Ensemble-based data assimilation for large scale simulations

Sebastian Friedemann

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Sebastian Alexander FRIEDEMANN

Thèse dirigée par **Bruno RAFFIN**, Directeur de Recherche,
Université Grenoble Alpes

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'École Doctorale **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Assimilation de données par ensemble pour les simulations à grandes échelles

Ensemble-based Data Assimilation for Large Scale Simulations

Thèse soutenue publiquement le **1er juillet 2022**,
devant le jury composé de :

Bruno RAFFIN
Directeur de Recherche, Inria Grenoble, Directeur de thèse
Mark ASCH
Professeur des Universites, Universite de Picardie Jules Verne, Rapporteur
Lars NERGER
Docteur en Sciences, Alfred-Wegener-Institut Helmholtz-Zentrum für Polar- und Meeresforschung, Rapporteur
Martin SCHREIBER
Professeur des Universites, Universite Grenoble Alpes, Président
Jean-Martial COHARD
Maitre de Conference, Universite Grenoble Alpes, Examinateur
Christian PEREZ
Directeur de Recherche, Inria Lyon, Examinateur

I dedicate this thesis to my mother, who always supported me and gifted me a book on chaos theory when I was about 15, maybe secretly hoping this would make me tidy up my room.

Acknowledgments

I would like to thank Bruno Raffin for making this Ph.D. thesis possible, supervising me, and especially for the valuable help and guidance he provided. I am also very thankful for the excellent collaboration with our EoCoE-2 project partners Kai Keller and Yen-Sen Lu. Kai Keller played an important role in the architecture design, implementation, and testing of the particle filter extension and server side checkpointing for EnKF of the developed framework, whereas Yen-Sen was always there to provide and exchange on Earth science use cases. Next, I would like to offer my special thanks to the research engineers Christoph Conrads and Théophile Terraz of the DATAMOVE team for the inspiring discussions and collaboration on the Melissa-DA architecture. Further, I would also like to gratefully acknowledge all the other EoCoe-2 project partners who allowed an enriching inter-institutional and international exchange, that finally provided motivating use cases and opens many further research directions. I would also like to thank Víctor Elvira from the University of Edinburgh for the productive discussions on adaptive particle filters. I am very thankful for all the members of the DATAMOVE and POLARIS teams at Inria Grenoble, who were always available to exchange on various topics providing a very productive work environment. I also want to thank all the colleagues, friends, flatmates, and family members that helped, not only to proofread this manuscript.

This project has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 824158 (EoCoE-2). This work was granted access to the HPC resources of IDRIS under the allocation 2020-A8 A0080610366 attributed by GENCI (Grand Equipement National de Calcul Intensif). The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS Supercomputer JUWELS at Jülich Supercomputing Centre (JSC). We also acknowledge PRACE for awarding us access to JUWELS at Jülich Supercomputing Centre (JSC), Germany.

Abstract / Résumé

Abstract

Prediction of chaotic and non-linear systems like weather or the groundwater cycle relies on a floating fusion of sensor data (observations) with numerical models to decide on good system trajectories and to compensate for non-linear feedback effects. Ensemble-based data assimilation (DA) is a major method for this concern. It relies on the propagation of an ensemble of perturbed model realizations (members) that is enriched by the integration of observation data. Performing DA at large scale to capture continental up to global geospatial effects, while running at high resolution to accurately predict impacts from small scales is computationally demanding. This requires supercomputers leveraging hundreds of thousands of compute nodes, interconnected via high-speed networks. Efficiently scaling DA algorithms to such machines requires carefully designed highly parallelized workflows that avoid overloading of shared resources. Fault tolerance is of importance too, since the probability of hardware and numerical faults increases with the amount of resources and the number of ensemble members.

Existing DA frameworks either use the file system as intermediate storage to provide a fault-tolerant and elastic workflow, which, at large scale, is slowed down by file system overload, or run large monolithic jobs that suffer from intrinsic load imbalance and are very sensible to numerical and hardware faults. This thesis elaborates on a highly parallel, load-balanced, elastic, and fault-tolerant solution, enabling it to run efficiently statistical, ensemble-based DA at large scale. We investigate two classes of DA algorithms, the ensemble Kalman filter (EnKF), and the particle filter algorithm with sequential importance resampling (SIR), and validate our framework under realistic conditions. Groundwater sensor data is assimilated using a regional hydrological simulation leveraging the ParFlow model. We efficiently run EnKF with up to 16,384 members on 16,240 compute cores for this purpose. A comparison with an existing state-of-the-art solution on the same domain, running 2,500 members on 20,000 cores, shows that our approach is about 50 % faster. We also present performance improvements running particle filter with SIR at large scale. These experiments assimilate cloud coverage observations into 2,555 members, i.e., particles, running the weather research and forecasting (WRF) model over the European domain. To manage the many experiments performed on various supercomputers, we developed a specific setup that we also present.

Keywords: Data Assimilation, Ensemble Based, In Situ Processing, EnKF, Particle Filter, High Performance Computing

Résumé

La prédiction de systèmes dynamiques non linéaires et chaotiques, comme la météo ou le cycle de l'eau, s'appuie sur la combination de données observées et de modèles numériques. L'assimilation de données (AD) par ensemble est une méthode majeure pour sélectionner les trajectoires du système et compenser les effets de rétroaction non linéaires. Elle s'appuie sur la propagation d'un ensemble de réalisations de modèles perturbés qui est enrichi par l'intégration d'observations. La réalisation de l'AD à diverses échelles est exigeante en terme de calcul. Par exemple, il faut capturer les effets géospatiaux continentaux et mondiaux avec une haute résolution pour prédire avec précision leurs impacts sur les petites échelles. Cela nécessite des super-ordinateurs exploitant des centaines de milliers de nœuds de calcul, interconnectés par des réseaux à haut débit. La mise à l'échelle efficace des algorithmes d'AD sur de telles machines nécessite des processus fortement parallélisés spécialement conçus pour éviter la surcharge des ressources partagées. La tolérance aux pannes est également importante. La probabilité de défaillances matérielles et numériques augmente avec la quantité de ressources utilisées et le nombre d'ensembles simulés.

Les cadres d'AD existants utilisent le système de fichiers comme stockage intermédiaire pour fournir un processus élastique et tolérant aux pannes. Ce procédé est ralenti à grande échelle par la surcharge du système de fichiers. Une autre approche courante s'appuie sur un code monolithique qui souffre d'un déséquilibre de charge intrinsèque et est très sensibles aux fautes numériques et matérielles.

Cette thèse présente un système fortement parallèle, équilibré dynamiquement en charge, élastique et tolérante aux pannes, lui permettant d'exécuter efficacement l'AD par ensemble. Nous étudions deux classes d'algorithmes d'AD, le filtre de Kalman d'ensemble (EnKF), et l'algorithme de filtre particulaire avec rééchantillonnage d'importance séquentiel (SIR). Nous validons notre approche dans des conditions réalistes. Des données de capteurs d'eau souterraine sont assimilées à l'aide d'une simulation hydrologique régionale utilisant le modèle ParFlow. Nous exécutons efficacement EnKF avec jusqu'à 16 384 membres sur 16 240 cœurs de calcul. Une comparaison avec une solution de l'état de l'art, avec 2 500 membres sur 20 000 cœurs, montre que notre approche est environ 50 % plus rapide. Nous présentons également des améliorations de performance en exécutant le filtre de particules avec SIR à grande échelle. Ces expériences assimilent des observations de couverture nuageuse avec 2 555 membres, correspondant ici à des particules, en exécutant le modèle de weather research and forecasting (WRF) sur le domaine européen. Afin de gérer les nombreuses expériences réalisées sur différents super-ordinateurs, nous avons mis en place une configuration spécifique que nous présentons également.

Contents

Acknowledgments iii

Abstract / Résumé v

Contents ix

1. Introduction 1

2. Data Assimilation 5

2.1. Overview . . . 6

2.2. Variational Data Assimilation . . . 8

2.3. Statistical Data Assimilation . . . 10

2.3.1. Kalman Filter . . . 10

2.3.2. The Ensemble Kalman Filter . . . 12

2.3.3. The Particle Filter . . . 15

2.4. Discussion . . . 20

3. State of the Art 23

3.1. Large Scale Data Assimilation . . . 24

3.1.1. Variational Data Assimilation . . . 24

3.1.2. Statistical Data Assimilation . . . 24

3.1.3. Hybrid Methods Mixing Statistical and Variational Methods . . . 26

3.1.4. PDAF – An Existing Large Scale DA Framework . . . 27

3.2. Anytime Particle Filters, Island Particle Filters and Adaptive Ensemble Sizes . . . 29

3.3. Data Assimilation and Machine Learning . . . 31

3.4. Ensemble Frameworks . . . 33

3.5. In Situ Computing Frameworks . . . 34

4. Ensemble Kalman Filtering at Large Scale **37**

4.1. Melissa-DA Architecture . . . 38
4.1.1. Overview . . . 38
4.1.2. Server . . . 40
4.1.3. Runners . . . 41
4.1.4. Launcher . . . 41
4.1.5. Fault Tolerance . . . 42
4.1.6. Dynamic Load Balancing . . . 43
4.1.7. Data Flow . . . 45
4.1.8. Code . . . 45

4.2. ParFlow . . . 48

4.3. Experimental Study . . . 50
4.3.1. The Challenge of Setting Up an Ensemble . . . 50
4.3.2. The First Assimilation Cycle . . . 51
4.3.3. Ensemble Propagation . . . 53
4.3.4. EnKF Update Phase . . . 53
4.3.5. Runner Scaling . . . 54
4.3.6. Fault Tolerance and Elasticity . . . 60
4.3.7. Ultra-Large Ensembles . . . 60
4.3.8. Comparison of Melissa-DA and PDAF . . . 62

4.4. Conclusion . . . 69

5. Large Scale Particle Filtering **71**

5.1. Architecture . . . 72
5.1.1. Runners . . . 73
5.1.2. Server . . . 74
5.1.3. Runners/Server Workflow . . . 75
5.1.4. Scheduling . . . 76
5.1.5. Cache Eviction Strategy . . . 79
5.1.6. Fault Tolerance . . . 79
5.1.7. Cache Implementation . . . 80
5.1.8. Speculative Propagation . . . 81

5.2. Experiments . . . 83
5.2.1. The WRF Use Case . . . 83
5.2.2. Runner Activity . . . 84
5.2.3. Server Activity . . . 86
5.2.4. State Transfers to/from PFS . . . 88

5.2.5. Fault Tolerance, Elasticity and Load Balancing 90
5.2.6. Scaling 91
5.2.7. Speculative Propagation 93
5.3. Conclusion 99

6. Reproducible HPC Experimentation – a Case Study **101**
6.1. Motivation of Reproducible Research 102
6.2. Challenges of Reproducibility in HPC Experimentation 103
6.3. Our Approach to Reproducible HPC Research 105
6.3.1. Overview 105
6.3.2. Lab Notebooks 108
6.3.3. Capturing the Environment 109
6.3.4. Version Control 110
6.3.5. Deployment 110
6.3.6. Continuous Integration 112
6.4. Conclusion 113

7. Conclusion and Perspectives **115**
7.1. Conclusion 116
7.2. Perspectives 118

A. Sources of Greenhouse Gas Emission **A1**

Bibliography **A3**

List of Figures **A15**

List of Tables **A16**

1 Introduction

Many systems whose understanding has a high impact on mankind, like weather and climate, streamflow in the water cycle or biological relations like the predator-prey cycle are governed by chaos. In many other fields, reaching from heart rhythm irregularities to traffic jam prediction, chaotic conditions can be found too. These systems are driven by inner feedback loops that create very different outcomes although using very similar initial conditions. The most popular example of such chaotic behavior is the *butterfly effect* coined by the meteorologist and mathematician Edward Norton Lorenz, 1972. As a metaphor, he claims that the trajectory of a tornado may be influenced by the wing flaps of a distant butterfly several weeks earlier.

The sensitivity to initial conditions that all those systems have in common renders their use for prediction difficult. Given a slightly perturbed input, a model will produce a more perturbed output due to the chaotic nature of the underlying system. To perform predictions further in time, typically, the model is applied on its own (perturbed) output over and over again, strongly amplifying inaccuracy. To avoid this, the intermediate outputs may be corrected using observation data. This process is called Data Assimilation (DA). It is used in a manifold of situations, not only for chaotic systems that motivated its development at the beginning. DA is applied from operational weather forecasts using more than hundred thousand compute cores[1], to sensor fusion for smartphone's inertial measurement units (IMUs) detecting how the gadget is oriented in 3D space by using only a very limited amount of compute resources (Yan et al., 2020). While in weather forecasting noisy data from satellites, airplanes, ground measuring stations and much more is assimilated every few hours, smartphone IMUs combine results of simple Newtonian equations with measurements from accelerometers, gyroscopes and more, many times per second.

Model resolutions need to be able to capture important small scale effects impacting larger scales. Thus, modeling continental, or even global scale, Earth science problems, for instance, for numerical weather prediction (NWP), demands a large amount of computation. Typically such models need to run on supercomputers. The fastest supercomputers are capable of performing multiple petaflops up to exaflops (more

[1] Actors like the European Centre for Medium-Range Weather Forecasts (ECMWF, 2021) or Meteo France rely on such machines (*TOP500* 2022). Largest DA runs even use millions of compute cores (Yashiro et al., 2020).

than $10^{15} - 10^{18}$ floating-point operations per second) (*TOP500* 2022). For this purpose, such machines leverage up to millions of compute cores that must be used in parallel to reach peak performance. Supercomputer software must be carefully created, keeping in mind load balancing, synchronization, and access to shared resources like the file system of all the parallel running compute cores, to name just a few requirements to reach top performance. This holds true also for Earth science modelling even if typically less resources (ten to hundred thousand cores) are used (Kurtz et al., 2016; Bauer, Thorpe, et al., 2015). DA at this large scale becomes increasingly challenging. While model states get larger since their resolution increases, also more observations are available for assimilation since the advent of Big Data (Bauer, Thorpe, et al., 2015; Bauer, Dueben, et al., 2021; H. Jain and R. Jain, 2017). Mankind starts logging all kinds of data – be it on personal devices like smartphones and smartwatches, cars, and buildings up to new satellites for remote sensing. All this differently sourced data shall improve model output relying on DA.

This thesis work will discuss how to map existing DA techniques to the heavy compute resources necessary to integrate observation data into recent large scale, high-resolution models to improve accuracy. Multiple concepts, proven to perform well in high-performance computing (HPC), like in situ computing and different caching hierarchies, are thus applied to large scale DA. We develop a modular online DA framework that runs highly parallelized but is nevertheless fault-tolerant, load-balanced, and elastic. Fault tolerance is necessary to recover from failures of parts of the used compute resources. These become more probable the larger the used set of compute resources. Elasticity permits changing the amount of used compute resources at runtime, freeing resources for higher prioritized tasks, or adding compute resources to advance the DA faster. The framework ensures that the addition and removal of resources may be performed without losing any progress at any time. Integrated load balancing equilibrates the workload among all resources to minimize waiting times of the compute cores. Many DA approaches leverage the file system to exchange data between the different components of the DA workflow. This is an important source of slowdown as the file system is shared between all compute resources on HPC systems and thus is often a subject of overcharge. In contrast, the proposed framework performs online DA, exchanging data between components directly over the network, entirely circumventing file I/O. Experiments on supercomputers show that our framework is scalable and efficient to run on at least 20,442 compute cores and outperforms existing approaches.

This thesis manuscript is based on our publications, reports and preprints. The research described in Chapter 4 just got accepted for publication at The International Journal of High Performance Computing Applications with the title "An elastic framework for ensemble-based large scale data assimilation" authored by Sebastian Friedemann and

Bruno Raffin. The work presented in Chapter 5 is a cooperative work together with Kai Keller (Barcelona Supercomputing Center), but also Yen-Sen Lu (Forschungszentrum Juelich), Bruno Raffin (Inria Grenoble) and Leonardo Bautista-Gomez (Barcelona Supercomputing Center). It was submitted to the IEEE Cluster 2021 conference but got rejected. A revised version is under active development. We encountered difficulties to publish our work. One reason is finding publishers on the interface between computer science and Earth science, since often audiences have either a Earth science or a computer science background and fundamentals of the other discipline must be introduced carefully – but are necessary to understand the principles motivating our proposals.

This work has been presented at the following international events, based on peer-reviewed abstract selection:

- International Symposium on Data Assimilation[2] (2022): "Melissa-DA: An Elastic and Fault-Tolerant Large-Scale Online Data Assimilation Framework"
- 19th Workshop on high performance computing in meteorology[3] (2021): "Elastic Large Scale Ensemble Data Assimilation with Particle Filters for Continental Weather Simulation"
- EnKF Workshop 2021[4]: "A new framework for elastic ensemble-based data assimilation at large-scale"

This thesis is financed by EoCoE-2 – the Energy-oriented Center of Excellence. Many collaborations, be it with the community of the flagship codes WRF or ParFlow, as well as the collaborations with the FTI team, have originated from EoCoE-2 meetings discussions.

The manuscript is structured as follows: Chapter 2 introduces the bases of DA. We then examine other research done in the field of large scale DA and in situ computing in Chapter 3. Relying on the presented methods and technologies, we introduce our framework to improve the large scale performance of ensemble Kalman filters (EnKF) in Chapter 4. Chapter 5 develops an extension of our proposal to particle filters that allow to perform DA in cases where EnKF struggles. To validate our approaches, large scale experiments on HPC platforms are performed. In Chapter 6 we discuss methodological considerations for such experiments, before a conclusion is drawn and perspectives are elaborated in Chapter 7.

[2] `https://isda-online.univie.ac.at`, retrieved the 18.03.2022
[3] `https://events.ecmwf.int/event/169`, retrieved the 18.03.2022
[4] `https://enkf.norceprosjekt.no/previous-workshops/enkf-workshop-2021-free-online-event-`, retrieved the 18.03.2022

2 Data Assimilation

This chapter introduces Data Assimilation (DA). Different DA methods adapted for varying use cases are presented. At the end of the chapter, the challenge of performing large scale DA is discussed.

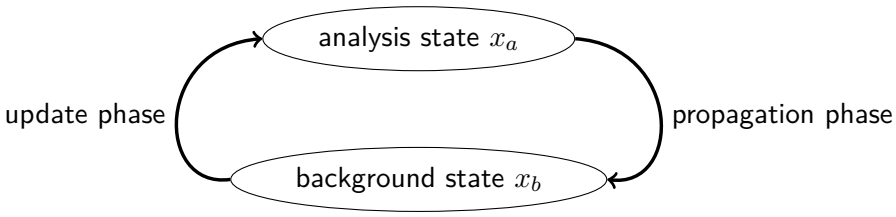


Figure 2.1: Data Assimilation flow. One propagation phase followed by one update phase is called assimilation cycle.

2.1 Overview

Data Assimilation (DA) tries to correct system state estimates returned by numerical models of typically chaotic systems, using observation data (Wikle and Berliner, 2007). The observation data is *assimilated* into the model. The model state can represent the atmospheric state in Numerical Weather Prediction (NWP), the soil moisture field when calibrating hydrological groundwater models or any other simulated measure that shall be corrected by DA. Observations in the geoscientific domain typically contain a mix of values from remote sensing instruments (satellites, airplanes) and in situ observations by ground-based observatories, buoys, airplanes, etc.

Let us introduce the formalism used in DA. Numerical models operate on the system state $x \in X \subseteq \mathbb{R}^n, n \in \mathbb{N}$ that cannot be directly observed. Consider a simple case of a model $\mathcal{M}$, i.e., a function, that, from an input state x_t, computes a new state x_{t+1}, also called the *background* (or *forecast*) *state* in DA. The background state x_{t+1} represents a new state at a point further in time $(t+1)$:

$$x_{t+1} = \mathcal{M}(x_t). \tag{2.1}$$

In the standard DA formalism, the *model operator* $\mathcal{M}$ fulfills the Markov property, taking the present state x_t as its *only* input to produce the next state x_{t+1}. The calculation of such background states happens during the *propagation phase*.

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

Let us assume the sum of these errors is unbiased (zero average). Let us also consider that time $t+1$ corresponds to a point in time where *observation data* y_{t+1} is available. Observations are provided by another source, typically a scientific measurement instrument. The observations are not necessarily of the same dimension, aligned with, or even

of the same nature as the model state. An observation operator $\mathcal{H}_t$ is introduced to project a model state vector x, into the observation space at time t:

$$\tilde{y}_t = \mathcal{H}_t(x). \tag{2.2}$$

The projection is usually defined this way as observation data are often of lower dimension than the model state. Like the background error, the observation error is also considered unbiased. Let the observation at time t be the projection of the true state plus an error ε_t with known probability density function (PDF) $P(\varepsilon_t)$:

$$y_t = \mathcal{H}_t(x_t^{\text{true}}) + \varepsilon_t. \tag{2.3}$$

Given the background state x_b from the last model iteration, the operator $\mathcal{H}_t$ and the observations y_t, the corrected system state, called *analysis state* x_a, can be computed. The calculation of meaningful analysis states is the core goal of DA. Assuming only one observation, the analysis state x_a will be between background state x_b and the observation y_t ($\mathcal{H}_t(x_a)$ will be between $\mathcal{H}_t(x_b)$ and y_t). If it is closer to the observation or the background state depends on the accuracy of each of them represented by background and observation error terms. Computing the analysis state x_a is called the *update phase*. One *propagation phase* followed by one *update (or analysis) phase* makes for one *assimilation cycle* (see Figure 2.1). For the next propagation phase, the model computes the state at time $t+2$ from the analysis state x_a, and not from the background state x_b (or x_{t+1}). Note that not only the observations y_t are time-dependent, but also the observation error, and the observation operator $\mathcal{H}_t$. For readability, the subscript t is omitted in most of the following equations.

Two main branches of methods to calculate the analysis state x_a exist, variational and statistical methods. While variational methods only provide corrected system states, i.e., analysis states, as output (e.g., *Tomorrow it rains.*), statistical methods additionally return uncertainty measures (e.g., rain forecast probability, *Tomorrow it rains with 60 % probability.*). Often the latter requires higher computational effort, but hybrid methods exist to combine the advantages of both.

We will briefly introduce variational methods (Section 2.2) that will help to give a basic understanding of the challenge behind DA. Then, we go into more detail in Section 2.3, explaining statistical methods which were implemented in the scope of this thesis.

For additional information on DA and methods, we recommend the reader the textbook by Asch et al., 2016.

2.2 Variational Data Assimilation

In this section, we dive briefly into variational DA. We will discuss 3D-Var, the most popular variational DA method. More details on this topic can for instance be found in Courtier, Andersson, et al., 1998 and Lorenc et al., 2000.

As mentioned, the challenge of DA consists in finding the analysis state x_a. Thus, observations and model forecasts must be taken into account. The analysis state will have minimal distances to the model forecast and the observations, where both distances are weighted by their uncertainty. When, for instance, the model error is larger than the instrument error of a specific state variable, the variable is shifted closer to the instrument's value in the analysis state. Assuming model and observation errors are unbiased, i.e., have zero mean, and follow Gaussian noise, they can be captured by their covariance only.

In 3D-Var, the model error covariance matrix is denoted as $\mathbf{B}$, and the observation error covariance matrix is denoted as $\mathbf{R}$. These matrices are used to weight the distance of a possible analysis state towards observations y and model output x_b. This leads to the 3D-Var cost function:

$$J(x) = \Big(x - x_b\Big)^{\mathrm{T}} \mathbf{B}^{-1} \Big(x - x_b\Big) + \Big(y - \mathcal{H}(x)\Big)^{\mathrm{T}} \mathbf{R}^{-1} \Big(y - \mathcal{H}(x)\Big). \tag{2.4}$$

Note that x must be transformed into observation space using $\mathcal{H}$ so that the distance to the observations is well defined: $y - \mathcal{H}(x)$. 3D-Var defines the analysis state x_a as the result from the minimization of $J(x)$:

$$x_a = \mathrm{argmin}_x \; J(x). \tag{2.5}$$

Typically, gradient descent is used to find the analysis state. Thus, the cost function must be differentiated. This is a major constraint of 3D-Var. The cost function needs to be formulated in a way that allows automatic differentiation, or its adjoint must be given manually. Especially advanced observation operators $\mathcal{H}$ may have no easily expressible adjoint.

Let us consider a linear observation operator $\mathcal{H}(x) \equiv \mathbf{H}x$. It follows:

- The analysis state estimate x_a is linear in the observed data set. To prove this, one reformulates the minimization problem in Equation (2.4) as $\frac{d}{dx} J(x) = 0$, which is a linear system.
- x_a is an unbiased estimate, $\mathrm{E}(x_a) = \mathrm{E}(x^{\mathsf{true}})$, since we assume that observation and model errors are Gaussian and unbiased.

- The estimate x_a is a *best* estimate having minimal variance among all linear and unbiased estimates as being the result of Equation (2.5).

Thus, the analysis state x_a, retrieved by Equation (2.5), is the best linear unbiased estimate (BLUE) for x^{true}.

Note that 3D-Var only permits assimilating observations every assimilation cycle. Measurements that fall to points in time in between need to be interpolated accordingly. The 4D-Var extension tackles this issue. Spatial and temporal distributions of observations are taken into account. For this purpose, the model becomes part of the 4D-Var cost function. To perform gradient descent, an adjoint of the model is required.

The classical formulation of variational DA assumes stationary model and observation errors. But these errors are not stationary as they vary over time. Weather models, for instance, can well predict calm atmospheric conditions one day, while the next day's turbulent situations are much more error-prone to predict. As we will see in the following chapter, statistical DA can handle such flow-dependent errors.

2.3 Statistical Data Assimilation

In the following, statistical DA is discussed. The methodology, vocabulary, and presented formalism will serve for the rest of this thesis, where statistical DA at large scale is considered. In comparison to variational DA, statistical DA allows the tracking of forecasts in a probabilistic way. Instead of one system state, a probability density function (PDF) $P(x_a)$ for the analysis state is obtained.

Statistical DA is based on Bayesian statistics. Events occur with a certain probability dependent on prior events. Following Bayes' theorem,

$$P(A|B)P(B) = P(A \cap B) = P(B|A)P(A), \tag{2.6}$$

the probability of events A and B coinciding, $P(A \cap B)$, can be calculated either by multiplying the probability of B by the conditional probability of A with the prior B or vice versa. Statistical DA searches the posterior PDF for the state vector x under the condition that y is observed. This matches the PDF of the analysis state. With Equation (2.6), this can be formulated as

$$P(x|y) = \frac{P(y|x)P(x)}{P(y)}. \tag{2.7}$$

$P(y)$ can be seen as a normalization factor, leading to:

$$P(x|y) \propto P(y|x)P(x). \tag{2.8}$$

$P(x)$ represents the knowledge of the simulated state x_b. $P(y|x)$ denotes the probability to measure y, given x would be the true state.

$$P(x_a) \propto P(y|x_b)P(x_b). \tag{2.9}$$

$P(y|x_b)$ can be obtained putting observations y in the error distribution (P_ε) of the used measurement instruments. When $P(x_b)$ is known, the analysis state can be computed. This is the DA update phase.

2.3.1 Kalman Filter

The Kalman filter (Kalman, 1960) uses Equation (2.9) for DA in a setting with Gaussian observation and model error distributions. It assumes linear model and observation operators. Kalman filters have their application in many fields of engineering and

science, be it for sensor fusion (S.-L. Sun and Deng, 2004) or to de-noise any kind of signals (Lakshmanan et al., 2012). We will introduce the Kalman filter in the following for the multidimensional case as it is the base for the ensemble Kalman filter (EnKF), a DA method that is widely used for large scale applications as those considered in this thesis.

Let n denote the dimension of the system state $x \in \mathbb{R}^n$, and let m denote the dimension of the observation vector $y \in \mathbb{R}^m$. Observation error and background states are represented by multivariate normal distributions. In n dimensional space such a distribution is defined as

$$\mathcal{N}(\mu, \Sigma). \tag{2.10}$$

by its mean $\mu \in \mathbb{R}^n$ and its covariance matrix $\Sigma \in \mathbb{R}^{n \times n}$. The PDF for observation and background states are identified as follows

$$y \sim \mathcal{N}(y, \mathbf{R}), \text{ and} \tag{2.11}$$

$$x_b \sim \mathcal{N}(\bar{x}_b, \mathbf{P}_b). \tag{2.12}$$

$\mathbf{P} \in \mathbb{R}^{n \times n}$ denotes the error covariance matrix of the state estimate. The indices b and a denote the background error returned by the last propagation phase or the analysis error updated by the update phase. The covariance matrix of the observation error is denoted $\mathbf{R} \in \mathbb{R}^{m \times m}$.

The multiplication of two multivariate normal distributions leads to another multivariate normal distribution (after normalization). Thus also the analysis state PDF $P(x_a)$ derived via Equation (2.9) follows the same distribution

$$P(x_a) \sim \mathcal{N}(\bar{x}_a, \mathbf{P}_a), \text{ where} \tag{2.13}$$

$$\mathbf{K} = \mathbf{P}_b \mathbf{H}^{\mathrm{T}} \left(\mathbf{H} \mathbf{P}_b \mathbf{H}^{\mathrm{T}} + \mathbf{R} \right)^{-1}, \tag{2.14}$$

$$\bar{x}_a = \bar{x}_b + \mathbf{K} \left(y - \mathbf{H} \bar{x}_b \right), \tag{2.15}$$

$$\mathbf{P}_a = \left(\mathbf{I} - \mathbf{K} \mathbf{H} \right) \mathbf{P}_b. \tag{2.16}$$

The calculation of its mean $\bar{x}_a$ and covariance $\mathbf{P}_a$ define the update phase of the Kalman filter. Using multivariate normal distributions and a linear observation operator $\mathbf{H}$, the increment between analysis and update state PDF is proportional to the *innovation*, the difference between observations and background state $y - \mathbf{H} x_b$. The factor of proportionality is expressed as the Kalman gain matrix $\mathbf{K} \in \mathbb{R}^{n \times m}$. $\mathbf{K}$ weights the uncertainty of the model results (background states) against the uncertainty of the observations.

The analysis state vector $\bar{x}_a$ returned by the Kalman filter is equivalent to x_a calculated by 3D-Var (Section 2.2) under the same assumptions: normal distributed observation and model error distributions with linear operators $\mathbf{M}$ and $\mathbf{H}$. Thus, the Kalman filter update phase also calculates the BLUE.

Assuming a linear model operator, the calculation of the DA propagation phase is analytically possible. The propagation of a linear model can be expressed as the matrix multiplication:

$$x_{t+1} = \mathbf{M}x_t, \tag{2.17}$$

with the matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ being the model operator. Transforming random variables that follow a multivariate normal distribution $\mathcal{N}(\bar{x}_a, \mathbf{P}_a)$ by matrix multiplication is straightforward:

$$\bar{x}_b = \mathbf{M}\bar{x}_a, \tag{2.18}$$

The propagation of the covariance matrix $\mathbf{P}_a$ is done by:

$$\mathbf{P}_b = \mathbf{M}\mathbf{P}_a\mathbf{M}^{\mathrm{T}}. \tag{2.19}$$

Note that Kalman filtering is also successfully used in settings with non-linear model and observation operators or non-Gaussian error PDFs, when good linear and Gaussian approximations exist (Extended Kalman Filter, Jazwinski, 1970). But this is not always the case. Especially in chaotic settings models and observation operators can hardly be linearized.

Current geoscience models that are subject to DA, as the ones presented in Chapter 1, are executed on state vectors with millions of entries. Running the Kalman filter at this scale is hardly possible as the calculation and memory necessary to represent the system covariance $\mathbf{P} \in \mathbb{R}^{n \times n}$ grows quadratically in the number of state dimensions n for the Kalman filter.

The ensemble Kalman filter, the Monte Carlo variant of the Kalman filter, which is presented in the following section, circumvents the necessity of linear operators and the direct calculation and representation of the covariance matrices.

2.3.2 The Ensemble Kalman Filter

We introduce the ensemble Kalman filter (EnKF) as a variant of the Kalman filter (Evensen, 2009; Evensen, 1994; Houtekamer and Mitchell, 1998).

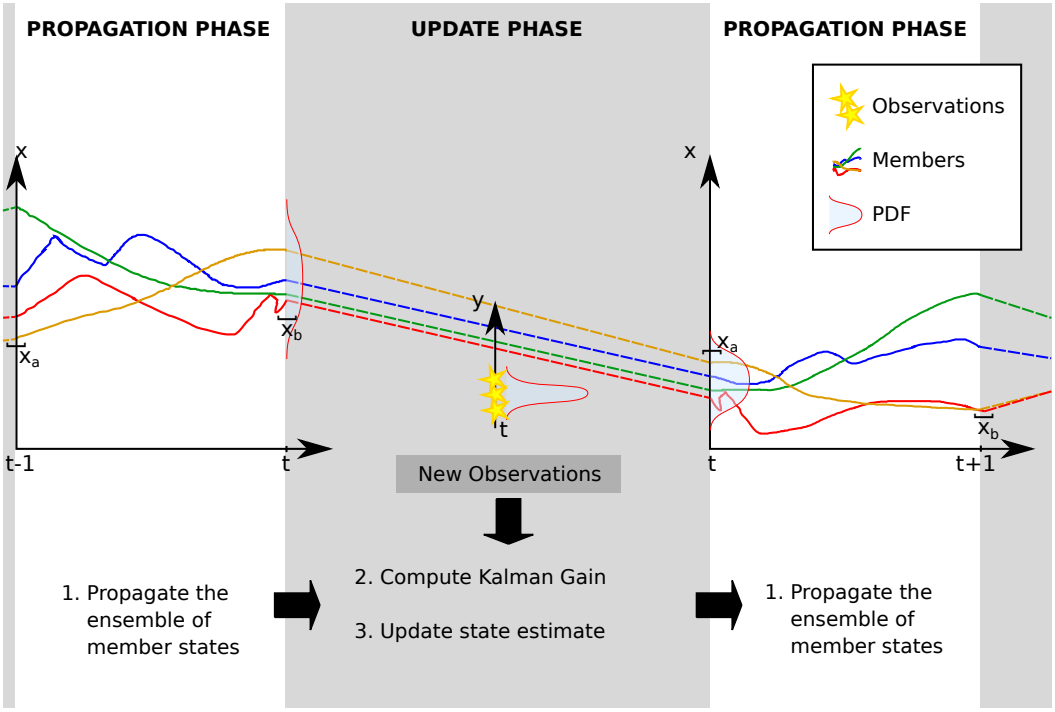


Figure 2.2: The ensemble Kalman filter workflow repeats assimilation cycles composed by a propagation and update phase.

The Kalman filter fails in high dimensional spaces. The storage needed for $\mathbf{P}$ grows quadratically with the number n of degrees of freedom of the system state. Typical models used with DA, like ParFlow and WRF that we will experiment with in Sections 4.3 and 5.2 have a model state vector with millions of dimensions. This leads to state error covariance matrices $\mathbf{P}$ that cannot be efficiently handled anymore in most computing facilities. The EnKF can solve DA problems in high dimensional spaces using a Monte Carlo approach. Instead of expressing the system state covariances explicitly, covariances are sampled stochastically.

An ensemble of $M \in \mathbb{N}$ perturbed model states is propagated. The covariance estimator of the ensemble of propagated states is given by:

$$\mathbf{P}_b = \frac{1}{M-1} \sum_{i \in [M]} (x_b^i - < x_b^i >)(x_b^i - < x_b^i >)^{\mathrm{T}}, \tag{2.20}$$

where $< x_b^i >$ denotes the mean over all members $i \in [M]$:

$$< x_b^i > = \frac{1}{M} \sum_{i \in [M]} x_b^i. \tag{2.21}$$

The terms used to calculate the Kalman gain **K** from Equation (2.14) can be rewritten as:

$$\mathbf{P}_b\mathbf{H}^{\mathrm{T}} = \frac{1}{M-1}\sum_{i\in[M]}(x_b^i - < x_b^i >)(\mathcal{H}(x_b^i) - < \mathcal{H}(x_b^i) >)^{\mathrm{T}}, \tag{2.22}$$

$$\mathbf{H}\mathbf{P}_b\mathbf{H}^{\mathrm{T}} = \frac{1}{M-1}\sum_{i\in[M]}(\mathcal{H}(x_b^i) - < \mathcal{H}(x_b^i) >)(\mathcal{H}(x_b^i) - < \mathcal{H}(x_b^i) >)^{\mathrm{T}}. \tag{2.23}$$

Then $P(x_a)$ is obtained analogously to the Kalman filter following Equations (2.13) to (2.15). In contrast to the Kalman filter, the model error is estimated by the ensemble. Thus, $\mathbf{P}_a$ does not need to be propagated, removing the constraint to linear models. Plugging Equations (2.22) and (2.23) into Equation (2.14), EnKF avoids the costly explicit calculation and representation of the covariance **P**.

Furthermore, EnKF circumvents the transposed formulation of the observation operator $\mathcal{H}$. Only the forward operator is used. These are major advantages compared to the classical Kalman filter and allow EnKF to be applied in situations where operator linearization is impossible.

To summarize, EnKF follows these steps (Figure 2.2):

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

2. The Kalman gain **K** is calculated from the background error $\mathbf{P}_b$ and the observation error **R**, inserting Equations (2.22) and (2.23) in Equation (2.14).

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

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

Various EnKF variants exist. The localized version, LEKF, for example, allows to speed up the covariance matrix estimation since covariances between spatially distant state vector entries are set to zero (Ott et al., 2004). Ensemble transform filters like the local ensemble transform filter (LETKF) reduce necessary computations further (Hunt et al., 2006). The different variants follow the same data flow. An ensemble of states is propagated and needs to be gathered at a central point to prepare for the next assimilation cycle. But varying calculations to transform the ensemble of background

states into analysis states are used. Since this thesis is concerned with data flow optimization, we do not detail further the different variants and how they transform background into analysis states. In Chapter 4 we present our approach to executing EnKF efficiently at the large scale. The integration of all these variants in our proposal is straightforward too.

EnKF and its variants have the limitation that probability density functions are only captured by their first two momentums, mean and variance. Especially for some observation errors, such simple statistics may not capture the error distribution imposed by measurement instruments. Furthermore, state updates produced by EnKF are linear in the innovation vector, which might be a miss-fit for very chaotic systems. A further type of DA methods, called particle filters, circumvents some of these limitations. We present particle filters in the next section.

2.3.3 The Particle Filter

Particle filters are another ensemble-based DA method. They circumvent the Gaussian assumption. Following the Monte Carlo approach, state PDFs are directly represented from histograms formed by members, also called *particles*. The different particles are weighted depending on their alignment with the observations. The weight $w_{i,t}$ of particle i is proportional to the probability that particle i equals the observed system state. Weights are computed comparing particles and observations using the observation error distribution P_ε:

$$w_{i,t} = P(y_t|x_{i,t})P(x_{i,t}) = P_\varepsilon\Big(y_t - \mathcal{H}(x_{i,t})\Big)w_{i,t-1}. \tag{2.24}$$

At the beginning, all particle weights are initialized equal:

$$w_{i,0} = \frac{1}{M}. \tag{2.25}$$

To convert the weighted particles into a PDF, the particle weights need to be normalized so that they sum up to one:

$$\hat{w}_{i,t} = \frac{1}{M}\frac{P(y_t|x_{i,t})}{P(y_t)} \approx \frac{w_{i,t}}{\sum_{j\in[M]} w_{j,t}}, \tag{2.26}$$

where $\hat{w}_{i,t}$ is called *normalized* weight of the i-th particle at time t. Then the analysis state PDF can be constructed according to Equations (2.7) and (2.9):

$$P(x_a) = \frac{P(y|x)P(x)}{P(y)} = \sum_{i\in[M]} \hat{w}_{i,t}\delta(x_t - x_{i,t}), \tag{2.27}$$

where δ is the Dirac measure. Thus, the analysis state PDF can be calculated at each update phase.

Particle filters suffer from ensemble degeneration and weight collapse (van Leeuwen et al., 2019, see also Figure 2.3). All particles spread more and more into the multidimensional space while the variance of the weights increases. Ultimately, only one particle has a normalized weight significantly differing from zero, rendering the analysis state PDF meaningless.

Sequential Importance Resampling (SIR)

The classical approach against ensemble degeneration consists in regularly resampling the particles based on their importance (SIR, Sequential Importance Resampling, Gordon et al., 1993; Liu et al., 2001). A new ensemble of M particles is drawn randomly, i.e., *resampled*, from the particles, each with its probability $\hat{w}_{i,t}$. This is also referred to as *multinomial* resampling. Low weighted particles are discarded, while high weighted ones can become the *parent* of multiple new particles (Figure 2.4).

After resampling, each resulting particle gets its weight reset to $w_{i,t} = \frac{1}{M}$. Particles inheriting the same parent may need to become stochastically perturbed if the model does not contain a stochastic component itself. Otherwise, all particles inheriting the same parent particle would propagate to the same state.

Different flavors of SIR exist. Some perform a resampling step after each propagation phase, while others make this dependent on criteria like the variance of the weights. For our use cases, we leverage SIR with resampling after each propagation phase throughout this work.

Residual Resampling (RR)

For resampling, the standard SIR particle filter draws particles at random (multinomial resampling) with respect to their weights. An alternative to multinomial resampling is residual resampling (RR) as described by Bolic et al., 2003. First, all particles that are *expected* to be redrawn according to their normalized weights are resampled deterministically. A particle is expected to be redrawn if its normalized weight, i.e., the probability to redraw the particle, leads to at least one expected occurrence when redrawing the full ensemble of M particles:

$$\hat{w}_{i,t} M > 1. \tag{2.28}$$

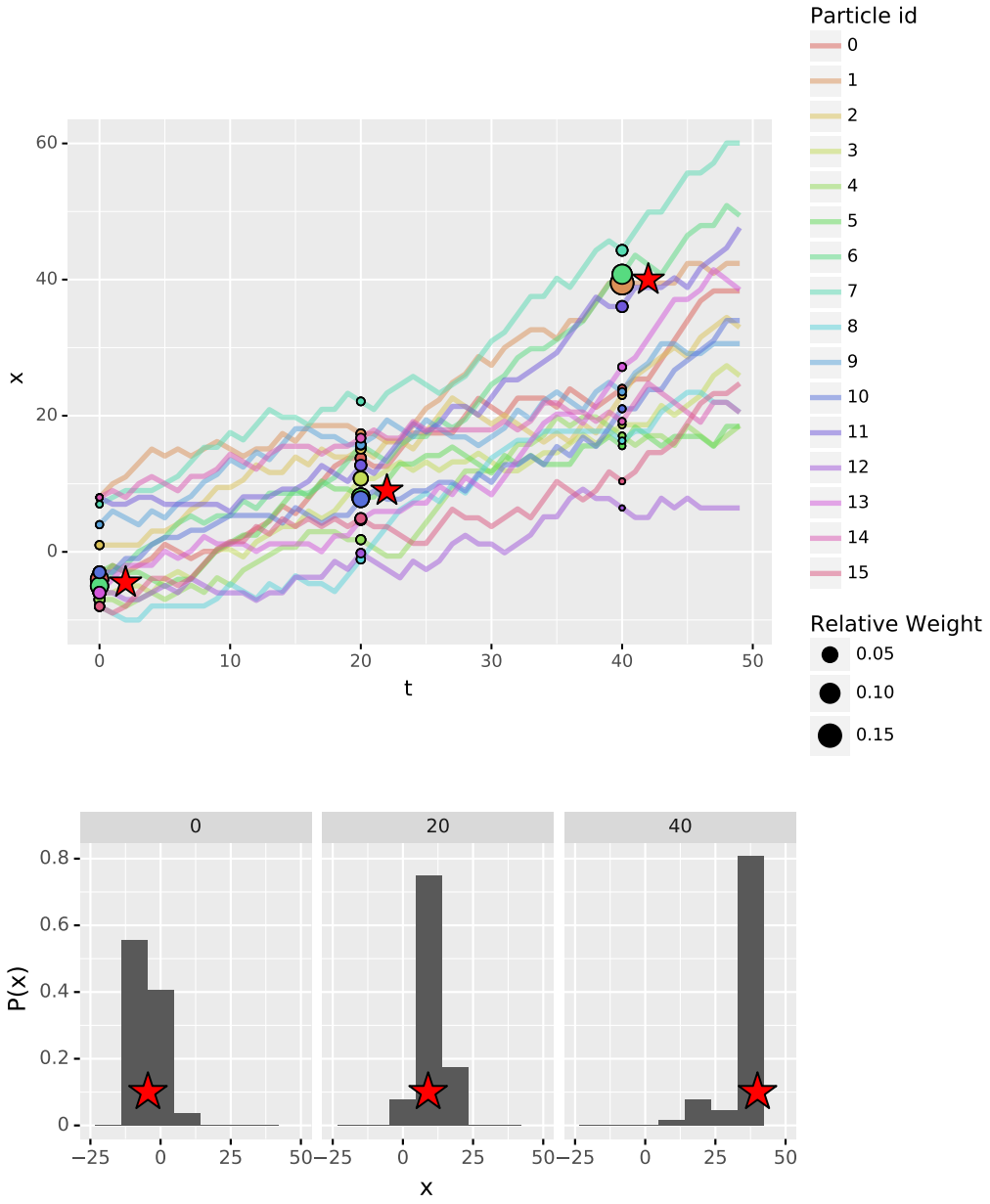


Figure 2.3: Importance sampling particle filter (without resampling). Red stars mark observations. *Top:* Particle view. *Bottom:* Corresponding histogram of the analysis state at different times.

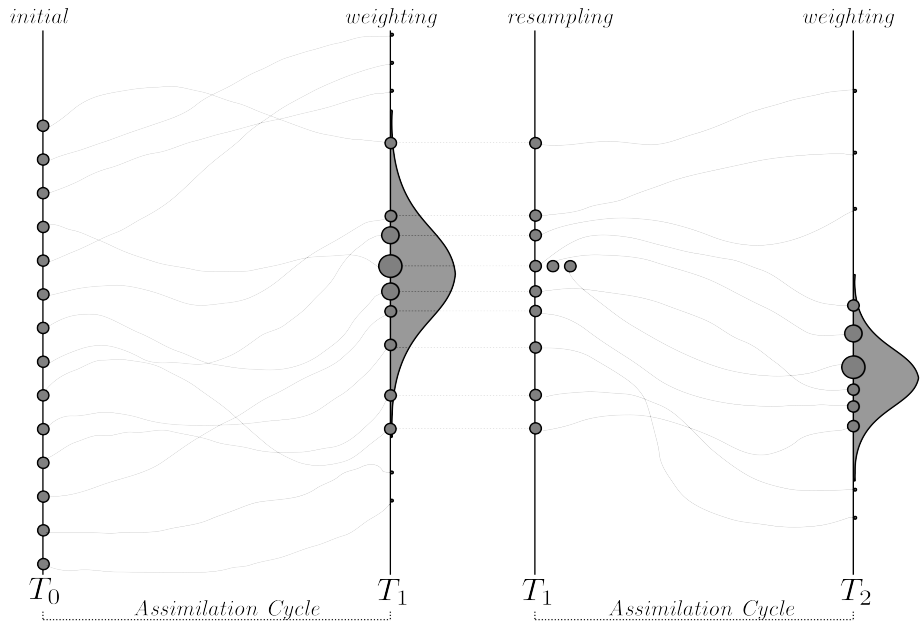


Figure 2.4: *(inspired by van Leeuwen, 2009, Figure 2)* Initially particles are uniformly sampled. They are propagated to T_1 where they are weighted taking into account observation data. Resampling leads to discard some particles with low weights (top and bottom), while others with high weights become parent of several ones (3 here).

Formulated differently, such particles are drawn with a probability larger $\frac{1}{M}$. These particles are redrawn $m_{i,t}$ times, with

$$m_{i,t} = \lfloor \hat{w}_{i,t} M \rfloor. \tag{2.29}$$

Second, to keep the ensemble size constant, $M^r = M - \sum_i m_{i,t}$ remaining particles are chosen similarly to multinomial resampling performed in the standard SIR. The normalized residual weights $\hat{w}^r_{i,t}$ define the probability to redraw the particles. They are calculated from the residual weights $w^r_{i,t}$, analogously to Equation (2.26):

$$w^r_{i,t} := \hat{w}_{i,t} - \frac{m_{i,t}}{M}, \tag{2.30}$$

$$\hat{w}^r_{i,t} = \frac{w^r_{i,t}}{\sum_{j \in [M^r]} w_{j,t}}. \tag{2.31}$$

The presented resampling methods allow to assimilate observations in the ensemble of particles without manipulating the particle state vectors. Only the composition of the ensemble is updated. Many other resampling techniques apart from residual resampling exist to perform particle weight (importance) based resampling. Surveys like the one of

Li et al., 2015 introduce them extensively. In Chapter 5 we will consider how to execute them efficiently at large scale.

In contrast, particle filter variants exist that do change the particle state vectors to perform the state update. Transportation particle filters, for instance, transform background particle states into analysis states by modifying the state vector entries, much like EnKF variants do (Reich, 2013). They can run efficiently at large scale following the approaches presented for the EnKF context in Chapter 4.

For an extensive review of different particle filter variants covering also combined forms of particle filters and EnKF or variational methods, refer to van Leeuwen et al., 2019.

2.4 Discussion

The various DA methods are suited to different use cases. Textbooks and survey articles like the ones by Asch et al., 2016; Evensen, 2009; van Leeuwen et al., 2019; Vetra-Carvalho et al., 2018; Carrassi et al., 2017 give an overview of these methods.

Variational methods convince with their compute efficiency. Only a cost function needs to be optimized. Statistical ensemble-based methods, in contrast, provide flow-dependent model errors and the possibility to rely on large ensembles that are expected to run efficiently on upcoming exascale machines (Schulthess et al., 2019). Furthermore, they do not require the definition of an adjoint model or observation operators. EnKF and particle filters are the most common statistical ensemble-based DA methods. Many variants of each of them exist. Together, they cover a wide range of applications. Thus, we focus on these two filtering methods in the following.

EnKF variants enable DA with a comparably small number of members but approximate model and observation errors as Gaussians which is not sufficient in several cases. Gaussians, for instance, cannot accurately capture the uncertainty of positive definite measures like precipitation (Lien et al., 2013). Particle filters do not approximate state and observation errors by any parametric, e.g., Gaussian PDF, permitting them to handle non-Gaussian PDFs. But particle filters necessitate more members, i.e., particles, to solve the same DA problem as EnKF at comparable accuracy (Pasetto et al., 2012). The number of necessary particles is growing exponentially with the effective problem dimension, while the necessary member count for EnKF filters grows linearly in the effective problem dimension (Majda and Tong, 2017).

Shifting system state vectors as performed by EnKF during the update phase is not always possible, especially in some non-linear settings, where it affects important invariants like energy or radiation budgets. The update phase of SIR particle filters, in contrast, conserves the state integrity of every particle. No state corrections at all, possibly unphysical, are introduced. Localized particle filters, e.g., as proposed by Poterjoy, 2016 circumvent the exponential growth in the particle count to adapt to systems of higher dimensions. This enables their use for non-linear high-dimensional problems, outperforming EnKF in such cases as shown by Poterjoy and J. L. Anderson, 2016.

The selection of a DA algorithm is very use-case dependent. When observation/model adjoint operators are available, and the system is not strongly non-linear, variational methods can perform very well. If adjoints are unknown or difficult to obtain, EnKF is easier to add to existing model codes, also due to existing tooling like PDAF (The Parallel Data Assimilation Framework by Lars Nerger and Hiller, 2013 that we will detail

in Section 3.1.4). Only for strongly non-linear and/or non-Gaussian settings particle filters are the adapted choice as they are more compute intense.

3 State of the Art

This chapter introduces state-of-the-art approaches for large scale DA in Section 3.1. Variational and statistical methods are currently used at large scale, each with its advantages and disadvantages. Possible improvements may evolve from recent work on flexible particle filter variants presented in Section 3.2. DA methods are affected by techniques from the quickly developing field of Machine learning (ML) too. In Section 3.3, mixed forms of ML and DA and the exchange of specific tools between the fields are discussed. Ensemble-based approaches are not only finding application in the field of DA. Many ensemble frameworks for the large scale exist already. Thus we also comment on existing tools and if they may be adapted for large scale DA in Section 3.4. But also in situ computing techniques can augment the performance of ensemble-based DA at large scale. The chapter ends with Section 3.5, discussing existing in situ computing frameworks.

3.1 Large Scale Data Assimilation

Methods like the ensemble Kalman filter (Evensen, 1994; Houtekamer and Mitchell, 1998; Burgers et al., 1998; D. Zhang et al., 2016), 3D/4D-Var (Lorenc et al., 2000; Courtier, Thepaut, et al., 1994) or particle filter (van Leeuwen, 2003), are most relevant to run at large scale. While giving accurate results, ways to parallelize them on HPC systems exist – be it by parallel propagation of different ensemble members, using parallel algorithms for matrix computations, or relying on parallelized model adjoint computation to minimize a cost function via gradient descent. Thousands of compute cores are leveraged for model propagation and to assimilate state vectors with millions of degrees of freedom. Thus, in this section variational methods, EnKF, and particle filter on existing large scale use cases are discussed.

3.1.1 Variational Data Assimilation

As explained in Section 2.2, variational DA (e.g., 3D-Var and 4D-Var) relies on minimizing a cost function evaluating the difference between the model state and the observations. Minimizing the cost function is typically done via gradient descent using the adjoints of the observation operator. When using 4D-Var, an adjoint of the model operator is necessary too. Although using gradient descent for the minimization is compute efficient, necessary adjoints are not always available or may require significant efforts not always accessible. Nowadays, large scale DA applications as used by Numerical Weather Prediction (NWP) operators typically rely on variational DA. For instance, the China Meteorological Administration uses 4D-Var to assimilate about 2.1 million daily observations into a global weather model with more than 7.7 million grid cells. As in L. Zhang et al., 2019, the DA itself is parallelized on up to 1,024 processes. But it is unclear if the possibility to execute statistical, ensemble-based methods with an increasing number of ensemble members will outperform variational methods in the future since the parallelization of variational methods on exascale is more challenging.

3.1.2 Statistical Data Assimilation

Statistical DA, as introduced in Section 2.3, takes a different approach relying on Bayesian statistics (Asch et al., 2016; Evensen, 2009). An important subcategory are ensemble-based methods. These run an ensemble of models to compute an estimator of some statistical variables used to express the analysis state PDF. The ensemble-based approach consumes more compute power as the number of members needs to be large

enough for the estimators to be relevant, but stands for its simplicity as it only requires the model and observation operator without adjoints as for the variational approach. But scaling the ensemble size can be challenging, especially when the model is already time-consuming and requires its own internal parallelization. Nevertheless, these methods are expected to become easier to access in the future regarding advances in Big Data and high performance computing (*TOP500* 2022; Reed and Dongarra, 2015).

Two main approaches exist, identified as *file-based* (offline) and *online*. For the file-based approach, the ensemble member's simulations, i.e., the member state propagations, are executed independently after loading each member's input from the file system. The resulting background states are saved back to files. Once all members executed for that assimilation cycle, an analysis code runs to load the observations and the states from the different members to produce the analysis states, also saved to files. The members can be started again from these states for the next assimilation cycle. This file-based approach is usually simple to set up, elastic, and fault-tolerant. All the files act as checkpoints isolating the impact of a failing component and making a restart easy. The number of concurrent simulations can vary from cycle to cycle depending on the supercomputer availability, providing elasticity at the granularity of a member. This approach is adopted by the EnTK (Ensemble Toolkit) framework (Balasubramanian, Turilli, et al., 2018), used to manage up to 4,096 members for DA on a molecular dynamics application (Balasubramanian, Jensen, et al., 2020) and by Toye et al., 2018 assimilating oceanic conditions in the Red sea with O(1,000) members. The OpenDA framework also supports this file-based model, relying on NetCDF files for data exchange for the NEMO ocean model in van Velzen et al., 2016. Miyoshi et al., 2014 run the offline approach at scale with 10,240 members for a LEKF (localized ensemble Kalman filter) on the K-computer in 2014. Also, Berndt, 2018 uses an offline approach to assimilate up to 4,096 members simulated by the WRF model on 262,144 processors with a particle filter. They moreover did a production run with 1,024 members for wind power prediction over Europe. A file-based approach may be the only option available when the full supercomputer is required to propagate just a fraction of members as in the work of Yashiro et al., 2020. They propagate 1,024 members to perform LETKF filtering with the NICAM model on 6 M Fugaku cores. However, relying on the machine I/O capabilities using files is a growing performance bottleneck. In 10 years the compute power leaped by a $134\times$ factor, from 1.5 PFlop/s peak on Roadrunner (*TOP500* 2022 $\#1$ in 2008) to 201 PFlop/s peak on Summit ($\#1$ in 2018), the I/O throughput for the same machines only increased by a $12\times$ factor, from 204 GB/s to 2,500 GB/s. This trend is expected to continue at exascale. For instance, the announced Frontier machine (2022)

expected to reach at least 1.5 ExaFlop/s should offer 7.5 times the compute power of Summit but only 2 to 4 times its I/O throughput[1].

Existing online approaches build one single application that encompasses the different simulation instances for the propagation and the update phase code. In that case, states stay in memory, and no intermediate file is needed. This further avoids to rerun model and update phase initialization code reserving and populating internal data structures for every single model propagation or each update phase respectively. C. Sun et al., 2021 show the benefit of online approaches compared to offline mode on a small scale already. They perform EnKF on WRF numerical weather simulations with up to 10 ensemble members on 3,200 compute cores. The online mode turns out up to 4 times faster than offline mode as it fixes the I/O issue and avoids repeated initialization of simulation and update phase codes, but leads to a large monolithic application. If implemented with classical message passing (MPI), load balancing, elasticity, and fault tolerance become challenging. They are not directly supported by MPI and need to be implemented explicitly. For instance, if one process in the monolithic application fails, the full application stops. Changing the number of CPUs used during execution to enable some elasticity is not supported by most MPI applications so far either. The full amount of resources needed must be allocated upfront and for the full duration of the execution. Requiring a large amount of resources, ensemble runs are sensitive to hardware and numerical faults. As the ensemble size increases, the probability that the model may execute in numerical domains it has not been well tested in will get more important too. The frameworks DART (Data Assimilation Research Testbed) by J. Anderson et al., 2009, PDAF (Parallel Data Assimilation Framework, see Section 3.1.4) by Lars Nerger and Hiller, 2013 and DAFCC1 (Data Assimilation framework based on C-Coupler2.0, version 1) by C. Sun et al., 2021, rely on this approach. PDAF, for instance, has been used for the regional Earth system model TerrSysMP using EnKF with up to 256 members (Kurtz et al., 2016). DAFCC1, Dart, and PDAF have an offline and online mode. DAFCC1, uses the existing C-Coupler2 to manage the data transfer between member propagation and Assimilation update. This is a straightforward way to parallelize the computations of the assimilation update and the member propagations differently.

3.1.3 Hybrid Methods Mixing Statistical and Variational Methods

As mentioned in Chapter 2, hybrid methods combining variational and statistical DA approaches exist. In some situations, they allow less compute intense calculations but

[1] see `https://www.olcf.ornl.gov/frontier`, retrieved the 21.03.2022

provide flow-dependent error terms (Bannister, 2017). Hybrid methods are used by some NWP actors. The ECMWF (European Centre for Medium-Range Weather Forecasts), the British Met office (Meteorological Office) and the Canadian Meteorological Centre rely on such approaches, using hundreds of ensemble members, to improve the estimate of the flow-dependent model error, leading to more accurate predictions (Bonavita et al., 2017; Clayton et al., 2013; Houtekamer, Buehner, et al., 2019). Hybrid methods are currently only rarely supported in existing frameworks. Only the latest PDAF release 2.0 from December 2021 supports some variants[2].

3.1.4 PDAF – An Existing Large Scale DA Framework

To detail the functioning of existing solutions for large scale DA, and, since we make use of PDAF in our proposed framework, this section presents PDAF more closely. At its core, PDAF provides a library with functions to perform the update phase of different DA methods. Among these are schemes like EnKF, Ensemble transform Kalman filters, smoothed and localized versions of each, particle filters, and since the latest version (V2.0) 3D-Var methods. Users of PDAF link it against their own code that will call the desired update phase implemented by PDAF. Users must provide callback functions to load background states and observations, execute the observation operator $\mathcal{H}$ or write back the analysis states as returned by PDAF. Two modes are possible. Background states can be loaded from files and analysis states are written back to the file system too (offline mode). Alternate calls to the model executable and the `PDAF_offline` executable, linked against PDAF, are necessary to run the DA workflow. First, the model executable is called, loading and propagating each member state and finally writing it back to disk as a background state. Once all members have been propagated, the `PDAF_offline` executable is called (the one linked against PDAF). It loads the background states, runs PDAF's implementation of the desired update scheme and the returned analysis states are stored back to disk. The next cycle begins by calling the model to propagate each member again. Note that the model code does not need to be instrumented for this approach. The model and the `PDAF_offline` executable can be executed as parallel code using MPI or OpenMP.

Alternatively, PDAF can be linked directly into the possibly parallel model executable and the update phase functions are called from there directly. This mode avoids transferring all states via the file system and is called online mode. PDAF supports time and space sharing in online mode. Multiple model instances can be executed in parallel (space sharing), in one big monolithic MPI run. Each of these model instances

[2] `http://pdaf.awi.de/trac/wiki/FeaturesofPdaf`, retrieved the 21.03.2022

can propagate multiple members in a row (time sharing). For example, assume that one model propagation is performed on 10 cores. To run DA with 100 members, one could reserve 1,000 compute cores, so all 100 members can run in parallel on different cores (space sharing). But PDAF also supports running such an ensemble on fewer, let's say, 200 compute cores. Only 20 models can run in parallel, so each of them has to perform 5 propagations per assimilation cycle (a mix of time and space sharing). Note that each model instance will advance the same 5 members at each assimilation cycle. PDAF does not perform data redistribution in online mode. The update phase can only be executed on as many cores as were used for each model instance since PDAF relies on the domain decomposition provided by the model. In contrast, in offline mode this is possible. The `PDAF_offline` executable and model can be parallelized on a different number of compute cores since data transfer is routed via a general encoding through the file system. PDAF is open source and comes with excellent documentation and examples for the integration with existing model codes for both modes.

3.2 Anytime Particle Filters, Island Particle Filters and Adaptive Ensemble Sizes

This section presents active research, trying to remove the resampling synchronization point persistent in particle filters. It also elaborates propositions to control the ensemble size for a better trade-off between spent compute power, over- and undersampling. The section ends up with a quick discussion how such approaches might be extended to EnKF variants.

Ensemble-based DA and especially particle filtering may require a large number of ensemble members (particles) to avoid undersampling. The necessary ensemble size grows exponentially with the effective system size (Snyder and Bengtsson, 2015; Fearnhead and Künsch, 2018; van Leeuwen, 2009). This effect is known as *curse of dimensionality*. For problems with millions of dimensions, as typical for geoscience applications (van Leeuwen et al., 2019), large amounts of compute resources are necessary. Different particle propagations can be executed independently from each other on different resources in parallel (space sharing). Nevertheless, the assimilation update (resampling) remains a necessary synchronization point and a source of scaling inefficiency. Different approaches exist to loosen the synchronization implied by resampling in particle filters. They are an active topic of research. *Anytime* or *asynchronous* particle filter techniques, for instance, allow to decide which particle is reused or thrown away at *any time*, *asynchronously*, one by one. Paige et al., 2014 uses the criterion if an arriving particle weight $w_{i,t}$ is better or worse than the iteratively calculated average over all particle weights received so far for decision making.

An other approach to loosen synchronization is based on the island particle model as proposed by Vergé et al., 2015. Particles are attached to groups, i.e., *islands*. Resampling happens within each island only, avoiding global synchronization. Estimates of statistical measures like mean and standard deviation computed within islands are biased towards global ones over all particles in all islands. To lower the bias of each island, some interactions between different islands are necessary. For this, the particles of pairs of randomly selected islands may be mixed before resampling. This adds some interaction but introduces only very limited synchronization. A further option is to attach weights to each island and resample whole islands, i.e., not sample particle by particle but instead duplicate or remove a whole group of particles (island). It is to explore, at which frequency different types of interaction between islands are necessary to keep bias low enough.

So called *alive* or *adaptive* particle filtering algorithms as presented by Jasra et al., 2013 and Elvira et al., 2016 are a further way to improve particle filter performance. They *adapt* the number of particles to keep the ensemble accurate enough, i.e., *alive*. This technique actively tackles under- and oversampling. As soon as an accuracy criterion is fulfilled, no more particle needs to be propagated. If results are not accurate enough, more particles are sampled. The computation time of each timestep can fluctuate but accuracy stays roughly constant. For traditional particle filters as described in Section 2.3.3, the opposite is the case. Since the number of particles is constant, one can give good estimates for the computational effort for each timestep but no guarantee on the ensemble accuracy. As described in the mentioned publications by Jasra et al. and Elvira et al., different metrics to decide if more particles are needed, are possible and one can start particles one by one or increase and decrease the ensemble size in steps of multiple particles at once.

EnKF can be interpreted as a special case of transport particle filters, where the particle ensemble is used to approximate the first and second order momentums of the background state distribution and the calculation of the ensemble transformation matrix is inherited from the Kalman filter formalism (van Leeuwen et al., 2019). Thus, the methods in this section have relevance for EnKF variants too. For instance, Lermusiaux and Robinson, 1999 experiment with an EnKF variant that varies the ensemble size.

3.3 Data Assimilation and Machine Learning

While Data Assimilation greatly improves the forecast skill of numerical models, there are remaining inaccuracies in posterior states. A manifold of sources, like incomplete or interpolated observations as well as numerical approximations in model and observation operators are responsible for those. Machine Learning (ML) was successfully used to reproduce time series of dynamic systems. As such they can also be trained to correct simulation errors (Um et al., 2021). Thus, there is increasing interest in leveraging ML tools to augment DA. In this section we discuss combined approaches between DA and ML.

Data Assimilation can for example be performed on the output of previously trained surrogate models as done by Tang et al., 2020. Surrogate models are machine learning models trained to imitate an existing simulation code. Often they reduce the input space but can return simulation results much quicker. They can also be interpreted as a way to compress model output with the capability to interpolate to never seen model results. Combining with DA, a surrogate, directly using observation data as input, can be created. The augmented model is typically less costly to execute in production than the original model and DA workflow.

DA can also be used to generate higher quality training data for ML models. Brajard et al., 2019 and Bocquet et al., 2020 propose to train surrogate models by alternating DA and machine learning steps. First an initial state guess is assimilated using a classical DA scheme. The assimilated state then is used to train a surrogate model. After some training, one can use the output of the surrogate to perform the DA followed by surrogate training again or the surrogate model forwards the state to the next assimilation cycle and another DA step is performed. The loop is closed. Farchi et al., 2021 use a similar approach to train a hybrid surrogate built from an existing meteorological model code but its error is corrected with the help of machine learning, leading to promising results.

Machine learning can also borrow techniques from DA. One example is proposed by Kovachki and Stuart, 2019. They train machine learning models, replacing the classical stochastic gradient descent with ensemble Kalman inversion (EKI), a gradient free optimization method. It can further be shown that there is an equivalence between some machine learning and DA techniques (Abarbanel et al., 2017; Bocquet et al., 2019). Both techniques try to minimize the discrepancy between model and observations. In some DA methods like 3/4D-Var, this is even done using a loss function similar to the one used in ML.

So far, such approaches have been validated at small scale with simple examples. But they have a big potential and are expected to lead to hybrid solutions using ML and DA

that will help to address more complex and larger problems. In this thesis we did not specifically address such combined ML/DA approaches as they are not yet mature for most realistic use cases. Nevertheless, the modularity of the framework makes it well amenable to more heterogeneous algorithms requiring the integration of ML/DA codes. We will discuss this at the end in Section 7.2 after our approach was introduced in more detail.

3.4 Ensemble Frameworks

Looking beyond DA frameworks, various Python based frameworks are supporting the automatic distribution of tasks, enabling to manage ensemble runs. Dask (Rocklin, 2015) is, for instance, used for hyperparameter search with Scikit-Learn, and Ray (Moritz et al., 2018) for reinforcement learning (Liang et al., 2018). But they do not support tasks that are built from legacy MPI parallel codes. Other frameworks such as Radical-Pilot (the base framework of EnTK) by Paraskevakos et al., 2018 or Parsl (Babuji et al., 2019) enable such features but are using files for data exchange. Other domain specific frameworks like Dakota (Elwasif et al., 2012), Melissa (Terraz et al., 2017) or Copernicus (Pronk et al., 2011) enable more direct support of parallel simulation codes but for patterns that require less synchronizations than the ones required for DA.

To our knowledge, map-reduce tools like Spark (Zaharia, 2014) and Flink (Alexandrov et al., 2014) have not been used for DA. Flink has been used for analyzing online the data of a parallel molecular dynamics simulations relying on its distributed stream processing capabilities (Zanúz et al., 2018). But performance was significantly below the one achieved with HPC specific in situ data processing tools like Damaris (Dorier et al., 2012) or FlowVR (Dreher and Raffin, 2014). Extending such approaches to DA would require to support an ensemble of simulations and efficient linear algebra operations on large matrices as needed for computing the Kalman gain in EnKF, for instance. Map-reduce is not well suited for such operations.

3.5 In Situ Computing Frameworks

File-based DA approaches are I/O intensive, limiting their performance. Online approaches enable to bypass I/O but rely on a monolithic architecture. In this section, we look at in situ processing, an approach to couple parallel simulations with online data processing to avoid I/O, and discuss related work.

HPC machines consist of thousands to hundreds of thousands of compute cores connected via high-speed networks, reaching a compute power of multiple PFlop/s. In contrast, HPC machine vendors are unable to scale the I/O performance of their platforms as much as the compute performance, leading to a severe performance bottleneck (Kunkel et al., 2014; Hu et al., 2016). One approach to alleviate this issue consists in bypassing the storage between different steps of a compute workflow. This approach is often referred to through the umbrella term *In Situ Processing*, as coined in Ma, 2009. The goal is to process the data as close as possible to the locus and time of data generation. Thus data can be processed *online* directly on the nodes that actually produced the data (in situ), minimizing data movements, or on nodes dedicated to data processing. This latter case is often referred to as *in transit* processing. In situ processing tasks are usually allocated on dedicated *helper cores* on the same compute node which generated the data. Dedicated nodes for in transit data processing are called *staging nodes*.

A number of frameworks have been developed to support in situ and in transit processing (Dorier et al., 2012; Docan et al., 2012; Zheng et al., 2013; Dreher and Raffin, 2014; Dreher and Peterka, 2017; Capul et al., 2018). Their main goal is to enable users to harness data processing capabilities to their data producing codes with minimal code intrusion, maximal performance, and flexibility. In situ processing has been successfully applied to various data processing operations like data indexing, compression, and computation of various high-level descriptors up to images and video production, which are eventually saved to disk but that can also be visualized *online* (Rivi et al., 2012). This effectively reduces the need of file system storage. For instance, the full state of the simulation is saved to disk at a low frequency, with intermediate high-level descriptors being computed in situ in between, while the simulation is running.

An example of one of the most basic in situ workflow is shadowed file writing (also called two-phase I/O). Instead of having each core of a parallel large scale application writing its share of the data to disk (Figure 3.1, *left*), execution is quicker when data is gathered on helper cores first, which then write larger chunks to disk in the background (Figure 3.1, *right*). This is for two reasons. First, storage devices typically used in HPC provide more bandwidth when fewer but larger chunks of data are accessed. Second, overlaps the in situ workflow file system access performed by the helper cores with useful computations

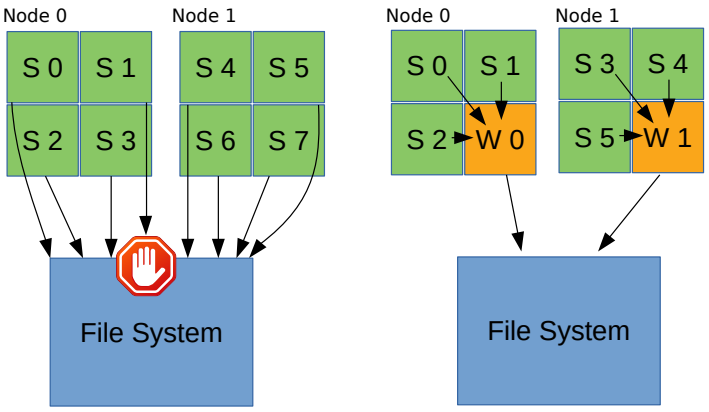


Figure 3.1: Shadowed file writing. *left:* The traditional way, all simulation cores (**S 0** - **S 7**) access the file system in parallel and need to wait for I/O completion. *right:* Shadowed file write where data is first gathered on one helper core per node (**W 0** and **W 1**). Then the simulation resumes while in parallel the helper cores write the data to the file system.

performed by the other cores. Even if some of the resources from the model are cut off from the application (helper cores), the performance impact of shadowed file writing typically stays very good due to the imperfect scaling of most applications on the large scale and the mentioned I/O acceleration. Dreher and Raffin, 2014, for instance, can accelerate the runtime of a molecular dynamics simulation with Gromacs using shadowed file writing by a factor of three.

4 Ensemble Kalman Filtering at Large Scale

This chapter introduces the Melissa-DA framework that was developed in the context of this thesis. Melissa-DA is an elastic, online, fault-tolerant, and modular framework for large scale ensemble-based DA. Details on its architecture and important design choices are given in Section 4.1. Section 4.2 presents the hydrological model code ParFlow used in the experiments. In Section 4.3 we present and analyze experimental results. This chapter is a refined and adapted edition of our research report presenting Melissa-DA (Friedemann and Raffin, 2020). A journal article on the topic was published recently at the International Journal of High Performance Computing Applications (Friedemann and Raffin, 2022).

4.1 Melissa-DA Architecture

4.1.1 Overview

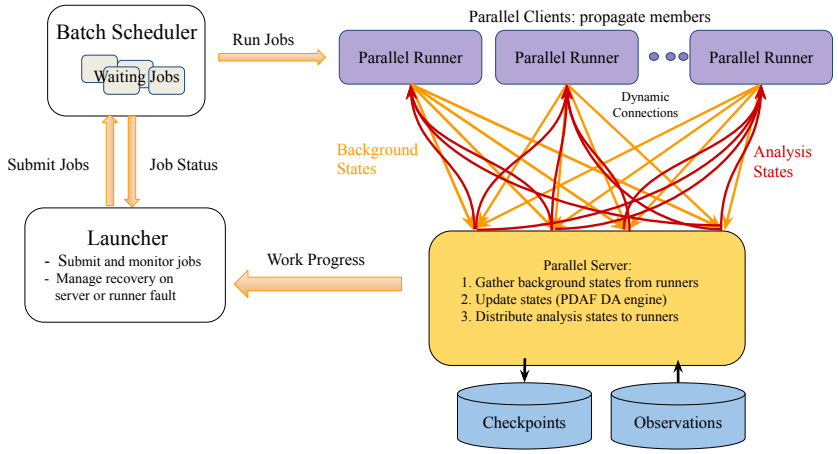


Figure 4.1: Melissa-DA three-tier architecture. The launcher supervises the execution in a tight link with the batch scheduler. The server distributes the members to propagate to the connected runners dynamically for balancing their workload. A fault-tolerance mechanism automatically restarts failing runners or a failing server.

Compared to the online and file-based approaches presented in Section 3.1.2, Melissa-DA takes a third way, keeping the flexibility of the file-based approach while avoiding intermediate files to bypass the I/O bottleneck (Figure 4.1). We present below a global overview motivating the Melissa-DA design choices that are then detailed in the next sections:

- **Client/Server model.** Melissa-DA relies on the standard client/server model extended to the parallel case where both the client and server are parallel codes with potentially different levels of parallelism. No intermediate files are required as all data exchanges occur through direct memory to memory communications between the server and the clients. Additionally, the client/server pattern makes the application more modular. In Melissa-DA, a client runs the simulation code for the propagation phase and the server the code for the update phase. Because the connection between a client and the server is dynamic, a client can be stopped (voluntarily or not) and started anytime. A client failure does not lead the server to fail, thus providing a sound base to support an efficient fault tolerance protocol. The number of running clients can evolve over time depending on the resources available on the supercomputer, making the application elastic.

- **Clients are runners.** A Melissa-DA client runs one simulation instance. Once done with the propagation of a given member, it sends the full member state to the server, and a new member propagation can start. The new member state to propagate is loaded from the server. Thus a client can propagate several members per propagation phase. We call such a client *runner*. Members can be propagated by any runner. We call this *member virtualization*. There are several benefits of enabling member virtualization:
  - No need to pay the full simulation starting cost for each member. Switching from one member to another in a runner is done by switching member states in memory, not requiring a full restart of the simulation code.
  - The time to propagate one member can vary, leading to load balancing issues, and inefficiency, as the update phase cannot start as long as not all members have been propagated. Dynamically distributing the members to the runners according to their workload enables to balance the workload between runners, improving the execution efficiency.
- **Server.** The server collects the states propagated by runners to compute the covariance and the Kalman gain matrices and updates each state to provide the new analysis states in transit. The server is also in charge of implementing the load balancing strategy, distributing the propagation work to runners following a list scheduling algorithm.
- **Launcher.** A Launcher executable overviews the full workflow progress. The launcher orchestrates the execution, interacting with the supercomputer's batch scheduler to request resources to start new jobs holding runners or the server, kill such jobs, monitor their status, and trigger job restarts in case of failure. It makes a single point of entry for the user to parameterize, control, and monitor the application.
- **Code modularity.** The runner, server, and launcher are separate codes that interact with each other over dynamic network connections. This makes the application very modular. For instance, changing the code of the server for switching to a different implementation of the update phase can be done without having to recompile the runner code. Turning an existing solver simulation code into a runner requires instrumenting it with the Melissa-DA API, but introduces no dependency to the server code into the solver code.

Putting all pieces together (Figure 4.1), the launcher starts and monitors runners and the server. During the propagation phase, the server distributes member states to the runners one by one. Runners propagate these member states and send them back to the

server as background states. The server then performs the assimilation of observations (update phase). This generates a new ensemble of member states (analysis states) used for propagation for the next assimilation cycle. We detail this workflow and its components in the following.

4.1.2 Server

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

The server needs to be linked against a user-defined function to initialize all the member states (Figure 4.2 `init_ensemble`). Other functions, e.g., to load the current assimilation cycle's observation data (`init_observations`) and the observation operator $\mathcal{H}$ (`observation_operator_H`) must be provided to the server for each DA study. In the current version user functions are called sequentially by the server, but we expect to support concurrent calls to `init_ensemble` and `init_observations` to further improve the server performance.

The current server embeds PDAF as a parallel assimilation engine. The server parallelization can be chosen independently from the runner parallelization. An N $\times$ M data redistribution takes place between each runner and the server to account for different levels of parallelism on the server and runner side. This redistribution scheme is implemented on top of ZeroMQ, an asynchronous networking library extending sockets (Hintjens, 2013). ZeroMQ supports a server/client connection scheme allowing dynamic addition or removal of runners.

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

decision is next shared amongst all the server ranks using nonblocking MPI broadcasts. This way, communication between the different server and runner ranks overlaps while other runner (-ranks) perform unhindered model integration.

4.1.3 Runners

Melissa-DA runners are based on the simulation code, instrumented using the minimalist Melissa-DA API. This API consists only of two functions: `melissa_da_init` and `melissa_da_expose` (Figure 4.2). `melissa_da_init` must be called once at the beginning to define the size of the member state per simulation rank. This information is then exchanged with the server, retrieving the server parallelization level. Next `melissa_da_init` opens all necessary connections to the different server ranks.

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

4.1.4 Launcher

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

The launcher typically runs on the supercomputer front node but can also be started on any other compute node for unattended DA studies or when the front node may not run long-running jobs. Since its computations are lightweight, the launcher resources can be oversubscribed by another task, e.g., for the Melissa-DA server. The launcher is the only part of the Melissa-DA application that interacts with the machine batch scheduler. The launcher requests resources for starting the server job, and as soon as the server is up, it submits jobs for runners.

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

The launcher prioritizes job submission within the same job allocation (if the launcher itself was started within such an allocation and free resources are left in it). Otherwise, the launcher can also submit jobs as self-contained allocations, e.g., by calling `srun` outside of any allocation on Slurm (Yoo et al., 2003) based supercomputers. In the latter case, the server job and some runner jobs maybe do not execute at the same time leading to inefficiency as the server is not well charged by enough runners or even waits for any runner to connect. For that reason, it is recommended to launch at least jobs for the server and some runners within the same allocation. This guarantees that they run at the same time, ensuring the Melissa-DA application operates efficiently, even if no further runner jobs can be executed. It is also possible to instruct the launcher to start jobs within different partitions.

4.1.5 Fault Tolerance

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

The server is checkpointed once during each assimilation cycle using FTI (fault tolerance interface, Bautista-Gomez et al., 2011). This enables the recovery from server crashes without user interaction. During the propagation phase, the server process asynchronously saves received member state parts on arrival to file (using threaded background checkpointing). Checkpoints are finalized before each update phase begins. So checkpointing does not impair the server reactivity for its other tasks and does not consume server resources during the update phase when the server is performing work on the critical path of the assimilation cycle.

The server sends heartbeats to the launcher. If missing, the launcher kills the runner and server jobs and restarts automatically from the last server checkpoint. Thus in the

case of a server crash, the application restarts from the last complete set of propagation states. In the worst case, only the last update phase and some member propagations of the current assimilation cycle that were not completely checkpointed yet need to be repeated.

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

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

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

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

4.1.6 Dynamic Load Balancing

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

Dynamic load balancing is a very desirable feature when the times to propagate different members varies. This is typically the case with solvers relying on iterative methods, but also when runners are started on heterogeneous resources, for instance, nodes with GPUs versus nodes without, or if the network topology impacts unevenly the data transfer time between the server and runners. The server has to wait for the last member to return its background state before being able to proceed with the update phase computing the analysis states. The worst case in terms of load balancing occurs when state propagation is fully parallel, i.e., when each runner is in charge of a single member. In that case, the runner idle time is the sum of the differences between each propagation time and the slowest one. As we target large numbers of members, each member potentially being a large scale parallel simulation, this can account for a significant resource underutilization. To reduce this source of inefficiency, Melissa-DA enables 1) to control the propagation concurrency level independently from the number of members, and 2) to dynamically distribute members to runners.

The Melissa-DA load balancing strategy relies on the Graham list scheduling algorithm (Graham, 1966). The server distributes the members to runners on a first come first serve basis. Each time a runner becomes idle, the server provides it with the state of one member to propagate. This algorithm is simple to implement, has a very low operational cost, and does not require any information on the member propagation time. The performance of the list scheduling algorithm is guaranteed to be at worst twice the one of the optimal scheduling that requires knowing the member execution time in advance (which is not the case here). More precisely the walltime T_{ls} (called makespan in the scheduling jargon) is bounded by the optimal walltime T_{opt}:

$$T_{\text{ls}} \leq T_{opt} \times (2 - \frac{1}{m}), \tag{4.1}$$

where m is the number of machines used, in our case the number of runners (Shmoys et al., 1991). This bound is tight, i.e., cannot be lowered, as there exist instances where this bound is actually met.

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

Also, the list scheduling algorithm is efficient independently of the number of runners, combining well with the Melissa-DA runner management strategy. While the number of expected runners can be statically defined by the user at start time, the actual number of executed runners depends on the machine availability and batch scheduler. Runners can start at different times, they may not all run due to resource limitations, some may

crash, and try to restart. With list scheduling, a runner gets from the server the next member to propagate as soon as connected and ready.

From this base algorithm, several optimizations can be considered. In particular, data movements could be reduced by trying to avoid centralizing all member states on the runner using decentralized extensions of list scheduling like work stealing (Blumofe and Leiserson, 1999). This could be beneficial when only a small part of the member states are changed during the update phase. This is left as future work.

4.1.7 Data Flow

Member states transit between runner and server memory directly. Transfers take place through parallel N $\times$ M communications (on top of ZeroMQ), where each runner process exchanges data with the corresponding server processes. A single state is thus never aggregated in a single process memory at any time. The full states that differ from member to member are gathered on the server, even though the EnKF algorithm only needs a subpart of each state for the assimilation update. Full states are transferred to the server to enable dynamic load balancing, a key component for improving execution efficiency, and to allow fault tolerance and elasticity, as presented before. On the server, the states are persisted asynchronously to storage using FTI (Section 4.1.5). The primary purpose is to enable restart from the last completed assimilation cycle on server crash. But as FTI can dump the state variables in HDF5 files, the checkpoints can also be used for post hoc analysis (often, users request to keep intermediate states). Centralizing the full states to the server increases the memory needs, which can often be satisfied by increasing the number of server nodes. Another option would be to leverage FTI multi-level checkpointing capabilities to offload data not fitting in memory to the file system or some fast persistent memory like burst buffers or NVRAM when available.

4.1.8 Code

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

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

A derived class calling the PDAF EnKF update phase methods was implemented and is linked against the user-defined methods to initialize the ensemble and observations and to apply the observation operator (Figure 4.2). From the PDAF perspective, the Melissa-DA server acts as a parallel simulation code assembling a "flexible assimilation system" with one model instance propagating all ensemble members sequentially online[1]. By handing over the server MPI communicator to PDAF, the update phase is parallelized on all server cores.

To let Melissa-DA support other assimilation algorithms implemented in PDAF (e.g., LEKF, LETKF, etc.[2]) only classes inheriting the `Assimilator` interface with calls to the desired PDAF filter update methods must be implemented.

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

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

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

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

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

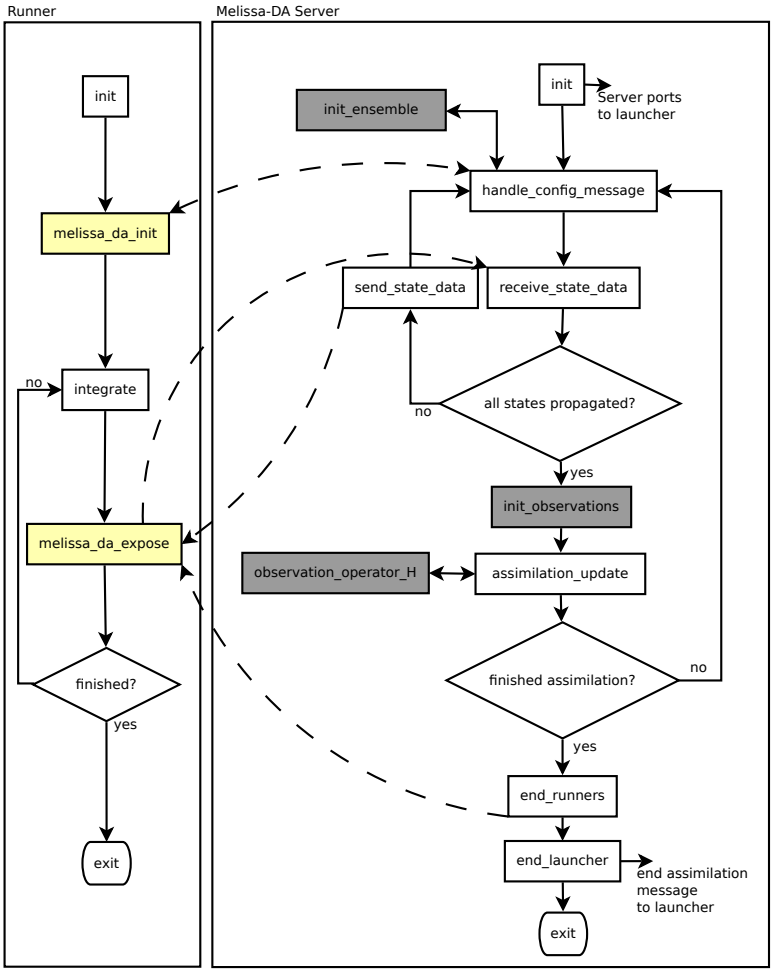


Figure 4.2: Melissa-DA runner and server interactions (fault tolerance part omitted for the sake of clarity). Dashed arrows denote messages that are exchanged between different components. Grey boxes are methods that need to be implemented by the user. Yellow boxes are Melissa-DA API calls that need to be introduced in the simulation code to transform it into a runner.

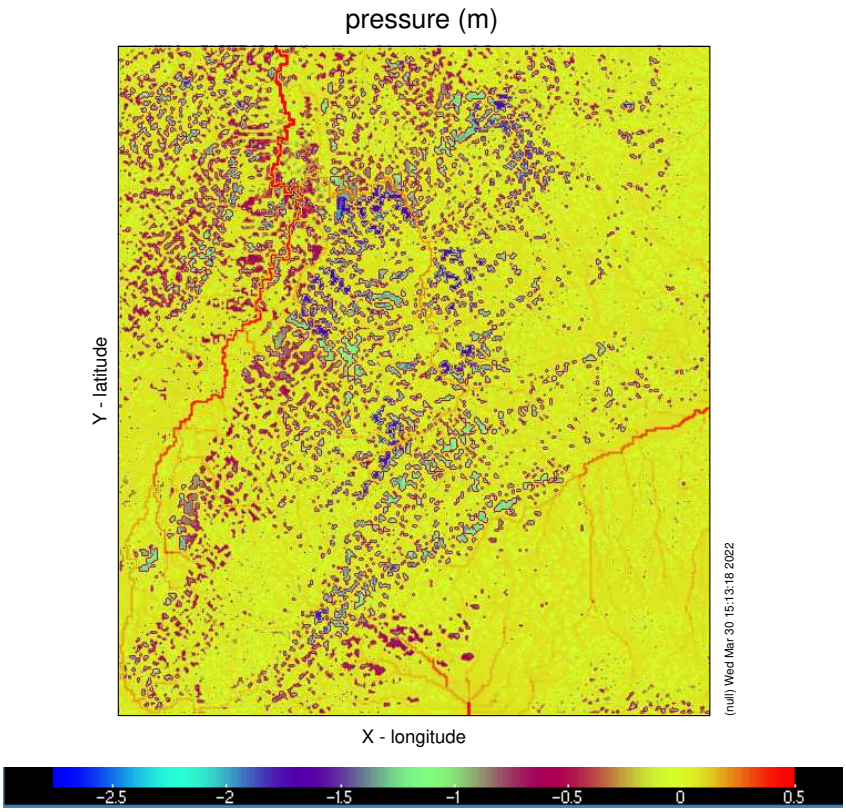


Figure 4.3: Pressure map at 5 cm depth used to initialize an ensemble member for DA on the Neckar catchment. Pressure values range from -2.76 to 0.5 m. The area covers 214 km $\times$ 242 km at 800 m horizontal resolution of the southwest of Germany.

4.2 ParFlow

This section introduces ParFlow, a hydrological model from the geoscience domain. ParFlow is adapted to run at a very large scale and is actively used in DA workflows, for example by Kurtz et al., 2016. Furthermore, it is the subject of some collaborations in the EoCoE-2 project. This makes it an ideal candidate to examine our proposals under realistic circumstances. It is a physically-based, fully coupled water transfer model for the critical zone (Ashby and Falgout, 1996; Jones and Woodward, 2001; Maxwell and Miller, 2005; S. J. Kollet and Maxwell, 2008; Maxwell, 2013). The critical zone is the part of the Earth that contains nearly all living organisms. It spans from bedrock to the top of the plant canopy. ParFlow simulates 3D groundwater and overland flow combined with plant processes. It is extensively used in water cycle research – be it to examine the impact of climate change, for agriculture predictions (soil moisture), or to estimate the impact of well water exfiltration. A typical ParFlow domain is depicted in Figure 4.3. ParFlow is part of various coupled simulation systems. For instance, it is coupled with the weather research forecasting model (WRF) or with the community land model (CLM) to insert

realistic water cycle components. It is also responsible for groundwater simulation in the regional Earth system model TerrSysMP. ParFlow is successfully applied from hillslope over catchment up to continental scale simulations (Lapides et al., 2020; Herzog et al., 2021; S. Kollet et al., 2018).

ParFlow is based on an iterative Krylov-Newton solver. This solver performs a changing number of iterations until a defined convergence tolerance is reached at each timestep. ParFlow relies on the Richards equation (Richards, 1931) to calculate soil water transport. The hydrological conductivity spans multiple orders of magnitude. Some materials, like sand, are very permeable, while, e.g., rock layers break the water flow. This leads to non-linearities hard to capture numerically and predict. Measuring groundwater flow remotely is challenging, and in situ measurements, e.g., from soil moisture sensors can only cover limited areas. DA is thus not only necessary to diminish numerical uncertainty but also to interpolate the few existing measurements in a sophisticated way. Typically, observations from soil moisture measuring networks are assimilated to speed up the model spin-up and improve simulation outputs.

4.3 Experimental Study

Experiments in Section 4.3.6, Section 4.3.7 and Section 4.3.8 were performed on the Jean-Zay supercomputer on up to 500 of the 1,528 scalar compute nodes. Each node has 192 GB of memory and two Intel Cascade Lake processors with 40 cores at 2.5 GHz. The compute nodes are connected through an Omni-Path interconnection network with a bandwidth of 100 Gb/s. The other experiments ran on the JUWELS supercomputer (2 Intel Xeon processors, in total 48 cores at 2.7 GHz and 96 GB of memory per compute node, EDR-Infiniband (Connect-X4)) (Jülich Supercomputing Centre, 2019).

For all experiments we keep nearly the same problem size. Experiments assimilating ParFlow simulations (Section 4.2) leverage $\approx$ 92.4 MiB per member state containing spatially distributed data for the pressure, density and saturation variables (4,031,700 cells in double precision each). This represents the Neckar catchment in Germany. A typical pressure map used to initialize an ensemble member is depicted in Figure 4.3. For our test, we were provided observations from 25 groundwater measuring sensors distributed over the whole catchment. Observation values were taken from a virtual reality simulation (Schalge et al., 2020). ParFlow runners are parallelized on one full node (40 processes for experiments on Jean-Zay and 48 processes for JUWELS respectively). For the experiments profiling the EnKF update phase (Section 4.3.4), a toy model from the PDAF examples, parallelized only on half of a node's cores, is used to save compute hours. For this experiment the member state size is smaller (4,032,000 grid cells, $\approx$ 30.8 MiB). Member initialization (`init_ensemble` function) uses an online approach relying on one initial system state and adding some uniform random noise for each member. Loading terabytes of data for the initial ensemble states is thus avoided. To further circumvent the influence of file system jitter, assimilation output to disk is deactivated if not stated differently.

4.3.1 The Challenge of Setting Up an Ensemble

One key factor in ensemble-based DA is the choice of the ensemble. The initial ensemble should represent the assumed system uncertainty and be large enough to accurately capture future changes in the uncertainty of the system state estimate. For that purpose, thousands of members can become necessary (Zhou et al., 2006; Xie and Dongxiao Zhang, 2010; Rasmussen et al., 2015). Their initialization can be challenging. Ensemble initialization is especially costly for models requiring to execute some time to reach a steady-state that can be used to define ensemble members and to kick off DA. Reaching this steady-state is called to *spin-up*. ParFlow, for instance, needs to simulate several

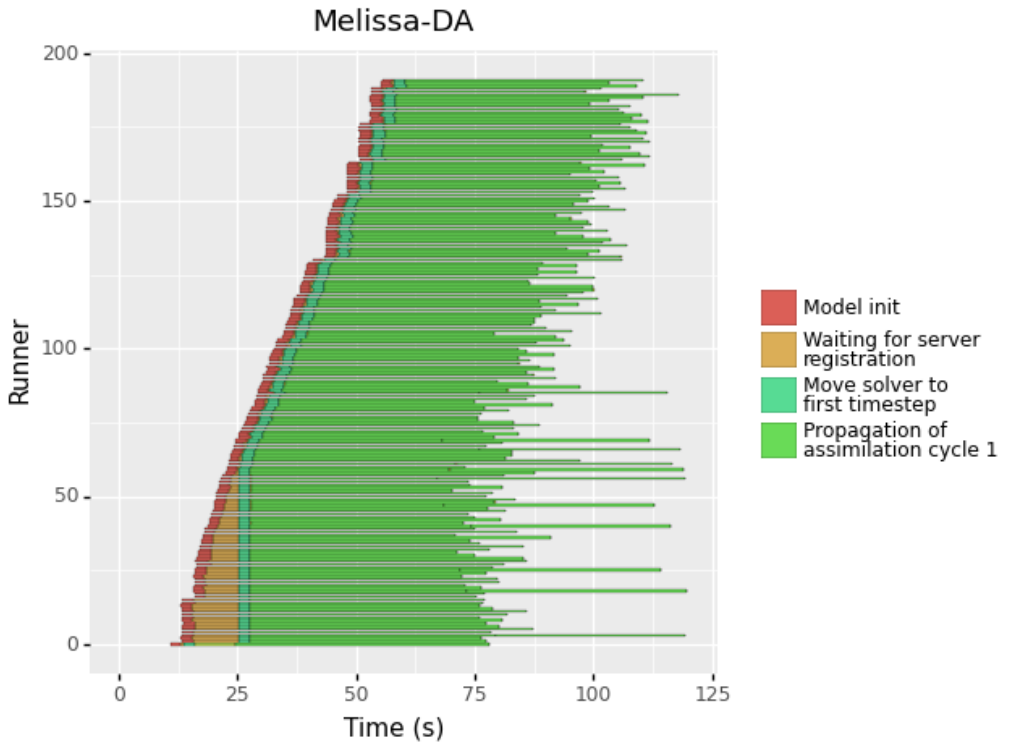


Figure 4.4: Traces of the startup of a Melissa-DA run with 1,000 members on 192 runners, Melissa-DA launcher starts at 0 s.

years to fill groundwater reservoirs, rivers, lakes, etc., with water content to spin-up. Often members may have different boundary conditions and soil parameters. Thus, one cannot simply reuse the output of one spin-up run and add randomized perturbation at the end. In the case of ParFlow, this could even generate physically wrong states. For the sake of compute time-saving, this technique is used in some of the following experiments to simplify stress-testing of Melissa-DA at a very large scale – even if that might void the interpretation of the physical assimilation outputs.

4.3.2 The First Assimilation Cycle

In the following experiments, we reserved all the nodes required for the run upfront (Slurm allocation) and then requested Slurm to start jobs into this allocation. The goal was to avoid introducing delays between job starts due to the machine load. For a production run, the upfront resource reservation is not required or can be done for the server and a few runners to ensure a minimal progress speed at start. However, this upfront reservation is not sufficient to ensure a synchronous start (Figure 4.4) due to the combination of several factors:

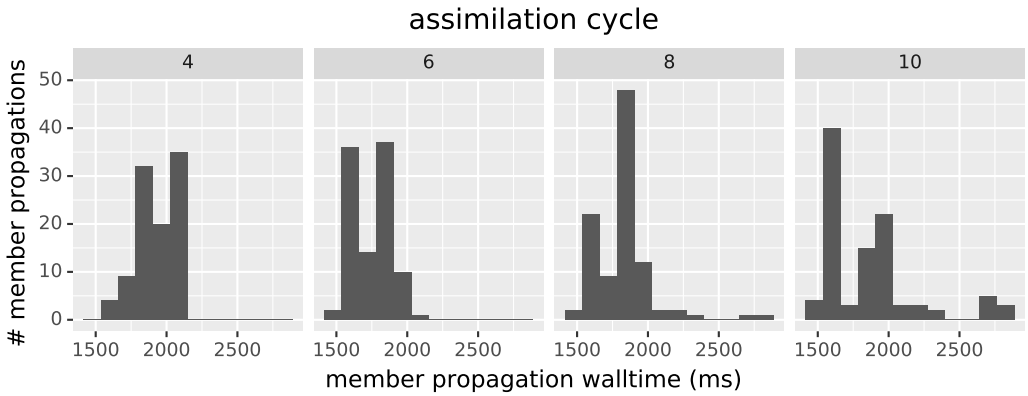


Figure 4.5: Histograms of propagation walltimes for 100 members during multiple assimilation cycles.

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

- The launcher submits runner jobs independently and sequentially.

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

- The server initializes its data structures holding the full state ensemble only after a first runner connection to get the full information on the N $\times$ M data redistribution scheme. Runners 0 to 60 have to wait ("Waiting for server registration" in Figure 4.4) before they can receive propagation tasks from the server (at 25 s after launcher start).

- ParFlow takes about 14 times longer to converge for member propagations of the first assimilation cycle compared to member propagations of later assimilation cycles. This behavior of ParFlow is due to misfitting initial data challenging the solver. For the case displayed in Figure 4.4, each initial member propagation takes 43 s on average, while later member propagations take about 3 s on average.

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

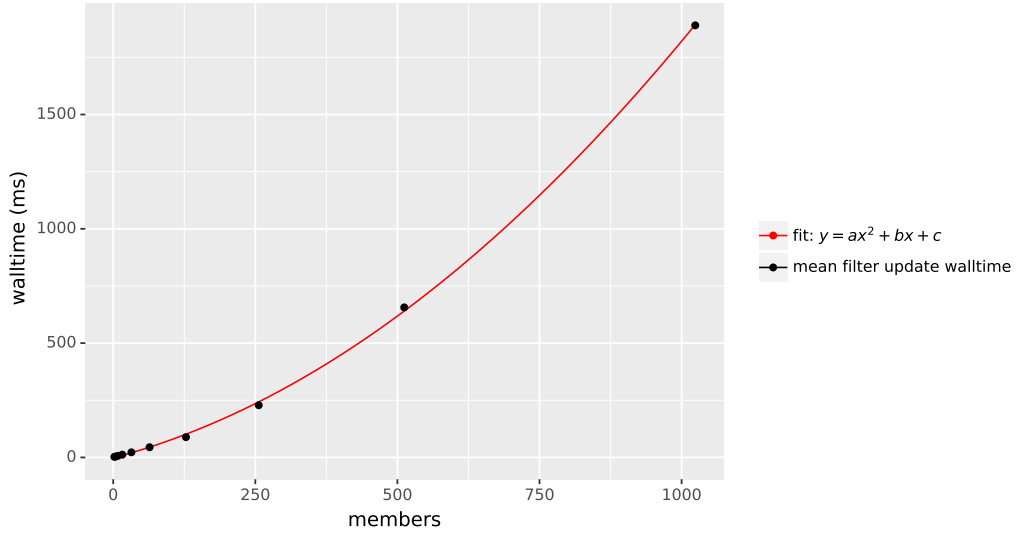


Figure 4.6: Assimilating 288 observations into about 4 M grid cells with up to 1,024 members on JUWELS. Mean over 25 update phase walltimes.

4.3.3 Ensemble Propagation

Figure 4.5 shows the walltime distribution of 100 ParFlow member propagations for multiple assimilation cycles. Member propagation times starting from the second cycle compare Section 4.3.2 can vary significantly from about 1.5 s to 2.5 s, with an average at 1.9 s. The main cause for these fluctuations is the Krylov-Newton solver used by ParFlow that converges with a different number of iterations depending on the member state. As detailed in Section 4.1.6, these variations can impair the execution efficiency. Melissa-DA mitigates this effect by dynamically distributing members to runners following a list scheduling algorithm.

4.3.4 EnKF Update Phase

Figure 4.6 displays the evolution of the update phase walltime depending on the number of members, using Melissa-DA with the PDAF implementation of EnKF. The mean is computed over 25 assimilation cycles, assimilating 288 observations each time. Standard deviation is omitted as not being significant ($< 6\%$ of the update phase walltime). The EnKF update phase is executed on 3 JUWELS nodes (144 cores in total). The EnKF update phase relies on the calculation of covariance matrices over M samples resulting in a computational complexity for the EnKF update phase of $O(M^2)$, with M being the number of ensemble members. This is confirmed by the experiments that fit a quadratic function. The walltime of the update phase also depends on the number of observations.

In the following experiments less observations are used, leading to an update phase of about only 1.1 seconds.

We also performed a strong scaling study (Figure 4.7, *top*), timing the update phase for 1,024 members while varying the number of server cores. The parallelization leads to walltime gains up to 576 cores. Computing the covariance matrix for the update phase is known to be difficult to efficiently parallelize. Techniques like localization enable to push the scalability limit. Localization is not used here as we run with a limited number of observations. As Melissa-DA relies on PDAF which supports localization, localization can be easily activated by changing the API calls to PDAF in the Melissa-DA `Assimilator` interface. Refer to Lars Nerger and Hiller, 2013 and L. Nerger et al., 2005 for further EnKF/PDAF scaling experiments.

Dimensioning the Melissa-DA server optimally depends on the assimilated problem dimensions, the used assimilation algorithm, and the target machine. It should be examined in a quick field study before moving to production. The results of such a field study can be seen in Figure 4.7, *bottom*. In the depicted case, fewer core hours are consumed when not using a large number of server nodes (and cores respectively), since these are idle during the whole propagation phase, outplaying the walltime advantage they bring during the update phase. For the next experiments we assume that the update phase is short compared to the propagation phase leading to the policy: Use the least server nodes able to fulfill memory requirements. Do not use all cores per server node if this would slow down the update phase as the domain is not efficiently splittable anymore (remember that the update phase uses domain distribution for parallelization). While, for instance, compute nodes on JUWELS leverage 48 compute cores, it can be faster to only use a fraction of them for the server. Splitting the domain adds communication overhead. At some point the speedup won from more parallelization of the update phase cannot amortize the increasing communication overhead anymore. Some supercomputers provide special large memory nodes that could be leveraged to run the Melissa-DA server too.

4.3.5 Runner Scaling

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

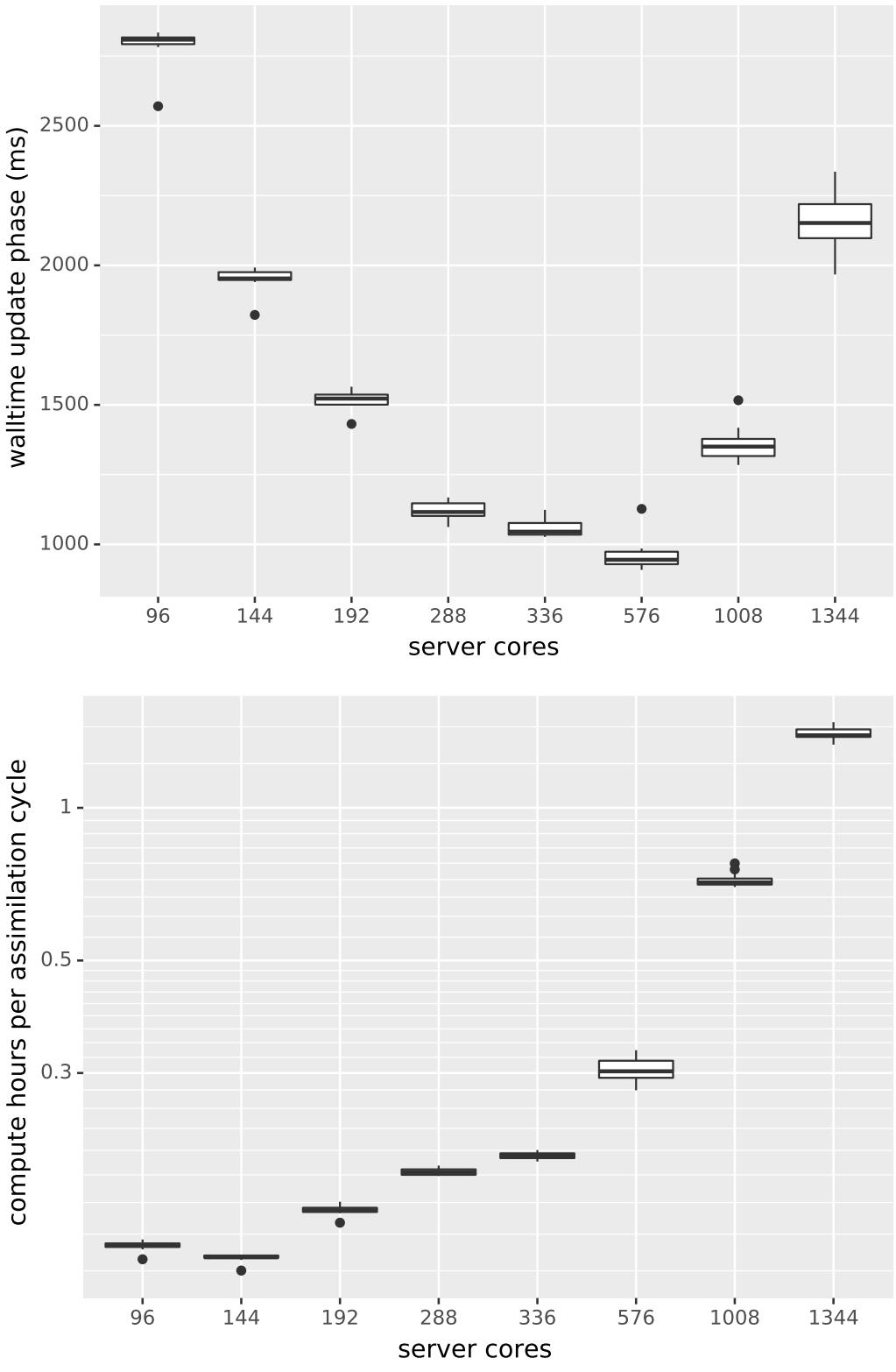


Figure 4.7: Boxplot of the update phase walltime (*top*) and the total compute hours per assimilation cycle (compromising update and propagation phase, *bottom*) when assimilating 288 observations into about 4 M grid cells with 1,024 members with a varying number of server cores on JUWELS. The latter graph can be used to evaluate server dimensioning.

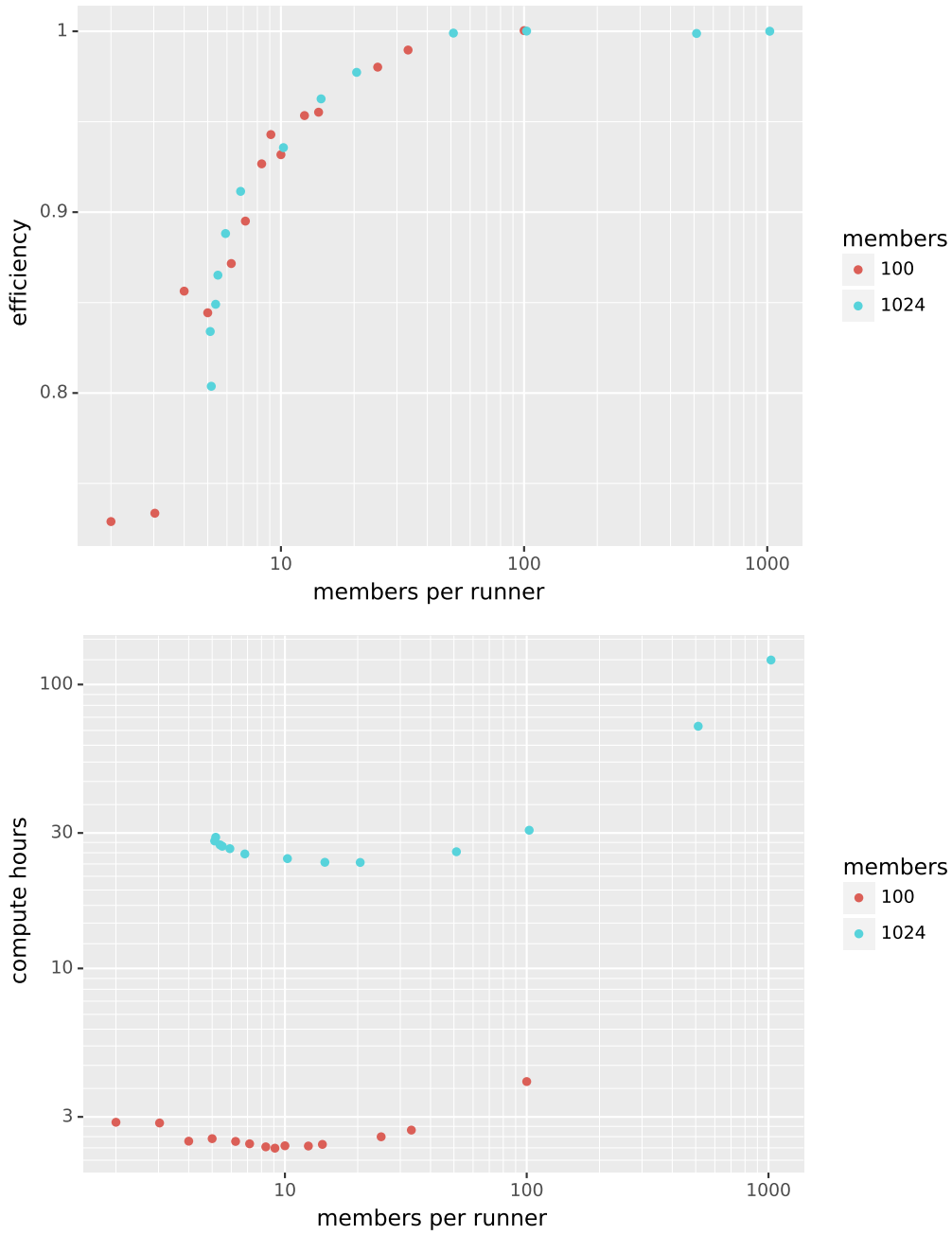


Figure 4.8: Efficiency of the propagation phase only (*top*) and total compute hours used per assimilation cycle (update and propagation phase) (*bottom*) for different numbers of runners while assimilating 25 observations into an about 4 M grid cell ParFlow simulation with 100 and 1,024 ensemble members.

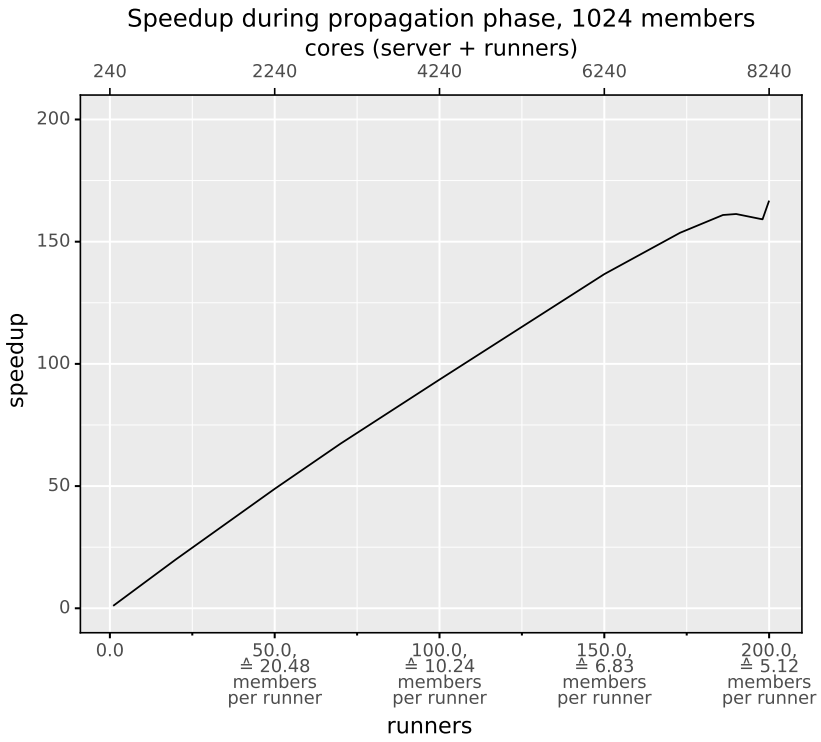


Figure 4.9: Speedup during propagation phase for a ParFlow DA problem, running on a varying number of resources. *Top x-axis*: cores used. *Bottom x-axis*: runners, members per runner.

We experiment with a varying number of runners for a fixed number of members (100 and 1,024), and a given server configuration (Figure 4.8). Plotted values result from an average obtained from eight executions, taking for each execution the time of the last two over three assimilation cycles. The efficiency of the propagation phase (Figure 4.8 *top*) is computed against the time obtained by running the members on a single runner. The compute hours are the total amount of consumed CPU resources (update and propagation phase, runners and server) during the assimilation cycles. For both plots, standard deviations are omitted as being small (relative standard deviations, $\frac{\text{standard deviation}}{\text{mean}}$, always smaller than 3 %). The server was scaled to meet the memory needs. Each runner executed ParFlow on 48 cores (1 node).

Efficiency during the propagation phase stays beyond 90 % when each runner propagates at least 7 or 8 members, 95 % for more than 10 members per runner and close to 100 % for 50 or 100 members per runner. The data can also be presented as speedup (Figure 4.9). The speedup flattens if runners get less than 7 members to propagate per propagation phase.

This demonstrates that Melissa-DA's load balancing algorithm maintains high efficiencies down to a relatively small number of members per runner. Obviously these levels of efficiency also depend on the distribution of propagation walltimes (see Section 4.3.3). Note that the update phase takes about 0.1 s and 1 s in the case of 100 and 1,024 members respectively, leading to scaling efficiencies for the full assimilation cycle decreased by at most 3 %. Also, the resources used for the server need to be considered. The total amount of compute hours (Figure 4.8 bottom) shows a U shape curve with a large flat bottom at about 9 members per runner for the 100 members case and at about 20 for the 1,024 members case. Changing the number of members per runner around those sweet spots changes significantly the efficiency of the propagation phase but slightly impacts the total compute hours: the efficiency variation is compensated by the impact on the runner idle time during the update phase that varies inversely (runners are mostly idle during the update phase). This also shows that changing the number of runners in these areas is useful, advocating for leveraging Melissa-DA's elasticity for adding or removing runners according to the machine availability.

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

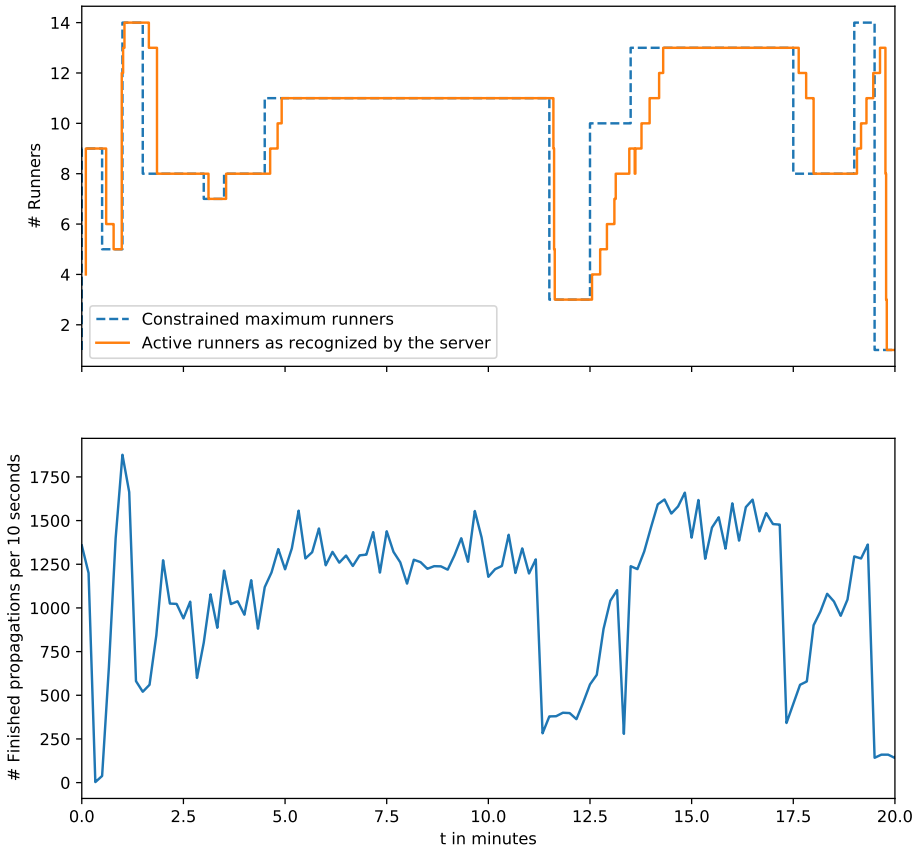


Figure 4.10: Elasticity with Melissa-DA: constraining the resources that may be used by runners (*top*), impacts the assimilation speed (*bottom*).

4.3.6 Fault Tolerance and Elasticity

In this experiment, we demonstrate the capability of Melissa-DA to have runners dynamically added or removed, giving Melissa-DA elasticity and the base of its fault tolerance protocol. The number of runners is limited by a target (dashed blue curve, Figure 4.10). The launcher periodically checks if more runners can be started. If so, it starts new runners. If too many resources are used, some runners are stopped. Runners are killed without notice, as in the case of an error. There is a delay between the launcher request to start a new runner and this runner being registered at the server (solid orange curve, Figure 4.10). Runner stops are also recognized after a delay by the server. Only if a runner did not respond for the last 10 seconds the server assumes it stopped. This runner timeout can be modified by the user. Having more or fewer runners impacts the speed at which the data assimilation runs, here indicated by the number of state propagations that finish per interval of 10 seconds (Figure 4.10, solid blue curve).

Notice that here we leverage the batch scheduler capabilities (Slurm). A single allocation encompassing all necessary resources is requested at the beginning. Next, jobs for the server or runners are allocated by Slurm within this envelope, ensuring a fast allocation. When the runners crash or are stopped, the restarted runners reuse the same envelope. A hybrid scheme is also possible, requesting a minimal first allocation to ensure the data assimilation progresses fast enough, while additional runners are allocated outside the envelope, but whose availability to accept members may take longer depending on the machine load. It is planned to rely in such situations on *best effort* jobs that are supported by batch schedulers like OAR (Capit et al., 2005). Best effort jobs can be deleted by the batch scheduler whenever resources to start other higher prioritized jobs are needed. This way Melissa-DA runners can fill up underutilized resources between larger job allocations on the supercomputer. All these variations on the allocation scheme require only minimal customization of the assimilation study configuration.

4.3.7 Ultra-Large Ensembles

We scale Melissa-DA to ultra-large ensembles, assimilating with up to 16,384 members. To save compute hours, only a few assimilation cycles ran on up to 448 compute nodes using up to 16,240 cores of the Jean-Zay supercomputer. When doubling both the ensemble size and the number of runners, the propagation phase's execution time stays roughly the same with an average number of about 20 members per runner up to 8,192 cores (Table 4.1). Dynamic load balancing leads to 96 % efficiency (or 4 % runner idle

Members	2,048	4,096	8,192	16,384
Amount of runners	100	200	400	400
Average members per runner	20.48	20.48	20.48	40.96
Cores per runner	40	40	40	40
Nodes per runner	1	1	1	1
Server cores	240	240	240	240
Server nodes	6	12	24	48
Ensemble state size (sum of all member state sizes) (TiB)	0.240	0.481	0.961	1.922
RAM theoretically available on server nodes (TiB)	1.125	2.25	4.5	9
Update phase walltime (ms)	5,339	11,872	28,368	52,893
Propagation phase walltime (ms)	41,555	41,757	42,295	83,898
Mean runner idle time (propagation + update phase) (ms)	361	704.8	1,441	1,255
Median runner idle time (propagation + update phase) (ms)	19.03	12.78	15.84	19.28
Std runner idle time (propagation + update phase) (ms)	1,518	3,058	6,294	7,799
Mean runner compute time (propagation phase) (ms)	1,948	1,952	1,986	1,961
Median runner compute time (propagation phase) (ms)	1,918	1,909	1,914	1,903
Std runner compute time (propagation phase) (ms)	155.9	165.2	273.9	248.6
Runner idle time during propagation phase (%)	3.968	4.227	3.819	4.468
Scaling efficiency for the propagation phase (%)	96.03	95.77	96.2	95.76
Scaling efficiency for the assimilation cycle (%)	85.1	74.57	57.58	58.74

Table 4.1: Large scale Melissa-DA runs. Scaling efficiency is computed against the walltime of the execution on a single runner.

time) during the propagation phase in this case. During the idle time, the runners are either waiting for new state data to arrive or the update phase to begin.

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

The update phase takes a considerable amount of time when using EnKF on such large member counts (Section 4.3.4). During this time the runners are idle, giving a total scaling efficiency of only about 59 % with 16,384 members. The scalability of the update phase is a well-known issue of DA. Classical solutions exist to reduce the update execution time, like relying on a localized EnKF filter (basically some terms of the covariance matrix are set to zero). Other less classical approaches to shorten the wait time during the update phase could be the iterative calculation of some parts of the EnKF update phase each time a new background state is received (Niño et al., 2012). Investigating such algorithms for the update phase is part of future work.

The server is spread on the minimum number of nodes necessary to fulfill the memory requirements to store the ensemble and to perform the update phase computations (Table 4.1). Since Melissa-DA and the underlying EnKF update phase parallelizations are based on domain decomposition, and the domain size does not change when the number of members increases, the number of allocated server processes is kept constant at 240 cores (otherwise communication cost between the server processes would overwhelm calculation).

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

4.3.8 Comparison of Melissa-DA and PDAF

We compare Melissa-DA to PDAF (Lars Nerger and Hiller, 2013; L. Nerger et al., 2005)[3], one of the most advanced frameworks for large scale ensemble based DA. Both frameworks are configured to run the same DA use case, assimilating the Necker catchment in Germany simulated by ParFlow as introduced at the beginning of Section 4.3.

PDAF, as detailed in Section 3.1.4, supports an offline file-based mode and an online mode. The latter is used here as it provides the best performance. In online mode PDAF

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

runs a single large MPI executable whose processes are split into runners, called *model tasks* in PDAF jargon, where:

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

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

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

Figure 4.11 compares the assimilation cycle times. Times are measured from the second cycle for the reasons explained in Section 4.3.2. Details on the first cycle can be found in Table 4.2. Two types of runs were performed: runs where runners perform no output to disk, and runs where runners write the member state to disk (state dump) after each propagation. This latter case introduces jitter in the propagation times, and so load imbalance, as the write time is sensitive to the file system load. This also matches a classical scenario when users require states to be saved for post hoc analysis. Melissa-DA outperforms PDAF except at smaller scale with 250 members. Melissa-DA is up to almost 3 times faster than PDAF at 2,500 members with state dumping.

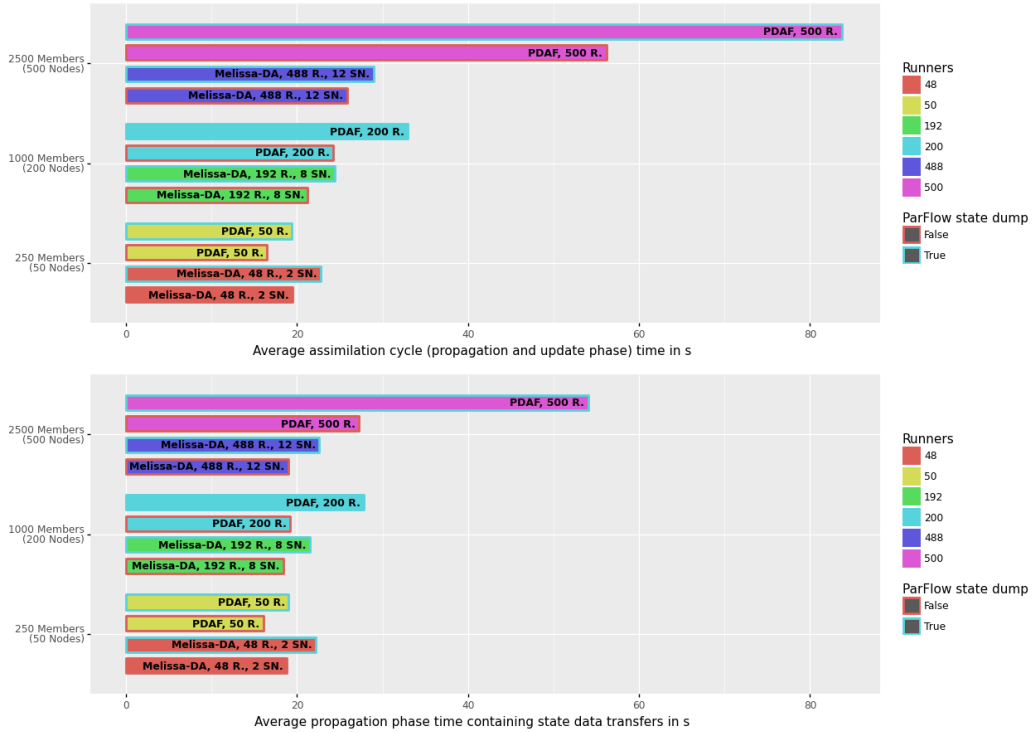


Figure 4.11: Comparison of the runtime of Melissa-DA and PDAF running the same DA problem on the same amount of resources. *Top*: full assimilation cycle time. *Bottom*: propagation phase time of each cycle. S.N.: number of server nodes, R.: number of runners. ParFlow state dump true if runners output each member state to disk.

Note, that Melissa-DA ran with server checkpointing activated. The server checkpoints contain the information to restore the full ensemble and could also be leveraged to output the full ensemble as we detail it in Section 4.1.7. This would make file output on the runner side useless. So we could compare Melissa-DA without ParFlow's state dump to PDAF with ParFlow's state dump activated as writing equivalent information, giving Melissa-DA an even larger performance advantage on large problem sizes.

The resources are split differently for PDAF and Melissa-DA. Melissa-DA performs the update phase on a server with 80, 320 and 480 cores for 250, 1,000 and 2,500 members, and 1,920, 7,680 and 19,520 cores are used for runners performing member propagations. PDAF uses always 40 cores for the update phase and 2,000, 8,000 and 20,000 cores for runners performing the member propagations. The update times are similar at 250 members (about 0.4 s) and reach 29 s for PDAF versus 6.5 s for Melissa-DA at 2,500 members. But the performance gain of Melissa-DA is not only related to the performance improvement of the update phase as visible when looking at the propagation times only (Figure 4.11 *bottom*) or when comparing execution traces (Figure 4.12).

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

1. The number of members, is chosen to be a multiple of the number of PDAF runners. Exactly 5 members are executed by every runner. That makes for an uneven number of members per runner for Melissa-DA. This is visible in Figure 4.12 where Melissa-DA needs to wait for the propagation of the few runners receiving 6 members before starting the update phase.

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

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

Allocating more resources to the server for Melissa-DA can be essential for performance. The trace of 500 members (Figure 4.13) shows that with 1 server node (40 cores, *right*), the server becomes a bottleneck impairing runner progress. At 4 server nodes (160 cores, *left*), runner idle time during the propagation phase is significantly reduced. We suspect that the ZeroMQ library used for the data transport but not designed and implemented to take full benefit of the underlying high-performance network is partly responsible for

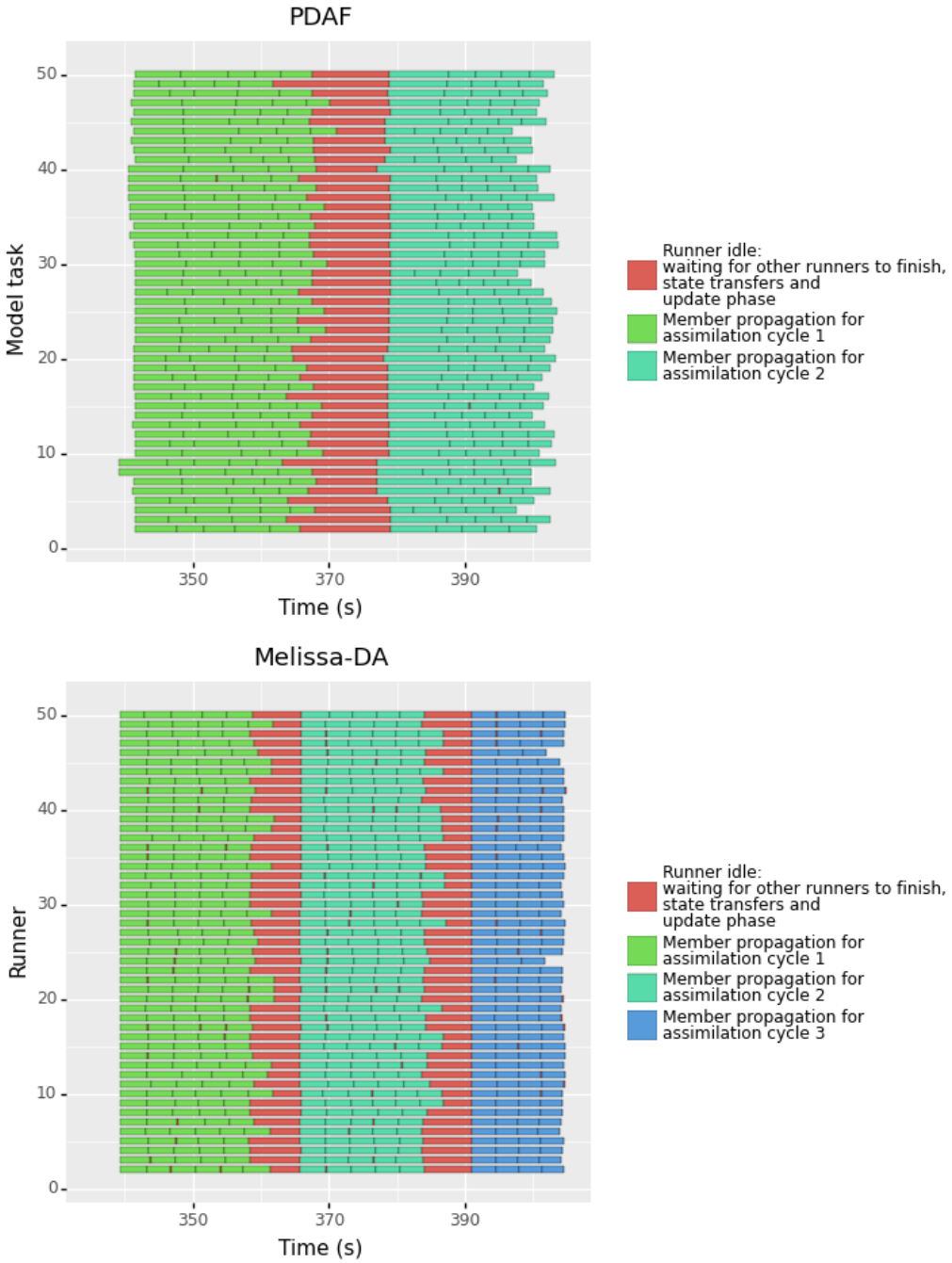


Figure 4.12: Traces of 50 PDAF and Melissa-DA runners. Both runs were performed on 200 nodes used as 200 runners (PDAF) or 192 runners and 8 server nodes (Melissa-DA) to propagate 1,000 members. Every runner in PDAF propagates exactly 5 members, while Melissa-DA runners propagate between 5 and 6 members per cycle. Every member state is dumped during propagation.

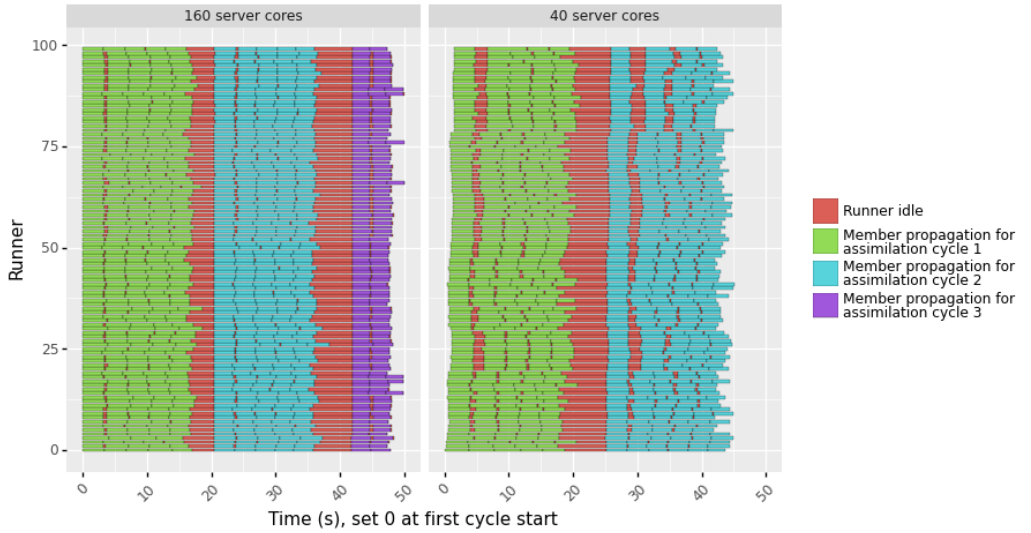


Figure 4.13: Traces of a Melissa-DA run with 500 members on 100 runners. The network buffers of one server node saturate and thus cannot send out all state data at once (*right*, 40 server cores). Adding extra server nodes fixes this issue (*left*, 160 server cores). Note that initialization time was cut off in the plot.

these idle times. Changing to native high-speed networking protocols for InfiniBand or Omni-Path hardware could bypass this bottleneck. `MPI_Comm_connect` could be used to build a fault-tolerant client-server architecture on top of these high-speed networks. Unfortunately, MPI distributions on the supercomputers we had access to did not fully support this feature yet. The usage of other high-speed networking libraries like libfabric[4] or libverbs[5] is to be evaluated in future.

In both cases (4 server nodes or 1 server node) we run 100 runners. So the 500 members can be evenly distributed. But even in this situation, the trace shows that dynamic load balancing is critical for performance. With one server node, some runners get 6 members to propagate, as some other runners experiencing significant delays only get 4 members.

We now compare startup times (Table 4.2, see "Start 2nd cycle"). PDAF benefits from being a single MPI executable, requiring a single request to the batch scheduler to start. Melissa-DA takes longer mainly due to the multiple requests done to the batch scheduler (Section 4.3.2), as well as its dynamic architecture. Melissa-DA is designed to have runners executed in an elastic mode, started independently as resources become available, rather than waiting to have the full set of resources reserved as required for PDAF.

[4] `https://github.com/ofiwg/libfabric`, retrieved the 25.01.2022
[5] `https://github.com/linux-rdma/rdma-core`, retrieved the 26.01.2022

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

Table 4.2: Startup times (seconds) of PDAF and Melissa-DA at 2500 members.

4.4 Conclusion

In this chapter, we introduced Melissa-DA, a framework for ensemble-based DA. A key feature is the introduced member virtualization that permits decoupling member propagation and resource allocation while avoiding file I/O. The framework can reduce load imbalance during the propagation phase by list scheduling member propagation tasks to different runners. A remaining source of imbalance is the synchronization point imposed by the state update. As soon as there is no member left to propagate for the current propagation phase, runners are idle, waiting for the end of the propagation phase and the server to finish the state updates. For large ensembles, this may take a significant amount of time. The server needs to store and process the whole ensemble. This is especially limiting when ensemble sizes or member state sizes overwhelm the server RAM. Offloading states to the local file system with associated performance impact could be used to circumvent this limitation.

Experiments proved the efficiency of the framework's fault-tolerance, elasticity, and load-balancing features. This enabled to scale EnKF up to 16,384 members for large scale DA with model state vectors of more than 4 M degrees of freedom. We also showed that the framework can compete with existing approaches at common scales, but outperforms them at large scale when processing multiple thousand members.

Experiments quantified the expected limitations of our approach. The update phase, adding synchronization between all members and blocking all runners is responsible for an important amount of runner idle times. This sensibly impacts the scaling efficiency of the framework. The effect worsens the larger the run in terms of members or state sizes. For our largest run, scaling efficiency reduced from 96% during the propagation phase to 59% for the update and propagation phases combined. To augment the efficiency we would need to avoid the centralization of states on the server and a reduction of the update phase calculations. But this is hardly possible with EnKF since it needs to centralize all states to compute and apply the state update. In the next chapter, we propose to adapt the Melissa-DA architecture to support a less centralized approach as allowed by particle filter DA algorithms.

5 Large Scale Particle Filtering

Particle filters allow to assimilate in very non-linear settings and work also when the model and observation errors cannot be approximated by Gaussian distributions. Nevertheless, much more members, i.e., particles, than for EnKF are needed to keep a comparable accuracy. Thus, particle filters are more compute-intensive.

The following Section 5.1 presents performed changes to enable particle filters in the Melissa-DA framework. A performance analysis of the particle filter implementation is shown in Section 5.2. The chapter is concluded in Section 5.3.

Note that the work presented here results from a collaborative effort with Kai Keller[1], Yen-Sen Lu[2], Bruno Raffin[3] and Leonardo Bautista-Gomez[1]. It was submitted to the IEEE Cluster 2021 conference but got rejected. A revised version is under active development.

The proposed particle filter implementation is completely integrated into the Melissa-DA framework. Its source code is available online under `https://gitlab.inria.fr/melissa/melissa-da`. More technical documentation is available there too.

[1] Barcelona Supercomputing Center
[2] Forschungszentrum Jülich
[3] Inria Grenoble

5.1 Architecture

Let us describe the architecture and design choices made to create the particle filter framework now. Particle filters propagate an ensemble of members, i.e., particles, that then is updated using observations, leading to the next assimilation cycle. But there are differences in the data dependency graph compared to other ensemble-based DA methods like EnKF. As described in Section 2.3.2, EnKF centralizes all member states after propagation to calculate and apply the state update to each member state. In consequence, the full ensemble of state vectors is processed at each update phase.

Particle filters, in contrast, do not require state centralization. After the propagation and weighting of all particles, only weights, i.e., one scalar representing each particle, need to be gathered for normalization. Next, the normalized weights are used as probability to resample a new set of parent particles from the propagated particles. The parent particles are propagated further in the next propagation phase. In contrast to EnKF, this neither necessitates centralizing data-heavy state vectors nor changing them. Instead, only the composition of the ensemble changes during the update phase. Refer to Section 2.3.3 for more details on the particle filter.

Our particle filter implementation extends Melissa-DA. Runners propagate particles during the propagation phase and compute locally the unnormalized particle weight. The server distributes propagation tasks and is responsible for the update phase. This enables features like load balancing, fault tolerance, and elasticity. The simple update phase in particle filters (performing only weight normalization and resampling) allows running a sequential server code that does not need to store all member states. It only needs to gather member weights and distribute propagation tasks to runners. A multi-level distributed cache is instead used by runners to store states and enable moving them between runners when needed.

The multi-level distributed cache leverages a runner, in-memory, *local cache* level and a *shared* or *distributed* level. The shared level relies currently on the shared parallel file system (PFS) available on supercomputers. Using the PFS as shared cache fulfills several purposes at once. First, its persistence allows to use states stored there as checkpoints for recovery from faults. Second, it enables to share states between multiple runners. Third, the PFS permits state offloading in situations where the union of all runner local caches is not large enough to store the full ensemble.

In the following, the changes to the different components of the Melissa-DA workflow are detailed.

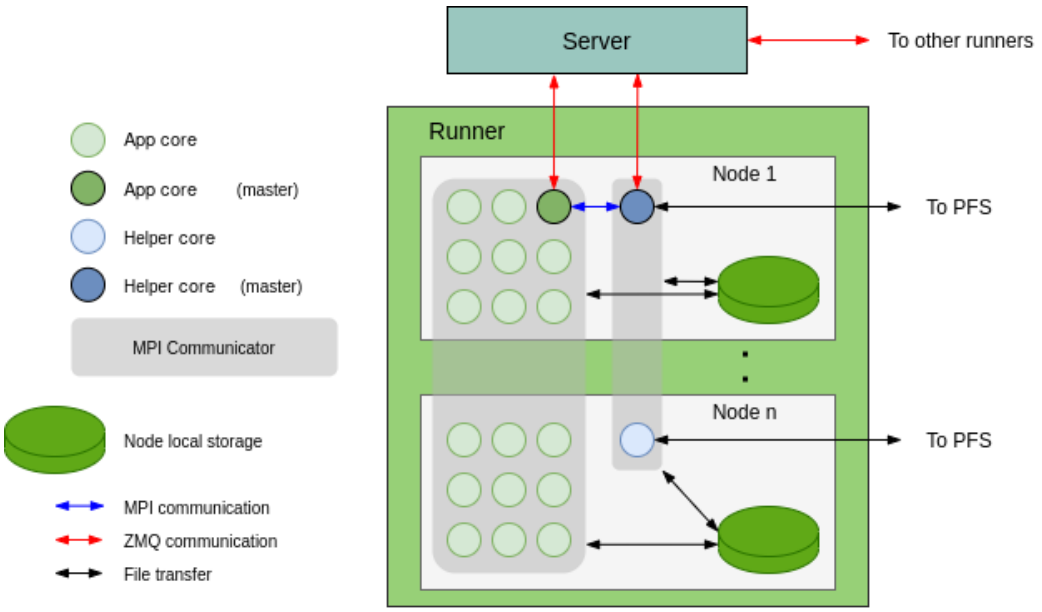


Figure 5.1: Runners/server architecture. The app cores perform the state propagation, the helper cores send propagated states to the PFS and prefetch the next scheduled states to the local cache in the background. Communications with the server combine MPI and ZeroMQ data exchanges.

5.1.1 Runners

Each runner executes one MPI parallelized model instance to propagate particles, following the same member virtualization principle as for the EnKF runners. The implementation follows an in situ workflow. MPI communicators of the model code are split into app cores and one helper core per node. While the app cores run the model to perform particle propagations and weight calculations, the helper cores manage the cache state loads and stores. This approach enables to overlap state loads and stores with state propagations and weight computations.

The node-local RAM disk is used as a local cache, and the parallel file system (PFS) is leveraged as shared cache. The helper cores copy the particles that result from propagation from the local cache into persistent storage shared between all runners, the shared cache. Then, the helper cores request the server which particle to propagate next. If it is not already available in the local cache, they prefetch it from the shared cache. Note that only app core rank 0 and helper core rank 0 directly contact the server. Messages are then distributed, using MPI, to other ranks of the same kind. As in the approach for EnKF, ZeroMQ is leveraged for the runner-server communication. This workflow is depicted in Figure 5.1.

Regularly, the helper core clears outdated particles of previous assimilation cycles from the local cache. If a local runner cache is still full, a helper core requests from the server which particle states are best to remove.

During the whole process, app cores are only interacting with the runner local cache to load and store particles. They do not write and read directly on the PFS. App cores wait only when a particle to be propagated next could not be prefetched by a helper core in time.

Runners also calculate the particle weight $w_{i,t}$ following Equation (2.24) after each particle propagation. Weight calculation is problem-dependent. Assimilating different types of observations requires different weight calculations. For that purpose, scientists can define a Python function that is called from the Melissa-DA API to calculate the weight. The Python function has the signature shown in Listing 5.1:

```
1 def calculate_weight(assimilation_cycle, particle_id, background_particle_state,
      assimilated_index, assimilated_varid, mpi_comm):
    return weight_as_float
```

Listing 5.1: Weight calculation interface

The `calculate_weight` function provides access to the underlying raw state data (`background_particle_state`). This function is called in parallel by every app core to permit the parallelization of weight calculation.

Alternatively, scientists can implement weight calculation within the model code and transmit the particle weight to Melissa-DA using the `melissa_set_weight` API call. Implementing weight calculation within the model context may make it easier to access the system state: the system state representation stays the same as used through the model code.

5.1.2 Server

In contrast to the EnKF server, which centralizes all member states to apply the state update, the particle filter server only needs to receive the weights of all particles to perform resampling and to distribute new work, i.e., particles to be propagated, to the runners.

For that purpose, runners connect to the server requesting particles to be propagated, prefetched, or removed from the local cache. They notify the server about the weights $w_{i,t}$ of each successfully propagated particle. Once the server received all particle weights $w_{i,t}$,

it resamples the particles for the next assimilation cycle. Only then outstanding job and prefetch requests are answered.

Currently, the server implements the two different resampling schemes that were introduced in Section 2.3.3, multinomial resampling, and RR, but this is easily extensible. Both schemes perform importance resampling, relying on particle weights to define at which probability a particle is resampled.

While the Melissa-DA server for EnKF is a parallel MPI application that handles the full ensemble state data using N $\times$ M data redistribution, the particle filter server is sequential and exchanges only lightweight messages. Next to some administrative messages to register runners and launcher, the particle filter server receives:

- particle weights (a scalar value),
- job propagation requests,
- prefetch requests, and
- delete requests from the runners.

All these messages contain only lightweight data identifying the requester and the request type, and possibly a scalar value for a particle weight. The server responds with small messages containing only particle and runner IDs.

5.1.3 Runners/Server Workflow

After the launcher started a runner, the runner requests from the server which particle to propagate (Figure 5.1). The server responds to this request following the scheduling policy (Section 5.1.4). Next, the server notifies the launcher about the successful runner start-up. The helper cores on the runner ensure that the particle state to be propagated is available in the local cache of the runner. If necessary, the app cores are blocked until the helper cores finish loading the required particle state from the PFS. Then, the app cores start propagation.

At the same time, the helper cores already request the next particle to be propagated from the server and load the particle state in the local cache if necessary. This mechanism avoids almost all blocking of the app cores, as we will show in Section 5.2.2. The only situation where runners are idle is at the end of each assimilation cycle. Prefetching and propagation activities are paused since the server cannot tell which particles will be resampled for the next cycle yet.

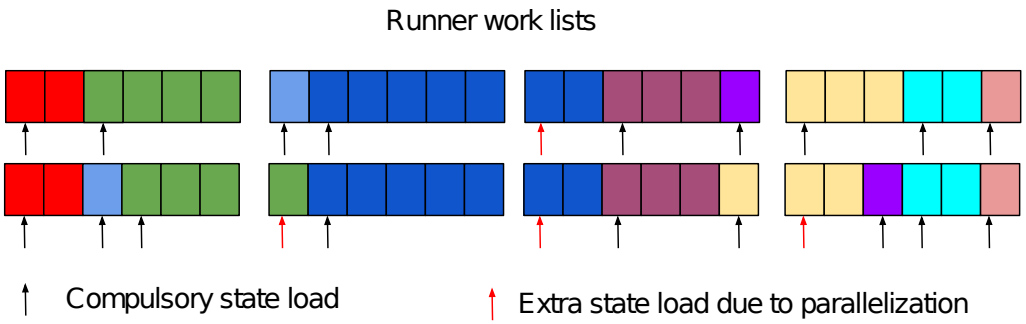


Figure 5.2: Two possible schedules of 24 propagation tasks of equal duration on four runners. All particles propagated from the same parent state have the same color (9 parents here). The top schedule is optimal with nine compulsory loads (one per parent) and one for the dark blue parent that cannot fit in one runner. The bottom schedule, with two more state loads, is a possible one that our online scheduling algorithm can produce. This is not optimal but still below the general $P + R - 1$ bound, as the algorithm ensures for even propagation runtimes that no more than $R - 1$ "color cuts" occur and avoids the same runner loading more than once a given parent state.

Each time the app cores finish particle propagation, they compute the unnormalized weight $w_{i,t}$ of the resulting particle and store it in the local cache. Finally, they send the unnormalized weight together with some metadata to the helper cores and request the next particle to propagate. Thanks to prefetching, this particle is normally available in the local cache already. Propagation goes on while the helper cores process the received information.

Upon reception of an unnormalized weight, the helper cores know that the according state is available in the local cache. They copy it to the shared cache (PFS). Once finished, unnormalized weight and metadata are forwarded to the server. Thus, the server only handles weights of particles that are checkpointed and available to all runners. The helper cores resume prefetching activity. Note that if a runner local cache is full, but some state still needs to be prefetched, the helper cores first request from the server which state to evict. The server responds to such requests following the eviction strategy described in Section 5.1.5.

5.1.4 Scheduling

This section presents the scheduling algorithm used by the server to distribute the particle propagations to the runners. The presented algorithm is first derived from a theoretical point of view on a simplified problem.

Let us first derive the theoretical base of the used scheduling algorithm. 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:

a) runners do not cache more than one particle,

b) the number of runners is constant, and

c) 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 particle states 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

$$s_i = \left\lceil \frac{\alpha_i}{\frac{M}{R}} \right\rceil. \tag{5.1}$$

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 5.2). 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

$$P \leq c^* \leq P + R - 1. \tag{5.2}$$

We can define a static schedule that respects this upper bound: distribute $\frac{M}{R}$ particles 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 with 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 (Graham, 1966; Shmoys et al., 1991): 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 knowing 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 (5.1), 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 should be split. To minimize the loads from the PFS, we try to copy parent states into s_i different runner caches at maximum. To support this algorithm, the server needs to know the parent state p_i currently propagated by each runner and each runner's cache. The following particle scheduling policy was implemented:

1. If a runner requests a state to be prefetched or propagated, try to schedule the same parent state p_i again (if $\alpha_i > 0$).

2. Otherwise, try in random order to schedule one of the states from the local runner cache (select p_i with $p_i \in$ runner cache, $\alpha_i > 0$).

3. If still no suitable parent state was found, select another parent state p_i with $\alpha_i > 0$ that is in no other runner cache yet and that has a maximal split factor s_i ($\operatorname{argmax}_i\ s_i$).

4. If still no fitting parent state was found, select the parent state with $\alpha_i > 0$ with maximal split factor s_i that fulfills the criterion $s_i > k_i$, where k_i denotes the number of runners that have parent state p_i in their local cache already. This ensures that p_i is not copied to more than s_i runners.

5. If all parent states with $\alpha_i > 0$ are in some runner's local cache and no particle with $s_i > k_i$ is found, select p_i that maximizes the split factor ($\operatorname{argmax}_i\ s_i$). Tests have shown that assimilation cycles finish faster this way due to prefetching, even if it adds some unnecessary PFS loads, violating the invariant $k_i \leq s_i$ in few cases.

After scheduling a new parent state, the server's knowledge about the runner caches is updated and α_i for the scheduled parent state p_i is decreased.

The rules 1 and 2 allow to select a state quickly in $O(\text{local runner cache size})$ steps. To select a state following rules 3 to 5, $O(d)$ split factors need to be tested, where $d < P$ denotes the number of different remaining parent states.

Using optimized data structures to avoid costly recalculation and ordering of split factors, this policy is fast in selecting the next particle for a runner. At the same time, the policy avoids state transfers assuming a reasonable cache eviction strategy (see Section 5.1.5).

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.

5.1.5 Cache Eviction Strategy

The number of particle states that can be kept in the local runner caches is limited by the node storage capacity. Typically, this is set by the available node memory when using the RAM disk for the cache. The helper cores interact with the server to perform particle eviction from the cache when required. The cache needs to provide at least two 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 explained in Section 5.1.3, each time a state has been stored in the cache by the app cores upon the successful propagation, the helper cores 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 helper cores check if the cache can fit the output particle state of the next propagation. If not, one particle state has to be evicted.

When an eviction is required, the server selects the state to evict from the 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 have been 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 5.2) 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 5.1.4.

5.1.6 Fault Tolerance

As for Melissa-DA with EnKF, the framework auto recovers from faults. The server or the launcher may detect runner crashes. If a runner times out, the server will regard

it as crashed, notifying the launcher to restart the runner and scheduling the runner's propagation task elsewhere in the meantime. In some situations, the launcher will detect a runner crash first since the runner job is not listed as up in the supercomputer scheduler anymore. In this case, the launcher restarts the runner directly and notifies the server. The server, again, reschedules the canceled particle propagation elsewhere.

Server faults are detected first by the launcher. A missing heartbeat or a server job that is listed as stopped in the batch scheduler of the supercomputer mark a server failure. In both cases, the launcher stops all remaining runners and restarts the server with a new set of runners. The assimilation will restart from the last server checkpoint recovering all necessary particles from the shared cache (PFS).

Due to missing heartbeats, the server can detect launcher failure. In that case, the server checkpoints all progress and stops. Runners will detect the server missing and stop too. Next, the user can restart Melissa-DA using the checkpoints and the particles stored in the shared cache.

5.1.7 Cache Implementation

Particles are stored locally in runner caches (typically leveraging the node-local RAM disk) or in a shared persistent cache, typically on the PFS. All runner local caches together form a *distributed cache*. The server has complete knowledge of all cache contents. Following the algorithms described in Sections 5.1.4 and 5.1.5, the server selects which runner loads, propagates, or evicts which particle. The runner local caches are further distributed among the different nodes allocated per runner.

The Fault Tolerance Interface (FTI) (Bautista-Gomez et al., 2011) is a multilevel checkpointing/restart library. Its multilevel checkpointing capabilities are leveraged to implement the different cache levels of the particle filter workflow. FTI also provides helper core functionality, originally, to overlap checkpointing with application computations. In Melissa-DA, we extend the FTI helper core functionality to overlap cache access times with propagation and update phase computations.

FTI supports checkpoints in multiple formats. For performance, FTI's binary format is often the best choice. But HDF5 is supported too. This format is readable by many tools that can be used to perform data analysis directly on the checkpoint files. Notice that all particle states are saved to the PFS, and so other state outputs for postmortem analysis purposes, often performed in the simulation code, can be turned off.

5.1.8 Speculative Propagation

To parallelize particle filters, Melissa-DA and many other implementations run multiple particle propagations and weight calculations on different resources (runners) at the same time. During the update phase, the weights of all propagated particles are gathered to perform the resampling. The particles for the next assimilation cycle are drawn (sampled) from the propagated particles of the previous cycle to avoid weight collapse. A resulting particle p_i is expected to be propagated $\hat{w}_{i,t}M$ times, where M denotes the ensemble size and $\hat{w}_{i,t}$ is the normalized particle weight as introduced in Section 2.3.3. The normalized particle weights $\hat{w}_{i,t}$ can only be computed after the full ensemble finished propagation and weighting. Thus, resampling adds a synchronization point to the particle filter workflow, leading to runner idle time.

To increase efficiency, we propose the speculative propagation of particles. Runners that would wait for the remaining particle propagations and resampling to finish, begin to propagate particles for the next propagation phase instead. When finally all particle weights are retrieved, the *final* resampling is performed, and runners receive particles as usual. No further speculative particle propagations are started. Speculative propagations are not interrupted and always run to completion. Some speculated particles may be finally not resampled in the ensemble for the next cycle, making their computation *useless*. To avoid wasting compute hours and blocking runners from doing useful propagations, the number of useless propagations must stay low. Therefore, it is crucial to predict as accurately as possible the number of times each particle is redrawn in the final ensemble.

Our heuristic for this purpose computes preliminary normalized weights from the particle weights available at the point in time where speculative propagations shall start. The normalized weights are then used for resampling. Multinomial resampling, as we use it in most of the following examples, is not appropriate. It draws each particle at random with the probability given by its normalized weight. Executing two times multinomial resampling on the same set of normalized weights produces different samples. Even with perfect knowledge, preliminary resampling would fail to predict the final sample.

Residual resampling (RR) limits the amount of randomly selected particles to the particles drawn from residual weights. Important particles that are *expected* to appear in the sample are deterministically put in it. For that reason, we select RR for the speculative particle filter. If the normalized weights do not change a lot between preliminary and final resampling, many of the important (high weighted) particles are sampled deterministically for speculative and non-speculative propagations.

The scheduling of speculative particle propagations follows the same strategies as described in Section 5.1.4. The server schedules speculative propagations for the next propagation phase whenever no particle propagation of the current propagation phase can be sent to an idle runner.

The modularity of Melissa-DA made it easy to extend the particle filter server, originally introduced in Section 5.1.2, with speculative particle propagation functionality. The speculative mode can be activated or deactivated by the user. The overall performance of speculative propagations in particle filters and to which extent it is influenced by errors in the heuristic for particle selection is experimented in Section 5.2.7.

5.2 Experiments

5.2.1 The WRF Use Case

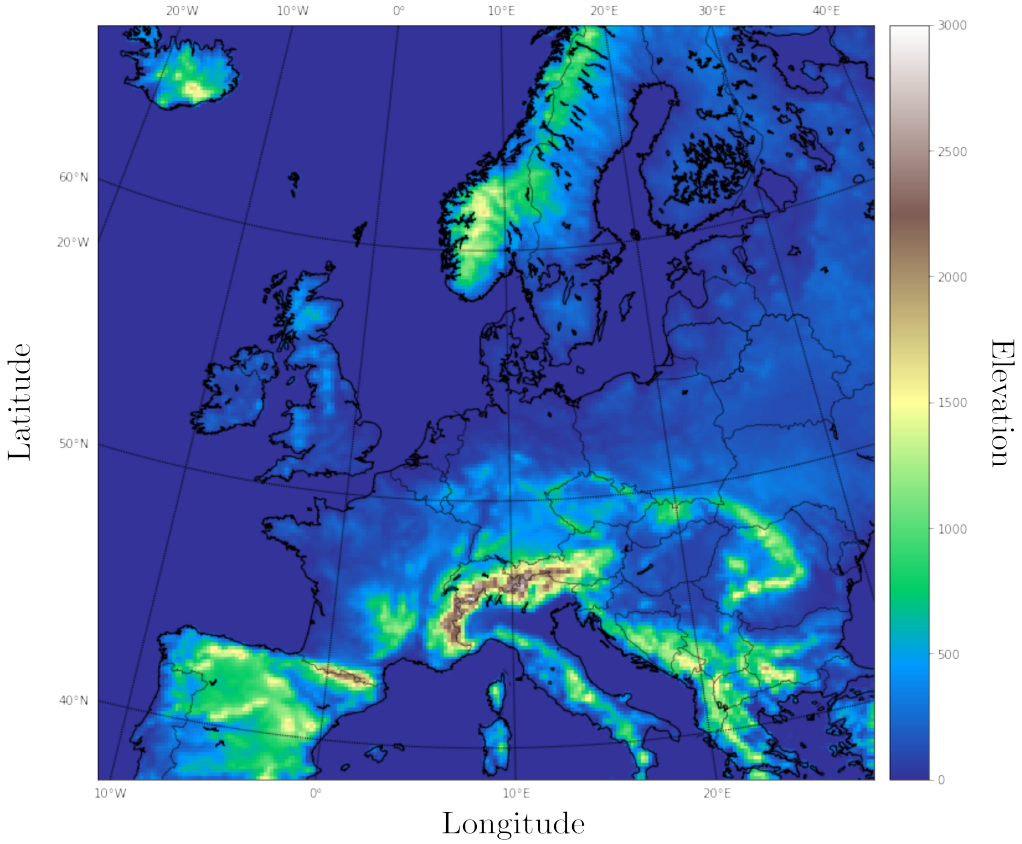


Figure 5.3: The topography of the target domain of Europe for the WRF simulation.

We rely on the Weather Research and Forecasting (WRF) model (Skamarock et al., 2008) for experimentation of our particle filter implementation. Besides being a real-world use case for particle filtering at large scale, it is also used by our EoCoE-2 partners Lu et al. that could provide us with interesting input and validation datasets. WRF is widely employed for weather prediction and reanalysis. WRF provides a solver for the fully compressible non-hydrostatic equations with complete Coriolis and curvature terms while supporting a large set of physics options. The challenge in meteorological modeling using, e.g., WRF emerges from turbulences in atmospheric fluxes producing chaotic behavior and not from non-linear permeability coefficients spanning multiple orders of magnitude as in the hydrological solver that we used before in Section 4.3. The used discretization schemes introduce further numerical errors responsible for chaotic perturbation (Ancell et al., 2018). The complexity of atmospheric modeling does not lower its difficulty in performing skillful prediction in climate modeling and weather forecasting (Goswami,

1996). WRF supports different simulation scales from large eddy simulations (100 m in horizontal resolution), to tropical cyclone (15 km horizontal resolution), and global domain (625 km in x- and 556 km in the y-direction). Benchmark tests on the AMD Epyc 7601 CPU for 12 km and 2.5 km resolution cases over the Continental U.S. domain (CONUS12km, CONUS2.5km) show nearly linear scaling (Kashyap et al., 2019).

We perform simulations on a domain covering most of Europe (See Figure 5.3) using 220×220 grid cells with a horizontal resolution of 15 km and 49 vertical levels with uneven thickness. This is a typical short-range weather forecasting setting. The 2018-07-19 was randomly chosen to simulate 48 hours by 100-second time steps. WSM6 microphysics, MYNN2 boundary layer physics, Grell-3 cumulus parameterization, Eta Monin-Obukhov similarity surface layer processes and the RUC land surface model were used. For more details on simulated clouds and precipitation, non-hydrostatics were employed. Input, initial, and boundary condition data are based on the reanalyzed *ERA5 hourly data* 2019. We assimilate cloud cover fraction measurements (CFRACT) from the EUMETSAT CMSAF satellite dataset by Stengel et al., 2014. The satellite observations are rescaled to match the simulation grid. A total of 48,400 cloud fraction values are assimilated each assimilation cycle. We chose an assimilation window of one hour. One assimilation cycle was set to 36 model time steps ($36 \times 100\,\mathrm{s} \mathrel{\widehat{=}} 1\,\mathrm{h}$) to assimilate all observation data and test our approach under high stress.

A snapshot of the meteorological state of this European domain accounts for 2.5 GiB of data. Writing the full output encompassing velocity components, perturbation potential temperature, cloud fraction, cloud water/ice mixing ratio etc., of the 2,555 particle ensemble for the 48 h simulation period produces almost 300 TiB of data.

If not stated differently, the data presented in the following results are from runs over this European domain with 2,555 particles using 20,442 compute cores on 512 Nodes of the Jean-Zay supercomputer to perform SIR with multinomial resampling. 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.

5.2.2 Runner Activity

Figure 5.4 shows an extract of a particle filter execution trace. The runner local caches are not used during the initial propagation. Up to 69 % of the parent states can be found in the runner local caches for the subsequent propagations. Only 31 % of the

Experimental Setup				
Particles	315	635	1,275	2,555
Number of runners	63	127	255	511
Number of nodes	64	128	256	512
App cores	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 state from PFS to runner cache (s)	2.1	2.1	2.4	4.1
Write state from runner 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 5.1: Experimental setting and performance overview at four different scales. The times are given as average in all cases.

propagated particles account for a PFS load. We will see in the following that most of these loads are performed by the helper cores in the background and thus do not affect the execution time.

As visible in Figure 5.4, multiple runners and propagations take place in parallel. The elasticity of the framework allows runners joining the application to start particle propagation as soon as they get ready. The workload gets distributed (balanced) between all available runners at any time.

After the second assimilation cycle, all runners joined. Each of them propagates 5 particles per assimilation cycle. The used WRF setup shows very even propagation times with only 10 % of maximal fluctuation. Figure 5.5 plots propagation times for one cycle. A single particle propagation takes between 24 and 26.5 seconds. During assimilation cycles, only a few places where runners pause propagation for a moment are visible. These correspond to situations where app cores need to wait for particle prefetching to finish. This is only the case in 6 % of the time.

The close-up view in Figure 5.6 shows the most important tasks performed by the application and helper cores of one runner. App cores are almost completely busy with model propagation and weight calculation. Their activity overlaps very well with the prefetching and particle state storing performed by the helper cores. This puts in light the benefit of the in situ approach. General idle periods are only visible at the end of each assimilation cycle when runners need to wait for the last propagations to finish. Only then the server can resample the next set of particles and distribute new work accordingly.

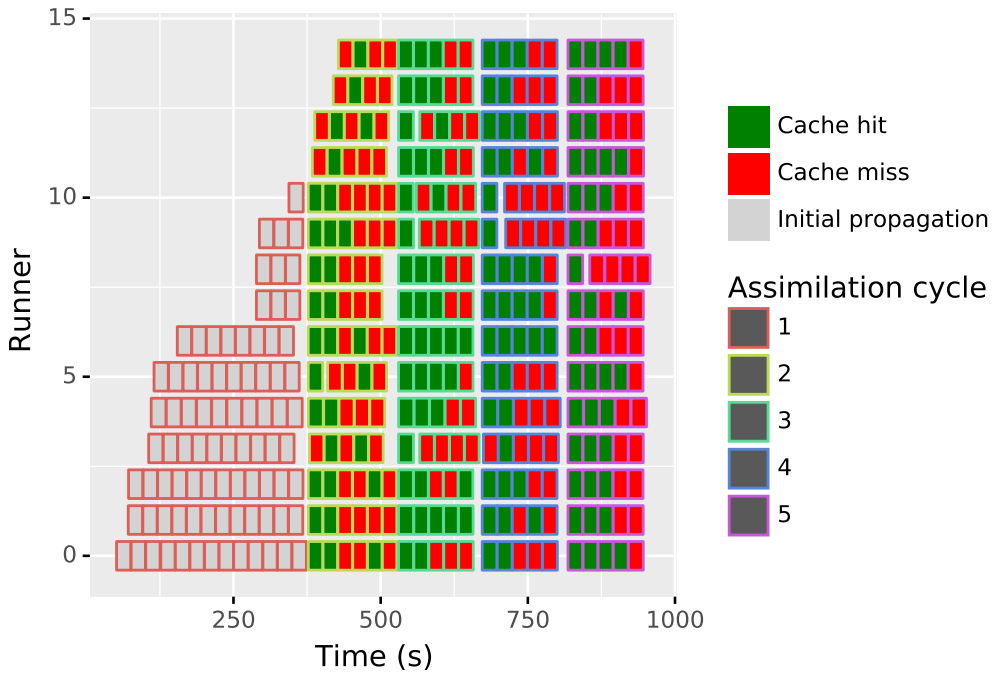


Figure 5.4: Gantt chart of particle propagations executed by 15 (out of 511) randomly selected runners over 5 assimilation cycles. Tasks are green when the associated parent state was already present in the runner cache and did not require a load from the PFS (red otherwise).

The load balancing algorithm keeps app cores busy for 88 % of their time. In 87 % of the time app cores propagate particles, and 1 % of the time is used to weight the propagation result against observations. The remaining 12 % are spent for server communication, including potential waiting time at the end of each cycle (Table 5.1) and the few waits for particle prefetches.

2.7 % of the compute resources are dedicated to helper cores and the server. In 94 % of all propagations, the states are entirely prefetched in the background by the helper cores. Otherwise, state transfers would take 4.1 s + 2.3 s for loading and storing each particle on the PFS. As a particle propagation takes roughly 25 s, this would increase the particle propagation times by 14 % (see the 2,555 particle case in Table 5.1).

5.2.3 Server Activity

The particle filter server is a sequential Python code. Its reactivity can become an issue. Preliminary versions of the server did not rely on optimized data structures to implement the scheduling policy proposed in Section 5.1.4 leading to slow server response times. In Figure 5.7 we show that the current version of the server fixes these issues and runs

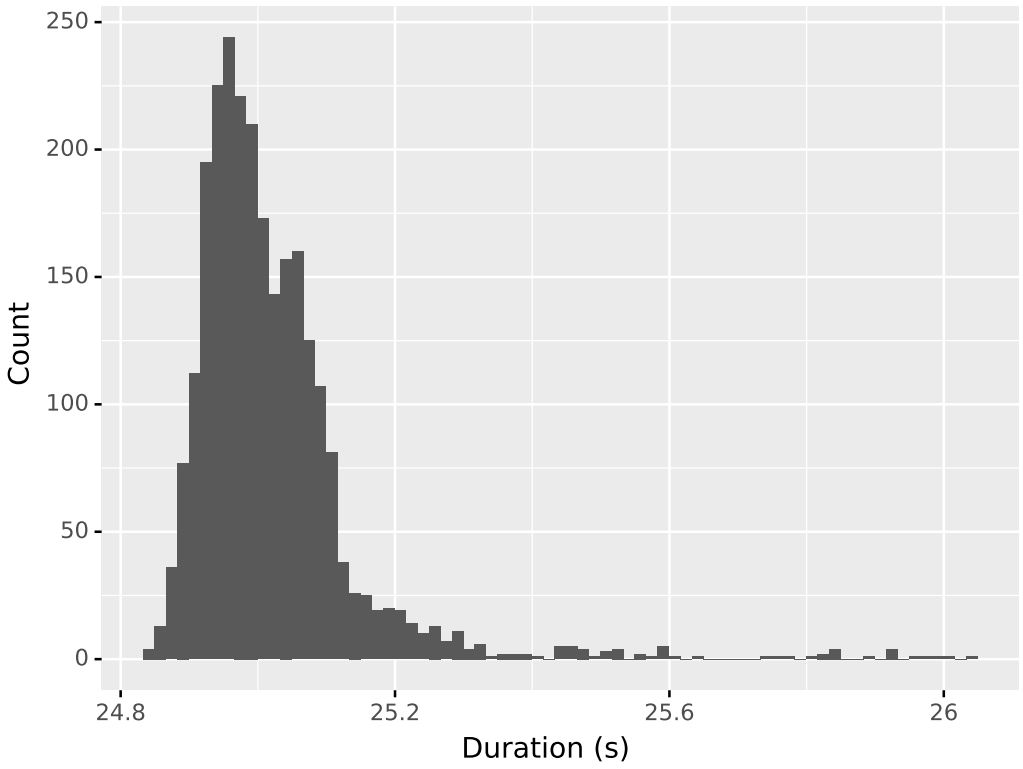


Figure 5.5: Histogram of 2,555 WRF model propagation times of one assimilaton cycle.

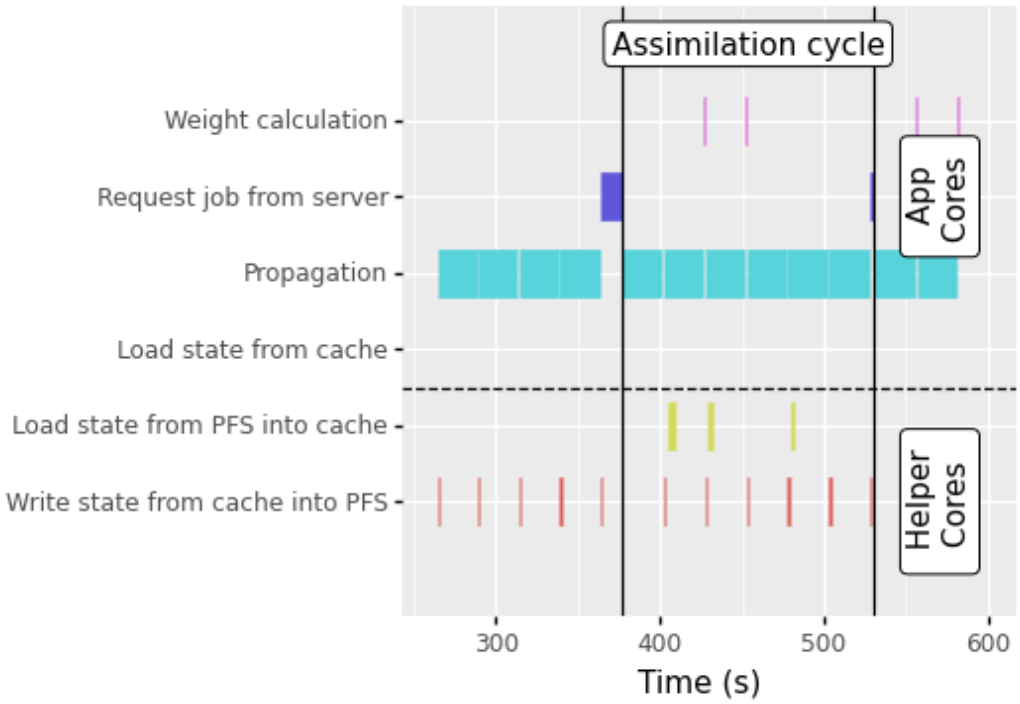


Figure 5.6: Trace detailing the activity of a runner throughout one assimilation cycle. Helper cores enable keeping app cores busy with particle propagation, except at the end of assimilation cycles when they wait for the server to finish particle resampling (dark blue). Some activities are so thin that they are not visible here (state copies from cache to model).

smoothly on the large scale (using 511 runners). The server responds within hundreds of microseconds. Only some job requests take up to seconds. The reason is that the first jobs of a new assimilation cycle can only be scheduled after the server received all weights of the previous cycle to perform weight normalization and resampling. The 511 runners charge the server with 676 requests per second at maximum. Easy optimizations, like adding parallelization, are at reach if the server needs to be accelerated.

5.2.4 State Transfers to/from PFS

Particles representing the European domain leverage 2.5 GiB of data each. The full ensemble of 2,555 particles accounts for about 6.2 TiB of data (Table 5.1).

Each runner locally caches up to five particles. Leveraging 511 runners, the whole ensemble ($2{,}555 = 511 \times 5$) of propagated states could fit in the runner caches. But runners need to store parent states of the previous propagation there too. Thus cache eviction and re-population are necessary. With the given cache size, from 1,024 to 1,563

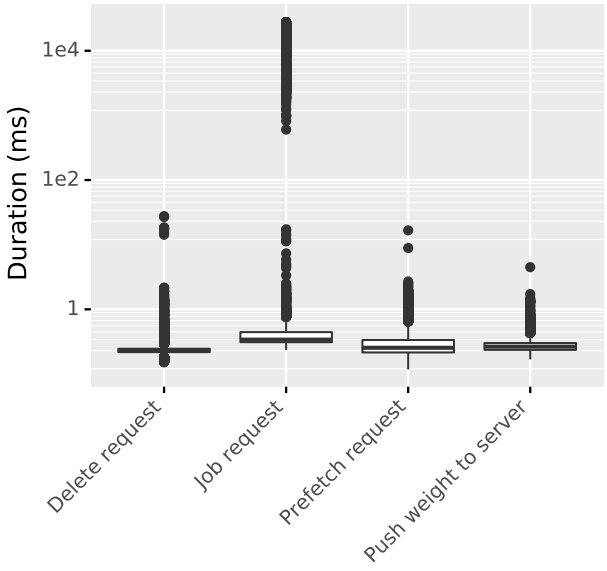


Figure 5.7: Server response times on various runner requests.

particles are loaded from the PFS each assimilation cycle. Between 1,594 and 1,629 different parent particles P were propagated. Thus the derived boundary $P + R - 1$ from Equation (5.2) is always met. Since the runner cache size is larger than 1, our implementation even undercuts the minimal number of P state loads.

Figure 5.8 shows the time to transfer states between runner local caches and the shared cache on the PFS. It is subject to significant variation and increases with the number of runners. While these numbers may be affected by other jobs on the supercomputer, they still suggest that Melissa-DA alone can notably stress the PFS already. However, the framework could completely overlap PFS access time with particle propagation on the tested machines Jean-Zay and JUWELS, as shown in Section 5.2.2.

Applications with propagation times faster than state loads would be impacted by PFS access. Notice that performing hourly resampling already leads to short propagation times in the WRF context. The motivation is to stress the framework. WRF production runs usually do not require such a high resampling frequency. Online data compression could be used to mitigate this issue. Alternatively, node-local persistent storage (SSD or NVRAM) could replace the PFS in the future, for instance, leveraging the functionality of GekkoFS by Vef et al., 2020, to create a distributed file system on top of node-local storage.

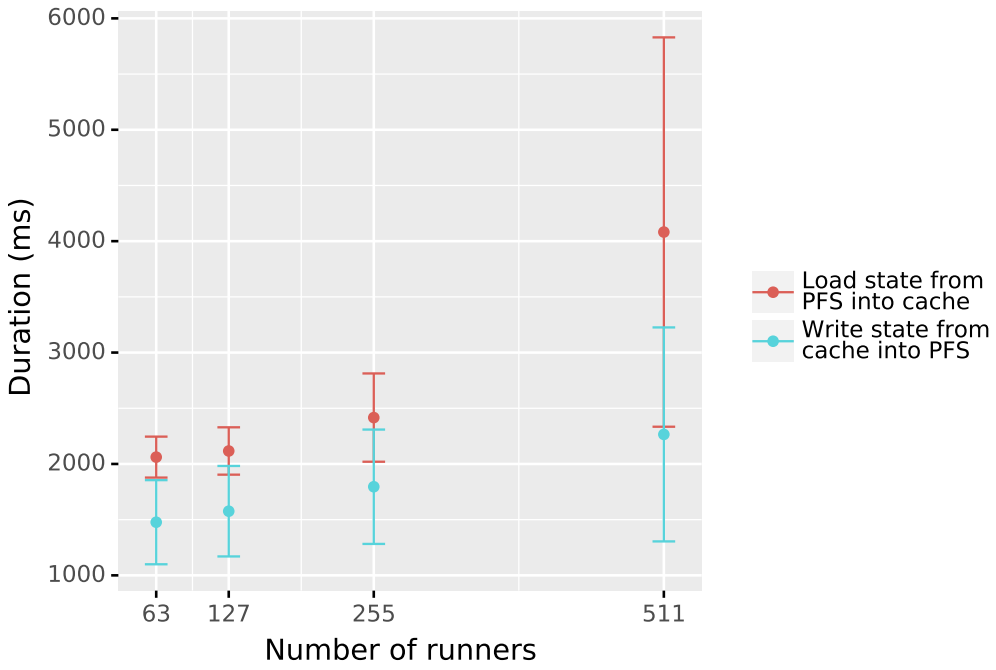


Figure 5.8: Mean time to load or store particle states of 2.5 GiB from/to the PFS with different numbers of runners.

5.2.5 Fault Tolerance, Elasticity and Load Balancing

To show fault tolerance and elasticity, 2 out of 63 runners were crashed in a run with 315 particles. The gantt chart of the run is shown in Figure 5.9. The crash of runner 53 leads to a large idle period. It crashed while performing one of the last particle propagations of the fourth assimilation cycle. All runners have to wait until the propagation timed out (the timeout was set to 60 s), runner 53 is acknowledged as unresponsive, and another runner (runner 44) takes over the missing particle propagation to finish the current cycle. Starting with the fifth assimilation cycle, runner 64 replaces the crashed runner. The second crash leads to less idle time since the dynamic load balancing works efficiently. The work stays well distributed among all runners, even after their number changed.

The particle propagation times are relatively even with at most a 10 % variability. Settings with more variability are possible when relying on different physics in WRF or with other simulation codes. It might also lead to more variability in the execution times if runners execute on heterogeneous resources. Some runners might propagate faster than others by leveraging GPUs, for instance. Testing in such contexts is part of future work.

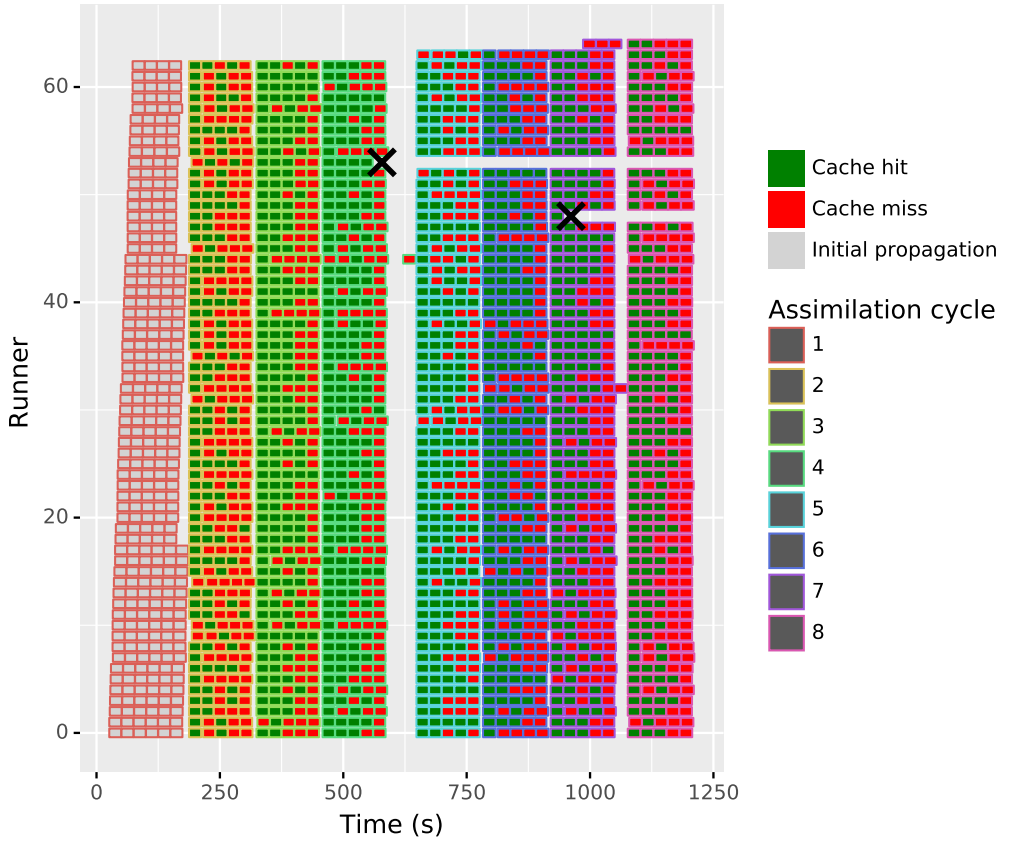


Figure 5.9: Gantt chart as in Figure 5.4. Two runners crashed (black cross) and 2 restarted (top 2 runners).

5.2.6 Scaling

With 63 runners, the framework reaches a strong scaling efficiency above 90 % compared to the case where one runner would execute all propagations sequentially (Figure 5.10). On average, at least 5 particles are executed per runner and assimilation cycle. Prefetching enables hiding the I/O costs. Thus, increasing the number of particles propagated by each runner allows to better amortize the cost of the synchronization during the resampling phase and positively influences the scaling efficiency.

Keeping the particle load constant at 5 particles per runner, a weak scaling study is performed (see Figure 5.11). The time of an assimilation cycle increases by 8 % from 52 to 511 runners. These good results suggest that Melissa-DA would also run efficiently at larger scales with more than 2,555 particles and 511 runners, but we did not manage to get access to allocations of more than 20,442 cores.

Running particle filters with WRF on a European domain for short-range weather prediction at this scale is an important advancement of the previous work done by

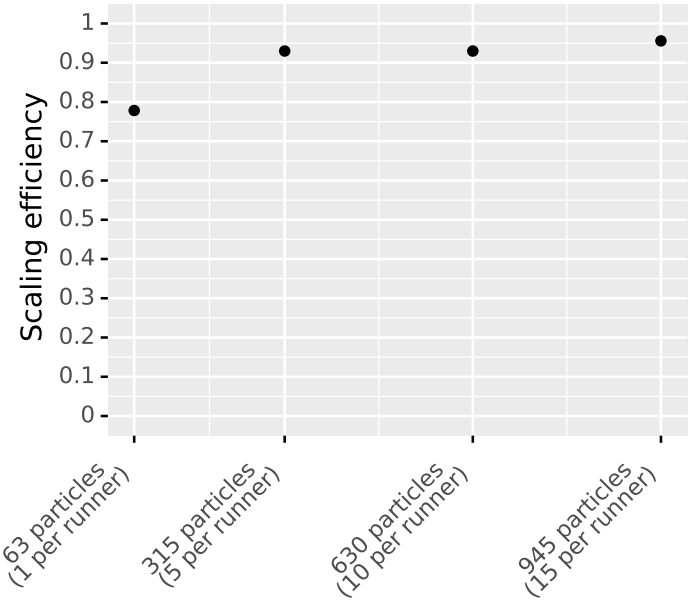


Figure 5.10: Strong scaling efficiency using different numbers of particles with 63 runners. One runner sets the reference case.

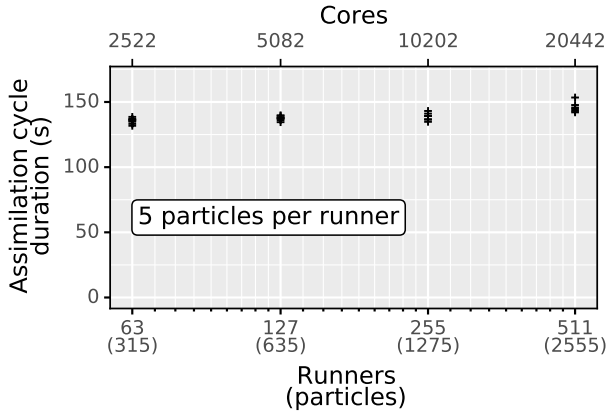


Figure 5.11: Weak scaling performance test: assimilation cycle duration for different numbers of runners, but always 5 particles per runner.

Berndt, 2018. Besides assimilating at a higher frequency, the proposed framework offers fault tolerance, automatic load-balancing, and elasticity while minimizing the file I/O and the time to calculate weights.

	Dummy use case	WRF use case
Particles	15	315
Particle state size	1 MiB	2.5 GiB
Runners	4	63
Prop. exec. time mean	0.484 s	22.373 s
Prop. exec. time std	0.297 s	1.503 s
Prop. exec. time min	0.001 s	18.976 s
Prop. exec. time max	0.996 s	24.925 s

Table 5.2: Statistical description of propagation runtimes for experiments traced in Figure 5.12 and Figure 5.13.

5.2.7 Speculative Propagation

The most important remaining source of inefficiency is at the end of each propagation phase. Runners finishing particle propagation earlier than others need to wait for the slowest propagation before resampling can start and new particles to propagate are distributed. At large scale, this may produce significant idle times. In the run depicted in Figure 5.4, the average idle time caused by the update phase, including waiting for the latest propagation and resampling until particle propagations start again, accounts for 13.5 s per runner. Note that a full assimilation cycle is only about 10 times as long (146 s on average, see Table 5.1). This idle time could be much shorter. The runner performing the last propagation only waits 1.4 s, accumulating weight and state transfers, resampling, and redistribution of propagation tasks. This gives an approximate lower bound of idle time for each runner if no load imbalance would occur between runners. The waiting time at the end of each propagation phase is responsible for 85 % of the total runner idle times. We experiment here with the speculative execution of particles for the next assimilation cycle. The implementation follows the description in Section 5.1.8. Residual resampling (RR) is used during the update phase, and speculative propagations are scheduled to runners following the policy introduced in Section 5.1.4.

Experiments on two settings were performed. First, a "dummy" model with a small state size of 1 MiB only, 15 particles, and with randomly chosen propagation times, uniformly distributed between 0.001 s - 1.0 s was assimilated. Second, the WRF model, like in Section 5.2, with 63 runners, propagating 315 members, has been tested with speculative particle propagations activated. The local cache on each runner was set to store up to 5 particles for both cases.

Figure 5.12 shows traces of two runs of the dummy model are shown. Runs without and with speculative particle propagation are shown side-by-side. As propagation times fluctuate a lot in this case (Table 5.2, $\frac{\text{std}}{\text{mean}} = 0.61$), and only few (4-5) members per

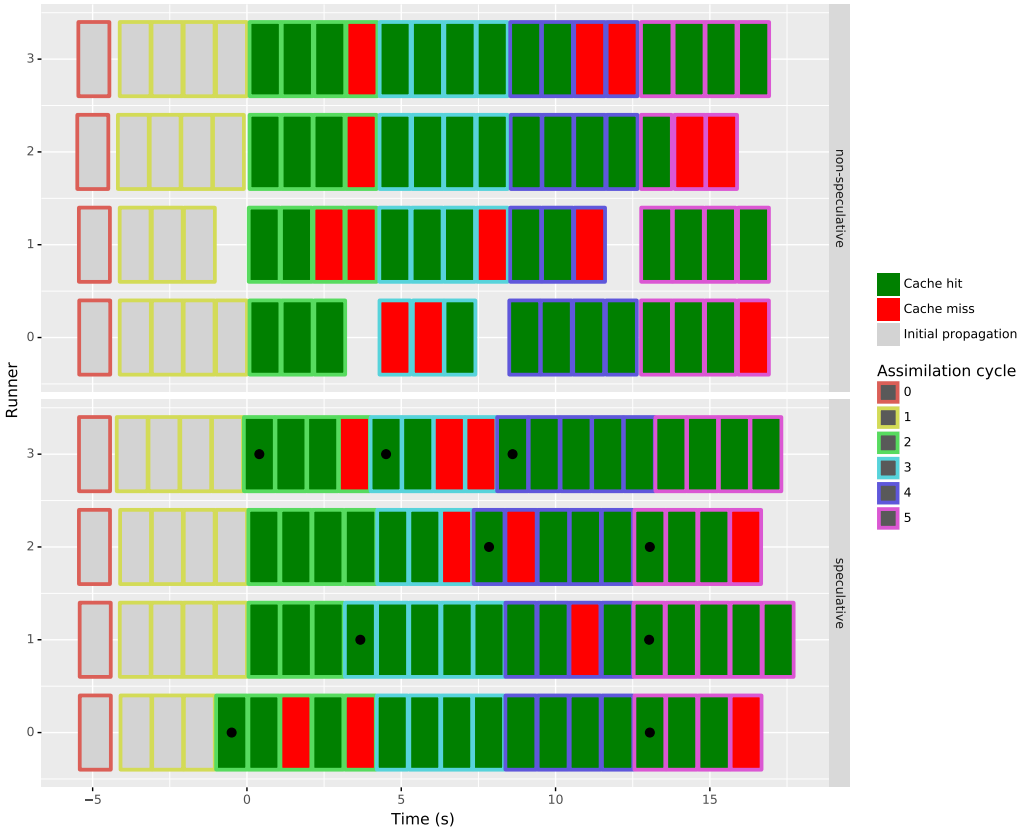


Figure 5.12: Non-speculative versus speculative scheduling on the dummy use case. Speculative propagations are marked with a black point.

runner are propagated, the introduced load imbalance at the end of each assimilation cycle is significant. Runners are idle 28 % of their time. Activating speculative scheduling reduces the idle time to 12 %. But, at the same time, between 20 % and 75 % of the speculatively executed particles turn out to be useless each assimilation cycle. Thus, 14 % more particles are propagated each cycle in the speculative case. Even if their propagation overlaps well with some of the idle time of the previous case, the speculative run cannot finish faster. The two cases have roughly the same runtime.

Regarding the WRF use case, execution time of the particle propagations is much more even than in the dummy case (Table 5.2, $\frac{\text{std}}{\text{mean}} = 0.07$). The smaller fluctuation among the different propagation tasks results in a workload better equilibrated between the different runners. The traces of a run with and without speculative scheduling are shown side-by-side in Figure 5.13. As already described in Section 4.3.2, startup times of Melissa-DA runs can vary a lot. Thus, the time axes were set to begin just when the last particle propagation of assimilation cycle 1 terminates. This way, the total runtimes of the following cycles are easier to compare visually. Again, between 30 % and 75 % of the speculatively propagated particles turn out to be useless. Thus, on average 12 % more particles are propagated each cycle compared to the non-speculative case, where each cycle runs exactly 315 propagations. The overlap with speculative propagations reduces the runner idle time from 8.5 % to 4.1 %.

Still, as for the dummy use case, this configuration for speculative propagations has no positive impact on the overall runtime. Many runners have to perform more than the 5 propagations that were necessary for the non-speculative case in each propagation phase to compensate for useless speculative propagations. Useless propagations that started just before the final resampling are responsible for longer runtimes. Only at the beginning, they do overlap idle time present in the non-speculative case, but later they delay the execution of useful work since there is no way to stop them beforehand. If some runners would propagate only 4 particles per cycle because of useful choices for speculative particles, this could equilibrate the introduced delay. Unfortunately, as visible in Figure 5.14, this is not the case. Too many of the speculatively selected particles are finally not useful, and in consequence, further particles need to be propagated.

The experiments show that the number of speculative but useless particle propagations is too high to make speculative propagation competitive in the dummy and WRF use cases. Nevertheless, these results are assumed to be very sensitive to the used setting, regarding, e.g., duration of particle propagations, weight calculation, or the particle to runner ratio. In the following, we discuss how changing the used setting could improve the performance of the speculative approach.

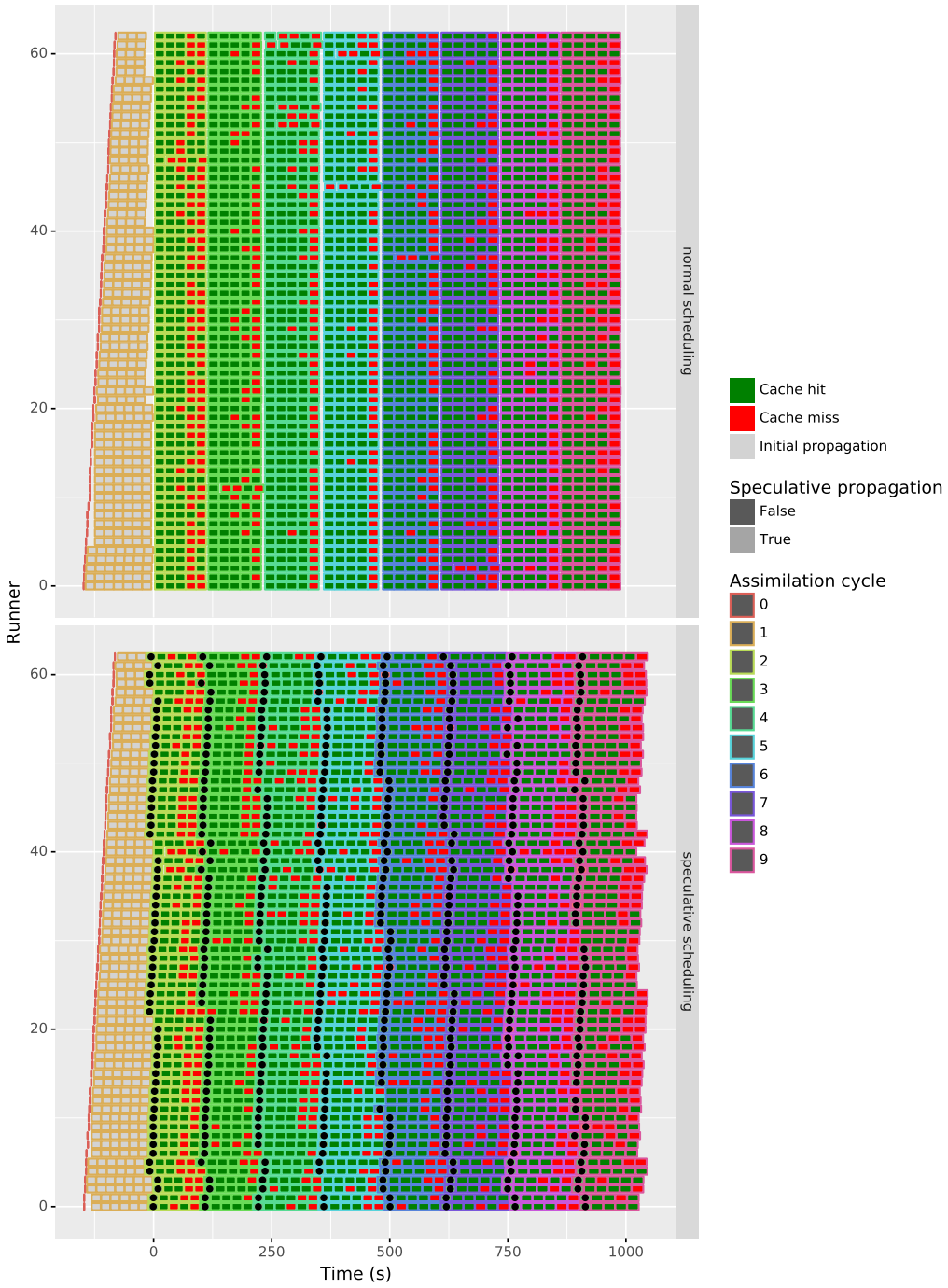


Figure 5.13: Non-speculative versus speculative scheduling on the WRF use case. Speculative propagations are marked with a black point.

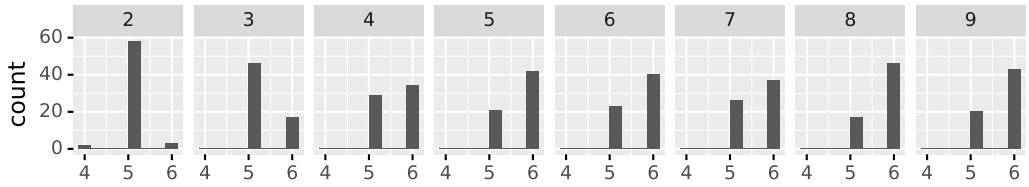


Figure 5.14: Propagations done per runner each assimilation cycle according to the trace in Figure 5.13.

One important aspect influencing the performance of our speculative particle filter is the condition under which speculative propagations are started. As described in Section 5.1.8, the server distributes propagation tasks for the next cycle speculatively as soon as no particle of the current cycle is left. But, as seen, for instance, in Figure 5.13, only the first speculative propagations overlap idle times well. Speculative propagations that were started later and turn out useless, in contrast, first overlap but then delay useful work. Thus, scheduling only the first speculative propagations that insert no delay might be a viable option. To diminish the number of useless particles further, one might only schedule very high-weighted particles speculatively, at the cost of more PFS loads. But this adds parameters to tune, which we did not investigate yet. The number of allowed speculative particles, and how often to ignore cache locality when scheduling them needs to be specified. We expect both parameters to be use case-specific.

Alternatively, one could improve the heuristic to predict the resampling, to reduce the number of useless speculatively propagated particles. As we mentioned in Section 5.1.8, the chosen resampling strategy may impact this. As a first guess, we use RR in our experiments. Nevertheless, resampling methods that are entirely deterministic and "stable", not changing much the resampled particles when only a few new weighted particles are added, might lead to further improvement. An existing resampling technique for this might be residual systematic resampling (RSR) as presented for example by Bolic et al., 2003. RSR works similarly to RR, but the remaining particles are selected depending on one random draw only. Compared to RR, where each particle of the remainder part is drawn randomly, RSR would reduce the fluctuation in the remainder part of the ensemble.

In each propagation phase, only the last particles per runner are executed speculatively for the next assimilation cycle. Putting more particles on each runner may thus decrease the overall ratio of speculative particle propagations to non-speculative ones, improving preliminary resamplings (of speculative particles) that now rely on a larger percentage of the ensemble's particles.

Another approach to improve the performance of speculative particle filters would be to recycle 'useless' particles, for example, to enrich the sample resolution of the analysis state PDF. For the latter, one could perform simple weight correction to include useless particles in the analysis state PDF following Equation (2.27). Particle weights from particles that were propagated too often need to be scaled down. Let $w_{j,t+1}$ be the weight of particle p_j where $p_j \in J$, with J being the set of child particles from parent particle p_i. As in Section 5.1.4, α_i denotes the number of times parent p_i was sampled, i.e., the intended count of its children ($\alpha_i \leq |J|$).

The child particles J were generated by independent propagations of the parent particle p_i. The corrected weights $w'_{j,t+1}$ can be computed by

$$w'_{j,t+1} = \frac{\alpha_i}{|J|} w_{j,t}. \tag{5.3}$$

After normalization according to Equation (2.26), they can be used to express the analysis state PDF $P(x_a)$.

All these strategies could be experimented changing only some parts of the particle filter server source code. Their exploration is left for future investigation. Examining the effect of a combination of these, calibrated for the DA problem to solve could be especially useful.

5.3 Conclusion

In this chapter, we presented an extension of the Melissa-DA framework that allows to run particle filters at a large scale in both, the count of particles and the number of state dimensions. While EnKF and many other ensemble-based DA methods shift member state vectors towards observations and therefore must centralize all member states during the update phase, the particle filter does not. The particle filter update phase changes only the composition of the full ensemble (resampling) to hinder weight collapse. This avoids heavy data exchange with a central server and reduces massively its memory requirements. Instead, a multilevel distributed cache architecture leveraging runner local RAM and the shared parallel file system is used to exchange particle states. State transfers rely on an in situ workflow to overlap with useful computations.

Note that leveraging the PFS might become a bottleneck, but this was not the case for the tested DA problems run on the JUWELS and Jean-Zay supercomputers. Whether a direct runner-to-runner state exchange is competitive is to be explored in the future. But relying on the PFS for particle storage permits the total size of all particle states to exceed the total RAM of all used runner resources.

The particle filter update phase performing resampling is less compute intense than, e.g., an EnKF state update. This reduces the runner idle time during the update phase. Due to the reduced computation effort and memory footprint, fewer resources are necessary for the server.

The framework could easily scale with 87 % efficiency to 2,555 particles executed on 20,442 compute cores. The remaining idle time comes mainly from runners that await the last particle propagations to finish before resampling can be done, and new propagation tasks for the next assimilation cycle are distributed. To circumvent this, we extended our framework to experiment with the possibility of speculative particle propagations. While decreasing the idle time of the workflow, this does not increase its performance for the tested configurations as more particles need to be propagated. We discussed possible adaptations to make speculative propagations efficient, but their validation is left to future investigations.

6 Reproducible HPC Experimentation – a Case Study

This Ph.D. work exposed me to the difficult problem of HPC experiments, how to drive and how to organize them to provide meaningful results and to enable as much reproducibility as possible. Here I present the methodology that I developed during my Ph.D. work. This methodology results from a mix of influences reaching back to the year 2016 when I had my first contact with research involving supercomputers, studying power dissipation of Intel CPUs for an internship at Regionales Rechenzentrum Erlangen (RRZE) up to the current day. Great influence had also the online course "Reproducible research: methodological principles for transparent science"[1] that I studied in 2020. It is not per se a scientific result, but I believe that sharing my experience and findings on that topic can be useful to others.

[1] `https://www.fun-mooc.fr/en/courses/reproducible-research-methodological-principles-transparent-scie`, retrieved the 01.04.2022

6.1 Motivation of Reproducible Research

Reproducibility is essential for comprehensible, transparent, and credible research. It is key to enable others and yourself to retrace the exact track of ideas and experiments that converged into hypotheses (Alston and Rick, 2021; Desquilbet et al., 2019). In the Information Age, where ideas, inputs, and algorithms may be stored digitally, and copying them is very cheap and highly efficient, research and especially the part of it running on information systems, should aim to be efficiently reproducible. Reinstalling and rerunning experiments and postprocessing pipelines up to chart generation should necessitate only a few manual steps.

Existing tooling helps to facilitate reproducibility. Digital notebook solutions like Jupyter notebooks[2], R-Studio notebooks[3], or org-mode[4] allow to display postprocessing and data analysis side-by-side with the research journal containing proofs, literature research and experiment documetation. Version control systems like Git[5] or subversion[6] allow to keep track and easily switch between different versions of source files, experimental setups, their input files, and postprocessing pipelines.

After introducing the special challenges concerning reproducibility in HPC experimentation in Section 6.2, we will detail the tools and methods key to enable reproducibility in our work in Section 6.3. In the last section of this chapter, Section 6.4, we conclude which of the challenges were solved using the introduced technologies and methods and which ones stay open issues. We further discuss how difficult reproducibility of some of our research effectively is.

[2] `https://jupyter.org`, retrieved the 01.04.2022
[3] `https://www.rstudio.com/blog/r-notebooks`, retrieved the 01.04.2022
[4] `https://orgmode.org`, retrieved the 01.04.2022
[5] `https://git-scm.com`, retrieved the 04.01.2022
[6] `https://subversion.apache.org`, retrieved the 04.01.2022

6.2 Challenges of Reproducibility in HPC Experimentation

Various computer hardware vendors produce a variety of different hardware used in HPC. Diverse processors, memory and network interfaces on different chipsets are assembled alone for the HPC machines listed on the *TOP500* 2022. Many HPC experiments are sensitive to the exact hardware configuration used. Heterogeneity within upcoming exascale machines is expected to increase, as stated by O'Brien et al., 2017 and Schulte et al., 2015. This will make it very important to capture the exact configuration of machines and the part of them that is used as a testbed for experiments. Only this way measurements are transparent, can be interpreted correctly, and possibly be reproduced.

Multiple users run their jobs on those systems concurrently and some resources like network and file systems are shared. As soon as those resources become saturated, slowdown can be observed (Skinner and Kramer, 2005; Freed et al., 2015). For instance, can the saturation of the network interface lead to race conditions and unpredictable delays of messages (Figure 4.13). Performance measurements on HPC are sensitive to the overall workload of a machine. Running multiple times the same experiments at moments where a machine is differently charged is crucial to obtain statistically validated results.

But, the statistical validation of results is challenging when it comes to research concerning the large scale, examining effects that are not visible on smaller runs. Single runs can take thousands to tens of thousands of core compute hours. For this work, for instance, more than 957,000 core compute hours on super computers were used (Appendix A) with largest runs consuming 60,000 core compute hours at once. Besides being extremely costly, repeating experiments at this scale quickly has non-negligible environmental impact. Researchers must carefully evaluate the trade-off between invested resources and the importance of possible results.

HPC experimentation is also sensitive to the used software stack, its version and its configuration. The installed operating system, as well as drivers, libraries, compilers, and compiler flags, can influence experiments or the behavior of postprocessing pipelines. Using drivers and compilers tailored to the installed hardware often brings performance improvements. Intel compilers, for instance, enable some optimizations to accelerate the execution of compiled code on Intel hardware. But such specialized software is often under restrictive licenses, limiting its availability or the availability of a specific version on many supercomputers.

While users of HPC facilities cannot influence the used operating system and installed drivers, they can install libraries, and sometimes also compilers manually in userspace. As software packages rely typically on complicated dependency hierarchies, it is a serious effort to manually compile and install them together with userspace applications necessary for experimentation. Installing necessary dependencies or selecting from different preinstalled versions available on many supercomputers is time-consuming, as different configurations need to be carefully tested for performance and inter-compatibility with other dependencies. Additionally, software updates on supercomputers require ongoing maintenance efforts to adapt to newly installed and removed versions of dependencies. We estimate that the installation of the Melissa-DA software stack on the two supercomputers Jean-Zay and JUWELS account for at least several hundred person-hours.

Reproducing large scale runs often leads to difficulties from a different direction too. Their input data, e.g., the boundary datasets of meteorological simulation, can easily account for hundreds of gigabytes to several terabytes. The same is true for the produced raw output data. Storing it for backups and public access is thus costly. Additionally, input data often comes from third-party data mining without permissive licensing. We experimented, for example, with the UMETSAT CMSAF satellite data[7], which is not freely accessible. While researchers have full control over the copyright of self-written source codes for experiment configuration and postprocessing, guaranteeing public access to terabytes of input and output data, possibly licensed by a third party, is very challenging. Raw output data of the experiments in this thesis can easily reach multiple terabytes of data. As visible in Table 5.1, storing for example 2,555 particle states of the European domain would produce 6.2 TiB per assimilation cycle and is therefore avoided in all cases. Only input and raw output data necessary to rerun experiments or finally used in analysis and visualization are archived. Note that this also contains information on the exact experiment setup, including used software and hardware, which is useful for reproduction. This alone accounts for at least 28 GiB of data in our case.

[7] `https://www.eumetsat.int`, retrieved the 01.04.2022, see also Stengel et al., 2014

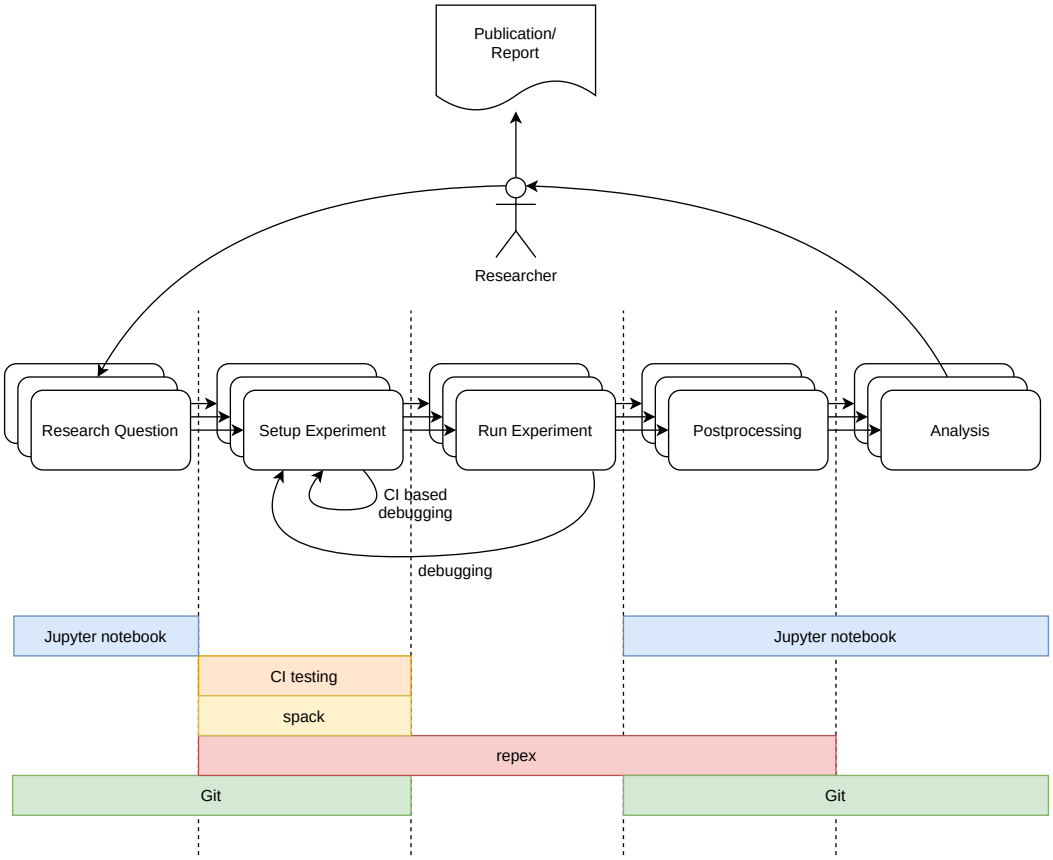


Figure 6.1: Experimentation workflow and used technologies in the context of this work.

6.3 Our Approach to Reproducible HPC Research

Many solutions to reproducible research exist. The ones adapted to the kind of research and experiments that shall be driven need to be carefully selected. Different aspects can be of importance. One might be interested in the data-heavy outputs of large scale simulation in one case. In contrast, during this work, we run performance measurements. Metadata of executions, i.e., the time it takes to write the output to disk and not the written output itself, is of interest. In the following, we discuss different key technologies that we used for reproducible research, starting with an overview and then going into more details about the different technologies.

6.3.1 Overview

Our experimentation workflow is shown in Figure 6.1. First, a research question is posed and noted down in the lab notebook (Jupyter notebooks in our case). Possible answers

are formulated as hypotheses to be tested. Experiments to falsify the hypothesis are set up. This may involve code changes. Continuous integration testing gives confidence that introduced changes do not break functionality and run on multiple architectures.

To perform large scale experiments deployed on HPC machines, software dependencies are handled by the Spack package manager in our case. The code, subject of experimentation, is installed manually since the experimental versions of it are not yet deployed on Spack.

Our own library, repex, is set up to manage input and output data of the experiment. Repex also stores software and hardware configurations necessary for reproducibility. As repex facilitates running multiple experiments concurrently, it is advisable to keep track of different running experiments in the lab notebook. Repex runs each experiment in its own folder on the scratch partition. While leveraging the fastest shared file system of the supercomputer, the scratch partition is often not persistent. For example, on JUWELS data gets cleared after 90 days from there.

While running the experiments, bugs in their setup that require code or configuration changes can appear. In this case, the experiments must be rerun. Next, after experiments run successfully, the postprocessing pipeline in the lab notebook is started. It downloads interesting parts of the results to the local machine. This includes information to transparently reproduce the experiments, as well as performance measurements. For our case, large binary input and output files are omitted. The raw data archived per run is in the order of several hundred megabytes for the experiments presented here. This contains traces of the run, logs, and information necessary for reproducibility. Since this data is now locally on the researcher's machine, its persistence is secured by regular full disk backups. Saving results of interest locally in combination with the automated file removal in the scratch partition make further data clean-ups unnecessary.

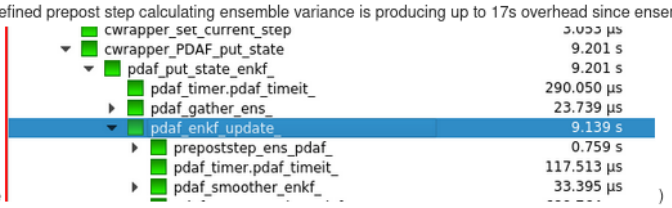
Any source code, be it to prepare, set up, or start runs, and the lab notebook containing postprocessing instructions are checked into Git repositories to track changes in these iteratively developing artifacts. In contrast, neither the result and input data stored using repex nor large binary for input or output files are checked into any version control system. Their content will not change. This data is only kept for archiving and as input of postprocessing making the ability to track different versions unnecessary.

Next, statistics, charts and tables for analysis of the results are generated within the lab notebook using the downloaded raw data. The researcher interprets generated analysis, leading to analysis refinements, the next research question, a report, or a publication.

Markdown Python 3

Influence of the prepost step

Further investigation using vampir and scorep traces on 16 members (traces when running with 32 members were corrupted all the time) Have shown that the user defined prepost step calculating ensemble variance is producing up to 17s overhead since ensemble variance is not needed here. (see)



Deactivating this code improves a lot the performance. The same function takes less than a second for 16 ensemble members now. Thus we are running the case with 1024 ensemble members now again to see if we become faster. Another easy to do improvement would be to run without Debug options. (just checked: ParFlow's timing flag works in this case as well)

[...]

One cycle takes about 4 seconds -> 4 * 48 = 192 $\frac{165}{192} \approx 0.86$

[...]

	Run name	Members	Server nodes	Server_Checkpointing	isPDAF	Time per cycle in s	Update times per cycle	Cycle time - Update time in s	max idle time in 2nd iteration	time per PFB file write	...	Runners
89	20211209_185734-test-r1i0n13	500	1	True	False	23.077614	3.002502	20.075112	2.539358	NaN	...	100
90	20211210_002636-test-r1i6n14	500	4	True	False	19.843631	0.993798	18.849833	0.463174	NaN	...	100

2 rows × 21 columns

[ ]:
```
def plot_trace_flexible_two(di1, di2):
    al = []
    ...
    df = pd.concat(al)
    p = ggplot(df[df['region'] != 'Iteration'])
    p += geom_rect(aes(xmin='start_time_s', xmax='end_time_s', ymin='runner_id-.4',
                                ymax='runner_id+.4', fill='factor(iteration)'), color= 'black', size=.1)
    ...
    return p

p = plot_trace_flexible_two('20211209_185734-test-r1i0n13', '20211210_002636-test-r1i6n14')
p + xlim(330, 378)
```

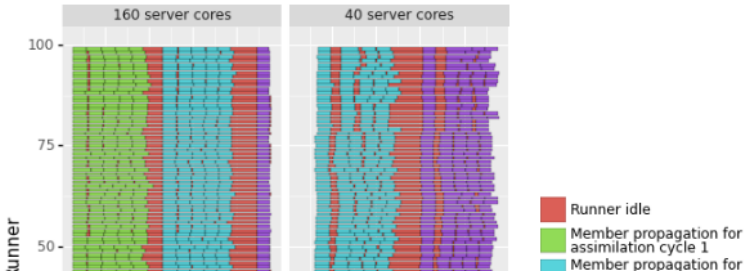


Figure 6.2: Example snippet from a Jupyter lab notebook. Formatted text, math formulas, external image data, tables, code and charts can be mixed.

6.3.2 Lab Notebooks

A well written comprehensible lab notebook is central to reproducible research as it keeps track of the ideas, experiments, and conclusions that were driven to get some results. Research questions are written there and possible answers are formulated as hypotheses that shall be validated through literature research, proofs or experimentation. In the scope of this research work, Jupyter notebooks with Python3 are used. While they provide an easier entry than, e.g., emacs' org-mode, they also enable to note down mathematical proofs, thanks to the supported LaTeX interface. Results from literature can be added as images and hyperlinks to files and websites can be included too. Also, experiments and postprocessing that do not require heavy computations can be performed directly from the lab notebook using the integrated Python or shell interpreter. As parts of the developed framework are written in Python too, code sharing between the framework and some postprocessing pipelines were possible, for instance, to reuse data structures that write and later interpret logs. Figure 6.2 displays some of the key features of Jupyter lab notebooks like text and math formatting, as well as in-line table or chart visualization and source code editing used for this work.

The nature of this work necessitates experiments executed at large scale HPC machines. Although many supercomputers make it possible to directly run Jupyter notebooks, we preferred to run our notebooks on a personal computer instead. This allows accessing our lab notebooks offline, for instance, while the supercomputer is unavailable during maintenance phases and shows better reactivity, especially when using interactive plots.

It is very difficult to entirely set up HPC experiments from within a lab notebook since often this involves compiling, copying, and changing many files on the supercomputer. We thus launched all experiments manually by connecting to the supercomputer, setting up the experiment, and launching the code there. An up to date list of currently running experiments was kept in the local lab notebook to track all running experiments. In contrast, postprocessing pipelines expressed within Jupyter notebooks are automated. The download and parsing of the results up to the generation of tables, charts, and statistics, are launched by a single function call, parametrized by the folder name of the experiment only. Such pipelines can easily be applied at any time to new experimental data to compare results. To avoid the redownload or reparsing, which can take several seconds to minutes on local computers, caching strategies are used. For instance, all results are downloaded via differential rsync[8] folder synchronization – thus never

[8] `https://rsync.samba.org`, retrieved the 16.02.2022

downloading existing experiment data twice. To avoid reparsing, Python's memoization techniques are used[9].

6.3.3 Capturing the Environment

As mentioned in Section 6.2, capturing the precise experimental environment is crucial for both understanding experiment results in the right context and permitting reproducibility. How this can be automated and how an environment is stored alongside other experiment inputs and outputs is discussed here. Different workflow-oriented tools like JUBE (Lührs et al., 2016), Cylc (Oliver et al., 2019) and ecFlow (*ecFlow* 2022) exist to capture the experimental environment. But, to keep the dependencies slim and avoid the installation on multiple machines and integration of existing tools in the experimental workflow, in the scope of this thesis, a minimalist library called *repex*[10] (Reproducible Experiments) was developed for experimental environment capturing. Repex is a single file library, which makes it easy to deploy. It can be called from Python3 or bash and takes a command executing the experiment, a list of input files, Git source repositories and CMake build directories as arguments. The states of all user-defined input files like start scripts and parameter files, build configurations and Git repositories are captured. Repex further captures operating system and kernel versions, some hardware information, used nodes, environment variables, and loaded environment modules[11]. Environment modules are used to separate different software stacks on supercomputers.

To make running multiple experiments in parallel as convenient as possible and to keep track of all experiments started through repex, each experiment will be run in its own folder containing a timestamp and some user-defined caption in its name. Thus, the folder name acts as a unique identifier for each experiment. The folder then contains all experiment inputs, the capture of the experiment's environment, including the Git revision numbers of used dependencies, and also the output. In the lab notebook, it is sufficient to call previously defined postprocessing pipelines on this folder name to inspect the experimental output. The output of the postprocessing typically leads to some discussion text and some questions that demand further experimentation, the cycle shown in Figure 6.1 restarts and thanks to repex, the output of the next experiment will be properly packaged in its own folder, conserving side-by-side the inputs and outputs of all runs.

[9] see `https://docs.python.org/3/library/functools.html#functools.cache`, retrieved the 04.01.2022
[10] `https://gitlab.inria.fr/sfriedem/repex`, retrieved the 31.03.2022
[11] `http://modules.sourceforge.net`, retrieved the 17.12.2021

6.3.4 Version Control

While lab notebooks allow very well to track ideas and experiments, it would be very annoying to track changes to source codes and other sets of input files there. Version control tools like subversion[12], Git[13] or Mercurial[14] exist for this purpose. They can easily keep track of the history of source code, input files, and scripts setting up the experiments. At any time, a user can restore older versions of files and folders, for example, to roll back some changes. Probably the most used[15] and a commonly well-understood version control system is Git. Git can handle even large codebases containing ten thousands of files[16], and allows to work offline in contrast to version control systems like subversion. Having the entire source history available offline also speeds up operations like showing changes to other versions of the repository.

Originally, Git was made to capture the state of small files that ideally contain text, as source code files do. Files containing binary data can be handled, but the performance of Git can be significantly impacted when handling large files. Efforts like Git LFS[17] or git-annex[18] were made to store large binary files within Git.

Git is used for version control in the scope of this work. There are different Git repositories for source code, lab notebooks, and experiment start scripts. It was not necessary to leverage Git solutions for large files like Git LFS or git-annex. Keeping different versions of large binary input and output files was not necessary in the scope of this work as these files did not evolve. At maximum a single version of them needs to be archived for later reproducibility or postprocessing, a functionality brought by repex.

6.3.5 Deployment

To deploy to supercomputers, software need to be configured to adapt to the machine's hardware and software stack. The paths of all dependencies and architecture-specific compile flags must be set. Originally this was done manually. This is a high effort demanding many iterations of trial and error. Some scientific software packages aimed at HPC ship with install scripts for different machines to simplify this process. The

[12] `https://subversion.apache.org`, retrieved the 04.01.2022
[13] `https://git-scm.com`, retrieved the 04.01.2022
[14] `https://mercurial-scm.org`, retrieved the 04.01.2022
[15] `https://trends.debian.net`, retrieved the 19.01.2022
[16] see the Linux kernel git repositories at `https://git.kernel.org`, retrieved the 04.01.2022
[17] `https://github.com/git-lfs/git-lfs`, retrieved the 04.01.2022
[18] `https://git-annex.branchable.com`, retrieved the 04.01.2022

ParFlow source code distribution[19], for instance, contains different build scripts for atlas-, mac- and tux-like infrastructures.

Alternative approaches that try to avoid machine-specific install scripts are package managers. For instance, Nix (Bzeznik et al., 2017), Guix[20] and Spack (Gamblin et al., 2015) find application in the HPC domain. Spack is tailored to multiuser supercomputers, allowing to install multiple software stacks side-by-side and facilitates switching between them. Many HPC software libraries and applications are already available in Spack, and userspace installation is possible too. Spack is also used by some administrators of supercomputers, for instance, on the Jean-Zay machine used for some of the experiments in the previous Chapters 4 and 5. Spack allows to mix packages installed by users and supercomputer administrators, shortening installation and compile times.

This motivated packaging the Melissa-DA framework as a Spack package too. The Spack package simplifies the deployment of Melissa-DA and its dependencies. In most cases, only the few commands shown in Listing 6.1 are necessary to install Melissa-DA using Spack.

```
$ git clone https://gitlab.inria.fr/melissa/spack
$ git checkout add-melissa-da-build2
$ source spack/share/spack/setup-env.sh
$ spack install melissa-da
$ spack load melissa-da
$
```

Listing 6.1: Installing and loading Melissa-DA using Spack.

For installations using Spack, it is often necessary to rely on the supercomputer's MPI version alongside the respective compilers. Configuring Spack with respect to this can be difficult for users new to a machine as it is not easy to decide which MPI and compilers work best in each situation.

One can also configure Spack to reuse existing package bases and system libraries. But especially system libraries are often difficult to integrate into Spack manually, and correctly announcing their specifications (used compile options and exact version numbers) is error-prone and possibly needs to be repeated after each system update. According to our experience, it is easier to use only the Spack packages provided by supercomputer administrators, or in doubt, build everything except for compiler and MPI versions through Spack. Building many Spack packages can take a long time, but this makes it more portable, requiring less human resources to adapt to varying infrastructures.

[19] `https://github.com/parflow/parflow/tree/v3.9.0/misc/build_scripts`, retrieved the 19.1.2022, see also Section 4.2 for a description of ParFlow

[20] `https://hpc.guix.info`, retrieved the 19.01.2022

6.3.6 Continuous Integration

To minimize the risk of breaking changes introduced by modifications in the software stack that is later run on HPC machines, Melissa-DA uses continuous integration (CI). This is an effective way to avoid bugs that could be expensive, for example, by deadlocking large scale runs costing thousands of compute hours.

Some problems may be easier to detect under special conditions, e.g., on architectures with different instruction sets or smaller or larger cache and memory sizes. Therefore CI is run on different architectures from Raspberry PI to a virtual cluster, ensuring that our code can be installed and executed on all of those. The virtual cluster is composed of multiple connected LXC Linux containers representing compute and front-end nodes. It runs a Red Hat Linux based operating system on an x86-64 architecture, as found on many real-world HPC systems like JUWELS and Jean-Zay. The virtual cluster can run on one physical host, e.g., the researcher's personal computer permitting quick testing and development cycles as there are no other users. Different schedulers like Slurm or OAR can be installed to test integration with them. Detailed documentation of the test setup can be found online[21].

The test suite is set up to run whenever code changes are uploaded to the source code repository of Melissa-DA[22]. After checking that the compilation and installation of all components works on different architectures, end-to-end testing is performed. The end-to-end testing encompasses DA studies with different degrees of parallelism (different numbers of runners, runner cores, and server cores), validated against reference results. The CI also performs forced component crashes to test the fault tolerance system.

[21] `https://gitlab.inria.fr/melissa/melissa-ci`
[22] `https://gitlab.inria.fr/melissa/melissa-da`

6.4 Conclusion

We experienced two main challenges for reproducibility in HPC experimentation. The first one concerns reproducing the experimental environment. Logging of the experimental environment and all code parts must be done extensively. In our case the repex library made this possible for experiments, and Git enabled to keep the history of experiment configuration files, source code, and even lab notebooks. Jupyter notebooks were used to document all undertaken research and to post-process the results up to chart and table generation. A bigger issue is the recreation of the necessary software environment on a different or even on the same machine at a later point in time to reproduce some experiments. In our case, continuous integration guaranteed that the code runs across different architectures. We chose Spack to simplify software dependency specifications and installations. This also adds transparency and reproducibility to the installation process. It still takes some effort to set up Spack on the HPC system of choice as it needs to be configured to rely on already installed MPI and compiler versions and possibly existing Spack package bases.

The second issue for reproducibility in HPC is resource availability. Input data may encompass multiple terabytes and possibly is restrictively licensed. While we provide all experimentation scripts under permissive open source licenses, the input data for the large DA runs on the Neckar catchment and on the European domain (Sections 4.3 and 5.2) are only made available upon request to save continuous hosting costs and overcome licensing issues. But to reproduce our research, access and enough compute hours on similar HPC machines are necessary. Additionally, it can be necessary to run experiments multiple times to compensate for fluctuations in measurements, e.g., due to varying machine loads. This can be extremely costly on the large scale, and the trade-off between invested resources and possible results must be carefully evaluated. For that reason, some experiments in this research work were not repeated multiple times with the exact same parameters at very large scale. Rather runs on a smaller scale help to be confident about the behavior of the setup. On very large scale, runs with different but comparable configurations were performed to explore a larger space of possible configurations while getting some confidence on measured performances if they behave as expected. For instance, some of the runs at largest scales on more than 10,000 compute cores presented in Sections 4.3.7 and 4.3.8 were not repeated for this reason.

Our workflow made it quickly possible to reproduce two years old experiments to test if experienced file system jitter persisted. The experiments were reproduced using the input data archived with repex. In contrast, letting collaborators reproduce our results on their machines required larger efforts. They needed to learn the tooling and adapt it

to their machines. Finally, Kai Keller from the Barcelona Supercomputing Center (BSC) successfully run Melissa-DA at scale on the Marenostrum and Fugaku machines, while our interns Anna Sekuła and Bartłomiej Pogodziński were able to execute Melissa-DA on machines of the Poznan Supercomputing and Networking Center (PSNC).

For research in general, and when planning experimental studies for supercomputers in particular, we recommend putting some efforts into the organization of a transparent and reproducible environment right from the beginning. The cost for this is to pay only once, and it provides great confidence in (experimental) results. In HPC experimentation, it can also enable testing of different parameter sets and code versions in parallel permitting to advance without confusing different experiment runs. In our experience, this quickly amortizes the extra effort at the beginning. Also, the initial effort to implement one-click solutions for postprocessing turned out to pay off quickly. These pipelines can be extended as time goes on, and more analytics are of interest.

For future HPC studies, an effective way to copy, compile, install and run experiment code on a supercomputer from within the lab notebook might be evaluated. While this would greatly improve transparency and automation, it remains questionable if there is much application for this. Often source code development and testing necessitate direct interaction with the supercomputer using a remote shell anyway.

A further topic of investigation might be the use of version control for large files. This might make sense in some use cases, especially when large input files are manipulated over multiple iterations, for instance, to improve the accordance of large scale simulation results with field observations.

7 Conclusion and Perspectives

7.1 Conclusion

In this work, a novel architecture for ensemble-based DA is developed. It is inspired by in situ and in transit workflows, common concepts to accelerate HPC workflows. Our architecture is designed to run massively parallel ensemble runs at exascale.

While existing offline approaches for ensemble-based large scale DA suffer from file system access times and repeated application initialization costs, online approaches avoid these issues. Nevertheless, existing online approaches run as large monolithic jobs that are very sensitive to faults. Their static scheduling of members to compute resources is responsible for load imbalance as fluctuation in the execution time of member propagations cannot be compensated dynamically. In contrast, our framework, Melissa-DA, permits the dynamic scheduling of member propagations to different model instances, called runners. Runners may accommodate a parallel large scale simulation code that executes different members sequentially. This "member virtualization" is key for load-balancing as it permits to schedule propagation tasks one-by-one to different resources, reacting to fluctuations within the execution time of different propagations. Member virtualization also allows runners to stop, crash, or (re-)start with minimal disruptions. This enables elastic executions that change the amount of used compute resources while running continuously, for instance, to react to evolving resource restrictions imposed by the supercomputer's batch scheduler or to auto-recover from faults without breaking the DA workflow.

Many ensemble-based DA algorithms manipulate member state vectors during the update phase. This requires centralizing all member states. Then state updates can be calculated and applied. EnKF variants, for instance, rely on such a workflow. Our architecture runs this workflow online, transferring states through direct connections to completely avoid file I/O.

But the centralization of member states is not always necessary. Many variants of particle filters require only the centralization of particle weights (one scalar per particle). Thus, our architecture employs a second mode where only weights are gathered, and state vectors are exchanged through a shared cache, possibly leveraging the file system. Member virtualization and cache-locality sensitive scheduling reduce the number of cache misses and permits to overlap computation and cache access nearly perfectly.

Both modes enable to overlap communication and computation thanks to a server component that runs on compute resources distinct from the runners, managing propagation scheduling, data transfers, and update phase in the background. Supporting these two modes, our proposal can be applied to nearly all ensemble-based DA methods, covering a wide range of different use cases.

Melissa-DA is shown to run EnKF for a hydrological use case with 16,384 members on 500 compute nodes (20,000 cores) and particle filtering with 2,555 particles on 20,442 cores with 87 % scaling efficiency for short-range weather forecasting on the European domain. Model state sizes account for 92 MiB and 2.5 GiB per member respectively. Thanks to its load-balancing and avoidance or overlapping of file system access, as well as communication and computation, Melissa-DA shows increased performance at very large scale compared to at least one existing framework.

Many of the properties that are key to execute ensemble-based DA with high efficiency at large scale rely on member virtualization. This feature is the base for load-balanced, fault-tolerant, and elastic runs. Further improvements are obtained from the overlap of useful computation and administrative tasks like communication and file I/O and using a multilevel cache in the case of particle filters. The proposed techniques can thus be key to adapt existing ensemble-based DA methods to the exascale era, where modeling is performed with finer resolution, larger domain sizes, and higher prediction accuracy requiring larger state vectors, and more members to be assimilated.

7.2 Perspectives

As shown in Section 4.3.4, the complexity of the EnKF update phase is quadratic in the number of members. At large scale, this leads to an important share of runtime spent during the state update when runners are idle. Using localization and other EnKF filter variants may decrease the necessary computations for the state update, shorten the waiting of the runners and increase efficiency.

Another approach could be based on the work of Niño et al., 2012. They propose the iterative calculation of the error covariance matrices for EnKF. Their algorithm could be implemented by the server. Besides receiving and sending state data during the propagation phase, the server could simultaneously start the heavy calculation necessary for the update phase. This would partly overlap the state update computation with member propagations and shorten the time runners wait for the update phase to finish. Terraz et al., 2017 performed ensemble-based iterative covariance computation for sensitivity analysis at a comparable scale already.

To reduce remaining file output time, in situ post-processing may be leveraged. Thanks to Melissa-DA's modularity, it would require only minor changes to perform post-processing steps in situ by implementing them within the DA update phase. Only higher-order results like ensemble mean and variance, instead of every member state, would be written to disk, diminishing the pressure on the file system.

Our particle filter implementation will experience an important slow down when state access times are higher than the time needed for one particle propagation. In this case, the full state access cannot be overlapped by propagations anymore. While this was not the case in our examples where propagation takes about 25 s and storing or loading the states (2.5 GiB size) takes less than 5 s, it might become an issue for DA problems on high dimensional state vectors where very short assimilation windows are chosen. Persistent RAM (e.g., NVRAM), which will be supported by upcoming supercomputers to some extent, might be used as shared cache instead of the PFS. This would enable to store states persistently to recover in case of failure. However, local persistent RAM is not shared between distant runners. Ephemeral distributed file systems like GekkoFS (Vef et al., 2020) could be used as they provide a shared virtual file system on top of NVRAM distributed on different nodes, i.e., runners.

The modularity of the Melissa-DA framework allows to run more advanced ensemble-based DA methods too. Among these count localized ensemble Kalman filters using a localized error covariance matrix (LEnKF) or the localized ensemble transform Kalman filter (LETKF). Also, methods like the ensemble transform Kalman filter (ETKF) or the Nonlinear Ensemble Transform Filter (NETF) can be easily set up with the Melissa-DA

framework, but testing stays future work. The data flows supported by Melissa-DA are common to different types of particle filters. Particle filters with resampling methods like stratified or systematic resampling could be implemented relying on the same data flow as for SIR particle filters. Transportation particle filters and some localized particle filters have a data flow similar to EnKF. Similar to EnKF, they could be implemented with the server centralizing all particle state vectors to change them accordingly.

Implementing one of these ensemble-based DA methods requires to redefine resampling or the update phase calculation. Only the function transforming a set of background states into analysis states for the workflow described in Chapter 4, or the function that samples the next cycle's ensemble from a set of weighted particles for the workflow of Chapter 5 is required respectively. Features like load-balancing, fault tolerance, elasticity and computation-communication overlap are intrinsic to Melissa-DA and do not need to be changed when adding further assimilation or resampling methods.

As mentioned in Section 5.2.7, speculative particle propagation opens possibilities for further development too. To improve its performance, adapting the used resampling strategy to prefer speculatively propagated particles might be an option. Alternatively, a way to "recycle" unnecessary propagated particles enriching the sample resolution could be examined. Different settings and DA problems should be tested too. Possibly the strategy to select particles for speculative execution could be fine-tuned to lower the count of unnecessary particles. A discriminator particle weight separating particles that may be propagated speculatively from particles that are too risky to propagate in advance might be set.

Alternative approaches to reduce the idle time runners experience before the end of each assimilation were presented in Section 3.2. Island particle filters reduce ensemble-wide synchronization, cutting groups of particles into islands. Particles within islands are synchronized at each cycle, but inter-island exchanges happen only sporadically. Anytime or asynchronous particle filters try to completely avoid any synchronization between particles by defining criteria to decide particle by particle if and how often it will be resampled for the next cycle. This would nicely coact with the implemented list scheduling of member propagations. At any time when a resource for model propagation requires new work, anytime particle filters would be able to provide the next task. The list of work to be scheduled would never be empty, blocking resources from performing propagation work, as it is the case when waiting for an assimilation cycle to finish. The foundations of such approaches were so far only validated with toy examples. Large scale efficiency remains to be demonstrated.

Another known issue of Monte Carlo methods including ensemble-based DA methods is that accuracy may vary over time when the ensemble size is kept constant. But

Melissa-DA does not require a constant ensemble size, as shown when we ran speculative particle propagations. Its workflow can easily be adapted to support, for instance, (alive) particle filters that adapt the ensemble size to fulfill accuracy requirements while avoiding oversampling too.

Island, anytime and alive particle filters look promising for peta- or exascale runs. Melissa-DA offers a sound base to implement these methods and to run them at according scales. While literature explores these methods primarily within the context of particle filters, we believe that similar extensions also have relevance for EnKF variants, as EnKF can, after all, be expressed as a special case of transport particle filters.

Hydrological groundwater simulators like ParFlow, which was used in Section 4.3, do not express interaction with the biosphere and atmosphere, for instance, responsible for evapotranspiration. ParFlow, for this purpose, is typically coupled with the community land model CLM (Dai et al., 2003). To assimilate a more realistic setting, CLM is about to be instrumented to run within Melissa-DA. Finally, also atmospheric conditions need to be considered to depict the full water cycle. Meteorologic models need to be included for this purpose. Thus, the Terrestrial Systems Modeling Platform TerrSysMP (Shrestha et al., 2014; Gasper et al., 2014) couples ParFlow, CLM and the meteorological model COSMO. Ultimately, it is planned to assimilate the full integrated Earth system model TerrSysMP at large scale using Melissa-DA. Instrumenting new models for the use with Melissa-DA can be an important effort that requires a deep understanding of the model code. Similar instrumentation is necessary to include new file I/O modules. PDI (Roussel et al., 2017) provides a generic interface for this purpose. After integrating PDI in a model code, different I/O modules can be plugged in. To simplify the instrumentation work that is needed to use an existing model with Melissa-DA, we collaborate with the authors of PDI, Bigot et al., to support Melissa-DA as a Plug-in in near future. Model codes supporting PDI for I/O could easily be launched as Melissa-DA runners then. To evaluate different ensemble-based DA methods for atmospheric particulate matter propagation, the chemistry part of the Ensemble for Stochastic Integration of Atmospheric Simulations ESIAS-chem (Franke et al., 2022) is about to be instrumented with PDI, ultimately running with the Melissa-DA PDI Plug-in to enable our DA framework.

As stated by O'Brien et al., 2017 and Schulte et al., 2015, future HPC machines are likely to contain a heterogeneous set of resources, leveraging CPUs, GPUs, FPGAs, and nodes with varying amounts of RAM, NVRAM, or SSD space. Its modular runner/server model paired with intrinsic load-balancing allows efficient Melissa-DA executions on such heterogeneous machines leveraging, e.g., nodes with accelerators or more memory for different components like server and runner. The N $\times$ M data redistribution

implemented by Melissa-DA enables the connection of components with different levels of parallelization. Load imbalance produced by tasks executed on a heterogeneous set of resources that progress at varying paces, is equilibrated by online list scheduling in Melissa-DA. In future, we want to investigate such settings.

Melissa-DA lets the user implement assimilation update phase modules for centralized workflows and weight calculation functionality for particle filter workflows in Python. In both cases, the full state data is exposed to the user-defined Python code. Machine learning is typically implemented using Python frameworks too, for instance, using PyTorch (Paszke et al., 2019) or TensorFlow (Martín Abadi et al., 2015). This enables the interception of the data flow to accomplish ML approaches described in Section 3.3. One such approach is online learning by streaming data from Melissa-DA directly into an ML framework. This may, for example, enable combined DA-ML approaches as proposed by Brajard et al., 2019 and Bocquet et al., 2020. Note that a similar workflow aimed at surrogate model training (without DA in the loop) is already under active development in another branch of the Melissa software family. This branch aims to online training of deep neural networks from ensemble simulations. Once a surrogate is trained, it can easily be used instead of a traditional solver code as runner within Melissa-DA. Thus, we believe that Melissa[1]/Melissa-DA is an excellent starting point to create distributed machine learning applications for the very large scale, especially when relying on very large state vectors. These may already necessitate parallelization to fit memory needs and may imply N $\times$ M data redistribution as provided by Melissa-DA.

DA techniques can also be used to solve various inverse problems, for instance, to calibrate model parameters (Ramgraber et al., 2019). Since Melissa-DA is able to run huge numbers of ensemble members necessary to explore high dimensional search spaces, we assume Melissa-DA excels in this kind of application too.

The workflow offered by Melissa-DA is common to many ensemble problems, be it for optimization algorithms or in the field of statistics. We thus imagine applying Melissa-DA to non-DA problems too. Particle swarm optimization (Bonyadi and Michalewicz, 2017), for instance, where the different runners execute a compute intense cost function of multiple particles in parallel. After gathering the results, the server moves the particles following the particle swarm optimization algorithms, and the next set of particles is evaluated until a stop criterion is reached. Melissa-DA can further be easily adapted to perform approximate Bayesian computation (Green et al., 2015) or even run sophisticated algorithms for Markov chain Monte Carlo (MCMC), as the one proposed by Robert et al., 2018 at the large scale.

[1] `https://gitlab.inria.fr/melissa`, retrieved the 10.03.2022

A Sources of Greenhouse Gas Emission

To give a general idea of the environmental impact of this work, this section tries to enumerate some of the major sources of greenhouse gas emissions produced by it. Converting the numbers into an actual amount of released greenhouse gas is left to the interested reader since it is out of reach for this thesis to find meaningful data for the conversion between, e.g., kilometers traveled by train and emitted grams of carbon dioxide.

The total number of CPU hours on HPC infrastructure as used by the author, including all intermediate and failed tests that were connected to this work is at about 957,603, split between the JUWELS and the Jean-Zay supercomputers.

This Ph.D. was a three-year effort summing up to about 630 workdays and to at least 4,400 hours at the personal computer – accounting for about 400 KWh of energy consumption, assuming an average power consumption of about 90 W for notebook computers and peripheral hardware like external screen and docking station. These numbers do not contain energy consumption for the production of all used materials, to run servers for backups, mails, or file exchange, and to maintain the workplaces (light, heating, air-conditioning, etc.). Note that in our case the impact of heating and air-conditioning of the workplace is limited due to the geothermal heating and cooling installed at the lab building[1].

Another major impact on energy consumption comes from business travels – even if their impact got cut in the second half of this thesis work due to the Covid-19 pandemics. All business trips were done by train, accounting for about 9,000 km.

Since the lab is within bike reach, no greenhouse gas was emitted for daily transport between home and workplace.

[1] `https://batiment.imag.fr/files/eco-conception-v2.pdf`, retrieved the 09.04.2022

Bibliography

Abarbanel, H. D. I., P. J. Rozdeba, and S. Shirman (Oct. 2017). "Machine Learning as Statistical Data Assimilation". In: *arXiv:1710.07276 [cs, stat]*. arXiv: 1710.07276. cit. on p. 31

Alexandrov, Alexander, Rico Bergmann, Stephan Ewen, et al. (Dec. 2014). "The Stratosphere Platform for Big Data Analytics". In: *The VLDB Journal* 23.6, pp. 939–964. DOI: 10.1007/s00778-014-0357-y. cit. on p. 33

Alston, Jesse M. and Jessica A. Rick (2021). "A Beginner's Guide to Conducting Reproducible Research". en. In: *The Bulletin of the Ecological Society of America* 102.2, e01801. DOI: 10.1002/bes2.1801. cit. on p. 102

Ancell, Brian C., Allison Bogusz, Matthew J. Lauridsen, and Christian J. Nauert (Mar. 2018). "Seeding Chaos: The Dire Consequences of Numerical Noise in NWP Perturbation Experiments". EN. In: *Bulletin of the American Meteorological Society* 99.3. Publisher: American Meteorological Society Section: Bulletin of the American Meteorological Society, pp. 615–628. DOI: 10.1175/BAMS-D-17-0129.1. cit. on p. 83

Anderson, Jeffrey, Tim Hoar, Kevin Raeder, et al. (Sept. 2009). "The Data Assimilation Research Testbed: A Community Facility". en. In: *Bulletin of the American Meteorological Society* 90.9. Publisher: American Meteorological Society, pp. 1283–1296. cit. on p. 26

Asch, Mark, Marc Bocquet, and Maëlle Nodet (2016). *Data assimilation: methods, algorithms, and applications*. Vol. 11. SIAM. cit. on pp. 7, 20, 24

Ashby, Steven F. and Robert D. Falgout (Sept. 1996). "A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations". In: *Nuclear Science and Engineering* 124.1, pp. 145–159. DOI: dx.doi.org/10.13182/NSE96-A24230. cit. on p. 48

Babuji, Yadu, Anna Woodard, Zhuozhao Li, et al. (2019). "Parsl: Pervasive Parallel Programming in Python". In: *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. cit. on p. 33

Balasubramanian, Vivek, Travis Jensen, Matteo Turilli, et al. (2020). "Adaptive Ensemble Biomolecular Applications at Scale". In: *SN Computer Science* 1.2, pp. 1–15. cit. on p. 25

Balasubramanian, Vivek, Matteo Turilli, Weiming Hu, et al. (2018). "Harnessing the Power of Many: Extensible Toolkit for Scalable Ensemble Applications". In: *IPDPS 2018*. cit. on p. 25

Bannister, R. N. (2017). "A review of operational methods of variational and ensemble-variational data assimilation". en. In: *Quarterly Journal of the Royal Meteorological Society* 143.703. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qj.2982, pp. 607–633. DOI: 10.1002/qj.2982. cit. on p. 27

Bauer, Peter, Peter D. Dueben, Torsten Hoefler, et al. (Feb. 2021). "The digital revolution of Earth-system science". In: *Nature Computational Science* 1.2, pp. 104–113. cit. on p. 2

Bauer, Peter, Alan Thorpe, and Gilbert Brunet (Sept. 2015). "The quiet revolution of numerical weather prediction". In: *Nature* 525.7567, pp. 47–55. cit. on p. 2

Bautista-Gomez, Leonardo, Seiji Tsuboi, Dimitri Komatitsch, et al. (Nov. 2011). "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*, pp. 1–12. cit. on pp. 42, 80

Berndt, Jonas (2018). "On the predictability of exceptional error events in wind power forecasting—an ultra large ensemble approach—". en. PhD thesis. Universität zu Köln. cit. on pp. 25, 92

Blumofe, Robert D. and Charles E. Leiserson (1999). "Scheduling multithreaded computations by work stealing". In: *J. ACM* 46.5, pp. 720–748. cit. on p. 45

Bocquet, Marc, Julien Brajard, Alberto Carrassi, and Laurent Bertino (Feb. 2019). "Data assimilation as a deep learning tool to infer ODE representations of dynamical models". en. In: *Nonlinear Processes in Geophysics Discussions*, pp. 1–29. DOI: 10.5194/npg-2019-7. cit. on p. 31

– (2020). "Bayesian inference of chaotic dynamics by merging data assimilation, machine learning and expectation-maximization". en. In: *Foundations of Data Science* 2.1. Company: Foundations of Data Science Distributor: Foundations of Data Science Institution: Foundations of Data Science Label: Foundations of Data Science Publisher: American Institute of Mathematical Sciences, p. 55. DOI: 10.3934/fods.2020004. cit. on pp. 31, 121

Bolic, M., P.M. Djuric, and Sangjin Hong (2003). "New resampling algorithms for particle filters". en. In: *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).* Vol. 2. Hong Kong, China: IEEE, pp. II–589–92. DOI: 10.1109/ICASSP.2003.1202435. cit. on pp. 16, 97

Bonavita, Massimo, Y. Trémolet, Elias Hólm, et al. (2017). *A Strategy for Data Assimilation*. DOI: 10.21957/tx1epjd2p. cit. on p. 27

Bonyadi, Mohammad Reza and Zbigniew Michalewicz (Mar. 2017). "Particle Swarm Optimization for Single Objective Continuous Space Problems: A Review". In: *Evolutionary Computation* 25.1, pp. 1–54. DOI: 10.1162/EVCO_r_00180. cit. on p. 121

Brajard, Julien, Alberto Carrassi, Marc Bocquet, and Laurent Bertino (May 2019). "Combining data assimilation and machine learning to emulate a dynamical model from sparse and noisy observations: a case study with the Lorenz 96 model". en. In: *Geoscientific Model Development Discussions* 44. arXiv: 2001.01520, pp. 1–21. DOI: 10.1016/j.jocs.2020.101171. cit. on pp. 31, 121

Burgers, Gerrit, Peter Jan Van Leeuwen, and Geir Evensen (June 1998). "On the Analysis Scheme in the Ensemble Kalman Filter". In: *Monthly Weather Review* 126. cit. on p. 24

Bzeznik, Bruno, Oliver Henriot, Valentin Reis, Olivier Richard, and Laure Tavard (Nov. 2017). "Nix as HPC package management system". In: *Proceedings of the Fourth International Workshop on HPC User Support Tools*. HUST'17. New York, NY, USA: Association for Computing Machinery, pp. 1–6. DOI: 10.1145/3152493.3152556. cit. on p. 111

Capit, Nicolas, Georges Da Costa, Yiannis Georgiou, et al. (2005). "A batch scheduler with high level components". In: *Cluster computing and Grid 2005 (CCGrid05)*. Cardiff, United Kingdom: IEEE. cit. on p. 60

Capul, Julien, Sébastien Morais, and Jacques-Bernard Lekien (2018). "PaDaWAn: a Python infrastructure for loosely coupled in situ workflows." In: *ISAV'2018*, pp. 7–12. cit. on p. 34

Carrassi, Alberto, Marc Bocquet, Laurent Bertino, and Geir Evensen (Sept. 2017). "Data Assimilation in the Geosciences - An overview on methods, issues and perspectives". en. In: *arXiv:1709.02798 [physics, stat]*. arXiv: 1709.02798. cit. on p. 20

Clayton, A. M., A. C. Lorenc, and D. M. Barker (July 2013). "Operational implementation of a hybrid ensemble/4D-Var global data assimilation system at the Met Office". en. In: *Quarterly Journal of the Royal Meteorological Society* 139.675, pp. 1445–1461. cit. on p. 27

Courtier, P., Erik Andersson, W.A. Heckley, et al. (Jan. 1998). *The ECMWF implementation of three dimensional variational assimilation. Part 1:Formulation*. Shinfield Park, Reading. DOI: 10.21957/unhecz1kq. cit. on p. 8

Courtier, P., J. Thepaut, and A. Hollingsworth (1994). "A strategy for operational implementation of 4D-Var, using an incremental approach". In: *Quarterly Journal of the Royal Meteorological Society* 120.519, pp. 1367–1387. DOI: 10.1002/QJ.49712051912. cit. on p. 24

Dai, Yongjiu, Xubin Zeng, Robert E. Dickinson, et al. (Aug. 2003). "The Common Land Model". In: *Bulletin of the American Meteorological Society* 84.8, pp. 1013–1024. DOI: 10.1175/BAMS-84-8-1013. cit. on p. 120

Desquilbet, Loic, Sabrina Granger, Boris Hejblum, et al. (May 2019). *Vers une recherche reproductible*. Ed. by Unité régionale de formation à l'information scientifique et technique de Bordeaux. Unité régionale de formation à l'information scientifique et technique de Bordeaux. cit. on p. 102

Docan, Ciprian, Manish Parashar, and Scott Klasky (2012). "DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows". English. In: *Cluster Computing* 15.2, pp. 163–181. DOI: 10.1007/s10586-011-0162-y. cit. on p. 34

Dorier, Matthieu, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf (Sept. 2012). "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O". en. In: *Proceedings of the IEEE Cluster 2012 conference*. cit. on pp. 33, 34

Dreher, Matthieu and Tom Peterka (2017). *Decaf: Decoupled dataflows for in situ high-performance workflows*. Tech. rep. Argonne National Lab.(ANL), Argonne, IL (United States). cit. on p. 34

Dreher, Matthieu and Bruno Raffin (May 2014). "A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations". en. In: *CCGrid - International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Science Press. cit. on pp. 33, 34, 35

ecFlow (Jan. 2022). URL: https://github.com/ecmwf/ecflow (visited on Feb. 15, 2022). cit. on p. 109

ECMWF (Sept. 2021). "IFS Documentation CY47R3 - Part II: Data assimilation". In: *IFS Documentation CY47R3*. IFS Documentation 2. ECMWF. DOI: 10.21957/t445u8kna. cit. on p. 1

Elvira, Víctor, Joaquín Míguez, and Petar M Djurić (2016). "Adapting the number of particles in sequential Monte Carlo methods through an online scheme for convergence assessment". In: *IEEE Transactions on Signal Processing* 65.7, pp. 1781–1794. cit. on p. 30

Elwasif, Wael R, David E Bernholdt, Sreekanth Pannala, Srikanth Allu, and Samantha S Foley (2012). "Parameter sweep and optimization of loosely coupled simulations using the DAKOTA toolkit". In: *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pp. 102–110. cit. on p. 33

ERA5 hourly data (2019). Type: dataset. cit. on p. 84

Evensen, Geir (1994). "Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics". en. In: *Journal of Geophysical Research: Oceans* 99.C5. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/94JC00572, pp. 10143–10162. DOI: 10.1029/94JC00572. cit. on pp. 12, 24

– (2009). *Data assimilation: the ensemble Kalman filter*. Springer Science & Business Media. cit. on pp. 12, 20, 24

Farchi, Alban, Patrick Laloyaux, Massimo Bonavita, and Marc Bocquet (2021). "Using machine learning to correct model error in data assimilation and forecast applications". en. In: *Quarterly Journal of the Royal Meteorological Society* 147.739, pp. 3067–3084. DOI: 10.1002/qj.4116. cit. on p. 31

Fearnhead, Paul and Hans Künsch (Mar. 2018). "Particle Filters and Data Assimilation". en. In: *Annual Review of Statistics and Its Application* 5.1. arXiv: 1709.04196, pp. 421–449. cit. on p. 29

Franke, Philipp, Anne Caroline Lange, and Hendrik Elbern (Feb. 2022). "Particle-filter-based volcanic ash emission inversion applied to a hypothetical sub-Plinian Eyjafjallajökull eruption using the Ensemble for Stochastic Integration of Atmospheric Simulations (ESIAS-chem) version 1.0". en. In: *Geoscientific Model Development* 15.3, pp. 1037–1060. DOI: 10.5194/gmd-15-1037-2022. cit. on p. 120

Freed, Jonathan, Saurabh Gupta, and Devesh Tiwari (Nov. 2015). *An Analysis of Network Congestion in the Titan Supercomputer's Interconnect*. en. URL: http://sc15.supercomputing.org/sites/all/themes/SC15images/src_poster/src_poster_pages/spost127.html. cit. on p. 103

Friedemann, Sebastian and Bruno Raffin (Nov. 2020). *An elastic framework for ensemble-based large-scale data assimilation*. Research Report RR-9377. Inria Grenoble Rhône-Alpes, Université de Grenoble. URL: https://hal.inria.fr/hal-03017033. cit. on p. 37

– (June 2022). "An elastic framework for ensemble-based large-scale data assimilation". en. In: *The International Journal of High Performance Computing Applications*. Publisher: SAGE Publications Ltd STM, p. 10943420221110507. DOI: 10.1177/10943420221110507. cit. on p. 37

Gamblin, T., M. LeGendre, M. R. Collette, et al. (Nov. 2015). "The Spack package manager: bringing order to HPC software chaos". In: *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1–12. DOI: 10.1145/2807591.2807623. cit. on p. 111

Gasper, F., K. Goergen, P. Shrestha, et al. (Oct. 2014). "Implementation and scaling of the fully coupled Terrestrial Systems Modeling Platform (TerrSysMP v1.0) in a massively parallel supercomputing environment – a case study on JUQUEEN (IBM Blue Gene/Q)". English. In: *Geoscientific Model Development* 7.5. Publisher: Copernicus GmbH, pp. 2531–2543. DOI: 10.5194/gmd-7-2531-2014. cit. on p. 120

Gordon, N.J., D.J. Salmond, and A.F.M. Smith (1993). "Novel approach to nonlinear/non-Gaussian Bayesian state estimation". en. In: *IEE Proceedings F Radar and Signal Processing* 140.2, p. 107. DOI: 10.1049/ip-f-2.1993.0015. cit. on p. 16

Goswami, B. N. (Nov. 1996). "The challenge of weather prediction". In: *Resonance* 2, pp. 8–15. DOI: 10.1007/BF02838775. cit. on p. 83

Graham, R. L. (1966). "Bounds for Certain Multiprocessing Anomalies". en. In: *Bell System Technical Journal* 45.9, pp. 1563–1581. cit. on pp. 44, 77

Green, Peter, Krzysztof Latuszyski, Marcelo Pereyra, and Christian Robert (2015). *Bayesian computation: a perspective on the current state, and sampling backwards and forwards*. en. DOI: 10.48550/ARXIV.1502.01148. cit. on p. 121

Herzog, Amelie, Basile Hector, Jean-Martial Cohard, et al. (2021). "A parametric sensitivity analysis for prioritizing regolith knowledge needs for modeling water transfers in the West African critical zone". en. In: *Vadose Zone Journal* 20.6, e20163. DOI: 10.1002/vzj2.20163. cit. on p. 49

Hintjens, Pieter (2013). *ZeroMQ, Messaging for Many Applications*. O'Reilly Media. cit. on p. 40

Houtekamer, P.L., Mark Buehner, and Michèle De La Chevrotière (2019). "Using the hybrid gain algorithm to sample data assimilation uncertainty". en. In: *Quarterly Journal of the Royal Meteorological Society* 145.S1, pp. 35–56. cit. on p. 27

Houtekamer, P.L. and Herschel L Mitchell (1998). "Data assimilation using an ensemble Kalman filter technique". In: *Monthly Weather Review* 126.3, pp. 796–811. cit. on pp. 12, 24

Hu, Wei, Guang-ming Liu, Qiong Li, Yan-huang Jiang, and Gui-lin Cai (Nov. 2016). "Storage wall for exascale supercomputing". In: *Frontiers of Information Technology & Electronic Engineering* 17.11, pp. 1154–1175. DOI: 10.1631/FITEE.1601336. cit. on p. 34

Hunt, Brian R., Eric J. Kostelich, and Istvan Szunyogh (Dec. 2006). "Efficient Data Assimilation for Spatiotemporal Chaos: a Local Ensemble Transform Kalman Filter". In: *arXiv: physics/0511236*. cit. on p. 14

Jain, Himanshi and Raksha Jain (Mar. 2017). "Big data in weather forecasting: Applications and challenges". en. In: *2017 International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*. Chirala, Andhra Pradesh, India: IEEE, pp. 138–142. DOI: 10.1109/ICBDACI.2017.8070824. cit. on p. 2

Jasra, Ajay, Anthony Lee, Christopher Yau, and Xiaole Zhang (2013). "The alive particle filter". In: *arXiv preprint arXiv:1304.0151*. cit. on p. 30

Jazwinski, Andrew H., ed. (1970). *Stochastic Processes and Filtering Theory*. Vol. 64. Mathematics in Science and Engineering. ISSN: 0076-5392. Elsevier. DOI: https://doi.org/10.1016/S0076-5392(09)60374-X. cit. on p. 12

Jones, Jim E. and Carol S. Woodward (July 2001). "Newton–Krylov-multigrid solvers for large-scale, highly heterogeneous, variably saturated flow problems". In: *Advances in Water Resources* 24.7, pp. 763–774. DOI: 10.1016/S0309-1708(00)00075-0. cit. on p. 48

Jülich Supercomputing Centre (2019). "JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre". In: *Journal of large-scale research facilities* 5.A135. DOI: 10.17815/jlsrf-5-171. cit. on p. 50

Kalman, Rudolph Emil (1960). "A New Approach to Linear Filtering and Prediction Problems". In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D, pp. 35–45. cit. on p. 10

Kashyap, Anre, Marc Baker, and Kevin Mayo (2019). *AMD EPYC and WRF Powering the Future of HPC*. Tech. rep. Advanced Micro Devices, Inc. cit. on p. 84

Kollet, Stefan, Fabian Gasper, Slavko Brdar, et al. (Nov. 2018). "Introduction of an Experimental Terrestrial Forecasting/Monitoring System at Regional to Continental Scales Based on the Terrestrial Systems Modeling Platform (v1.1.0)". en. In: *Water* 10.11, p. 1697. DOI: 10.3390/w10111697. cit. on p. 49

Kollet, Stefan J. and Reed M. Maxwell (Feb. 2008). "Capturing the influence of groundwater dynamics on land surface processes using an integrated, distributed watershed model". en. In: *Water Resources Research* 44.2, W02402. DOI: 10.1029/2007WR006004. cit. on p. 48

Kovachki, Nikola B. and Andrew M. Stuart (Sept. 2019). "Ensemble Kalman Inversion: A Derivative-Free Technique For Machine Learning Tasks". In: *Inverse Problems* 35.9. arXiv: 1808.03620, p. 095005. DOI: 10.1088/1361-6420/ab1c3a. cit. on p. 31

Kunkel, Julian Martin, Michael Kuhn, and Thomas Ludwig (2014). "Exascale storage systems: an analytical study of expenses". In: *Supercomputing frontiers and innovations* 1.1, pp. 116–134. cit. on p. 34

Kurtz, Wolfgang, Guowei He, Stefan J Kollet, et al. (Apr. 2016). "TerrSysMP-PDAF (version 1.0): a modular high-performance data assimilation framework for an integrated land surface-subsurface model." In: *Geoscientific Model Development* 9.4, pp. 1341–1360. DOI: 10.5194/gmd-9-1341-2016. cit. on pp. 2, 26, 48

Lakshmanan, S., C. Mythili, and V. Kavitha (Apr. 2012). "Kalman Filtering Technique For Video Denoising Method". en. In: *International Journal of Computer Applications* 43.20, pp. 10–13. DOI: 10.5120/6218-8707. cit. on p. 11

Lapides, Dana A., Cy David, Anneliese Sytsma, David Dralle, and Sally Thompson (2020). "Analytical solutions to runoff on hillslopes with curvature: numerical and laboratory verification". en. In: *Hydrological Processes* 34.24, pp. 4640–4659. DOI: 10.1002/hyp.13879. cit. on p. 49

Lermusiaux, P. F. J. and A. R. Robinson (July 1999). "Data Assimilation via Error Subspace Statistical Estimation.Part I: Theory and Schemes". en. In: *Monthly Weather Review* 127.7. Publisher: American Meteorological Society Section: Monthly Weather Review, pp. 1385–1407. cit. on p. 30

Li, Tiancheng, Miodrag Bolic, and Petar Djuric (May 2015). "Resampling Methods for Particle Filtering: Classification, implementation, and strategies". In: *Signal Processing Magazine, IEEE* 32, pp. 70–86. DOI: 10.1109/MSP.2014.2330626. cit. on p. 19

Liang, Eric, Richard Liaw, Robert Nishihara, et al. (2018). "RLlib: Abstractions for distributed reinforcement learning". In: *International Conference on Machine Learning*, pp. 3053–3062. cit. on p. 33

Lien, Guo-Yuan, Eugenia Kalnay, and Takemasa Miyoshi (Dec. 2013). "Effective assimilation of global precipitation: simulation experiments". In: *Tellus A: Dynamic Meteorology and Oceanography* 65.1. Publisher: Taylor & Francis _eprint: https://doi.org/10.3402/tellusa.v65i0.19915, p. 19915. DOI: 10.3402/tellusa.v65i0.19915. cit. on p. 20

Liu, Jun S., Rong Chen, and Tanya Logvinenko (2001). "A Theoretical Framework for Sequential Importance Sampling with Resampling". en. In: *Sequential Monte Carlo Methods in Practice*. Ed. by Arnaud Doucet, Nando de Freitas, and Neil Gordon. Statistics for Engineering and Information Science. New York, NY: Springer, pp. 225–246. DOI: 10.1007/978-1-4757-3437-9_11. cit. on p. 16

Lorenc, A. C., S. P. Ballard, R. S. Bell, et al. (Oct. 2000). "The Met. Office global three-dimensional variational data assimilation scheme". en. In: *Quarterly Journal of the Royal Meteorological Society* 126.570, pp. 2991–3012. DOI: 10.1002/qj.49712657002. cit. on pp. 8, 24

Lorenz, Edward N. (1972). "Predictability: Does the Flap of a Butterfly's Wings in Brazil Set Off a Tornado in Texas?" en. In: OCLC: 971887091. Kbh.: L & R Uddannelse. cit. on p. 1

Lührs, Sebastian, Daniel Rohe, Alexander Schnurpfeil, Kay Thust, and Wolfgang Frings (2016). "Flexible and Generic Workflow Management". In: *Parallel Computing: On the Road to Exascale*. Publisher: IOS Press, pp. 431–438. DOI: 10.3233/978-1-61499-621-7-431. cit. on p. 109

Ma, Kwan-Liu (2009). "In Situ Visualization at Extreme Scale: Challenges and Opportunities". In: *Computer Graphics and Applications, IEEE* 29.6, pp. 14–19. DOI: 10.1109/MCG.2009.120. cit. on p. 34

Majda, Andrew J. and Xin T. Tong (May 2017). "Performance of Ensemble Kalman filters in large dimensions". en. In: *arXiv:1606.09321 [math, stat]*. arXiv: 1606.09321. cit. on p. 20

Martín Abadi, Ashish Agarwal, Paul Barham, et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. cit. on p. 121

Maxwell, Reed M. (Mar. 2013). "A terrain-following grid transform and preconditioner for parallel, large-scale, integrated hydrologic modeling". In: *Advances in Water Resources* 53, pp. 109–117. DOI: 10.1016/j.advwatres.2012.10.001. cit. on p. 48

Maxwell, Reed M. and Norman L. Miller (June 2005). "Development of a Coupled Land Surface and Groundwater Model". In: *Journal of Hydrometeorology* 6.3, pp. 233–247. DOI: 10.1175/JHM422.1. cit. on p. 48

Miyoshi, Takemasa, Keiichi Kondo, and Toshiyuki Imamura (July 2014). "The 10,240-member ensemble Kalman filtering with an intermediate AGCM: 10240-MEMBER ENKF WITH AN AGCM". en. In: *Geophysical Research Letters* 41.14, pp. 5264–5271. cit. on p. 25

Moritz, Philipp, Robert Nishihara, Stephanie Wang, et al. (Oct. 2018). "Ray: a distributed framework for emerging AI applications". In: *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, pp. 561–577. cit. on p. 33

Nerger, L., W. Hiller, and J. Schröter (Sept. 2005). "PDAF - THE PARALLEL DATA ASSIMILATION FRAMEWORK: EXPERIENCES WITH KALMAN FILTERING". en. In: *Use of High Performance Computing in Meteorology*. Reading, UK: WORLD SCIENTIFIC, pp. 63–83. DOI: 10.1142/9789812701831_0006. cit. on pp. 54, 62

Nerger, Lars and Wolfgang Hiller (June 2013). "Software for ensemble-based data assimilation systems—Implementation strategies and scalability". In: *Computers & Geosciences*. Ensemble Kalman filter for data assimilation 55, pp. 110–118. DOI: 10.1016/j.cageo.2012.03.026. cit. on pp. 20, 26, 54, 62

Niño, Elias D., Adrian Sandu, and Jeffrey L. Anderson (Jan. 2012). "An Effcient Implementation of the Ensemble Kalman Filter Based on Iterative Sherman Morrison Formula". en. In: *Procedia Computer Science*. Proceedings of the International Conference on Computational Science, ICCS 2012 9, pp. 1064–1072. cit. on pp. 62, 118

O'Brien, Kenneth, Lorenzo Di Tucci, Gianluca Durelli, and Michaela Blott (Mar. 2017). "Towards exascale computing with heterogeneous architectures". en. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. Lausanne, Switzerland: IEEE, pp. 398–403. DOI: 10.23919/DATE.2017.7927023. cit. on pp. 103, 120

Oliver, Hilary, Matthew Shin, David Matthews, et al. (July 2019). "Workflow Automation for Cycling Systems". In: *Computing in Science Engineering* 21.4. Conference Name: Computing in Science Engineering, pp. 7–21. DOI: 10.1109/MCSE.2019.2906593. cit. on p. 109

Ott, Edward, Brian R. Hunt, Istvan Szunyogh, et al. (Jan. 2004). "A local ensemble Kalman filter for atmospheric data assimilation". In: *Tellus A: Dynamic Meteorology and Oceanography* 56.5, pp. 415–428. DOI: 10.3402/tellusa.v56i5.14462. cit. on p. 14

Paige, Brooks, Frank Wood, Arnaud Doucet, and Yee Whye Teh (July 2014). "Asynchronous Anytime Sequential Monte Carlo". In: *arXiv:1407.2864 [stat]*. arXiv: 1407.2864. cit. on p. 29

Paraskevakos, Ioannis, Andre Luckow, Mahzad Khoshlessan, et al. (2018). "Task-parallel analysis of molecular dynamics trajectories". In: *Proceedings of the 47th International Conference on Parallel Processing*, pp. 1–10. cit. on p. 33

Pasetto, Damiano, Matteo Camporese, and Mario Putti (Oct. 2012). "Ensemble Kalman filter versus particle filter for a physically-based coupled surface–subsurface model". en. In: *Advances in Water Resources* 47, pp. 1–13. DOI: 10.1016/j.advwatres.2012.06.009. cit. on p. 20

Paszke, Adam, Sam Gross, Francisco Massa, et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, et al. Curran Associates, Inc., pp. 8024–8035. cit. on p. 121

Poterjoy, Jonathan (Jan. 2016). "A Localized Particle Filter for High-Dimensional Nonlinear Systems". EN. In: *Monthly Weather Review* 144.1. Publisher: American Meteorological Society Section: Monthly Weather Review, pp. 59–76. DOI: 10.1175/MWR-D-15-0163.1. cit. on p. 20

Poterjoy, Jonathan and Jeffrey L. Anderson (May 2016). "Efficient Assimilation of Simulated Observations in a High-Dimensional Geophysical System Using a Localized Particle Filter". EN. In: *Monthly Weather Review* 144.5. Publisher: American Meteorological Society Section: Monthly Weather Review, pp. 2007–2020. DOI: 10.1175/MWR-D-15-0322.1. cit. on p. 20

Pronk, S., G. R. Bowman, B. Hess, et al. (Nov. 2011). "Copernicus: A new paradigm for parallel adaptive molecular dynamics". In: *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–10. cit. on p. 33

Ramgraber, Maximilian, C. Albert, and Mario Schirmer (Nov. 2019). "Data Assimilation and Online Parameter Optimization in Groundwater Modeling Using Nested Particle Filters". In: *Water Resources Research* 55. DOI: 10.1029/2018WR024408. cit. on p. 121

Rasmussen, J., Henrik Madsen, K. Jensen, and Jens Refsgaard (July 2015). "Data assimilation in integrated hydrological modeling using ensemble Kalman filtering: evaluating the effect of ensemble size and localization on filter performance". In: *Hydrology and Earth System Sciences* 19. DOI: 10.5194/hess-19-2999-2015. cit. on p. 50

Reed, Daniel A. and Jack Dongarra (June 2015). "Exascale computing and big data". In: *Communications of the ACM* 58.7, pp. 56–68. DOI: 10.1145/2699414. cit. on p. 25

Reich, Sebastian (Jan. 2013). "A non-parametric ensemble transform method for Bayesian inference". In: *arXiv:1210.0375 [math]*. arXiv: 1210.0375. cit. on p. 19

Richards, L. A. (Nov. 1931). "Capillary conduction of liquids through porous mediums". In: *Physics* 1.5. Publisher: American Institute of Physics, pp. 318–333. DOI: 10.1063/1.1745010. cit. on p. 49

Rivi, Marzia, Luigi Calori, Giuseppa Muscianisi, and Vladimir Slavnić (Jan. 2012). *In-situ Visualization: State-of-the-art and Some Use Cases*. cit. on p. 34

Robert, Christian P., Victor Elvira, Nick Tawn, and Changye Wu (Apr. 2018). "Accelerating MCMC Algorithms". en. In: *arXiv:1804.02719 [stat]*. arXiv: 1804.02719. cit. on p. 121

Rocklin, Matthew (2015). "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: *Proceedings of the 14th Python in Science Conference*. Ed. by Kathryn Huff and James Bergstra, pp. 130–136. cit. on p. 33

Roussel, Corentin, Kai Keller, Mohamed Gaalich, Leonardo Bautista Gomez, and Julien Bigot (Sept. 2017). "PDI, an approach to decouple I/O concerns from high-performance simulation codes". URL: https://hal.archives-ouvertes.fr/hal-01587075 (visited on June 25, 2020). cit. on p. 120

Schalge, Bernd, Gabriele Baroni, Barbara Haese, et al. (Mar. 2020). "Presentation and discussion of the high resolution atmosphere-land surface subsurface simulation dataset of the virtual Neckar catchment for the period 2007-2015". English. In: *Earth System Science Data Discussions*. Publisher: Copernicus GmbH, pp. 1–40. cit. on p. 50

Schulte, Michael J., Mike Ignatowski, Gabriel H. Loh, et al. (July 2015). "Achieving Exascale Capabilities through Heterogeneous Computing". en. In: *IEEE Micro* 35.4, pp. 26–36. DOI: 10.1109/MM.2015.71. cit. on pp. 103, 120

Schulthess, Thomas C., Peter Bauer, Nils Wedi, et al. (Jan. 2019). "Reflecting on the Goal and Baseline for Exascale Computing: A Roadmap Based on Weather and Climate Simulations". en. In: *Computing in Science & Engineering* 21.1, pp. 30–41. DOI: 10.1109/MCSE.2018.2888788. cit. on p. 20

Shmoys, D.B., J. Wein, and D.P. Williamson (1991). "Scheduling parallel machines on-line". en. In: *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*. San Juan, Puerto Rico: IEEE Comput. Soc. Press, pp. 131–140. cit. on pp. 44, 77

Shrestha, P., M. Sulis, M. Masbou, S. Kollet, and C. Simmer (Sept. 2014). "A Scale-Consistent Terrestrial Systems Modeling Platform Based on COSMO, CLM, and ParFlow". EN. In: *Monthly Weather Review* 142.9. Publisher: American Meteorological Society Section: Monthly Weather Review, pp. 3466–3483. DOI: 10.1175/MWR-D-14-00029.1. cit. on p. 120

Skamarock, W. C., J. B. Klemp, J. Dudhia, et al. (2008). *A Description of the Advanced Research WRF Version 3*. Tech. rep. No. NCAR/TN-475+STR. University Corporation for Atmospheric Research. DOI: 10.5065/D68S4MVH. cit. on p. 83

Skinner, D. and W. Kramer (2005). "Understanding the causes of performance variability in HPC workloads". en. In: *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* Austin, TX, USA: IEEE, pp. 137–149. DOI: 10.1109/IISWC.2005.1526010. cit. on p. 103

Snyder, Chris and Thomas Bengtsson (2015). "Performance Bounds for Particle Filters Using the Optimal Proposal". en. In: *MONTHLY WEATHER REVIEW* 143, p. 1. cit. on p. 29

Stengel, M., A. Kniffka, J. F. Meirink, et al. (Apr. 2014). "CLAAS: the CM SAF cloud property data set using SEVIRI". In: *Atmos. Chem. Phys.* 14.8, pp. 4297–4311. cit. on pp. 84, 104

Sun, Chao, Li Liu, Ruizhe Li, et al. (May 2021). "Developing a common, flexible and efficient framework for weakly coupled ensemble data assimilation based on C-Coupler2.0". In: *Geoscientific Model Development* 14, pp. 2635–2657. DOI: 10.5194/gmd-14-2635-2021. cit. on p. 26

Sun, Shu-Li and Zi-Li Deng (June 2004). "Multi-sensor optimal information fusion Kalman filter". en. In: *Automatica* 40.6, pp. 1017–1023. DOI: 10.1016/j.automatica.2004.01.014. cit. on p. 11

Tang, Meng, Yimin Liu, and Louis J. Durlofsky (July 2020). "A deep-learning-based surrogate model for data assimilation in dynamic subsurface flow problems". en. In: *Journal of Computational Physics* 413, p. 109456. DOI: 10.1016/j.jcp.2020.109456. cit. on p. 31

Terraz, Théophile, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin (2017). "Melissa: Large Scale In Transit Sensitivity Analysis Avoiding Intermediate Files". In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. Denver. cit. on pp. 33, 118

TOP500 (June 2022). URL: https://www.top500.org/lists/top500/2022/06/ (visited on Aug. 7, 2022). cit. on pp. 1, 2, 25, 103

Toye, Habib, Samuel Kortas, Peng Zhan, and Ibrahim Hoteit (2018). "A fault-tolerant HPC scheduler extension for large and operational ensemble data assimilation: Application to the Red Sea". In: *Journal of Computational Science* 27, pp. 46–56. cit. on p. 25

Um, Kiwon, Robert Brand, Yun, et al. (Jan. 2021). "Solver-in-the-Loop: Learning from Differentiable Physics to Interact with Iterative PDE-Solvers". In: *arXiv:2007.00016 [physics]*. arXiv: 2007.00016. cit. on p. 31

van Leeuwen, Peter Jan (Sept. 2003). "A Variance-Minimizing Filter for Large-Scale Applications". In: *Mon. Wea. Rev.* 131.9, pp. 2071–2084. cit. on p. 24

– (Dec. 2009). "Particle Filtering in Geophysical Systems". In: *Mon. Wea. Rev.* 137.12, pp. 4089–4114. DOI: 10.1175/2009mwr2835.1. cit. on pp. 18, 29

van Leeuwen, Peter Jan, Hans R Künsch, Lars Nerger, Roland Potthast, and Sebastian Reich (2019). "Particle filters for high-dimensional geoscience applications: A review". In: *Quarterly Journal of the Royal Meteorological Society* 145.723, pp. 2335–2365. cit. on pp. 16, 19, 20, 29, 30

van Velzen, Nils, Muhammad Umer Altaf, and Martin Verlaan (May 2016). "OpenDA-NEMO framework for ocean data assimilation". en. In: *Ocean Dynamics* 66.5, pp. 691–702. cit. on p. 25

Vef, Marc-André, Nafiseh Moti, Tim Süß, et al. (Jan. 2020). "GekkoFS — A Temporary Burst Buffer File System for HPC Applications". en. In: *Journal of Computer Science and Technology* 35.1, pp. 72–91. DOI: 10.1007/s11390-020-9797-6. cit. on pp. 89, 118

Vergé, Christelle, Cyrille Dubarry, Pierre Del Moral, and Eric Moulines (2015). "On parallel implementation of sequential Monte Carlo methods: the island particle model". In: *Statistics and Computing* 25.2, pp. 243–260. cit. on p. 29

Vetra-Carvalho, Sanita, Peter Jan van Leeuwen, Lars Nerger, et al. (Jan. 2018). "State-of-the-art stochastic data assimilation methods for high-dimensional non-Gaussian problems". en. In: *Tellus A: Dynamic Meteorology and Oceanography* 70.1, pp. 1–43. DOI: 10.1080/16000870.2018.1445364. cit. on p. 20

Wikle, Christopher K. and L. Mark Berliner (June 2007). "A Bayesian tutorial for data assimilation". en. In: *Physica D: Nonlinear Phenomena*. Data Assimilation 230.1, pp. 1–16. DOI: 10.1016/j.physd.2006.09.017. cit. on p. 6

Xie, Xianhong and Dongxiao Zhang (June 2010). "Data assimilation for distributed hydrological catchment modeling via ensemble Kalman filter". en. In: *Advances in Water Resources* 33.6, pp. 678–690. DOI: 10.1016/j.advwatres.2010.03.012. cit. on p. 50

Yan, Wenlin, Qiuzhao Zhang, Lijuan Wang, et al. (Sept. 2020). "A Modified Kalman Filter for Integrating the Different Rate Data of Gyros and Accelerometers Retrieved from Android Smartphones in the GNSS/IMU Coupled Navigation". In: *Sensors* 20. DOI: 10.3390/s20185208. cit. on p. 1

Yashiro, H., K. Terasaki, Y. Kawai, et al. (Nov. 2020). "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)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1–10. DOI: 10.1109/SC41405.2020.00005. cit. on pp. 1, 25

Yoo, Andy B., Morris A. Jette, and Mark Grondona (2003). "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 44–60. cit. on p. 42

Zaharia, Matei (Feb. 2014). "An Architecture for Fast and General Data Processing on Large Clusters". PhD thesis. EECS Department, University of California, Berkeley. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html. cit. on p. 33

Zanúz, Henrique C., Bruno Raffin, Omar A. Mures, and Emilio J. Padrón (Nov. 2018). "In-transit molecular dynamics analysis with Apache flink". en. In: *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. Dallas Texas USA: ACM, pp. 25–32. cit. on p. 33

Zhang, D., H. Madsen, M. E. Ridler, et al. (Oct. 2016). "Multivariate hydrological data assimilation of soil moisture and groundwater". In: *Hydrol. Earth Syst. Sci.* 20.10, pp. 4341–4357. cit. on p. 24

Zhang, Lin, Yongzhu Liu, Yan Liu, et al. (2019). "The operational global four-dimensional variational data assimilation system at the China Meteorological Administration". en. In: *Quarterly Journal of the Royal Meteorological Society* 145.722, pp. 1882–1896. cit. on p. 24

Zheng, F., H. Zou, G. Eisnhauer, et al. (2013). "FlexIO: I/O middleware for Location-Flexible Scientific Data Analytics". In: *IPDPS'13*. Boston. cit. on p. 34

Zhou, Yuhua, Dennis McLaughlin, and Dara Entekhabi (Aug. 2006). "Assessing the Performance of the Ensemble Kalman Filter for Land Surface Data Assimilation". en. In: *Monthly Weather Review* 134.8. Publisher: American Meteorological Society Section: Monthly Weather Review, pp. 2128–2142. DOI: 10.1175/MWR3153.1. cit. on p. 50

List of Figures

2.1. Data Assimilation flow 6
2.2. The ensemble Kalman filter workflow 13
2.3. Importance sampling particle filter 17
2.4. Sequential importance resampling particle filter 18

3.1. Shadowed file writing 35

4.1. Melissa-DA three-tier architecture 38
4.2. Melissa-DA runner and server interactions 47
4.3. ParFlow pressure map 48
4.4. Traces of the Melissa-DA startup 51
4.5. Histograms of propagation walltimes for 100 members during multiple assimilation cycles 52
4.6. EnKF update phase scaling 53
4.7. Update phase walltime and compute hours per assimilation cycle for different server core counts 55
4.8. Efficiency of propagation phase and compute hours per assimilation cycle 56
4.9. Speedup during propagation phase for a ParFlow DA problem, running on a varying number of resources 57
4.10. Elasticity with Melissa-DA 59
4.11. Melissa-DA - PDAF comparism 64
4.12. Traces of PDAF and Melissa-DA 66
4.13. Traces of two Melissa-DA runs 67

5.1. Runners/server architecture 73
5.2. Two possible schedules of 24 propagation tasks of equal duration on four runners 76
5.3. The topography of the target domain of Europe for the WRF simulation. 83
5.4. Gantt chart of particle propagations executed by 15 (out of 511) randomly selected runners over 5 assimilation cycles 86
5.5. Histogram of 2,555 WRF model propagation times of one assimilaton cycle 87
5.6. Trace detailing the activity of a runner throughout one assimilation cycle 88
5.7. Server response times on various runner requests 89
5.8. Mean time to load or store particle states 90

5.9. Gantt chart with faults 91
5.10. Strong scaling efficiency using different numbers of particles with 63 runners 92
5.11. Weak scaling performance test 92
5.12. Non-speculative versus speculative scheduling on the dummy use case . 94
5.13. Non-speculative versus speculative scheduling on the WRF use case . . 96
5.14. Propagations done per runner with activated speculative propagations . 97

6.1. Experimentation workflow and used technologies 105
6.2. Example snippet from a Jupyter lab notebook 107

List of Tables

4.1. Large scale Melissa-DA runs 61
4.2. Startup times of PDAF and Melissa-DA 68

5.1. Experimental setting and performance overview at four different scales . 85
5.2. Statistical desciption of the dummy use case propagation runtimes . . . 93

Abstract

Prediction of chaotic and non-linear systems like weather or the groundwater cycle relies on a floating fusion of sensor data (observations) with numerical models to decide on good system trajectories and to compensate for non-linear feedback effects. Ensemble-based data assimilation (DA) is a major method for this concern. It relies on the propagation of an ensemble of perturbed model realizations (members) that is enriched by the integration of observation data. Performing DA at large scale to capture continental up to global geospatial effects, while running at high resolution to accurately predict impacts from small scales is computationally demanding. This requires supercomputers leveraging hundreds of thousands of compute nodes, interconnected via high-speed networks. Efficiently scaling DA algorithms to such machines requires carefully designed highly parallelized workflows that avoid overloading of shared resources. Fault tolerance is of importance too, since the probability of hardware and numerical faults increases with the amount of resources and the number of ensemble members.

Existing DA frameworks either use the file system as intermediate storage to provide a fault-tolerant and elastic workflow, which, at large scale, is slowed down by file system overload, or run large monolithic jobs that suffer from intrinsic load imbalance and are very sensible to numerical and hardware faults. This thesis elaborates on a highly parallel, load-balanced, elastic, and fault-tolerant solution, enabling it to run efficiently statistical, ensemble-based DA at large scale. We investigate two classes of DA algorithms, the ensemble Kalman filter (EnKF), and the particle filter algorithm with sequential importance resampling (SIR), and validate our framework under realistic conditions. Groundwater sensor data is assimilated using a regional hydrological simulation leveraging the ParFlow model. We efficiently run EnKF with up to 16,384 members on 16,240 compute cores for this purpose. A comparison with an existing state-of-the-art solution on the same domain, running 2,500 members on 20,000 cores, shows that our approach is about 50 % faster. We also present performance improvements running particle filter with SIR at large scale. These experiments assimilate cloud coverage observations into 2,555 members, i.e., particles, running the weather research and forecasting (WRF) model over the European domain. To manage the many experiments performed on various supercomputers, we developed a specific setup that we also present.

Keywords: Data Assimilation, Ensemble Based, In Situ Processing, EnKF, Particle Filter, High Performance Computing