E-Infrastructures

H2020-INFRAEDI-2018-1

INFRAEDI-2-2018: Centres of Excellence on HPC

EoCoE-II

Energy oriented Center of Excellence : toward exascale for energy

Grant Agreement Number: INFRAEDI-824158

D5.2

Melissa-DA: First Stable Version

Project and Deliverable Information Sheet

EoCoE	Project Ref:	INFRAEDI-824158
	Project Title:	Energy oriented Centre of Excellence
	Project Web Site:	http://www.eocoe.eu
	Deliverable ID:	D5.2
	Lead Beneficiary:	INRIA
	Contact:	Bruno Raffin
	Contact's e-mail:	Bruno.Raffin@inria.fr
	Deliverable Nature:	Report and Code
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M24 01/01/2021
	Actual Date of Delivery:	M29 15/06/2021
	EC Project Officer:	Evangelia Markidou

* - The dissemination level are indicated as follows: PU – Public, CO – Confidential, only for members of the consortium (including the Commission Services) CL – Classified, as referred to in Commission Decision 2991/844/EC.

Document Control Sheet

Document	Title :	Melissa-DA: First Stable Version
	ID :	D5.2
	Available at:	http://www.eocoe.eu
	Software tool:	LaTeX
Authorship	Written by:	Bruno Raffin, Sebastian Friedemann, Yen-Sen Lu, Kai Keller
	Contributors:	Bibi Naz, Harrie-Jan Hendricks Franssen, Anna Sekula, Bartłomiej Pogodziński, Christoph Conrads, Leonardo Bautista-Gomez
	Reviewed by:	Sebastian Lührs, Mathieu Lobet

Contents

1. Overview

1.1 General Objectives of WP5

The WP5 is one of EoCoE technical challenges called *Ensemble Runs*. The goal is to efficiently run large simulation ensembles on coming pre-exascale and exascale systems, in particular for data assimilation. The objective pursued here is to provide a flexible and maintainable way of executing simulation ensembles in multiple jobs on a given supercomputer while enabling communication between ensemble members to allow ensemble management and data assimilation processes. For that purpose WP5 targets developing an elastic exascale-ready framework for ensemble runs extending the Melissa approach developed at INRIA [24].

Two out of five EoCoE-II Scientific Challenges (Meteorology for Energy and Water for Energy) integrate some support for ensemble runs for data assimilation or sensitivity analysis, usually relying on one monolithic big MPI job or file-based solutions, like ESIAS developed during EoCoE-I. WP5 relies on a novel developed framework, called Melissa-DA, to empower the Meteorology and Water EoCoE-II applications to benefit from next generation exascale machines for large ensemble runs. WP5 builds on top of WP4 (I/O) work re-using the PDI interface for data access and the FTI library for fault tolerance.

1.2 WP5 Work Progress

The WP5 is structured in 3 tasks:

- Task 5.1: Ensemble Run Framework Development, M1-M30, Task leader INRIA,
- Task 5.2: Ensemble Runs for Meteorology. M6-M36, Task leader: FZJ,
- Task 5.3: Ensemble Runs for Water, M6-M36, Task leader: FZJ,

with three deliverables:

- D5.1 - Architecture Specification and prototype codes for the framework and the 2 applications with initial documentation as well as an early usability and performance evaluation. (M18) (Report + code DEM, PU) (INRIA),
- D5.2 - M24 - First stable version of the framework code and the two applications adapted to the framework. Report will include documentation as well as performance evaluation. (M24) (Report + code DEM, PU) (FZJ),
- D5.3 - M34 - Final code release for the framework and the three applications, with the associated documentation. Report on testing at large scale. (M36) (Report + code DEM, PU) (CEA),

The WP5 contributions includes as of today:

- Previous period (deliverable D5.1):
  - A first functional code implementing **Melissa-DA**, the Melissa extension for data assimilation.
  - A toy use case based on Lorentz equation, as well as an advanced use case based on the ParFlow hydrology simulation code.
  - Integration of the PDAF data assimilation engine into Melissa server, enabling to support different assimilation algorithms.

  - Experiments on supercomputers with ParFlow simulations, EnKF assimilation, running up to 1024 members.
  - Identification of ambitious use cases with code and datasets for the Weather and Hydrology applications.
  - Submission of two research papers related to Melissa-DA results

- Current period (this D5.2 deliverable):
  - Following remarks on D5.1, switch to a single (public) code repository for Melissa-DA: `https://gitlab.inria.fr/melissa/melissa-da`)
  - Consolidation of the Melissa-DA code
  - Large scale Melissa-DA experiment with ParFlow simulations, EnKF assimilation, propagating 16,384 members on 16,240 cores.
  - Development of a variation of Melissa-DA specifically targeting data assimilation with a Particle Filter
  - Support of the WRF (Weather Research and Forecasting Model) for enabling assimilation with a Particle Fitler using Melissa-DA
  - Large scale Melissa-DA experiment with WRF simulations, Particle Filter assimilation, running up to 2,555 members on 20,442 cores.
  - Communications:
    * Revision of paper under submission at International Journal of High Performance Computing Applications.
    * Two paper submissions at IEEE Cluster 2021 and HPCS 2021.
    * Oral presentation at https://enkf.norceprosjekt.no/home/enkf-workshop-2021-free-online-event-
    * Joint organisation with WP4 of the https://hpcda.github.io/

This document is the D5.2 deliverable from the EoCOE-II project from WP5. Task 5.3 (Meteorology use case) was initially delayed due to hiring difficulties and so we focused on the water use case (Task 5.2) during the first period (D5.1). For the current period most efforts focused on Task 5.3 and so this deliverable D5.2 mainly focus on adapting Melissa-DA to enable large scale DA with particle filter for the WRF/ESIAS meteo code. We have today reached a point where both use cases have progressed equally, and the first period delay has been absorbed. Notice that this deliverable is about 5 months late compared to the initial schedule, these extra months enabled us to get tangible results with the second use case presented in the following sections.

WP5 developments are done by INRIA (lead and Melissa-DA core development), PSNC (core Melissa-DA development), FZJ (use cases) and CEA (PDI and Melissa-DA integration). BSC is contributing to Melissa-DA fault-tolerance mechanism with FTI (work attached to WP4).

2. Code Repository and Management

Taking into consideration the remarks from reviewers of D5.1, we changed our code organization. Instead of relying on two repositories (one private and one public), we now have only public repos under the Melissa group: `https://gitlab.inria.fr/melissa`. This group contains the repositories of Melissa for sensibility analysis (the original Melissa development), Melissa for data assimilation, an the melissa-ci repository which contains everything related to Melissa continuous integration workflows.

Melissa-DA is released under the BSD-3 licence; the latest code and documentation are included in the repository (`https://gitlab.inria.fr/melissa/melissa-da`) As an effort to offer high-quality code, we properly manage issues, have an extended test suite, and employ continuous integration (CI). We run in the CI multiple containers for testing Melissa on various Linux distributions on the x86-64 architecture including CentOS which is very similar to the RedHat Enterprise Linux distributions found on most supercomputers. We are in the process of extending our CI setup to run tests on ARM CPUs because we expect a more diverse choice of CPUs for supercomputers in the future. Also notice that we made specific efforts to build *virtual cluster setups* by running several linked containers with the OAR and SLURM batch schedulers. This enables us to test Melissa in the CI with a multi-node cluster configuration.

3. Scalability Targets

Note that a one-to-one direct comparison between Pre-EoCoE-II and Melissa based EoCoE-II target runs should be made very carefully as the numbers given below are focused on scalability only and do not reflect the gains in efficiency and resilience. *Hero run* denotes a large scale run pushing the scalability in number of members of the system. *Production run* denotes a realistic run involving coupled codes and full observation datasets. Notice that we are facing difficulties to have the necessary access rights to reserve very large number of cores for super large scale tests.

Meteorology - ESIAS (WRF+EURAD-IM)

- Pre-EoCoE-II status: production runs with 4096 ensemble with 262,144 CPU cores
- **EoCoE-II targets:**
  - Hero run (WRF): 15 000 members on 40 000 CPU cores. **Status: achieved 2,555 members on 20,442 cores (5/2021)**
  - Production run:
    - Target: analysis of a mineral dust event limiting the solar power production in a water-cloud free sky coupling WRF and EURAD-IM within the Melissa
    - 512 members, 65,000 CPU cores, JUWELS machine
    - **Status: planned**

Water - TerrSysMP (ParFlow-CLM)

- Pre-EoCOE-II status:
  - Hero run (ParFlow, CLM and TSMP-PDAF): 256 members on 132,768 CPU cores
  - Production runs :
    - Neckar use-case (Parflow, CLM and TSMP-PDAF): 64 ensemble members on 4,608 CPU cores.
    - Europe use-case (CLM and TSMP-PDAF): 20 ensemble members on 1,920 CPU cores.

- **EoCoE-II targets**:
  - Hero run (Parflow, Melissa): 15,000 members on 40,000 CPU cores. **Status: goal reached with 16,384 members on 16,240 cores (10/2020)**
  - Production runs:
    - Neckar use-case (Parflow, CLM and Melissa)
      - 512 members on 36,864 CPU cores, JUWELS machine
      - **Status: planned**
    - Europe use-case (Parflow, CLM and Melissa):
      - 512 members on 36,864 CPU cores, JUWELS machine
      - **Status: planned**

4. Melissa-DA for Particle Filters

We present in the following sections the approach developed with Melissa-DA to efficiently support data assimilation with particle filters. We also detail the work done to support the WRF/ESIAS (meteo) use-case and the associated experiments. This text is based on a article recently submitted.

This work focuses on *Particle Filters*, which are part of ensemble-based statistical DA methods. Particle filters are of growing importance when the model does not comply with the linear hypothesis associated to the more classical DA methods like Ensemble Kalman Filters (EnKF). Particles, also called realizations, samples or members, correspond to states of the numerical model, drawn from a given probability distribution. The particles are each individually propagated forward in time through the numerical model to the next timestep when observation data is available. Particles are then weighted according to their likelihood with the observation probability density function. The process repeats after resampling the particles (i.e., selecting a subset of particles) according to their weight following methods like Stochastic Importance Resampling (SIR). The goal of resampling is to keep a representative sample of particles, discarding particles that took trajectories too unlikely (low weight), while generating new ones with high weights.

The goal is to propose a software infrastructure pushing the limits of particle filters by the use of massive computing resources offered by the soon to come exascale era [19]. The ability to handle a very large number of particles is critical for high dimension models as encountered especially in geoscience applications. Scaling efficiently the number of particles to multiple thousands is challenging taking into consideration that each instance of the numerical model used to propagate one particle is a parallel code and that computing particle weights requires processing large amount of data.

The key to achieve this goal is the virtualization of particle propagation. We turn a numerical model code instance into a *runner* capable of propagating several particles one after the other with low overheads. Each runner is coupled with a node-local distributed state cache enabling fast particle switching. The particle's states are shared between runners through the parallel file system where they are persisted. Prefetching states in cache enables to overlap state loads with particle propagation. Runners are running as independent parallel codes, each one in its own batch scheduler allocation for flexibility and resilience purpose. Runners connect dynamically to a server on startup and provide it with the weight of each particle they compute. This server handles particle weight normalization, resampling and work distribution to runners. The association of these different features complemented with a fault-tolerance mechanism, leads to an elastic and resilient

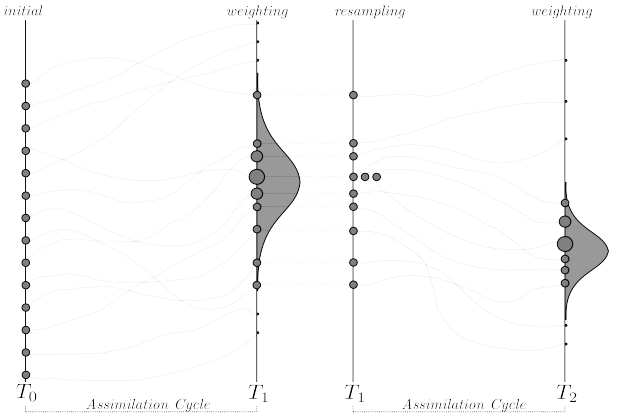

Figure 1: Initially particles are uniformly sampled. They are propagated to T_1 where they are weighted taking into account observation data. Resampling leads to discard some particles with low weights (top and bottom), while others with high weights become parent of several ones (3 here).

framework minimizing data movements while enabling dynamic load balancing. Particle virtualization enables to decouple 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. These features are important to ensure the execution can proceed efficiently to completion while leveraging the potential of exascale machines.

In comparison, classical existing approaches for ensemble-based DA either rely on temporary files to store particle states that are later read to compute their weights and resample them. Performance is impaired by the intensive use of the file system, a supercomputer performance bottleneck that is expected to worsen with next generation machines. The lack of particle virtualization requires to run one numerical model instance per particle with the associated init time overheads. Another classical approach consists in building a large MPI application that encompass the full workflow. Temporary files are avoided, but the resulting monolithic MPI code makes it difficult to ensure efficient elasticity and low cost fault-tolerance.

The proposed approach is experimented with a realistic use-case relying on the Weather Research and Forecasting (WRF, version 3.7.1) model [21]. The WRF model is a widely used weather model for practical forecasting and research purpose. Our approach enables to run 2,555 particles using 20,442 compute cores for WRF simulations on Europe with 87% efficiency.

5. Data Assimilation and Particle Filters

We remind the background on particle filters for data assimilation. Refer to [27] for an extensive survey on the topic. Particle filters use a set of particles to represent the probability distribution of a process state given noisy and partial input or models relying on heuristics. Particle filters distinguish themselves from other DA methods like EnKF or 3/4DVar by supporting flow dependent errors, nonlinear state/space models, and the distributions of observation errors and initial states can follow any probability distribution. These properties motivate their use for weather forecasting [7] similar to the experiments in this paper.

Let $\mathcal{M}$ be a numerical model, that *propagates* a model state x in time, i.e., computes

the next time step (t) from a model state x_{t-1}:

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

Let y_t be an observation obtained by measures on the real system (satellite images, buoys, ground measuring stations...). This data is often not homogeneous to the state of the numerical model. The operator $\mathcal{H}$ projects a model state into an observation state. We consider that the observation is the projection of the true state plus an error ϵ whose distribution p_ϵ is known:

$$y_t = \mathcal{H}(x_t^{\text{true}}) + \epsilon_t \tag{2}$$

DA acts by *assimilation cycles* (Figure 1). First the M particles $x_{i,t-1}$ are propagated through the model $\mathcal{M}$ up to the next time step when observations are available (t for simplicity here). Then the normalized particle weight $\hat{w}_{i,t}$ that approximates the probability $p(x_{i,t}|y_t)$ is computed for each particle state $x_{i,t}$:

$$\hat{w}_{i,t} = \frac{1}{M}\frac{p(y_t|x_{i,t})}{p(y_t)} \approx \frac{w_{i,t}}{\sum_{0\leq j<M} w_{j,t}}, \tag{3}$$

where

$$w_{i,t} = p(y_t|x_{i,t}) = p_{\epsilon_t}(y_t - \mathcal{H}(x_{i,t})). \tag{4}$$

Finally, the probability density function of $p(x_t|y_t)$ is approximated by:

$$p(x_t|y_t) = \sum_{0\leq i<M} \hat{w}_{i,t}\delta(x_t - x_{i,t}), \tag{5}$$

where δ is the Dirac measure.

Especially for high dimensional problems, particle filters tend to suffer from weight degeneration, i.e., one normalized weight is close to one and all the others to zero, making the particle sample meaningless. To avoid this issue, one classical approach consists in resampling the particles based on their importance (SIR, Sequential Importance Resampling) before starting the next cycle. A new ensemble of M particles is drawn, each one with a probability $\hat{w}_{i,t}$. This leads to discard low weight particles while high weight ones can become the *parent* of multiple new particles (Figure 1). Particles from the same parent state may need to be slightly perturbed if the model does not contain a stochastic component, so they do not propagate to the same state.

6. Architecture

From a data dependency point of view, the propagation and weight computation of each particle can be performed in parallel, while weight normalization and resampling require to aggregate all particle weights (see Section 5). The proposed framework relies on a runners/server architecture that directly derives from these dependencies to implement the particle filter (Figure 2). In a nutshell, the *runners* propagate states and compute the unnormalized weights. The *server* gathers the weights from runners, normalizes and resamples them, and dynamically distributes state propagation work to runners. The various runner instances are independent jobs that connect to the server at startup. The server and runners communicate by direct message exchanges. In the following we further detail the architecture and the interactions between the different components.

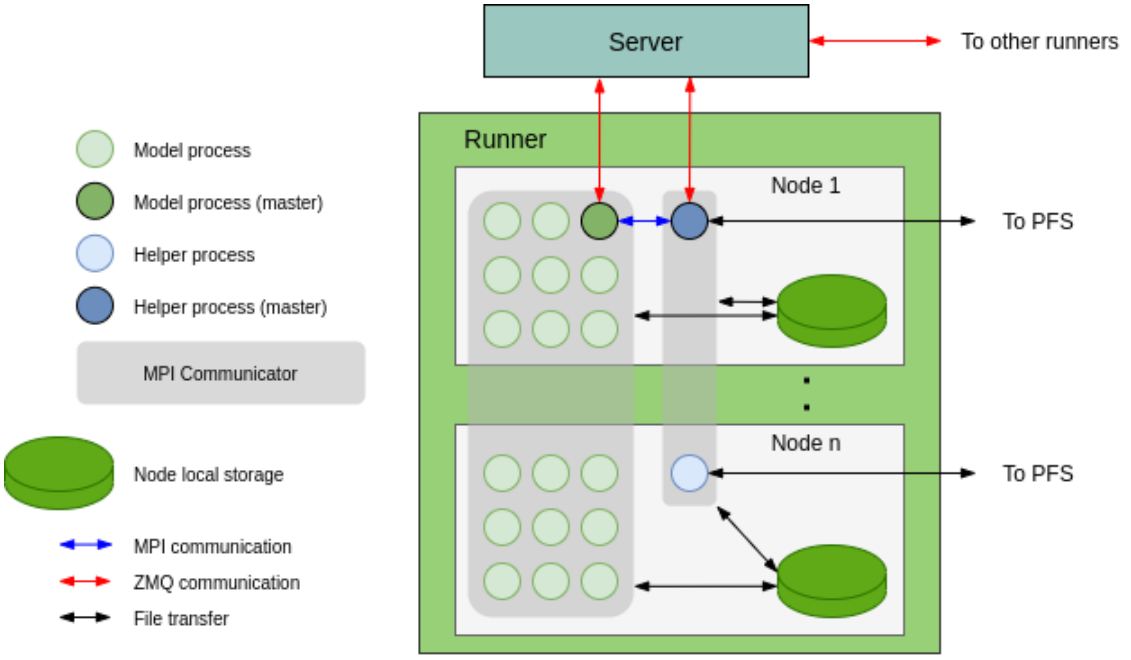

Figure 2: Runners/server architecture. The model processes perform the state propagation, the helper processes send propagated states to the PFS and prefetch next scheduled states to the local cache in the background. Communications with the server combine MPI and ZMQ data exchanges.

6.1 Runners

Runners are built from the simulation code, often an advanced parallel code or even a coupling of several parallel codes, with significant start times to load and build the different internal data structures. To avoid paying the cost of a restart for each state propagation, we augment the simulation code with a mechanism to store and load particle states. A runner can load a state, propagate it, store the produced state, load another state and so on. This is the base of the particle virtualization mechanism mentioned in the introduction.

Particle states are stored and loaded from a state cache associated to each runner. This state cache is distributed among the runner nodes and can leverage any node-local storage device (including the RAM disk). The cache is maintained by one *helper process* per node. Helper processes are in charge of moving states between the cache and the parallel file system (PFS). These operations are performed *asynchronously* while the *model processes* propagate the states, enabling to overlap the associated I/O costs.

The model processes also compute the associated (unnormalized) weight $w_{i,t}$ after propagating a particle state $x_{i,t}$ to $x_{i,t+1}$. For that purpose each runner also needs to load the observation data y_t, only once per cycle as y_t is shared amongst all particles (see Section 5). The observations are loaded by the model processes. Notice that the size of observation data is typically much smaller than the size of a particle state.

A typical DA run relies on several runners to propagate states. Each runner is submitted on its own job allocation and thus, runs on different resources. Runners can be dynamically added or removed at runtime. This can occur on a runner fault. The runner is then restarted by the fault-tolerance algorithm, or for elasticity purpose if resource usage needs to be adapted under the control of an external resource manager.

Storing the states to the PFS is necessary for load balancing and fault tolerance. The same parent state can be propagated several times. By storing states in the PFS, several runners can load the same parent state to perform the different associated propagations in parallel. State storage on the PFS is also used for fault tolerance, since states remain available for propagation after runner failures.

6.2 Server

Runners send unnormalized weights to the server when they finish a particle propagation. Note that a weight is a simple scalar value and therefore, results in very small data transfers. Once all unnormalized weights $w_{i,t}$ are gathered, the server normalizes them and starts the resampling process. Based on their normalized weight $\hat{w}_{i,t}$, some particles are selected to become parent particles for the next cycle, in some cases to be propagated several times, while others are discarded (Figure 1). Notice that we keep a constant number of M particles from one cycle to the other.

The server is next responsible to schedule the propagation work to runners in a sequence that minimizes direct loads from the PFS (particle sate cache misses) and particle state transfers in general. For this, the server relies on lightweight communications with the runners' helper processes. These additional communications take place asynchronously, meanwhile the particle propagations are performed by the model processes. They enable the server to maintain a view on the runners' cache content and to control cache evictions and state prefetching. The scheduling policy is explained with more detail in Section 6.5 and the cache prefetching and eviction strategy in Section 6.7.

6.3 Launcher

The launcher bundles the framework execution. This is the user entry point to configure and start the application. The launcher starts first and is next responsible to start and monitor the runner and server instances. The launcher can also kill or start instances at runtime, if required for elasticity or fault-tolerance purpose. The launcher is the only component that interacts with the job scheduler of the supercomputer (e.g., slurm), to monitor job status, or to request new resources when new server or runner instances are required.

6.4 Runners/server workflow

Once a runner job has started, the model processes request the server a particle id to propagate (Figure 2). Upon the first request, the server notifies the launcher about the successful runner startup. Afterwards, the server selects a particle id following the scheduling policy (Section 6.5) and replies to the runner to trigger the particle propagation. The model processes then check the location of the corresponding particle state. If the state is in the local cache, the model can start with the propagation immediately. Otherwise, the model processes request the helper processes to load the state into the cache. The model processes are blocking until the helper processes finish the state transfer from the PFS. Once the state is in the local cache, the model can begin the propagation.

Once a particle propagation finishes and its unnormalized weight is computed by the model processes, the particle state is stored to the local cache by the model processes. Afterwards, the weight and some additional metadata are sent to the helper processes and the model processes request the next job from the server. The helper processes, after

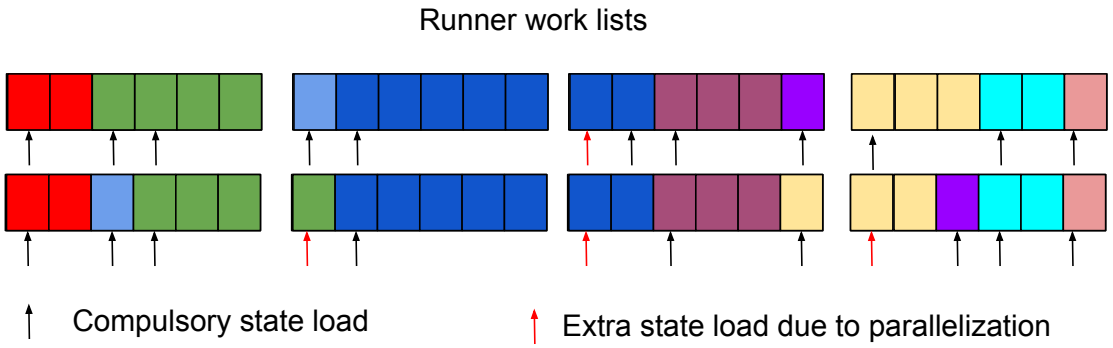

Figure 3: Two possible schedules of 24 propagation tasks of equal duration on 4 runners. All particles propagated from the same parent state have the same color (9 parents here). Top schedule is optimal with 9 compulsory loads (one per parent), and one for the dark blue parent that cannot fit in one runner. The bottom schedule, with 2 more sate loads, is a possible one that our on-line scheduling algorithm can produce. This is not optimal but still bellow the general $P + R - 1$ 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 state.

receiving the weight from the model processes, copy the state from the local cache to the PFS and forward the weight to the server. This ensures that the server only handles weights with safely stored states.

To avoid blocking transfers from the PFS (i.e., cache misses), the helpers prefetch states to the local cache each time a weight is sent to the server. The prefetching is coordinated by the server that gives the helpers the next state to propagate, in parallel with the current propagation performed by the model processes. Prefetching is suspended at the end of each new propagation cycle, as propagation work for the next cycle is known only once the server has resampled particles. Prefetching proved to be very efficient for overlapping particle state loads with propagation (Section 7.2). The server and runners also interact for removing states from the cache once full (Section 6.7).

6.5 Scheduling

In this section we present the scheduling algorithm used by the server to distribute the particle propagations to the runners. Let R be the number of runners. Let $(p_i)_{0 \leq i < P}$ be the P parent particle states selected for the next assimilation cycle. The total number of particles to be propagated is $M = \sum_{0 \leq i < P} \alpha_i$, where α_i is the number of times the parent p_i needs to be propagated.

We first derive a lower and upper bound for the minimum number of particle state loads per assimilation cycle c^* in the 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. In these conditions, each runner needs to propagate $\frac{M}{R}$ particles. Because each parent state needs to be loaded at least once, the number of compulsory state loads is P. If $\alpha_i = 1$ for all $0 < i \leq P$, i.e., every parent state is only used for one job, then $c^* = P$. Otherwise, parallelizing the propagation can require some parent particles to be loaded on more than one runner, accounting for some extra state loads beyond the compulsory ones. Indeed, each p_i needs to be loaded into at least s_i runners where

$$s_i = \left\lceil \frac{\alpha_i}{\frac{M}{R}} \right\rceil. \tag{6}$$

But as we have R runners, the list of M particles to propagate is split at most $R-1$ times, and so these extra state loads are at most $R-1$ (Figure 3). This occurs if all particles are propagated from a single parent ($\alpha_0 = M$ and $\alpha_i = 0$ for $i \neq 0$): $c^* = R$. Thus, in the general case the minimum number of state loads c^* is tightly bounded by

$$P \leq c^* \leq P + R - 1. \tag{7}$$

We can define a static schedule that respects this upper bound: distribute $\frac{M}{R}$ particle per runner, where each parent state p_i is given to no more than $\lceil \frac{\alpha_i}{\frac{M}{R}} \rceil$ runners, and by imposing that runners do not switch to the next state without finishing all propagations associated to the current one first. But this static schedule is not suitable in our case as the number of runners can vary during executions, and the time it takes to propagate a given particle state is unknown and can be uneven. Our extension to a dynamic case relies on dynamic list scheduling to ensure an efficient load balancing [12,20]: when idle, a runner requests work from the server that returns a particle to propagate. The execution time using the list scheduling algorithm is guaranteed to be at worst twice as long as the optimal schedule that requires to know the particle propagation time in advance (in our case particle propagation times are unknown in advance). The assigned propagation may require a state load. To ensure a low number of loads, we augment the list scheduling algorithm with a parent state distribution algorithm. The scheduling policy is based on the split factor s_i defined in Equation (6), but recomputed each time needed with the updated values α_i and M of the remaining work to do, and the current number of active runners. The split factor tells us among how many runners at most one parent state can be split. To support this algorithm, the server needs to know the parent state p_i currently propagated by each runner, and for each parent state i, the number of runners that are currently propagating one of its instances. The particle distribution algorithm works as follows:

1. If $\alpha_i > 0$ for the last parent state, p_i, propagated by the runner, decrease α_i and assign p_i again.
2. Otherwise, select another parent state p_j and compute the associated split factor s_j. By default any particle could be selected, but as runners are coupled with a state cache, priority goes to particles whose state is already loaded in the runner cache (see Section 6.7).
3. If s_j runners are already scheduled to propagate parent state p_j go back to 2).
4. Otherwise, assign p_j, decrease α_j by one and register p_j as being propagated by this runner.

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

In conditions of even propagation time and static runners, this algorithm behaves like the static schedule and so respects the upper bound of Equation (7).

6.6 Fault tolerance

The fault tolerance of the framework relies on the fast recognition of failures from any of the three components. Runner failures are detected in two different ways. Runner crashes can be recognized by the launcher, which is monitoring their execution using the

cluster scheduler. Unresponsive runners are also detected by the server using due dates for the assigned particle propagations. After recognizing the violation of a due date, the server notifies the launcher and the launcher terminates this runner execution. In either case, when a runner failure has been detected, a new instance is scheduled by the launcher. The new runner connects to the server and is incorporated in the runner team.

Server failures are detected either directly, when the server crashes, again relying on the schedulers functionality, or due to timeouts, when not responsive anymore. The timeouts are implemented using heartbeats exchanged between launcher and server. On server failure, the launcher terminates all runner instances and restarts server and runners.

A launcher failure is detected by the server monitoring the launcher through a heartbeat connection. In case of a missing heartbeat, the server checkpoints the current state and shuts down. Runners detect the server crash on a timeout on the connection to the server and shut down as well. The application can next be restarted from where it has left off.

6.7 Cache eviction strategy

The number of particle states that can be kept in the cache is limited by the node storage capacity, the node memory when using a RAM disk to implement the cache. The helper processes interact with the server to implement the eviction of particle states from the cache when required. The cache needs to provide at least 2 slots, one to store the resulting particle state from the current propagation and one to store the next scheduled parent state loaded by prefetching. As we explained in Section 6.4, each time a state has been stored in the cache by the model processes upon the successful propagation, the helper processes copy it to the PFS. These states can potentially be selected for eviction, since they are safely stored. Thus, after copying the particle state, the helpers check if the cache can fit the next propagation's output particle state. If not, one particle state has to be evicted.

When an eviction is required, the server selects the state to evict from cache in the following order:

1. A discarded state from the previous assimilation cycle.
2. A parent state from the current cycle for which all associated particles have already been propagated and all weights received.
3. The propagated state from the current cycle with the lowest weight.
4. A randomly selected state.

The states for cases 1 and 2 can safely be removed from the cache, since those states will not be needed anymore for future propagations. In case 3, we select the state with the lowest weight, as this is the least likely state to serve as a parent state in the next cycle. Experiments (Section 7) show that this cache management strategy, coupled with the scheduling algorithm, leads to a number of loads from the PFS that is below the lower bound derived in Section 6.5.

6.8 Implementation details

The runner local caches built up a *distributed cache* where the server has knowledge of the content of each local cache instance and selects the states propagated on each runner accordingly. Each runner local cache, again, is distributed among the nodes allocated for the runner. The local caches are implemented using the Fault Tolerance Interface (FTI) [6]. FTI is a multilevel checkpoint/restart library. FTI supports checkpointing into the Hierarchical Data Format (HDF5). Storing checkpoints on the PFS into shared HDF5 files allows further post processing for data analyses with a manifold of tools supporting HDF5. However, FTI also can store into one file per process using an opaque binary format, providing fast IO throughput. This approach is used here to optimize for performance. We take advantage of the dedicated processes (called *heads* in FTI jargon) that FTI provides for the checkpoint post processing. At initialization FTI splits the application's global MPI communicator into one for the application and another one for the head processes. The latter ones take over the role of the helper processes in our framework. In FTI, the head processes are used to perform certain tasks for the checkpoint creation in the background, asynchronously to the application execution.

FTI provides several levels of reliability for the checkpoints. The 1st level is a checkpoint local to the node and the 4th level is a checkpoint on the PFS. To store states to the local cache, we perform a level 1 checkpoint. We extended the library to enable duplicating local checkpoints into global checkpoints, which corresponds to a copy of states from the local cache to the PFS. That did not require significant implementation efforts, since the library already implemented such a feature, only that this was performed automatically. We just added the possibility to trigger the copy manually.

The communication between helper and model processes relies on asynchronous MPI messages. Communications with the server are implemented in two steps for efficiency purpose. Only rank 0 (*master*) of the application (i.e., model) communicator and the rank 0 (*master*) of the helper process communicator communicate with the server. As a dynamic connection is needed, each master connects to the server using a socket through the ZMQ library. Information that needs to be propagated between helper or model processes relies on MPI collective communications in the associated communicator (see Figure 2).

7. Experiments

7.1 WRF use case

Experiments rely on the widely used WRF model [21]. The core of WRF is based on solving fully compressible non-hydrostatic equations with complete Coriolis and curvature terms, and a large set of physics options. The simulation domain covers most Europe (See Figure 4) as 220 by 220 grid cells with horizontal resolution of 15 km and 49 vertical levels with uneven thickness to perform short–range weather forecasting. We randomly choose 2018-07-19 to simulate 48 hours by time steps of 100 seconds. The model employs the WSM6 microphysics, MYNN2 boundary layer physics, Grell-3 cumulus parameterization, Eta Monin-Obukhov similarity surface layer processes, and RUC land surface model. We also employ non-hydrostatics to have more details in simulated cloud and precipitation. The input initial and boundary condition is based on the reanalyzed ERA5 dataset from the European Center for Medium-Range Weather Forecasts (ECMWF). Data assimilation is performed using the cloud cover fraction (CFRACT). For each particle the cloud cover fraction is compared with its pendant obtained from the EUMETSAT CMSAF satellite data [23]. The observation data is available hourly, so we perform assimilation cycles

over 36 model time steps ($36 \times 100\,\mathrm{s} \mathrel{\widehat{=}} 1\,\mathrm{h}$) to assimilate all observation data testing our approach under high stress.

If not stated differently, the data presented in the following results are from a run over this European domain with 2,555 particles using 20,442 compute cores on 512 Nodes of the Jean-Zay supercomputer. Each compute node of Jean-Zay is 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 with each other while an IBM Spectrum Scale (ex-GPFS) parallel file system with SSD disks (GridScaler GS18K SSD) is used for persistent file storage. To capture the meteorological state of the European domain, each particle state accounts for 2.5 GiB of data.

Writing the full output of the 2,555 particle ensemble for the 48 h simulation period would produce almost 300 TiB of data. Post processing this amount of data is also a challenge. Future work will consider extending our framework to enable on-line data processing using approaches such as [24] for computing statistics. The experiments performed for this paper, from early test runs to large ones, account for about 220,000 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 (s)	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 state from PFS to cache (s)	2.1	2.1	2.4	4.1
Write 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)	0.30-0.4	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.

7.2 Runner activity

The excerpt of an execution trace (See Figure 5) shows the different particle propagations undertaken by some of the runners. During initial propagation the runners' local cache is not yet used but during later assimilation cycles it is visible that many of the parent particle states (up to 69 % per assimilation cycle) are already found in the cache of the according runner avoiding PFS loads.

The close-up view in Figure 6 shows the sequence of the most important tasks done by one runner's model and helper processes. It can be observed that model processes are almost completely occupied performing model propagation and weight calculation, overlapping very well the helper processes prefetching and particle state storing. This puts in light the benefit of using helper processes. The developed caching, prefetching and scheduling machinery can keep model processes busy with particle propagation most of

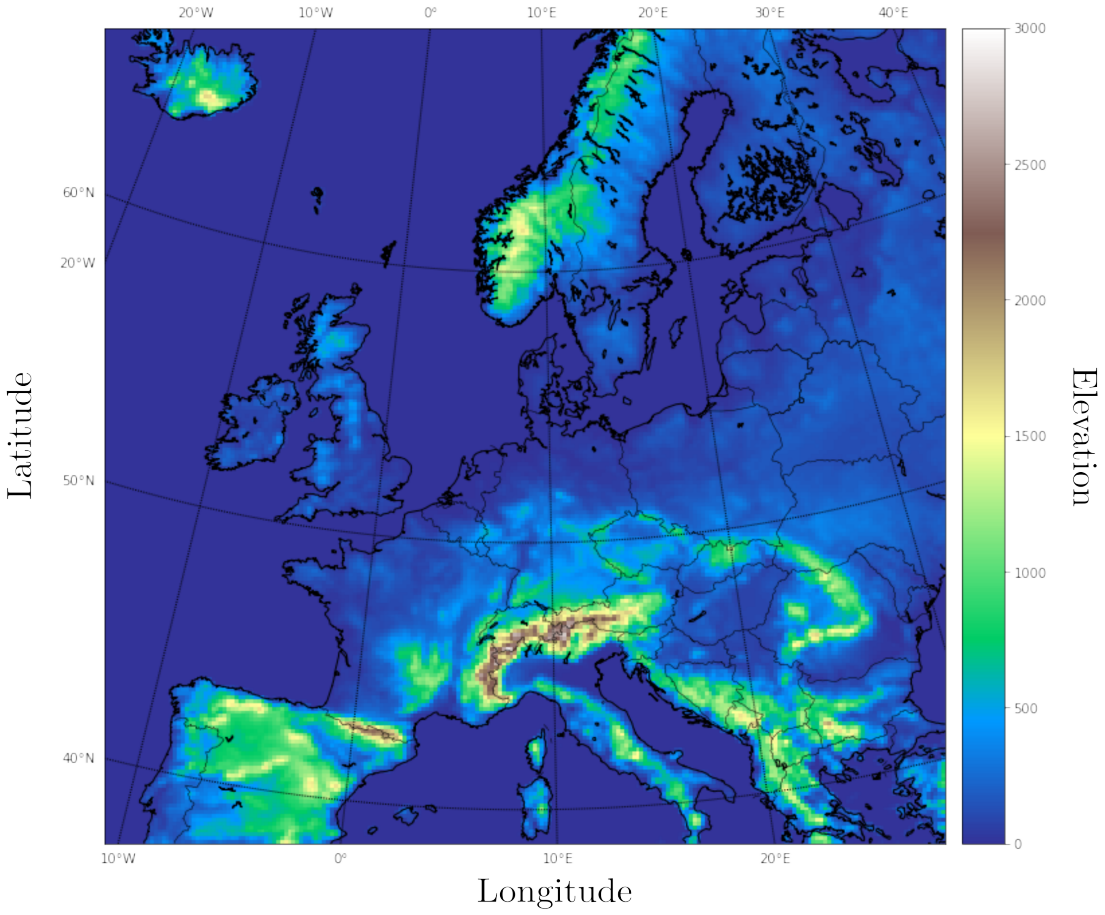

Figure 4: The topography of the target domain of Europe for the simulation.

the time, while, in parallel, helpers interact with the PFS. Some general idle periods can only be observed between assimilation cycles when runners are waiting for the server to normalize weights, resample and start to distribute work again.

Propagations take place in parallel thanks to using several runners (Figure 5) and workload is well balanced. After all runners joined (after assimilation cycle 2), they propagate 5 particles each per assimilation cycle since the time it takes to propagate particles with the used WRF setup is very even showing only 10% of fluctuation at maximum. For a few cases during an assimilation cycle, some thin idle periods can be observed between particle propagations when model processes need to wait for the prefetching to complete.

The benefits of elasticity are also visible. At the beginning of a study, runners can start propagating as soon as ready even if others take significantly more time to join. A single particle propagation takes between 24 and 26.5 seconds, keeping model processes busy 87% of their time, otherwise performing weight calculation (1%) or communications with the server (12% that includes potential waiting time at the end of each cycle) (Table 1). By using one helper process per runner node and one node for the server ($\widehat{=}$ 2.7% of the total compute resources) 94% of the state loads are completely performed in the background lowering I/O costs. Loading and writing states would otherwise cost 4.1 s + 2.3 s each 25 s, increasing each particle propagation time by 14% (Table 1, 2,555 particle case).

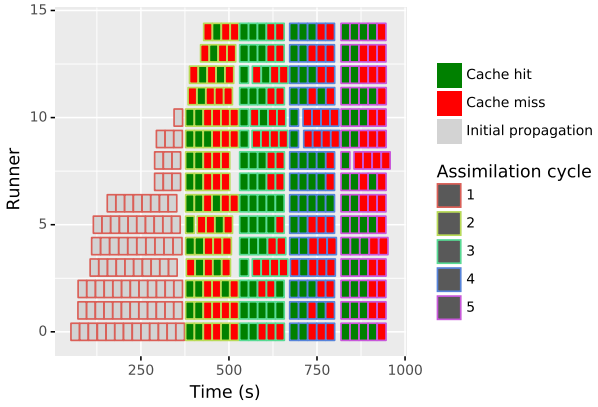

Figure 5: Gantt chart of particle propagations executed by 15 (out of 511) randomly selected runners over 5 assimilation cycles. Tasks are green when the associated parent state was already present in the runner cache and did not require a load from the PFS (red otherwise).

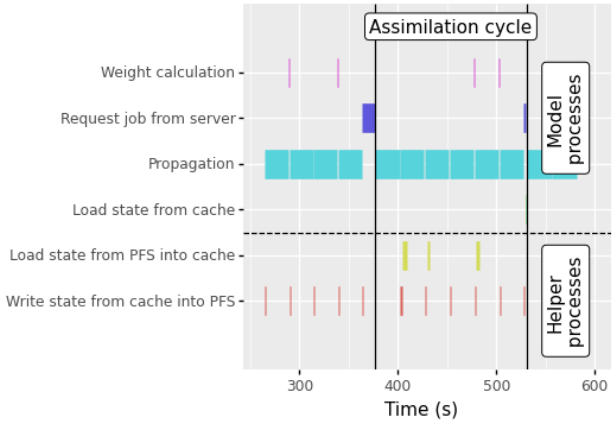

Figure 6: Trace detailing the activity of a runner over the course of an assimilation cycle. Helper processes enable to keep model processes busy with particle propagation, except at the end of assimilation cycles when they wait for the server to finish particle resampling (dark blue). Some activities are so thin that they are not visible here (state copies from cache to model). they can become idle

7.3 Server activity

We measure the reactivity of the server upon runner requests (Figure 7). Response time is in the order of a few hundred microseconds except for some job requests that take up to seconds. As already mentioned, these correspond to job requests at the end of an assimilation cycle that wait for the server to gather all weights and to normalize and resample particles before starting the next cycle. Using 511 runners, the server is loaded with 676 requests per second at maximum. Thus, 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).

7.4 State transfers to/from PFS

Each particle state leverages 2.5 GiB of data, leading to write about 6.2 TiB to the PFS for each assimilation cycle for the run with 2,555 particles (Table 1). Using a cache size of 5 particle states, between 1,024 and 1,563 particle states are loaded from the PFS into runner caches, saving 38% of the state loads necessary if each propagation would require

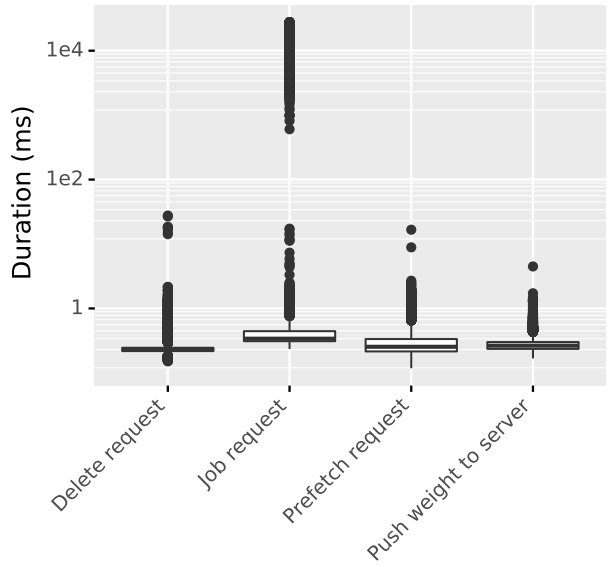

Figure 7: Server response times on runner requests.

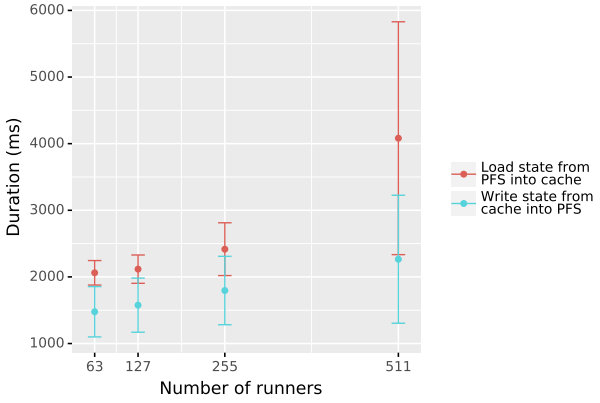

Figure 8: Mean time to load or store particle states of 2.5 GiB from / to the PFS with different numbers of runners.

a state load. The P value (number of parent states) per cycle is between 1,594 and 1,629, with parent particles used to propagate up to 5 particles each. The scheduling algorithm, without caching, is expected to achieve less than $P + R - 1$ loads (compare Equation (7)). Due to the runner's local caches even the minimal number P of state loads is undercut (See Section 6.5). The time to load or to store a state from the PFS can vary significantly and increases with the number of runners (Figure 8), showing that our application alone can stress the PFS (these numbers may also be impacted by other jobs). However, all tests performed on the Jean-Zay and the JUWELS supercomputers show that our framework enables to overlap the PFS access time with particle propagation (Section 7.2).

Applications where propagations are faster than state loads, would be impacted by PFS access. Notice that here we already have short propagation times in the WRF context as we perform hourly resampling. This is done to stress the framework, but production runs usually do not require such high frequency. We plan to extend our approach to use node-local persistent storage when available (SSD or NVRAM) instead of the PFS,

to bypass the PFS. But this requires significant changes to enable state sharing between runners and fault-tolerance.

7.5 Fault tolerance, elasticity and load balancing

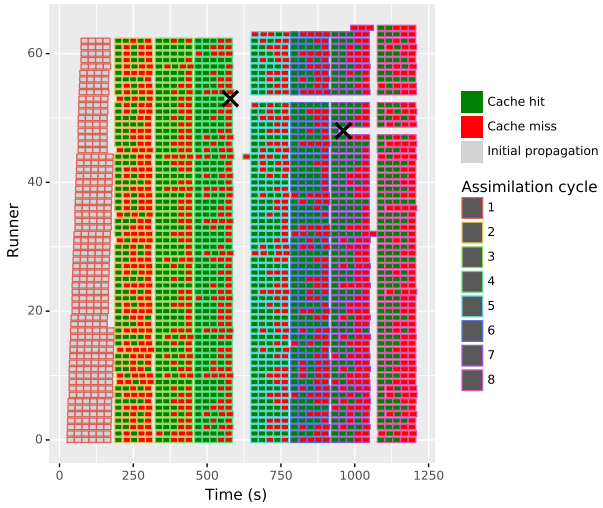

Figure 9: Gantt chart as in Figure 5. Two runners crashed (black cross) and 2 restarted (top 2 runners).

We test fault tolerance and elasticity on an execution with 63 runners where we crash 2 runners (Figure 9). 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 situation: just when propagating the last particle of the current cycle, leading to a significant idle period. The server needs to wait for the timeout (set to 60 s) to acknowledge that runner 53 is unresponsive and to start redistributing the particle it was working on to runner 44. As there is no work left except this single particle, all runners are idle meanwhile. The second crash does not lead to such idle period as the other runners are kept busy with propagation work. This Gantt also shows that the dynamic load balancing algorithm is efficient as the work load is kept well distributed amongst runners, even when their number varies. The same mechanism is used for elasticity to dynamically adapt the number of runners.

The particle propagation time is relatively even with at most a 10% variability. Situations with more variability are possible using different physics in WRF, with other simulation codes, or, if runners execute on heterogeneous resources - some runners propagating faster than others by leveraging GPUs for instance. Testing in such contexts is part of our future work.

7.6 Scaling

Testing strong scaling efficiency with a constant number of runners (63) and increasing the number of particles (Figure 10) shows that efficiency is above 90% with at least an average of 5 particles per runner. As prefetching enables to overlap I/O with propagations, increasing the number of particles per runner mainly enables to better amortize the cost of the synchronization associated with resampling. Taking this particle load, we next test weak scaling, increasing the number of runners (Figure 11). The time an assimilation cycle takes increases slowly (by 8%) from 62 to 511 runners. These good results make us confident that the framework can scale efficiently beyond 2,555 particles and 511 runners,

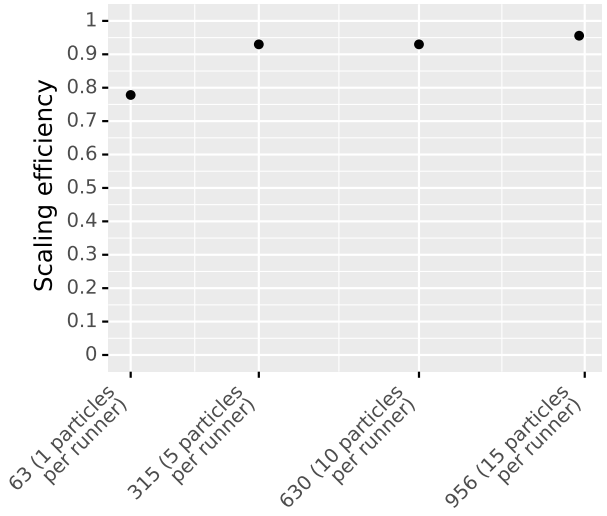

Figure 10: Strong scaling efficiency using different numbers of particles with 63 runners. One runner sets the reference case.

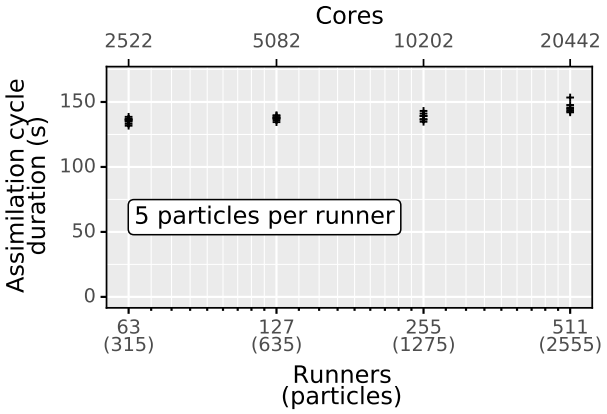

Figure 11: Weak scaling performance test: assimilation cycle duration for different numbers of runners, but always 5 particles per runner.

but we did not manage to get access to more than 20,442 cores.

Particle filtering with WRF on a European domain for short–range weather prediction at this scale is an important advancement of the previous work done by Berndt et. al. [7]. Besides assimilating at a higher frequency, the proposed framework offers fault tolerance, automatic filtering and elasticity while minimizing the file I/O and the time to calculate weights.

8. Related Work

The DA domain encompasses a large variety of techniques and algorithms, like nudging [18], kriging [30], ensemble Kalman Filter [33], ensemble maximum likelihood filter [34], or particle filter [26]. Refer to [2,9] for an overview. We focus here on statistical DA relying on an ensemble run of the model to compute a statistical estimator (co-variance matrix for EnKF, PDF for particle filters).

To aggregate the data produced by all members (i.e., particles) two main groups of

approaches are used. Either the data is stored to files and then processed in a second step (off-line mode), or the data is processed on-line usually within a large MPI code in charge of running the members and data processing.

Frameworks relying on the off-line mode include EnTK [4], with the largest published DA use cases reaching 4,096 members for a molecular dynamics application with an EnKF filter [3]. OpenDA also follows this model, using NetCDF for data exchange with the NEMO code [28]. DART supports both [1], with reports of large scale DA in off-line mode in [25] (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 to request a new resource allocation launching a new instance of the model code with all the associated start-up costs.

NVRAM is expected to become standard on supercomputers in a near future, providing node-local persistent storage capabilities. This technology can enable to loosen the I/O bottleneck by storing intermediate files in NVRAM. Today NVRAM is not yet available on large machines, but SSDs are present on some. They are used for state storage in [31], but without specific fault tolerance mechanism. So if a node fails and the node-local storage becomes unavailable, the lost member states need to be recomputed. We expect to support node-local storage instead of the parallel file system for our approach in future work.

The on-line mode avoids the I/O bottleneck. PDAF [17], which supports both modes, has for instance been used on-line for the assimilation of observations into the regional earth system model TerrSysMP. DA was based on EnKF with up to 256 members [14]. ESIAS uses on-line DA via particle filters with up to 4,096 particles on a wind power simulation on Europe [7]. Notice that we work with the same WRF component of ESIAS in this paper, using a configuration with similar domain but at higher spatial resolution and with more advanced and more time consuming physics. We also find ad hoc MPI codes for on-line DA as in [15] (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. We experimented with a more flexible architecture supporting these features in [11], with the largest runs reaching 16 k members, 16 k cores for DA with EnkF for the hydrology code Parflow. But in that work, focused on the EnKF DA, the server centralizes all the model states to enable dynamic load balancing. This eases the implementation as checkpointing is centralized on the server, but impacts performance as it leads to intensive data movements, the states being sent back and forth between runners and server. Here, we propose an alternate architecture for particle filters, relying on distributed caching and checkpointing to suppress the server bottleneck and significantly reduce data movements. Notice that we rely on filters that do not compute internal state corrections. Extending our approach to such particle filters [27] would likely require to aggregate more than just the particle weights to the server. This will be addressed in future work.

The framework proposed here is validated with SIR particle filters, but many variations exist and are active research topics. One challenge is the need for growing exponentially the particle number with the dimension of the problem [10, 22]. This is particularly

acute for geoscience use cases that, as in this paper, work in high dimensional spaces. The survey [27] gives an extensive overlook of DA by particle filters for geoscience and ways to cope with dimensionality issues. Particle filters, as used here, require a synchronization point at the end of each assimilation cycle. For our framework, this is the major remaining source of efficiency. Loosening this synchronization point requires revisiting the particle filtering algorithm, an active topic of research [8, 13, 16, 29].

9. Summary and Perspectives

We developed an operational first version of Melissa-DA with two flavours, one for data assimilation relying on the PDAF assimilation engine, and one on particle filters. The particle filter implementation is a result of a strong collaboration with WP4. We support the two expected use cases, Meteorology and Water for Energy with runs reaching 16k members for the ParFlow/Water use case, and 1500 members for the WRF/Meteorology one. These early results, at already a significant scale, demonstrate the benefits of having a modular, elastic, and fault tolerant architecture for large scale data assimilation. It has been tested on three large supercomputers (JUWELS, Jean-Zay and MareNostrum).

Future work will focus on:

- Code and documentation consolidation
- Integration of both Melissa-DA flavors
- Support of PDI (WP4) for further easing the integration of new simulation code and data assimilation engines
- Investigations to further improve performance on two fronts:
  - Leverage on-node storage such as NVRAM that will likely become common on future supercomputers
  - Alternative analysis strategies that do not impose a (costly) synchronization point, which will become a bottleneck at very large scale
- Support of coupled codes (e. g., ParFlow-CLM) for production runs
- Publications and communications

10. Acknowledgments

This work was granted access to the HPC resources of IDRIS under the allocation 2020-A8 A0080610366 attributed by GENCI (Grand Equipement National de Calcul Intensif). The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC). We also acknowledge PRACE for awarding us access to JUWELS at Jülich Supercomputing Centre (JSC), Germany.

11. References

[1] Jeffrey Anderson, Tim Hoar, Kevin Raeder, Hui Liu, Nancy Collins, Ryan Torn, and Avelino Avellano. The Data Assimilation Research Testbed: A Community Facility. *Bulletin of the American Meteorological Society*, 90(9):1283–1296, September 2009. Publisher: American Meteorological Society.

[2] Mark Asch, Marc Bocquet, and Maëlle Nodet. *Data assimilation: methods, algorithms, and*

applications, volume 11. SIAM, 2016.

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

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

[5] Peter Bauer, Peter D. Dueben, Torsten Hoefler, Tiago Quintino, Thomas C. Schulthess, and Nils P. Wedi. The digital revolution of earth-system science. *Nature Computational Science*, 1(2):104–113, February 2021.

[6] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: High performance Fault Tolerance Interface for hybrid systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, November 2011.

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

[8] Víctor Elvira, Joaquín Míguez, and Petar 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):1781–1794, 2016.

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

[10] Paul Fearnhead and Hans Künsch. Particle Filters and Data Assimilation. *Annual Review of Statistics and Its Application*, 5(1):421–449, March 2018. arXiv: 1709.04196.

[11] Sebastian Friedemann and Bruno Raffin. An elastic framework for ensemble-based large-scale data assimilation. Research Report RR-9377, Inria Grenoble Rhône-Alpes, Université de Grenoble, November 2020.

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

[13] Ajay Jasra, Anthony Lee, Christopher Yau, and Xiaole Zhang. The alive particle filter. *arXiv preprint arXiv:1304.0151*, 2013.

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

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

[16] Lawrence M Murray, Sumeetpal Singh, Pierre E Jacob, and Anthony Lee. Anytime monte carlo. *arXiv preprint arXiv:1612.03319*, 2016.

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

[18] Valentijn R. N. Pauwels, Rudi Hoeben, Niko E. C. Verhoest, and François 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):88–102, September 2001.

[19] Thomas C. Schulthess, Peter Bauer, Nils Wedi, Oliver Fuhrer, Torsten Hoefler, and Christoph Schar. Reflecting on the Goal and Baseline for Exascale Computing: A Roadmap Based on Weather and Climate Simulations. *Computing in Science & Engineering*, 21(1):30–41, January 2019.

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

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

[22] Chris Snyder, Thomas Bengtsson, and Mathias Morzfeld. Performance Bounds for Particle Filters Using the Optimal Proposal. *MONTHLY WEATHER REVIEW*, 143:12, 2015.

[23] M. Stengel, A. Kniffka, J. F. Meirink, M. Lockhoff, J. Tan, and R. Hollmann. Claas: the cm saf cloud property data set using seviri. *Atmos. Chem. Phys.*, 14(8):4297–4311, April 2014.

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

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

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

[27] Peter Jan Van Leeuwen, Hans R Künsch, Lars Nerger, Roland Potthast, and Sebastian Reich. Particle filters for high-dimensional geoscience applications: A review. *Quarterly Journal of the Royal Meteorological Society*, 145(723):2335–2365, 2019.

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

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

[30] John L. Williams and Reed M. Maxwell. Propagating subsurface uncertainty to the atmosphere using fully coupled stochastic simulations. *Journal of Hydrometeorology*, 12(4):690–701, 2011.

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

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

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

[34] Sara Q. Zhang, Milija Zupanski, Arthur Y. Hou, Xin Lin, and Samson H. Cheung. Assimilation of precipitation-affected radiances in a cloud-resolving wrf ensemble data assimilation system. *Monthly Weather Review*, 141(2):754–772, 2013.