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

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

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

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

[a] *Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France*
[b] *Barcelona Supercomputing Center, 08034 Barcelona, Spain*
[c] *Jülich Supercomputing Centre, Forschungzentrum Juelich, 52428 Juelich, Germany*
[d] *Institute of Energy and Climate Research – Troposphere (IEK-8), Forschungszentrum Jülich, 52428 Juelich, Germany*

Abstract

Several ensemble-based Data Assimilation (DA) methods rely on a propagate/update cycle, where a potentially compute intensive simulation code propagates multiple states for several consecutive time steps, that are then analyzed to update the states to be propagated for the next cycle. In this paper we focus on DA methods where the update can be computed by gathering only lightweight data obtained independently from each of the propagated states. This encompasses particle filters where one weight is computed from each state, but also methods like Approximate Bayesian Computation (ABC) or Markov Chain Monte Carlo (MCMC). Such methods can be very compute intensive and running efficiently at scale on supercomputers is challenging.

This paper proposes a framework based on an elastic and fault-tolerant runner/server architecture minimizing data movements while enabling dynamic load balancing. Our approach relies on runners that load, propagate and store particles from an asynchronously managed distributed particle cache permitting particles to move from one runner to another in the background while particle propagation proceeds. The framework is validated

[*] These authors contributed equally to this work
[**] 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)

with a bootstrap particle filter with the WRF simulation code. We handle up to 2,555 particles on 20,442 compute cores. Compared to a file-based implementation, our solution spends up to 2.84 less resources (cores×seconds) per particle.

Keywords: 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 the form of a Probability Density Function (PDF). **Particle Filters (PF)** are 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** based on their likelihood given the observations. The weights are used to generate a new sample of particles representing the state estimate including information obtained by the observations. The process repeats while observations are available.

PF 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 stand by their ability to work with nonlinear and/or non-Gaussian state space models and non-Gaussian observation errors in opposition to techniques like Ensemble Kalman Filtering (EnKF) [7]. But the versatility of PF comes at the price of high sensitivity to the curse of dimensionality [8, 9, 10]: the number of particles needs to grow exponentially with the effective dimension of the assimilation problem. Dynamical systems running parallel high-dimensional numerical model solvers, as some geoscience applications, would quickly require a very large number of particles to avoid undersampling.

Addressing this, the research community has been, and still is, very active in investigating PF variations that would be less sensitive to the curse of dimensionality. Localization schemes and equal weight PF show promising results with reasonable sample sizes, that, in some aspects outperform

traditional methods such as the Local Transform Ensemble Kalman Filter (LETKF) [11]. The filter degeneration is not observed in those more advanced PF methods [1, 11, 12]. The same discussion as for other ensemble-based DA methods still holds true for such PF: the number of particles has to be large enough to provide results with sufficient skill.

Large scale DA with PF, leveraging localization [13, 14] or implicit equal weight particle filtering (IEWPF) and its variants [11, 12, 15, 16] have been successfully used in a number of geoscience studies.

Supercomputers, reaching Exascale, have the computing power to run a very large number of particles. But using such resources efficiently, time-, and energy-wise is challenging. Applications need to limit the use of the parallel file system, a classical supercomputer bottleneck, and favor instead in situ data processing as well as local data storage to reduce data movements, and asynchronicity to overlap tasks whenever possible. Applications also need to be flexible to adapt to changes during execution, requiring support for resilience, elasticity, and dynamic load balancing.

In the landscape of ensemble-based DA, we can separate the different approaches depending on the data from the particles (also called ensemble members) that needs to be aggregated, to be combined with the observations: Some methods aggregate the full or a significant part of the particle states, or only very reduced representatives. EnKF needs the full ensemble member states, while certain PF variants only require aggregating one weight per particle state. From a parallel computing architecture point of view, the associated data dependencies and thus data movements are so different that it is worth investigating specific solutions for each category. In previous work, we developed a framework for EnKF and alike solutions [17]. In this paper, we focus on the PF category. Analyzing the actual filter to be used with this criteria will guide the user towards the best-suited approach for maximum performance.

Existing large-scale approaches for ensemble-based DA 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 and the repeated startup of model instances. They keep particles

in memory and exchange them via network communication. The majority of the existing implementations do so by running a large MPI application that encompasses the full workflow. While saving I/O overheads, this approach loses flexibility. For instance, a numerical fault during a single particle propagation that crashes a single compute core may stop the entire application. Thus existing online approaches, as will be detailed in the related work section (Section 2), usually do not support fault tolerance or dynamic load balancing.

In this paper, we develop an alternate approach that leads to a highly-efficient yet flexible framework, which might even be applied to further large scale ensemble computations. The key to achieving 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. We exploit the property of certain PF variants, that particle states do not need to be centralized, and therefore, couple each runner to a node-local distributed state cache enabling fast loads and stores of particles. The caches are asynchronously persisted in the file system for checkpointing and load balancing between runners. Asynchronous prefetching of particles into the cache enables overlapping particle loads with 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 decoupling the resource allocation from the number 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.

To strain our framework and to show its use case, we use the Weather Research and Forecasting Model (WRF) [18] with the simplest possible PF, the bootstrap PF [19]. WRF is a widely used weather model for operational forecasting and research, enabling testing with the complexity of a production solver. The bootstrap PF, though known to fail for high dimensional problems, due to the filter degeneracy and impoverishment [20], has the ad-

vantage of its simplicity and enables us to focus on the development of our framework. 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: We review related work in Section 2, Section 3 introduces the bootstrap PF. Section 4 presents the architecture of our proposed approach, while Section 5 is dedicated to experiments. The paper ends with a suitability analysis to other methods that have compatible data flows such as the IEWPF, Markov chain Monte Carlo (MCMC) methods, and Approximate Bayesian computation (ABC) Section 6 and a conclusion in Section 7.

2. Related Work

The DA domain encompasses a large variety of techniques and algorithms, like nudging [21], kriging [22], Ensemble Kalman Filter [7], ensemble maximum likelihood filter [23], or particle filter [1]. For an overview, we refer to [24, 25]. 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 via histogram for PF).

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 (offline mode), or the data is processed online usually within a large MPI code in charge of running the members and data processing. Frameworks relying on the offline mode include EnTK [26], with the largest published DA use cases reaching 4,096 members for a molecular dynamics application with an EnKF filter [27]. OpenDA also follows this model, using NetCDF for data exchange with the NEMO code [28]. DART supports both [29], with reports of large scale DA in offline mode in [30] (about 1,000 members with an oceanic code), or [31, 32] (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 requesting a new resource allocation and 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 [31], but without a specific fault tolerance mechanism. So if a node fails and the node-local storage be-

comes 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 online mode avoids the I/O bottleneck. PDAF [33], which supports both modes, has for instance been used online for the assimilation of observations into the regional Earth system model TerrSysMP. DA was based on EnKF with up to 256 members [34]. In [35] ESIAS uses online DA via particle filters with up to 4,096 particles on a wind power simulation for Europe. Notice that we work with the same WRF component of ESIAS in this paper, using a configuration on a similar domain but at a higher spatial resolution and with more advanced and more time-consuming physics. We also find ad hoc MPI codes for online DA as in [36] (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 [37], various particle propagation scheduling are analyzed, but, at a limited scale (6 compute nodes and 300 particles).

In [17], the authors propose an online processing framework relying on a runner/server architecture for EnKF with support for load-balancing, fault-tolerance and elasticity. Experiments use up to 16k members, 16 k cores for DA with EnKF for the hydrology code Parflow. The approach proposed relies on centralizing the member states on the server nodes, which is required by EnKF to compute the covariance matrix, and next update these states before redistribution to runners. In this paper we pivot on this architecture towards a decentralized state storage model relying on a distributed state cache for reducing data movements (subsection 6.1). This approach fits filters where state processing can be performed mainly locally and only a reduced amount of data needs to be centralized on the server. This is most of the time the case for algorithms from the family of PF.

3. Particle Filters

In this section, we give a brief introduction to the PF formalism, focusing on properties that we exploit in our proposal. For a comprehensive introduction, we refer to [1, 19]. 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:

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

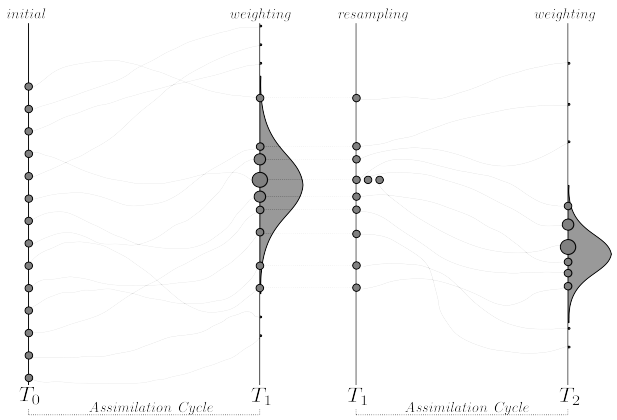

Figure 1: Initially particles are uniformly sampled. They are propagated to T_1 where they are weighted taking into account observation data. Resampling leads to discarding some particles with low weights (top and bottom), while others with high weights become parents of several ones (three here). (Inspired by [38], Figure 2)

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 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 PDF, $p(\mathbf{x}_t)$ is the prior PDF, $p(\mathbf{y}_t|\mathbf{x}_t)$ is the probability 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 $\mathbf{y}_t$.

3.1. The Bootstrap Particle Filter

In the bootstrap particle filter, the prior $p(\mathbf{x}_t)$ is estimated assuming equal weights for each of the P particles:

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

The likelihood $p(\mathbf{y}_t|\mathbf{x}_t)$ is assumed to be known, estimated when calibrating the sensor or from the error of the used observation operator. 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:

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

estimated using Equation 4 with:

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

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

$$p(\mathbf{x}_t|\mathbf{y}_t) \approx \sum_{p=0}^{P-1} \hat{w}_{p,t}\, \delta(\mathbf{x}_t - \mathbf{x}_{p,t}), \tag{8}$$

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

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

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

$$w_{p,t} = p(\mathbf{y}_t|\mathbf{x}_{p,t})\, w_{p,t-1}. \tag{10}$$

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

Especially for high-dimensional models, bootstrap particle filters suffer from weight degeneration, i.e., one normalized weight is close to one and all the others are close to zero. A classical approach weakening ensemble degeneration is Sequential Importance Resampling (SIR) [39, 40]. The procedure of SIR consists of resampling particles from the posterior (Equation 8) after weight calculation; P particles are randomly drawn, i.e., *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

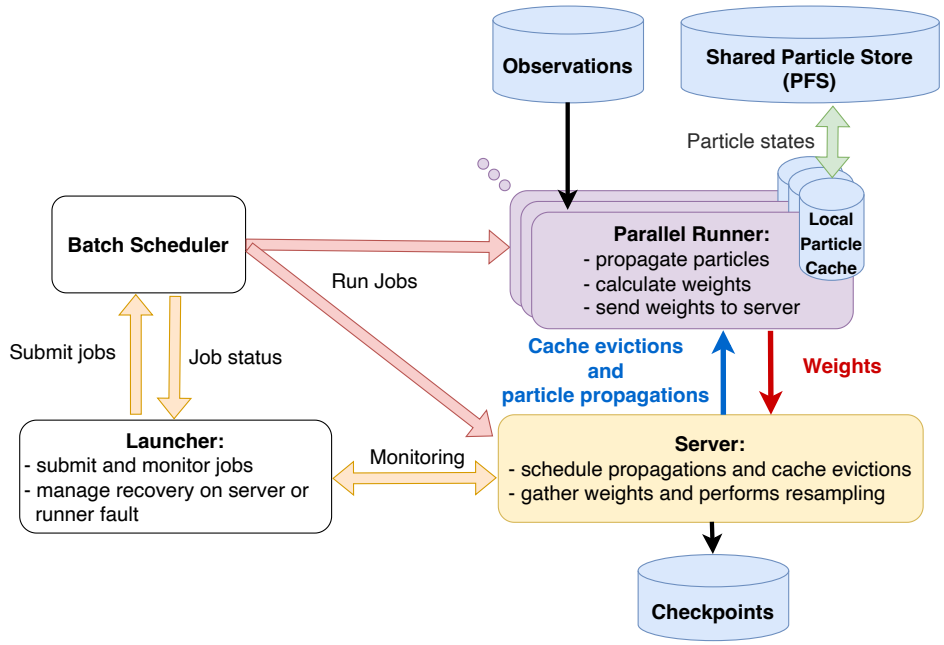

Figure 2: Architecture overview.

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 α_q is the number of occurrences of q in P. The particles q are hereinafter called **parent particles**.

The resampled particles are then all assigned the same weight of $w_{p,t} = 1/P$ again so that the formalism can be applied at each filter step anew. 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.

4. 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 3, 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 state: $\mathbf{x}_{p,t} = \mathcal{M}(\mathbf{x}_{p,t-1})$,

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

$$w_{p,t} = p_\epsilon(\mathbf{y}_t - \mathcal{H}(\mathbf{x}_{p,t})). \quad (11)$$

2. A second lightweight step that requires gathering all unnormalized 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., ensuring 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 the missing states from the file system. 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 malleability) is the ability to run under changing resource availability, here a varying number of runners.

In the following, we detail this design: the runners (Section 4.1), the server (Section 4.2), the distributed cache (Section 4.3), the workflow between these components (Section 4.4), the particle propagation scheduling (Section 4.5), the job monitoring (Section 4.6), and the fault tolerance protocol (Section 4.7) before ending with additional implementation details (Section 4.8).

4.1. 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 4.3. 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 $\mathbf{x}_{p,t}$.

Turning an existing simulation code into a runner relies on a minimalist API, but requires a deep understanding of the simulation code and data structures. The code needs to be instrumented to save the current particle state at each process and load a new one. This includes all variables that define a particle state, but also all additional variables required to reset the simulation state to the time step the newly loaded particle needs to start from, including all time-dependent boundary conditions and random number generators that need to be seeded differently for each particle.

4.2. Server

The server is entrusted with multiple tasks. First, it is responsible for scheduling the particle propagations to the runners (Section 4.5). 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 4.3). 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 exchanged between a runner and the server. The runners send the particle id (a single integer) and the corresponding weight (a single float), and the server responds with the particle id next to propagate.

4.3. Distributed Particle Cache

To allow multiple propagations of one particle on different runners, it is necessary to make them globally available. A straightforward approach is to store particles on global storage. However, on supercomputers, global storage

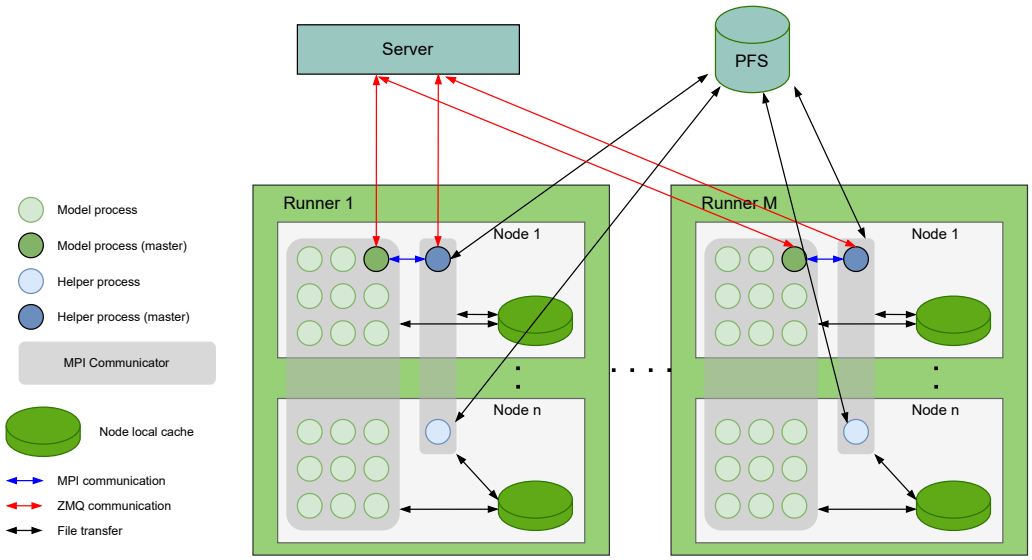

Figure 3: Internal runner architecture and interactions with the server and global storage (PFS). Communications with the server combine MPI and ZMQ data exchanges.

is subject to large throughput variability due to the high workload and 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 their 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 to global storage asynchronously, enabling to overlap the associated I/O costs. Figure 4 shows this in a schematic Gantt chart. With the exception of initial state propagations, states can be stored and loaded from the persitant storage in the background, as well as the requests to the server. Neither one is blocking the processors that perform *useful* state propagation and weighting work. Also notice that persisting particles to global storage acts as a particle checkpoint used by the fault tolerance

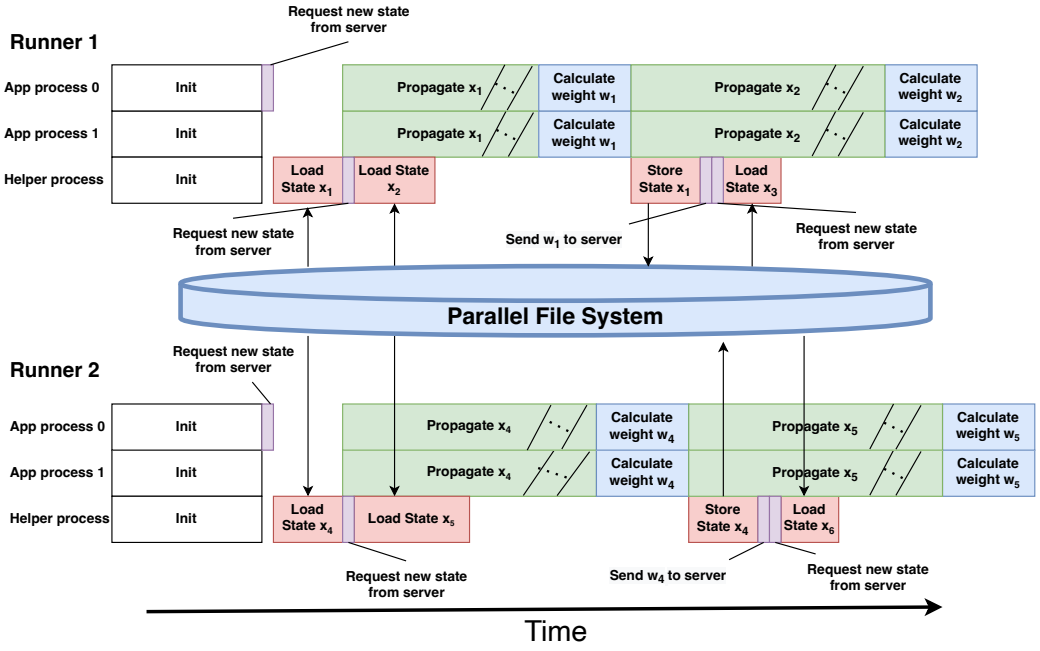

Figure 4: Schematic Gantt diagram showing the activity of two runners (initialization followed by two assimilation cycles). Focus on how the helper process asynchronous loads and stores enable to shadow parallel file system access. Propagation and calculation work (filled green and blue) overlap administrative work for state requests and file system loads and stores (filled violet and red)

protocol (Section 4.7).

Caching particles 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. Dynamic load balancing (Section 4.5) enables to turn the time gain of particle caching into a global performance gain.

As explained in Section 4.4, 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 particles 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.

4.4. 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 following a scheduling policy described in Section 4.5. 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 in 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 send 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 workload, it also takes care of ensuring coherency between runners. Globally prefetching proved to be very efficient for overlapping particle state loads with propagation (Section 5.2).

4.5. 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 minimize the global particle loads, i.e. transfers of particles from global to local storage.

4.5.1. 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 be the number of parent particles q, and α_q

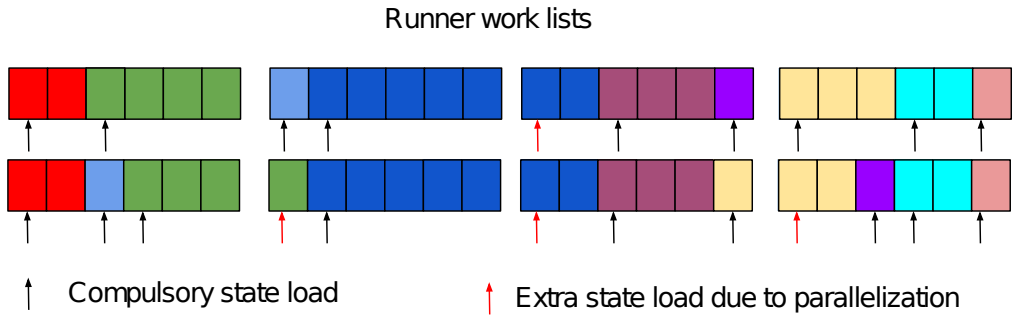

Figure 5: Two possible schedules of 24 propagation tasks of equal duration on 4 runners. All particles propagated from the same parent particle state have the same color (9 parents here). The top schedule is optimal with 9 compulsory loads (one per parent), and one for the dark blue parent that cannot fit in one runner. The bottom schedule, with 2 more state 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.

the number of times the particle q needs to be propagated. The total number of particles to propagate is:

$$P = \sum_{0 \le q < Q} \alpha_q. \tag{12}$$

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

$$s_q = \left\lceil \frac{\alpha_q}{\frac{P}{R}} \right\rceil. \tag{13}$$

Distributed to R runners, the list of P particles is cut $R - 1$ times. Con-

sequently, 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

$$Q \leq c^* \leq Q + R - 1. \tag{14}$$

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 with the current one.

4.5.2. Dynamic 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 [41, 42]. 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 13). However, we adapt the static scheduling to the dynamic case by recomputing s_q each time with the updated values of α_q, P, and R. To implement this algorithm on the server, we need bookkeeping of the number of runners R_q currently propagating particle q, and the number α_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 α_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 14.

4.5.3. 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 4.5.2. 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 the strategy of Section 4.5.2, favoring particles with no previous propagation. The rationale is to start as soon as possible with new parent particles and, once in a cache, propagate them as often as required, and intend to reduce the need for splitting. The last item departs from the strategy of Section 4.5.2, 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.

4.6. 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 4.7), or for elasticity purposes.

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 resource allocation, but the execution progress becomes very dependent on the machine's availability. The user may need more control over 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 [43], or OAR containers[44]. For even more flexible schemes, we plan to support workflow pilot-based schedulers like Radical-Pilot [45] or QCG-PilotJob [46].

4.7. 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 ongoing 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.

4.8. 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 [17]. The distributed cache implementation relies on the Fault Tolerance Interface (FTI) [47]. FTI is a multilevel checkpoint-restart library supporting asynchronous checkpointing to the global storage. One of the main modifications performed to FTI is related to its event loop. Events are triggered using MPI communications 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 that FTI provides and to implement the cache management 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 purposes. Only rank 0 (master) of the application (i.e., model) communicator and 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 [48]. The ZMQ library simplifies the implementation of asynchronous messaging by providing various communication patterns on top of different transport protocols like TCP. Information that need to be propagated between helper or model processes relies on MPI collective communications in the associated communicator (Figure 3).

The framework code with support for EnKF and Particle Filter is available at `https://gitlab.inria.fr/melissa/melissa-da` and is part of the Melissa framework for large ensemble runs `https://gitlab.inria.fr/melissa`.

5. Experiments

The experiments in this section are meant to evaluate the computational performance of our framework. We use WRF to demonstrate the potential of our approach within the context of a complex production code. WRF is used with the bootstrap PF for a very limited number of cycles to fit our compute hours budget, and our tests are focused on the scalability, efficiency, and

reliability and not the numerical quality of the assimilation. Many studies have shown that the bootstrap PF fails for high-dimensional DA applications. In our case, the short operation time did not lead to filter degeneration, but this is an important point to consider when projecting our experimental results to real production runs, which will need to rely on advanced PF methods more robust to degeneration and impoverishment on long runs.

We also push the number of particles to thousands to test scalability beyond standard DA practice that usually runs at a few hundreds as a trade-off between compute costs and assimilation quality.

For access to further technical details and reproducibility purpose the instrumented code of WRF is publicly available at `https://gitlab.inria.fr/melissa/wrf-melissa-da`.

5.1. WRF Use Case

Experiments rely on an established Numerical Weather Prediction (NWP) system – the Weather Research and Forecasting Model (WRF, V3.7.1)[18]. The core of Weather Research and Forecasting Model (WRF) is based on solving fully compressible non-hydrostatic equations with complete Coriolis and curvature terms, and a large set of physics options. The simulated spatial 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 100-second time steps. The model employs the following physics options: WSM6 microphysics [49], MYNN2 boundary layer physics [50], Grell-3D cumulus parameterization, which is an improved version of [51], Eta Monin-Obukhov similarity surface layer processes, which is based on the Monin-Obukhov similarity theory [52] using Zilitinkevich thermal roughness length [53], and the RUC land surface model [54]. Also, the 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.

DA is performed using the Cloud fraction variable (CFRACT), which represents the fraction of cloud coverage over the 2D plane of the modeled grid. The particle weights are determined by comparison with the observed cloud mask obtained from the EUMETSAT Level-2 satellite data, which is

validated to have good agreement with ground-based observation [55]. Following the method in [56], the simulated cloud fraction (a floating-point number between 0 and 1, unitless) is converted into a binary cloud mask (0 for cloud-free and 1 for cloudy) and rescaled, to match the format and grid of the observed cloud mask data.

The observation data is available hourly, thus, one assimilation cycle comprises 36 model time steps ($36 \times 100\,\mathrm{s} \mathrel{\widehat{=}} 1\,\mathrm{h}$).

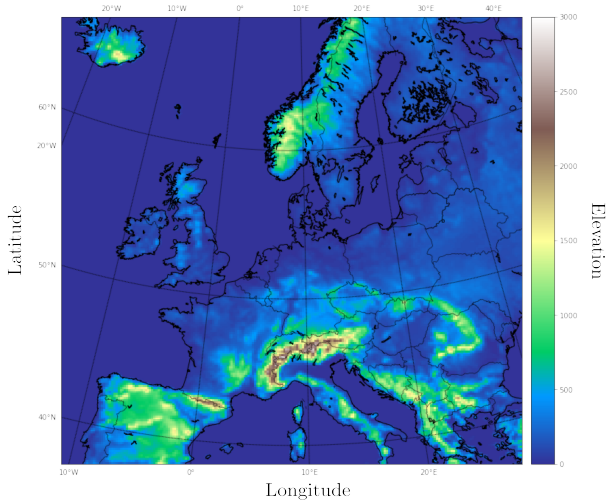

Figure 6: The topography of the target domain of Europe for the simulation.

The experiments presented in this article leverage our proposed 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 assimilation cycles) corresponds to an aggregated size of about 300 TiB. The experiments performed for this article, including small beta-stage experiments, account for about 900,000 CPU

hours split between the JUWELS, Jean-Zay, and Marenostrum supercomputers.

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.

5.2. 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.

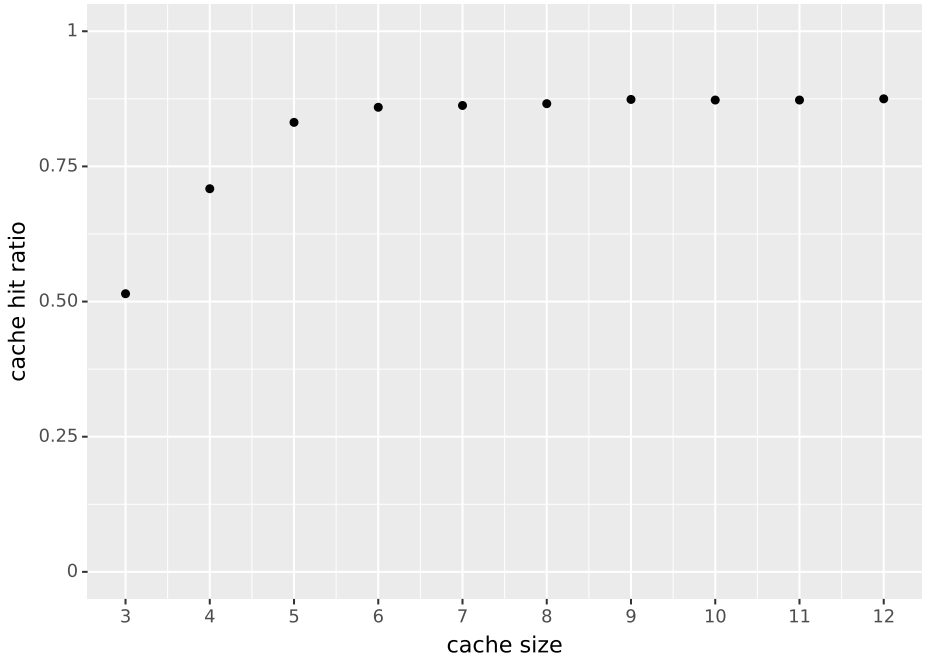

Figure 7: Cache hit ratio for different cache sizes on each runner. In total 128 particles run on 32 runners. The first and last assimilation cycles were disregarded to remove warm up effects and not fully recorded cycles.

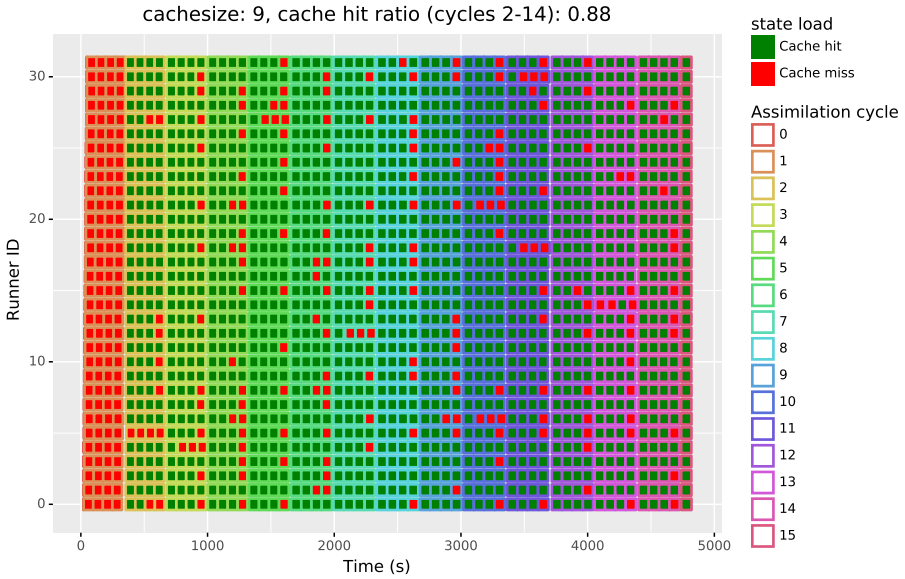

Figure 8: Gantt chart of particle propagations executed by the 32 runners over 15 assimilation cycles. Tasks are green if the associated parent particle state was already present in the runner cache and did not require a load from the PFS (red otherwise). For more than 97% of the cache misses (red), particle prefetching (Figure 4) is effective to load it from storage with enough anticipation not to delay its propagation.

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 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 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

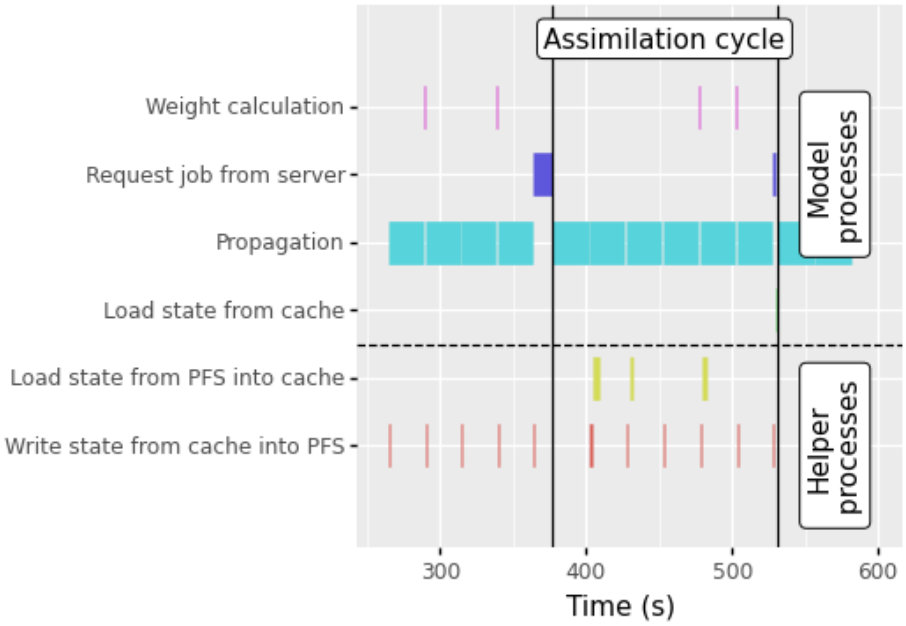

Figure 9: Trace detailing the activity of a runner over the course of an assimilation cycle. Helper processes enable to keep model processes busy with particle propagation, except at the end of assimilation cycles when they wait for the server to finish particle resampling (dark blue). Some activities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

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 variability 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 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 propagation. 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 scheduler.

5.3. 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, i.e. resampling. 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).

5.4. 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 fewer loads than the $Q + R - 1$ upper bound expected with static scheduling and no cache (Equation 14).

The access times to the Parallel File System (PFS) (load/store) vary significantly and increase with the number of runners (Figure 12), showing that our application alone can stress the PFS [1]. Each particle is associated with

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

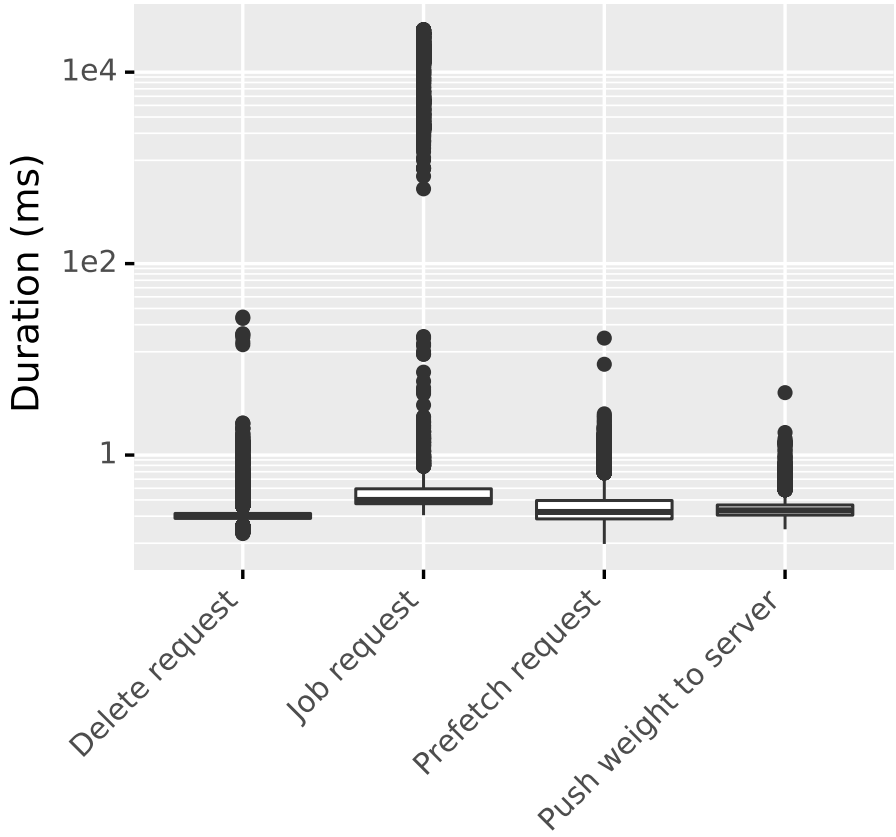

Figure 10: Server response times on runner requests.

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 5.2). 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. The time to perform the local loads and stores from the cache shows a constant average independent of the number of runners (Figure 13).

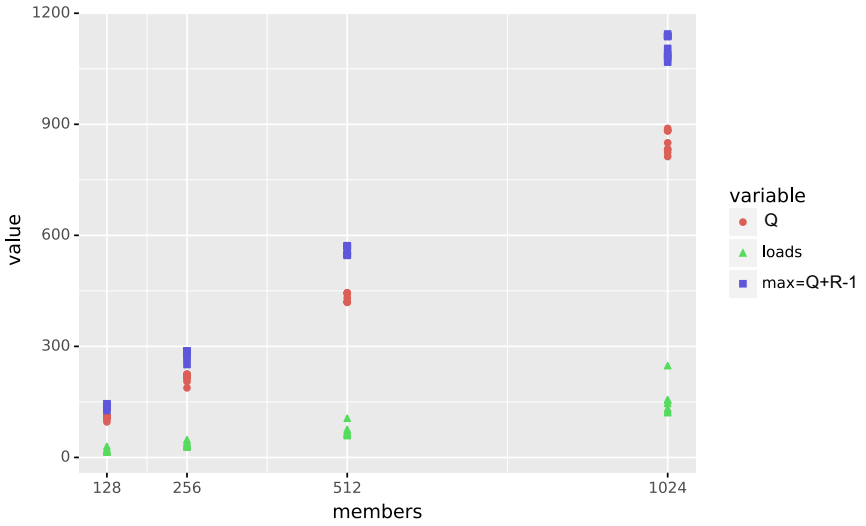

Figure 11: Number of parent particles Q, particles loads from the PFS to the cache, and $Q + R - 1$ upper bound from Equation 14 for different ensemble sizes, a cache size of 9 particles with 4 particles per runner.

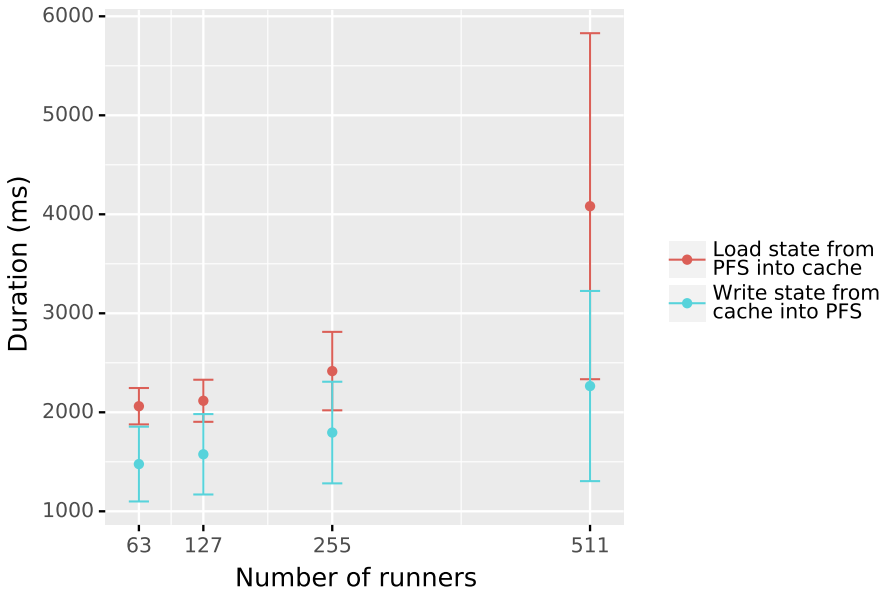

Figure 12: Mean time (and standard deviation bars) to load or store particle states of 2.5 GiB from/to the PFS with different numbers of runners.

5.5. 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. 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 in 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

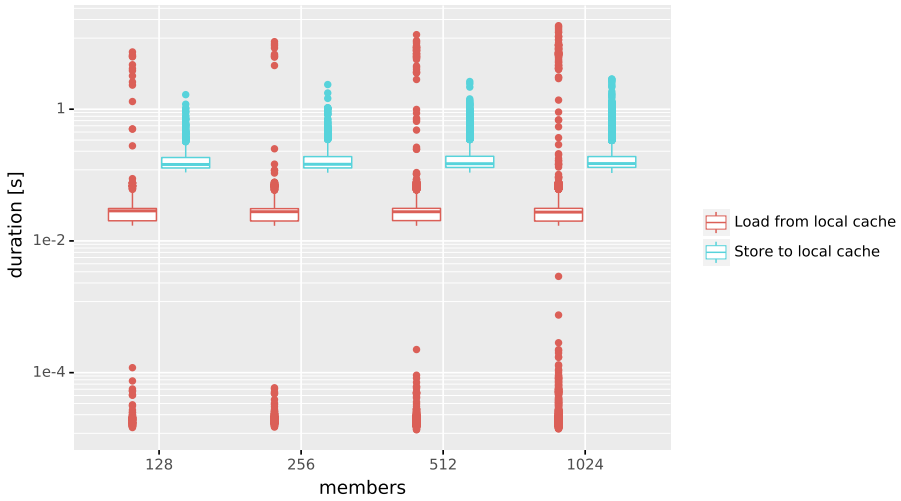

Figure 13: Box plot of the time spent for loads and stores from/to the local cache with different numbers of particles.

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 workload is kept well distributed amongst runners, even when their number varies.

5.6. 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 increases only by 8% from 63 to 511 runners, indicating an efficient scaling behavior of the framework up to production scale (Figure 16).

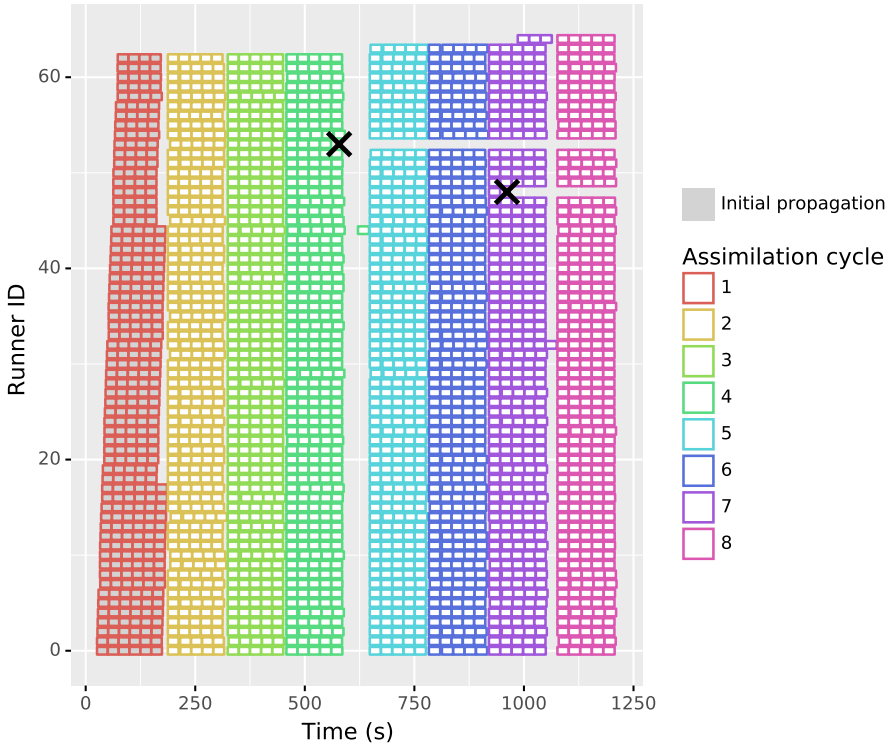

Figure 14: Gantt chart of particle propagations executed by the 63 runners over 8 assimilation cycles. After runners #48 and #53 crashed (black cross), two new ones restarted (top 2 runners #63 and #64).

	This Publication			ESIAS-met			ESIAS-met/This Publication
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 our approach and ESIAS-met (file-based) runs.

5.7. Comparison to a File-based Approach

We compare our approach with the file-based approach ESIAS-met [56] using the same simulation code WRF (V3.7.1), and the same input data. For the same number of particles, both approaches use a very different number of cores (Table 2). With ESIAS-met, each particle propagation requires to start a dedicated instance of WRF. Each time it includes the cost of

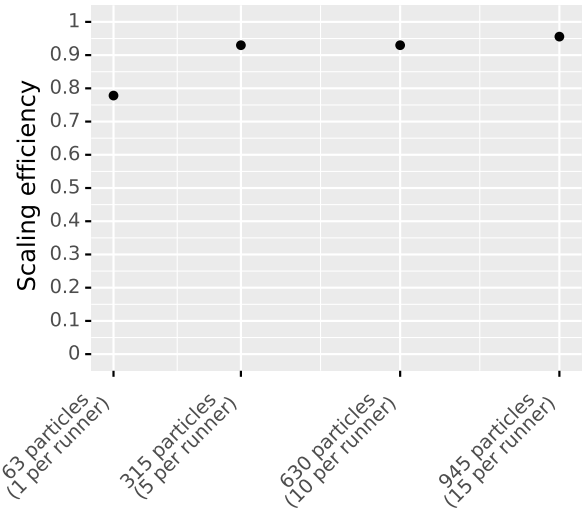

Figure 15: Scaling efficiency using different numbers of particles with 63 runners. One runner sets the reference case.

loading and storing the particle state from/to a file. At 1,024 particles ESIAS-met uses 12,288 cores while our approach just needs 3,072 cores as runners propagate several particles each. ESIAS-met's execution time is thus shorter, as highly parallelized, but, the resource usage (core×second/particle/cycle) is 2.84 times lower for our approach due to the combined strategies developed to improve efficiency. The gain increases with the number of particles, showing that our approach is particularly beneficial when targeting large ensemble sizes.

5.8. Experiment Discussions

In Section 5.1 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. Not all these intermediate states need to be saved and our framework can easily be extended with an extra server dedicated to computing summary statistics for instance as in [57].

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. For instance, some runners could propagate faster than others by leveraging GPUs.

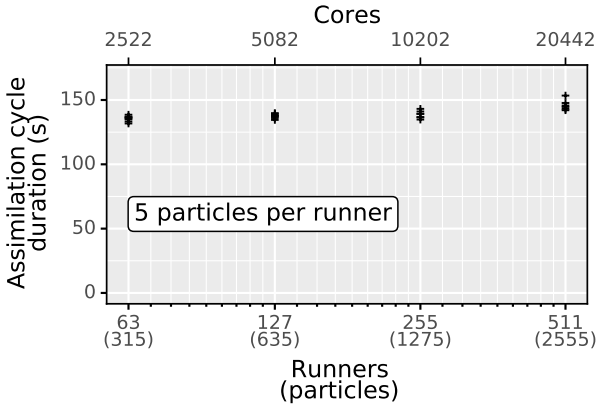

Figure 16: Weak scaling performance test: assimilation cycle duration for different numbers of runners, but always 5 particles per runner.

We only considered the case where the propagation time is longer than the time for loading states from the PFS. Reducing the assimilation cycle length may lower the time to calculate the model propagation as well. If the propagation time becomes shorter than the time necessary for state loads and stores (I/O) cannot be further overlapped by model propagations. This will reduce the efficiency of our proposal. To minimize state loading and storing times further, permitting shorter assimilation cycles with large model states, we are evaluating approaches leveraging node-local persistent storage as a globally shared storage layer. Solutions for this are readily available in the form of distributed ad hoc file-systems [58] 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 with each other.

6. Beyond the (bootstrap) PF

We presented our framework and experiments using the bootstrap PF. The PF was chosen for its simplicity to keep the paper focused on the architectural design. In this section, we discuss support for other ensemble-based methods.

6.1. Distributed versus Centralized State Store

In a previous work [17], we developed a solution for DA where a large fraction of the particle states needs to be aggregated to compute the update.

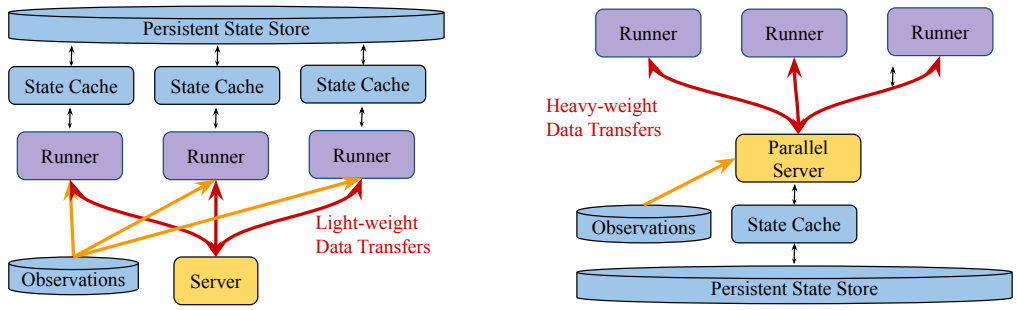

Figure 17: Distributed state store (left - this paper) versus centralized state store (right - [17]). In both cases the server is in charge of 1) computing the update phase and 2) load balancing the particle propagation to the different workers. The centralized approach is well adapted to filters requiring to gather a significant portion of the states (e.g. EnkF filters), while the distributed approach is adapted to filters where only a lightweight fraction or a summary of each state is required for the update phase.

This is typically the case for DA methods from the EnKF family, where computing the Kalman Gain requires computing the sample covariance from the ensemble of states. Let us call it the *centralized* design, compared to the architecture of this paper that we call the *distributed* design (Figure 17). Both frameworks rely on the same software base and share various components. We stress here the main differences. The centralized design gathers all the particle states on the server. When all particle propagations are done for the current cycle, the server computes the state update and redistributes dynamically the modified states to runners to load-balance the propagations for the next cycle. For checkpointing purposes, the server persists the states to the parallel file system [59]. Because states are memory-consuming and the associated computations are significant, the server is in that case a parallel code running on several nodes. For instance, we support PDAF [33] as DA engine for the server. Refer to [17] for details. The centralized design is very generic and can support a wide variety of ensemble-based DA methods. But moving back and forth the particle states is time-consuming, and actually not always required by the data dependencies of the used method, as in the case of the bootstrap PF. Therefore, the distributed design proposed here enables saving these state movements, caching the states locally at each runner. Hence, by analyzing the data dependencies associated with a given DA method, one can identify the most suitable design and improve efficiency. To

	Distributed Design (this work)	Centralized Design [17]	Conventional Online Approach	Offline Approach
Model initialization only once	+	+	+	-
State Exchange	File System	Network	Network	File System
Load-balancing	+	+	-	+
Fault Tolerance	+	+	-	+
State Movements	Local cache + PFS	Gather/Scatter	Gather/Scatter	PFS

Table 3: Comparison of the properties of existing approaches. We assume a conventional online approach that gathers all member (i.e. particle) states for the DA update (i.e. resampling) and scatters them again afterwards.

comply with the distributed design, DA methods need to fulfill the following properties:

(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) The reduction during the DA update only depends on weights $w_{p,t}$ or similar scalar variables, and not on the particles and associated states.

(c) The scalar values that need to be aggregated during the update phase, i.e., the 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.

(d) The states $\mathbf{x}_{p,t}$ associated with the particles p remain unchanged during the filter update, or the particle's state transformation only depends on local information, such as observations, or $\mathbf{x}_{p,t-1}$ and $w_{p,t-1}$.

We can compare the centralized and distributed designs with the traditional online and offline approaches. As shown in Table 3, the two new designs mix features from offline and conventional online approaches. This work can either be seen as an online approach that circumvents gathering the full state ensemble for the DA update step, or as an offline approach that avoids simulation code startup costs thanks to runners as well as I/O delays using a distributed cache and prefetching. In addition, our framework supports dynamic load-balancing enabling it to adapt to execution variabilities,

even in the presence of failures. In the following, we make a deeper analysis of methods that are good candidates to be ported to the framework for the distributed workflow.

6.2. Particle Filters Analysis

We discuss how various PF methods fit the different properties (subsection 6.1) and so the distributed state store proposed in this paper. Implicit Equal Weights Particle Filter (IEWPF) is discussed appart in Section 6.2.1.

Property (a) is typically fulfilled by most PF variants. Property (c) is fulfilled if we can compute the weights according to Equation 10. If the method allows decoupling the weight calculation from the filter update, property (b) is fulfilled. Finally, property (d) is fullfilled when particles are either withdrawn or selected, but not changed, as for the bootstrap PF with SIR, as well as other flavors of SIR and resampling algorithms, like residual or stratified resampling [60, 61]. If the states are transformed but no global information is used for the transformation, property (d) is fulfilled as well, as for the IEWPF discussed in the following section.

However, the properties exclude certain classes of PF. For instance, PF that use localization [13, 14, 62], as the transformation of the particle depends on segments of other particle states from the ensemble, and the ensemble mean. Furthermore, hybrid particle filters like the weighted EnKF [63] are not supported due to the necessity of computing the Kalman Gain which involves the sample covariance matrix of the ensemble states. The centralized design is well adapted for these types of filters.

Other PF methods that try to reduce the impact of the centralized update could fit the distributed design with some adaptations. This include Island PF that groups particles hierarchically to reduce ensemble-wide synchronization [64]. To remove any remaining synchronization, anytime or asynchronous PF decide on a particle-by-particle basis if it is resampled [65]. For constant ensemble sizes, the DA accuracy may vary over time. Thus adaptive resampling of fewer or more particles might be another extension for our framework to guarantee result quality at optimal runtimes, avoiding oversampling [66, 67].

6.2.1. Implicit Equal Weights Particle Filter

IEWPF [11], in contrast to the bootstrap PF, does not suffer from weight degeneration, as all the particles are assigned the same weight, and therefore, each particle has the same probability to be drawn. In fact, most IEWPF

implementations do not perform resampling at all. Instead, the particle state undergoes a transformation to fulfill the equal-weights-condition, and the transformed ensemble is kept.

The idea behind the IEWPF is based on drawing the particles from a Gaussian-shaped proposal distribution, $q(\boldsymbol{\xi})$, instead of the original one. For this, the particle state is transformed by:

$$\mathbf{x}_{p,t} = \mathbf{x}^a_{p,t} + \alpha^{1/2}_{p,t}\mathbf{P}^{1/2}\boldsymbol{\xi}_{p,t} \tag{15}$$

$$\boldsymbol{\xi}_{p,t} \sim N(0, \mathbf{P}), \tag{16}$$

where $\mathbf{x}^a_p$ is the mode and $\mathbf{P}$, the covariance of $\boldsymbol{\xi}_{p,t}$, is a measure of the width of the optimal proposal distribution $p(\mathbf{x}_{p,t}|\mathbf{x}_{p,t-1}, \mathbf{y}_t)$. The scalar $\alpha_{p,t}$ is obtained during each update step by solving the equal-weights-condition:

$$w_{p,t} = \frac{p\left(\mathbf{x}_{p,t} \mid \mathbf{x}_{p,t-1}, \mathbf{y}_t\right) p\left(\mathbf{y}_t \mid \mathbf{x}_{p,t-1}\right)}{q(\xi)} \left\| \frac{\mathrm{d}x}{\mathrm{d}\xi} \right\| \cdot w_{p,t-1} \stackrel{!}{=} \hat{w}_t, \tag{17}$$

where $\hat{w}_t$ is the target weight, and $\left\| \frac{\mathrm{d}x}{\mathrm{d}\xi} \right\|$ the absolute value of the determinant of the Jacobean from the transformation in Equation 15. Note that $p(\mathbf{x}_{p,t}|\mathbf{x}_{p,t-1}, \mathbf{y}_t)$, as well as $\mathbf{P}$ and $\mathbf{x}^a_{p,t}$ only depend on the corresponding particle state and observations. The transformation in Equation 15 ensures that with the appropriate $\alpha_{p,t}$, determined by solving Equation 17, each particle takes on the same distance to the observations, hence, obtains the same weight as the other particles. In opposition to the boostrap filter, IEWPF performs a state update, but this update (Equation 15) only requires local information, observations and the scalar $\alpha_{p,t}$ obtained from Equation 17, fullfiling the property b) and d). The only centralized operation is the computation of the target weight (Equation 17) that also relies on scalar values, fullfilling property c). Thus IEWPF is suitable for the distributed design.

6.3. MCMC and ABC

Our framework is well suited for a certain class of PFs as we have explained in the preceding sections. However, it is not constrained to PFs and geoscience applications. In the following two paragraphs we want to introduce two important techniques that can be ported to our framework as well.

6.3.1. MCMC

In the last decade, Markov Chain Monte Carlo (MCMC) algorithms played an important role in solving high dimensional integrals in statistics, econometrics, physics, and computing science [68, 69, 70, 71]. MCMC, for instance, is the only known general solution to calculate the volume of a d-dimensional convex body [72].

MCMC is a probabilistic method. Its solutions take the form of a PDF. The MCMC algorithm samples N first guesses $x_0^1, x_0^2, \ldots, x_0^N$ and propagates them M times through a Markov transition $\mathcal{M}(\cdot)$. Note that the state variables x may be scalars up to high-dimensional state vectors. The stationary distribution of the resulting Markov processes, i.e., of all samples x_n^i after applying multiple times the Markov transition, must converge against the PDF of the integral in question $P(x^*)$ [73].

$$x_{n+1}^i = \mathcal{M}(x_n^i) \tag{18}$$

$$P(x^*) \sim \frac{1}{N} \lim_{n\to\infty, N\to\infty} \sum_{i=0}^{N} \delta(x_n^i - x^*), \tag{19}$$

with δ defined as the Dirac delta function.

Basic MCMC methods, as PF too[2], rely on two-phase workflows. After sampling multiple first guesses x_n^i, each is propagated repeatedly through the given Markov transition until stationary. Note that this can take different iteration counts and therefore different runtimes per initial first guess, leading to load imbalance when parallelized naively. Sampling and propagation of new first guesses x_0^i is repeated until convergence, e.g., of a distribution histogram that represents the solution of the integral as a PDF $P(x^*)$.

Multiple types of Markov processes, like Metropolis-Hasting or Hamiltonian Monte Carlo processes, can be used in MCMC. Refer to surveys like [74] for more details on these methods and how to find suitable Markov processes with stationary distributions converging against the searched integral.

The approach developed in this paper can be adapted to MCMC algorithms in high-dimensional spaces with computationally expensive transition functions. Instead of instrumenting the simulation code to become a runner, the runner can be used to embed the execution of Markov processes. After a first guess propagated through the transition $\mathcal{M}(\cdot)$ until it converged against stationarity, the resulting state vector x_n^i or a summary statistic of it can be

[2]Note that PF can also be seen as an MCMC method itself.

sent to the server which then decides if the runner should propagate a new initial guess or if the run is finished. The load balancing between runners intrinsic to our framework will enable to cope with variations in computation times of different propagations due to different iteration counts needed until different Markov processes are completed. The distributed state store will enable handling large state vector sizes for x_n^i beyond the memory capacity of the supporting compute nodes.

6.3.2. ABC

Another important tool used for Bayesian inference to sample from impossible or difficult-to-compute i.e., intractable PDFs, is Approximate Bayesian Computation (ABC) [75, 76]. It finds applications from cognitive science [77] over genetics [78] to finance [79].

ABC infers

$$\pi(\theta|y) = \frac{\pi(y|\theta)\pi(\theta)}{\pi(y)}, \tag{20}$$

where likelihood $\pi(y|\theta)$ and $\pi(y)$ are unknown. ABC draws θ from the prior $\pi(\cdot)$ and accepts it with the probability $\pi(y|\theta)$. Accepted θ thus are independent draws from the posterior distribution $\pi(\theta|y)$. For ABC to work there must be a way to simulate y:

$$y_{\text{sim}} = \mathcal{M}(\theta). \tag{21}$$

Instead of the evaluation of the unknown $\pi(y|\theta)$ one can thus draw θ from the prior $\pi(\cdot)$ and then accept it if $y = y_{\text{sim}}$. If y is continuous rarely any θ are accepted. Thus an approximate version is used accepting θ if y and y_{sim} are close. For instance, accept θ if $|y - y_{\text{sim}}|$ is smaller than a given threshold value.

ABC can be implemented within the framework as follows: each runner simulates with a different parameter drawn from the prior distribution $\pi(\theta)$. Only if the result y_{sim} is accepted, it is sent to the server. Otherwise, the runner draws another input parameter and tries again.

7. Conclusion

In this article, we proposed an architecture for handling very large ensembles for a variety of PF. The architecture is designed to address the challenge of exascale computing that will allow massive ensemble runs [80]. This is

part of a larger effort to develop an open source solution, called Melissa, for supporting online data processing of large ensemble runs for different DA schemes as well as sensibility analysis and deep surrogate training[3].

The proposed 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 workload. 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-met file-based approach, our proposal improves resource usage 2.83 times at 1,024 particles.

We plan to extend the distributed cache and fault tolerance algorithm of the framework to fully avoid the centralized file system and only rely on node-local SSDs for particle storage for further performance improvements. Future work also includes experimenting with other PF such as IEWPF, adaptive or anytime PF, MCMC and ABC.

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 allocations 2020-A8 A0080610366 and 2021-A10 A0100610366 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.

[3] `https://gitlab.inria.fr/melissa`

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: European Symposium on Programming (ESOP 2021), Springer International Publishing, 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 1809.10756, 2018.

[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.

[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.

[7] G. Evensen, Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics, Journal of Geophysical Research: Oceans 99 (C5) (1994) 10143–10162.

[8] F. Daum, J. Huang, Curse of dimensionality and particle filters, in: 2003 IEEE Aerospace Conference Proceedings (Cat. No.03TH8652), Vol. 4, 2003, pp. 4_1979–4_1993.

[9] C. Snyder, T. Bengtsson, P. Bickel, J. Anderson, Obstacles to high-dimensional particle filtering, Monthly Weather Review 136 (12) (2008) 4629–4640.

[10] C. Snyder, T. Bengtsson, M. Morzfeld, Performance Bounds for Particle Filters Using the Optimal Proposal, Monthly Weather Review 143 (2015) 12.

[11] M. Zhu, P. J. van Leeuwen, J. Amezcua, Implicit equal-weights particle filter, Quarterly Journal of the Royal Meteorological Society 142 (698) (2016) 1904–1919.

[12] P. Wang, M. Zhu, Y. Chen, W. Zhang, Implicit equal-weights variational particle smoother, Atmosphere 11 (4) (2020) 338.

[13] S. Kotsuki, T. Miyoshi, K. Kondo, R. Potthast, A local particle filter and its gaussian mixture extension implemented with minor modifications to the LETKF, Geoscientific Model Development 15 (22) (2022) 8325–8348.

[14] R. Potthast, A. Walter, A. Rhodin, A localized adaptive particle filter within an operational NWP framework, Monthly Weather Review 147 (1) (2019) 345–362.

[15] P. Wang, M. Zhu, Y. Chen, W. Zhang, Y. Yu, Ocean satellite data assimilation using the implicit equal-weights variational particle smoother, Ocean Modelling 164 (2021) 101833.

[16] H. H. Holm, M. L. Sætra, P. J. van Leeuwen, Massively parallel implicit equal-weights particle filter for ocean drift trajectory forecasting, Journal of Computational Physics: X 6 (2020) 100053.

[17] 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.

[18] 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).

[19] J. V. Candy, Bootstrap Particle Filtering, IEEE Signal Processing Magazine 24 (4) (2007) 73–85.

[20] T. Li, T. P. Sattar, S. Sun, Deterministic resampling: unbiased sampling to avoid sample impoverishment in particle filters, Signal Processing 92 (7) (2012) 1637–1645.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 536–545.

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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).

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

[34] 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.

[35] 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).

[36] 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.

[37] 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.

[38] P. J. van Leeuwen, Particle filtering in geophysical systems, Monthly Weather Review 137 (12) (2009) 4089–4114.

[39] 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) 107–113.

[40] 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.

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

[42] 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.

[43] The slurm workload manager - `https://slurm.schedmd.com/`, retrieved 30.06.2023.

[44] The oar resource and task manager - `http://oar.imag.fr/`, retrieved 30.06.2023.

[45] A. Merzky, M. Turilli, M. Titov, A. Al-Saadi, S. Jha, Design and performance characterization of radical-pilot on leadership-class platforms, IEEE Transactions on Parallel and Distributed Systems 33 (04) (2022) 818–829.

[46] Qcg-pilotjob - `https://github.com/psnc-qcg/QCG-PilotJob`, retrieved 30.06.2023.

[47] 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.

[48] P. Hintjens, ZeroMQ, Messaging for Many Applications, O'Reilly Media, 2013.

[49] S.-Y. Hong, J.-O. J. Lim, The WRF single-moment 6-class microphysics scheme (WSM6), Asia-Pacific Journal of Atmospheric Sciences 42 (2) (2006) 129–151.

[50] M. Nakanishi, H. Niino, An improved mellor-yamada level-3 model: Its numerical stability and application to a regional prediction of advection fog, Boundary-Layer Meteorology 119 (2) (2006) 397–407.

[51] G. A. Grell, D. Dévényi, A generalized approach to parameterizing convection combining ensemble and data assimilation techniques, Geophys. Res. Lett. 29 (14) (2002) 38–1–38–4.

[52] A. S. Monin, A. M. Obukhov, Basic laws of turbulent mixing in the surface layer of the atmosphere, Contrib. Geophys. Inst. Acad. Sci. USSR 151 (163) (1954) 163–187.

[53] S. S. Zilitinkevich, Non-local turbulent transport: Pollution dispersion aspects of coherent structure of connective flows, WIT Transactions on Ecology and the Environment 9 (1995) 53–60.

[54] T. G. Smirnova, J. M. Brown, S. G. Benjamin, J. S. Kenyon, Modifications to the rapid update cycle land surface model (ruc lsm) available in the weather research and forecasting (wrf) model, Monthly Weather Review 144 (5) (2016) 1851–1865.

[55] R. A. Roebeling, S. Placidi, D. P. Donovan, H. W. J. Russchenberg, A. J. Feijt, Validation of liquid cloud property retrievals from seviri using ground-based observations, Geophysical Research Letters 35 (5) (2008).

[56] Y.-S. Lu, G. H. 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, GMD 16 (3) (2023) 1083–1104.

[57] 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, pp. 1–14.

[58] 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.

[59] K. Keller, A. C. Kestelman, L. Bautista-Gomez, Towards zero-waste recovery and zero-overhead checkpointing in ensemble data assimilation, in: 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), 2021, pp. 131–140.

[60] T. Li, M. Bolic, P. M. Djuric, Resampling Methods for Particle Filtering: Classification, implementation, and strategies, IEEE Signal Processing Magazine 32 (3) (2015) 70–86.

[61] M. Bolic, P. Djuric, Sangjin Hong, New resampling algorithms for particle filters, in: 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'03), Vol. 2, IEEE, Hong Kong, China, 2003, pp. II–589–92.

[62] T. Bengtsson, C. Snyder, D. Nychka, Toward a nonlinear ensemble filter for high-dimensional systems, Journal of Geophysical Research: Atmospheres 108 (D24) (2003).

[63] N. Papadakis, E. Mémin, A. Cuzol, N. Gengembre, Data assimilation with the weighted ensemble kalman filter, Tellus A: Dynamic Meteorology and Oceanography 62 (5) (2010) 673–697.

[64] 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.

[65] B. Paige, F. Wood, A. Doucet, Y. W. Teh, Asynchronous Anytime Sequential Monte Carlo, in: 27th International Conference on Neural Information Processing Systems (NIPS'14), Vol. 2, 2014, p. 3410–3418.

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

[67] 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.

[68] D. A. Levin, Y. Peres, E. L. Wilmer, Markov Chains and Mixing Times, American Mathematical Soc., Providence (2009).

[69] D. P. Landau, K. Binder, A Guide to Monte Carlo Simulations in Statistical Physics, Cambridge university press, 2014.

[70] D. Grana, L. de Figueiredo, K. Mosegaard, Markov chain Monte Carlo for seismic facies classification, GEOPHYSICS (2023) 1–67.

[71] P. Abbaszadeh, H. Moradkhani, H. Yan, Enhancing hydrologic data assimilation by evolutionary Particle Filter and Markov Chain Monte Carlo, Advances in Water Resources 111 (2018) 192–204.

[72] C. Andrieu, C. Andrieu, An Introduction to MCMC for Machine Learning, Machine Learning 50 (2003) 5–43.

[73] A. E. Gelfand, A. F. M. Smith, Sampling-Based Approaches to Calculating Marginal Densities, Journal of the American Statistical Association 85 (410) (1990) 398–409.

[74] C. P. Robert, W. Changye, Markov Chain Monte Carlo Methods, a survey with some frequent misunderstandings, arXiv:2001.06249 [stat] (Jan. 2020).

[75] D. B. Rubin, Bayesianly Justifiable and Relevant Frequency Calculations for the Applied Statistician, The Annals of Statistics 12 (4) (1984) 1151–1172.

[76] K. Csilléry, M. G. Blum, O. E. Gaggiotti, O. François, Approximate Bayesian Computation (ABC) in practice, Trends in Ecology & Evolution 25 (7) (2010) 410–418.

[77] H. Yadav, D. Paape, G. Smith, B. W. Dillon, S. Vasishth, Individual Differences in Cue Weighting in Sentence Comprehension: An Evaluation Using Approximate Bayesian Computation, Open Mind 6 (2022) 1–24.

[78] T. Thorne, P. D. W. Kirk, H. A. Harrington, Topological approximate Bayesian computation for parameter inference of an angiogenesis model, Bioinformatics 38 (9) (2022) 2529–2535.

[79] J. Dyer, P. Cannon, J. D. Farmer, S. Schmon, Black-box Bayesian inference for economic agent-based models, arXiv:2202.00625 [cs, econ, stat] (Feb. 2022).

[80] 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.