathering the full ensemble of states at the central instance for the assimilation of observations. In our novel proposal for PF, we exploit certain properties of particle filters to suppress the server bottleneck and significantly reduce data movements.

## 7. Conclusion

In this article we proposed an architecture for handling very large ensembles for particle filters. The architecture was designed to address the challenge of exascale computing that will allow massive ensemble runs [50]. The architecture is based on a server/runner model where runners support a distributed cache and virtualization of particle propagation, while the server aggregates the weights computed by the runners and ensures the dynamic balancing of the work load. Particle propagation is virtualized so the required number of runners is decoupled from the particle number. With the addition of a distributed checkpointing mechanism, the architecture supports dynamic changes in the number of runners during execution for fault tolerance and elasticity. Experiments with the WRF weather simulation code show that our framework can run at least 2,555 particles on 20,442 cores with a 87% scaling efficiency. Dynamic particle-propagation scheduling and caching enable to avoid 88% of the global I/O operations. Compared to the ESIAS file based approach, Melissa improves resource usage 2.83 times at 1024 particles.

Future work includes experimenting with adaptive or localized particle filters as well as combining particle and Kalman filter. We also plan to extend the distributed cache and fault tolerance algorithm to fully avoid the centralized file system and only rely on node-local SSDs for particle storage.

## Acknowledgement

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). The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC). We acknowledge the access to the meteorological input data from the Meteocloud of SDL Climate Science, JSC. We also acknowledge PRACE for awarding us access to JUWELS at Jülich Supercomputing Centre (JSC), Germany.

## References

[1] P. J. Van Leeuwen, H. R. Künsch, L. Nerger, R. Potthast, S. Reich, Particle filters for high-dimensional geoscience applications: A review, Quarterly Journal of the Royal Meteorological Society 145 (723) (2019) 2335–2365.

[2] D. Lundén, J. Borgström, D. Broman, Correctness of sequential monte carlo inference for probabilistic programming languages., in: ESOP, 2021, pp. 404–431.

[3] F. Ronquist, J. Kudlicka, V. Senderov, J. Borgström, N. Lartillot, D. Lundén, L. Murray, T. B. Schön, D. Broman, Universal probabilistic programming offers a powerful approach to statistical phylogenetics, Communications biology 4 (1) (2021) 1–10.

[4] J.-W. van de Meent, B. Paige, H. Yang, F. Wood, An Introduction to Probabilistic Programming, arXiv, 2018. doi:10.48550/ARXIV.1809.10756.
URL https://arxiv.org/abs/1809.10756

[5] J. F. G. de Freitas, M. Niranjan, A. H. Gee, A. Doucet, Sequential Monte Carlo Methods to Train Neural Network Models, Neural Computation 12 (4) (2000) 955–993. doi:10.1162/089976600300015664.

[6] P. M. Blok, K. van Boheemen, F. K. van Evert, J. IJsselmuiden, G.-H. Kim, Robot navigation in orchards with localization based on Particle filter and Kalman filter, Computers and Electronics in Agriculture 157 (2019) 261–269. doi:10.1016/j.compag.2018.12.046.

URL https://www.sciencedirect.com/science/article/pii/S0168169918315230

[7] P. Bauer, P. D. Dueben, T. Hoefler, T. Quintino, T. C. Schulthess, N. P. Wedi, The digital revolution of earth-system science, Nature Computational Science 1 (2) (2021) 104–113.

[8] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. Barker, M. G. Duda, J. G. Powers, A description of the advanced research wrf version 3, Tech. Rep. No. NCAR/TN-475+STR, University Corporation for Atmospheric Research (2008).

[9] J. V. Candy, Bootstrap Particle Filtering, IEEE Signal Processing Magazine 24 (4) (2007) 73–85, conference Name: IEEE Signal Processing Magazine. doi:10.1109/MSP.2007.4286566.

[10] N. Gordon, D. Salmond, A. Smith, Novel approach to nonlinear/non-Gaussian Bayesian state estimation, IEE Proceedings F Radar and Signal Processing 140 (2) (1993) 107. doi:10.1049/ip-f-2.1993.0015.
URL https://digital-library.theiet.org/content/journals/10.1049/ip-f-2.1993.0015

[11] J. S. Liu, R. Chen, T. Logvinenko, A Theoretical Framework for Sequential Importance Sampling with Resampling, in: A. Doucet, N. de Freitas, N. Gordon (Eds.), Sequential Monte Carlo Methods in Practice, Statistics for Engineering and Information Science, Springer, New York, NY, 2001, pp. 225–246. doi:10.1007/978-1-4757-3437-9_11.
URL https://doi.org/10.1007/978-1-4757-3437-9_11

[12] M. Bolic, P. Djuric, Sangjin Hong, New resampling algorithms for particle filters, in: 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03)., Vol. 2, IEEE, Hong Kong, China, 2003, pp. II–589–92. doi:10.1109/ICASSP.2003.1202435.
URL http://ieeexplore.ieee.org/document/1202435/

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

[14] D. Shmoys, J. Wein, D. Williamson, Scheduling parallel machines online, in: [1991] Proceedings 32nd Annual Symposium of Foundations of

Computer Science, IEEE Comput. Soc. Press, San Juan, Puerto Rico, 1991, pp. 131–140.

[15] The slurm workload manager - https://slurm.schedmd.com/.

[16] The oar resource and task manager - http://oar.imag.fr/.

[17] A. Merzky, M. Turilli, M. Titov, A. Al-Saadi, S. Jha, Design and performance characterization of radical-pilot on leadership-class platforms (2021). doi:10.48550/ARXIV.2103.00091.
URL https://arxiv.org/abs/2103.00091

[18] Qcg-pilotjob - https://github.com/psnc-qcg/QCG-PilotJob.

[19] S. Friedemann, B. Raffin, An elastic framework for ensemble-based large-scale data assimilation, The international journal of high performance computing applications 36 (4) (2022) 543–563.
URL https://hal.inria.fr/hal-03017033

[20] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, S. 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, 2011, pp. 1–12.

[21] J. Berndt, On the predictability of exceptional error events in wind power forecasting —an ultra large ensemble approach—, Ph.D. thesis, Universität zu Köln (2018).

[22] Y.-S. Lu, G. Good, H. Elbern, Optimization of weather forecasting for cloud cover over the european domain using the meteorological component of the ensemble for stochastic integration of atmospheric simulations version 1.0 1–31Publisher: Copernicus GmbH. doi:10.5194/gmd-2022-118.
URL https://gmd.copernicus.org/preprints/gmd-2022-118/#discussion

[23] T. Terraz, A. Ribes, Y. Fournier, B. Iooss, B. Raffin, Melissa: Large scale in transit sensitivity analysis avoiding intermediate files, in: International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17), Denver, 2017.

[24] A. Brinkmann, K. Mohror, W. Yu, P. Carns, T. Cortes, S. A. Klasky, A. Miranda, F.-J. Pfreundt, R. B. Ross, M.-A. Vef, Ad Hoc File Systems for High-Performance Computing, Journal of Computer Science and Technology 35 (1) (2020) 4–26. doi:10.1007/s11390-020-9801-1. URL https://doi.org/10.1007/s11390-020-9801-1

[25] A. Doucet, S. Godsill, C. Andrieu, On sequential monte carlo sampling methods for bayesian filtering, Statistics and computing 10 (3) (2000) 197–208.

[26] S. C. Surace, A. Kutschireiter, J.-P. Pfister, How to avoid the curse of dimensionality: Scalability of particle filters with and without importance weights, SIAM review 61 (1) (2019) 79–91.

[27] C. Snyder, T. Bengtsson, M. Morzfeld, Performance Bounds for Particle Filters Using the Optimal Proposal, MONTHLY WEATHER REVIEW 143 (2015) 12.

[28] P. Fearnhead, H. Künsch, Particle Filters and Data Assimilation, Annual Review of Statistics and Its Application 5 (1) (2018) 421–449, arXiv: 1709.04196.

[29] C. Vergé, C. Dubarry, P. Del Moral, E. Moulines, On parallel implementation of sequential monte carlo methods: the island particle model, Statistics and Computing 25 (2) (2015) 243–260.

[30] A. Jasra, A. Lee, C. Yau, X. Zhang, The alive particle filter, arXiv preprint arXiv:1304.0151 (2013).

[31] V. Elvira, J. Míguez, P. M. Djurić, Adapting the number of particles in sequential monte carlo methods through an online scheme for convergence assessment, IEEE Transactions on Signal Processing 65 (7) (2016) 1781–1794.

[32] V. R. N. Pauwels, R. Hoeben, N. E. C. Verhoest, F. P. De Troch, The importance of the spatial patterns of remotely sensed soil moisture in the improvement of discharge predictions for small-scale basins through data assimilation, Journal of Hydrology 251 (1) (2001) 88–102.

[33] J. L. Williams, R. M. Maxwell, Propagating subsurface uncertainty to the atmosphere using fully coupled stochastic simulations, Journal of Hydrometeorology 12 (4) (2011) 690–701.

[34] D. Zhang, H. Madsen, M. E. Ridler, J. Kidmose, K. H. Jensen, J. C. Refsgaard, Multivariate hydrological data assimilation of soil moisture and groundwater, Hydrol. Earth Syst. Sci. 20 (10) (2016) 4341–4357.

[35] S. Q. Zhang, M. Zupanski, A. Y. Hou, X. Lin, S. H. Cheung, Assimilation of precipitation-affected radiances in a cloud-resolving wrf ensemble data assimilation system, Monthly Weather Review 141 (2) (2013) 754–772.

[36] van Leeuwen, J. P., A variance-minimizing filter for large-scale applications, Mon. Wea. Rev. 131 (9) (2003) 2071–2084.

[37] M. Asch, M. Bocquet, M. Nodet, Data assimilation: methods, algorithms, and applications, Vol. 11, SIAM, 2016.

[38] G. Evensen, Data assimilation: the ensemble Kalman filter, Springer Science & Business Media, 2009.

[39] V. Balasubramanian, M. Turilli, W. Hu, M. Lefebvre, W. Lei, G. Cervone, J. Tromp, S. Jha, Harnessing the power of many: Extensible toolkit for scalable ensemble applications, in: IPDPS 2018, 2018.

[40] V. Balasubramanian, T. Jensen, M. Turilli, P. Kasson, M. Shirts, S. Jha, Adaptive ensemble biomolecular applications at scale, SN Computer Science 1 (2) (2020) 1–15.

[41] N. van Velzen, M. U. Altaf, M. Verlaan, OpenDA-NEMO framework for ocean data assimilation, Ocean Dynamics 66 (5) (2016) 691–702.

[42] J. Anderson, T. Hoar, K. Raeder, H. Liu, N. Collins, R. Torn, A. Avellano, The Data Assimilation Research Testbed: A Community Facility, Bulletin of the American Meteorological Society 90 (9) (2009) 1283–1296, publisher: American Meteorological Society.

[43] H. Toye, S. Kortas, P. Zhan, I. Hoteit, A fault-tolerant hpc scheduler extension for large and operational ensemble data assimilation: Application to the red sea, Journal of Computational Science 27 (2018) 46 – 56.

[44] H. Yashiro, K. Terasaki, Y. Kawai, S. Kudo, T. Miyoshi, T. Imamura, K. Minami, H. Inoue, T. Nishiki, T. Saji, M. Satoh, 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), IEEE Computer Society, Los Alamitos, CA, USA, 2020, pp. 1–10.

[45] H. Yashiro, K. Terasaki, T. Miyoshi, H. Tomita, Performance evaluation of a throughput-aware framework for ensemble data assimilation: The case of nicam-letkf, Geoscientific Model Development 9 (7) (2016).

[46] L. Nerger, W. Hiller, Software for ensemble-based data assimilation systems—implementation strategies and scalability, Computers & Geosciences 55 (2013) 110–118.

[47] W. Kurtz, G. He, S. J. Kollet, R. M. Maxwell, H. Vereecken, 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) (2016) 1341–1360.

[48] T. Miyoshi, K. Kondo, T. Imamura, The 10,240-member ensemble Kalman filtering with an intermediate AGCM: 10240-MEMBER ENKF WITH AN AGCM, Geophysical Research Letters 41 (14) (2014) 5264–5271.

[49] F. Bai, F. Gu, X. Hu, S. Guo, Particle routing in distributed particle filters for large-scale spatial temporal systems, IEEE Transactions on Parallel and Distributed Systems 27 (2) (2015) 481–493.

[50] T. C. Schulthess, P. Bauer, N. Wedi, O. Fuhrer, T. Hoefler, C. Schar, Reflecting on the Goal and Baseline for Exascale Computing: A Roadmap Based on Weather and Climate Simulations, Computing in Science & Engineering 21 (1) (2019) 30–41.