Towards Zero-Waste Recovery and Zero-Overhead Checkpointing in Ensemble Data Assimilation

Kai Keller*, Adrian Cristal Kestelman†, Leonardo Bautista-Gomez‡
Barcelona Supercomputing Center (BSC-CNS)
Barcelona, Spain
Email: *kai.keller@bsc.es, †adrian.cristal@bsc.es, ‡leonardo.bautista@bsc.es

***Abstract*—Ensemble data assimilation is a powerful tool for increasing the accuracy of climatological states. It is based on combining observations with the results from numerical model simulations. The method comprises two steps, (1) the propagation, where the ensemble states are advanced by the numerical model and (2) the analysis, where the model states are corrected with observations. One bottleneck in ensemble data assimilation is circulating the ensemble states between the two steps. Often, the states are circulated using files. This article presents an extended implementation of Melissa-DA, an in-memory ensemble data assimilation framework, allowing zero-overhead checkpointing and recovery with few or zero recomputation. We hide the checkpoint creation using dedicated threads and MPI processes. We benchmark our implementation with up to 512 members simulating the Lorenz96 model using 10^9 gridpoints. We utilize up to 8 K processes and 8 TB of checkpoint data per cycle and reach a peak performance of 52 teraFLOPS.**

I. Introduction

In *High-Performance Computing* (HPC), numerical weather and climate simulations belong to the applications with the highest demand for computing resources. Those applications run at full scale on the world's largest supercomputers. Yet, the degree of resolution is nowhere near saturation. Terasaki, Miyoshi et al. have performed studies in 2015, using about 5,700 nodes of the K-Computer at RIKEN reaching 720 teraFLOPS [20], and in 2020 on the Fugaki supercomputer on more than 130,000 nodes reaching 79 petaFLOPS [28]. The amount of memory needed for such simulations already is in the Petabyte regime. High resolution weather and climate prediction towards less than 10km is expected to run at full scale on exascale systems [21].

An important part of numerical weather prediction (NWP) is *Data Assimilation* (DA) [25]. Some popular models using ensemble based DA are TOPAZ [8], NICAM-LETKF [25], CESM-DART [14], EC-EARTH [12] and IFS (ECMWF) [7]. DA combines numerical models and real world observations to achieve the most accurate description of the current system state. One cycle in ensemble data assimilation involves two steps: (1) *propagation* and (2) *analysis*. During the propagation, the current estimate of the state and the flow-dependent error covariance is *propagated* from t_i to t_{i+1}. The resulting state is called the *background state* (a.k.a. model or forecast state). The propagation is performed with the numerical climate model. During the analysis, the background state is improved by *assimilating* real world observations. The resulting state is called *analysis state* and represents the new best estimate of the true state. Propagation and analysis are repeated until the desired accuracy is reached or all the available observations are consumed.

DA is used for a number of reasons, an important one is operational NWP. This refers to continuous operated prediction systems (e.g., short-term weather forecasting or the prediction of extreme weather events). A key aspect of operational forecasting frameworks is the timely availability of the results, which becomes ever more challenging due to the increased resolution and grid size.

Melissa-DA [11] is a novel in-memory ensemble DA framework providing a simple interface for connecting numerical climate models. Traditionally, in-memory ensemble DA implements the state circulation with MPI. In Melissa-DA, the states are circulated via TCP leveraging the ZMQ [2] library. The framework is based on a server-runner architecture. Each runner and the server are executed on different resources (i.e., different cluster jobs). The server distributes the analysis states to the runners; the runners propagate the states until the next timestep and return the background states to the server. The server then performs the analysis and redistributes the analysis states for the next cycle. The ZMQ layer introduces elasticity into the framework: Runners can be dynamically added and removed, and each component of the framework can fail without affecting the others.

In this work, we present an effortless framework protection leveraging checkpoint/restart. The checkpoints fulfill the purpose of resiliency *and* hold the climate model results in form of shared HDF5 checkpoint files. The files contain both the background and analysis states. We use dedicated threads and MPI processes to overlap the checkpoint creation with the framework execution. We demonstrate that the overhead is effectively hidden behind the framework's normal operation (*zero-overhead* checkpoint). Our implementation manages further to recover from failures with none or few recomputations (*zero-waste* recovery). In addition, we derive a model that predicts the average cost of failures during continuous operation.

In the following sections, we provide a short introduction to the most important concepts necessary to understand this article II, present our implementation III, acknowledge related work to better understand the topic and introduce different concepts IV, present our experimental methods and goals V, the results of our experiments VI, discuss the results VII, and eventually conclude the article VIII.

 DOI 10.1109/HiPC53243.2021.00027

II. Background

A. Data Assimilation and the Ensemble Kalman Filter

An essential part of climate research is making predictions and reanalyses of environmental systems using numerical models. The governing equations of the systems are typically nonlinear and behave chaotically (i.e., are very sensitive to initial conditions). Therefore, to make accurate predictions, initial states near to the true state are necessary. The observational data, however, is sparse and afflicted with uncertainties. Consequently, observational data alone is insufficient for reliable predictions. A mean to reduce uncertainty is data assimilation. DA combines the error probability distributions for observation and model states to decrease the uncertainty. *Kalman Filtering* (KF) is among the most common techniques for DA. The foundation of the formalism is represented by the state space equations:

$$x_t = \mathcal{M}x_{t-1} + q_t, \quad q_t \sim \mathcal{N}(0, Q_t) \tag{1}$$

$$y_t = \mathcal{H}x_t + r_t, \quad r_t \sim \mathcal{N}(0, R_t) \tag{2}$$

Here, x_t represents the true state, $\mathcal{M}$ the model operator, y_t the observed state and $\mathcal{H}$ the observation operator. q_t and r_t are the respective errors, which are assumed to be *unbiased* and *Gaussian*. Hence, they follow a normal distribution with zero mean and covariance matrices Q_t and R_t.

The *Ensemble Kalman Filter* (EnKF), approximates the covariance matrices with the statistical moments of an ensemble of states, thus, the error covariance information is contained in the ensemble. This reduces the effective dimension of the covariance matrix from $N \times N$ to $N \times M$, with N being the state dimension and M the number of ensemble members. The method has been established in climate science over the past decades and has shown good results, despite the gaussian assumption. Besides the EnKF, there are various other techniques for DA. For instance, 4D-Var, particle filters or hybrids of EnKF and 4D-Var. Detailed introductions to the individual methods can be found in many textbooks and articles (for instance, [15], [27], [10], [9]).

B. Melissa-DA

Melissa-DA (DA for data assimilation) is a spin-off from Melissa, initially used for sensitivity analysis (SA) [26]. SA and ensemble based DA share some similarities. Both are based on sampling outcomes from model simulations and extract information about the system by statistical means; however, the goals and the underlying formalisms are very different. In both cases, though, the law of large numbers dictates a sufficient ensemble size to achieve accurate results. For DA, a 1K member ensemble, operating on a state with 10^9 variables, comprises at least 7.5 TB of memory (double precision), just for representing the ensemble states. In fact, for the whole ensemble, represented by background and analysis states, we need at least 15 TB. As mentioned earlier, the ensemble needs to be circulated between the propagation and analysis steps. In most cases, this takes place through the IO layer (a.k.a. *offline mode*). In that case, the propagation and analysis are performed on separate binaries and the ensemble is circulated through files. Another possibility is using the same binary for propagation and analysis while circulating the states through MPI (a.k.a. *online mode*). Both methods have certain advantages and drawbacks. The first method is intrinsically fault tolerant, since the last state ensemble is available on the files. However, the IO layer introduces a bottleneck. The second method provides better performance but misses the intrinsic fault tolerance, and introducing resiliency might again add considerable overheads.

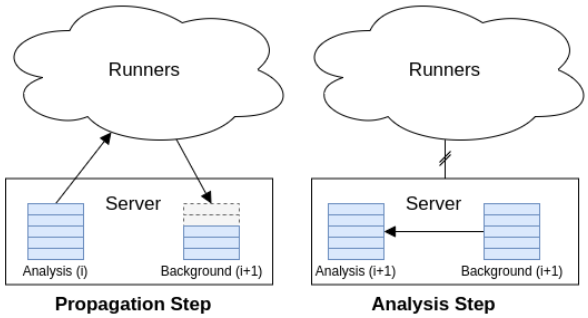


Fig. 1: Server-runner concept of Melissa-DA. Each iteration, the server distributes the analysis ensemble to the runners, which in turn compute the background state as input for the next analysis step.

Melissa-DA takes an intermediate approach. As in the first method, the propagation and analysis steps run in separate binaries, however, are executed simultaneously. The ensemble is circulated through the network using ZMQ instead of MPI. Melissa-DA is based on a server-client architecture. The clients (called *runners*) compute the background states and the server gathers the states and performs the analysis step. Leveraging ZMQ mitigates the effort of protecting the framework since each module can fail without affecting the others. Furthermore, it provides a high level of elasticity, as runners can be added and removed during the runtime. The state ensemble is exchanged between the server and runners directly, without depending on the file system. The server acts as a task scheduler during the propagation step, distributing the analysis state ensemble to the runners. The runners compute the background state and send it back to the server. Once the background state ensemble is complete, the server performs the data assimilation and thereupon distributes the analysis state ensemble back to the runners for the next iteration. The number of runners is adjustable and is typically chosen to be smaller than the ensemble size, thus, each runner will receive several states from the server during one epoch. Figure 1 visualizes the concept.

C. Asynchronous Checkpointing and Elastic Recovery

With asynchronous checkpointing, we refer to the checkpoint creation in two stages, while the second stage is performed *asynchronously* to the application. We refer to the first stage as **pre-processing** and to the second stage as **post-processing**. Asynchronous checkpointing can be applied to checkpoint techniques that involve further actions besides

storing the data to the file system. For instance, in partner-checkpointing, the checkpoint is stored locally on node storage and a copy is send to the partner node. Our extensions rely on the *Fault Tolerance Interface* (FTI) [5] checkpoint library. FTI provides several checkpoint types with different tradeoffs between speed and reliability. All the levels can be performed asynchronously, most importantly here, the creation of shared HDF5 files. During the pre-processing, the checkpoint data is stored on local node storage. During post-processing, FTI worker-processes consolidate the data to a shared HDF5 file on the *parallel file system* (PFS) in the background. The recovery from the shared file can be performed with a different number of processes [16] (a.k.a., elastic recovery). We will need this feature to recover the background states on the server side, as we will explain in the next Section (Section III-B).

III. Implementation

Our implementation involves modifications in all modules of the Melissa framework, the **Launcher**, **Server** and **Runner**. In the following paragraphs, we will gradually introduce the three modules along with our modifications.

A. Launcher

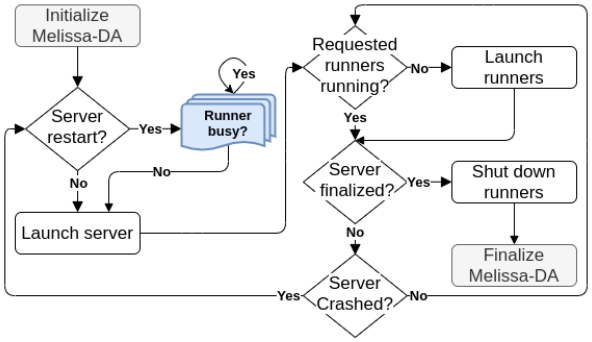


Fig. 2: Launcher workflow. Upon a runner failure, the launcher starts a new runner instance. Upon server failures, the launcher waits until the runners completed their computations and checkpoints and restarts the framework.

The launcher is the monitoring unit of the Melissa-DA framework. It starts the server and runner instances and monitors their operation. The launcher takes an essential role in our protection mechanism. The server and the runners are monitored using (1) the cluster scheduler (checking the job status) and (2) through timeouts or heartbeats. When the launcher notices the failure of runners, it manages their restart, and in case of a server failure, the restart of both the server and runner instances. Initially, upon server failures, the runners were immediately terminated, independently of the point of their execution. We intercepted this mechanism to allow a gradual shutting down of the runner instances. With our additions, the runners can finish computing the current background state and are gracefully shut down after storing the state to the PFS. With this, unless the server fails during the assimilation step or the checkpoint creation, the framework can resume execution where it has been left off. This leads to a smooth transition from failure to recovery with none or a minimum of recomputations (*zero-waste recovery*). Figure 2 shows the restart mechanism in detail.

B. Server

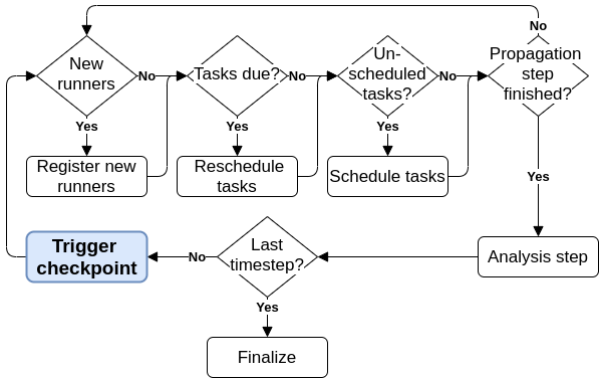


Fig. 3: Server mainloop showing the mechanism to register new runners, the scheduling and checkpointing.

The server workflow is divided into a propagation step followed by an analysis step. During the propagation step, the runners generate the background state ensemble, which serves as input for the next analysis step. This involves state transfers between server and runners, as the analysis is performed by the server. The server schedules and sends the analysis states as propagation tasks to the runners and the runners return the propagated analysis states (i.e., background states). When the whole ensemble has been propagated, the server performs the data assimilation, generating the analysis ensemble for the next iteration. Melissa-DA intrinsically provides basic fault tolerance to the framework, as the launcher detects and restarts failed runners (see Figures 2 and 3 for details). The server, however, still needs protection against failures. To provide server FT, we checkpoint the analysis states on the server. The checkpoints are created after each analysis step. In Section II-C we introduced the asynchronous HDF5 checkpoint creation in FTI. We leverage this feature to protect the analysis state ensemble asynchronously. We dedicate several of the server's MPI processes to FTI (further on called *heads*) to perform the post-processing in the background. Moreover, we added thread support to the remaining server processes to perform the pre-processing stage asynchronously as well. In that way, we can move the entire checkpoint creation into the background.

C. Runner

The runners apply the numerical climate model to propagate the analysis states to the next observation timestep. Each runner may need very large allocations comprising several nodes, depending on the complexity of the model (for instance 512 nodes for the NICAM atmospheric model [28]). In order to interface with the Melissa-DA framework, the simulation model needs to implement two API functions: (1) `melissa_init` and (2) `melissa_expose`. The state exchange between runner and server takes place during `melissa_expose`. The

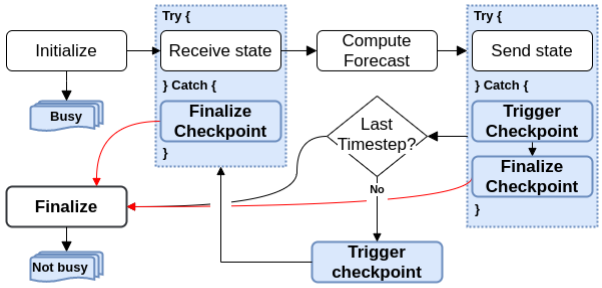


Fig. 4: Flowchart of the Melissa-DA runner workflow.

function essentially involves a send and receive operation. Runners receive analysis states from the server and return background states. Besides the checkpoint of the analysis states on the server, we perform checkpoints of the background states on the runners. Each runner operates on only one state at a time and thus, checkpoints only one state at a time. Since all runners execute simultaneously, the background state ensemble can, in a way, be checkpointed in parallel. To further improve the checkpoint performance on the runners, the when of the checkpoint creation is essential. The runners are idle between the send and the receive operation for some time, waiting for the next analysis state to arrive. This is where we place the checkpoint creation. We will see in the evaluation section that we can hide the cost for checkpointing entirely in that way. However, depending on the size of the states, the synchronous checkpoint creation may still exceed the idle time of the runners. Therefore, we apply asynchronous checkpointing for the runners as well. However, only for the post-processing, recall that we leverage threads for the asynchronous pre-processing on the server side. Our evaluation shows that the time for the inline pre-processing is much less than the runner's idle time, hence, it is not necessary to move the pre-processing to the background using threads. Figure 4 shows a diagram of the runner workflow. As we can see there, the runner-server interaction is performed within try blocks. If the runners detect uncommonly long send or receive calls, a server failure is assumed and the runners shut down gracefully after completing the checkpoint.

D. Recovery

With the presented checkpointing scheme, we hide the checkpoint overhead behind the framework's execution and we recover, in most cases, back to the point where the failure has occurred. The server typically uses a different domain decomposition than the runners. Since the recovery of both the analysis and the background state ensemble takes place on the server, we need to recover from the background states elastically, as the runners created them. We leverage the elastic recovery API for shared HDF5 files from FTI for this (see Section II-C).

IV. Related Work

In this section, we present research connected to our work. We focus on works in numerical climate modelling, selecting works that are similar to ours, or involve ideas that could be used for improvements. The first research we present is a fault-tolerant scheduler for the MITgcm-DART ensemble system [3]. The second work comprises a comprehensive overview of available FT mechanisms to protect numerical climate models [6].

A. Fault Tolerance for DART-MITgcm with Decimate

Decimate [18] is a Slurm extension aiming to facilitate the handling of HPC applications that include submission of multiple jobs. It provides mechanisms to include prolog and epilog to groups of jobs and allows inspecting the output by so-called *checker functions* at the end of the job execution. Toye et al. used Decimate to implement a scheduler providing automatic recovery and rescheduling of failed jobs for DART/MITgcm, the *Massachusetts Institute of Technology ocean general circulation model* (MITgcm) on the *Data Assimilation Research Testbed* (DART) [13]. Besides failures that occur either upon random soft errors or hardware failures, Toye et al. also include the handling of failures due to filter errors (e.g., collapsed ensemble states). According to Toye et al. about 3% of failures can be related to filter or other numerical errors. The failure detection in the presented scheduler extension is based on two scenarios. The first scenario includes hard failures. The second scenario includes filter errors, unphysical outcomes or numerical errors. The checker function takes the decision on which scenario has occurred based on the model-simulation output. If no output is found or incomplete, the framework decides for the first scenario. If an output was generated and contains unexpected or unphysical results, the framework chooses the second scenario. Depending on the scenario, the framework triggers the user defined failure handling.

B. Fault Tolerance Methods for Numerical Climate Models

Benacchio et al. [6] represents a comprehensive collection of FT methods in NWP on software and hardware level. We are more interested in the software level here, as it has a connection to our work. An engaging method presented is *interpolation-restart*. It describes the restart on a subset of the simulation data by interpolating the missing data from the available. This can be applied to models with data dependencies to reduce the amount of data in the checkpoint files. The authors also mention lossy compression for the checkpoint creation. Three methods are discussed in some detail *zero backup*, lost data is initialized with zeros, *multigrid backup*, where essentially only grid-points are stored that can be used to interpolate the missing points upon recovery, and *SZ compression*. Other methods discussed in the article (referred to as *system resiliency*) rely on fault-tolerant MPI implementations such as ULFM [4] and Fenix [24]. Again other rely on replication or message logging.

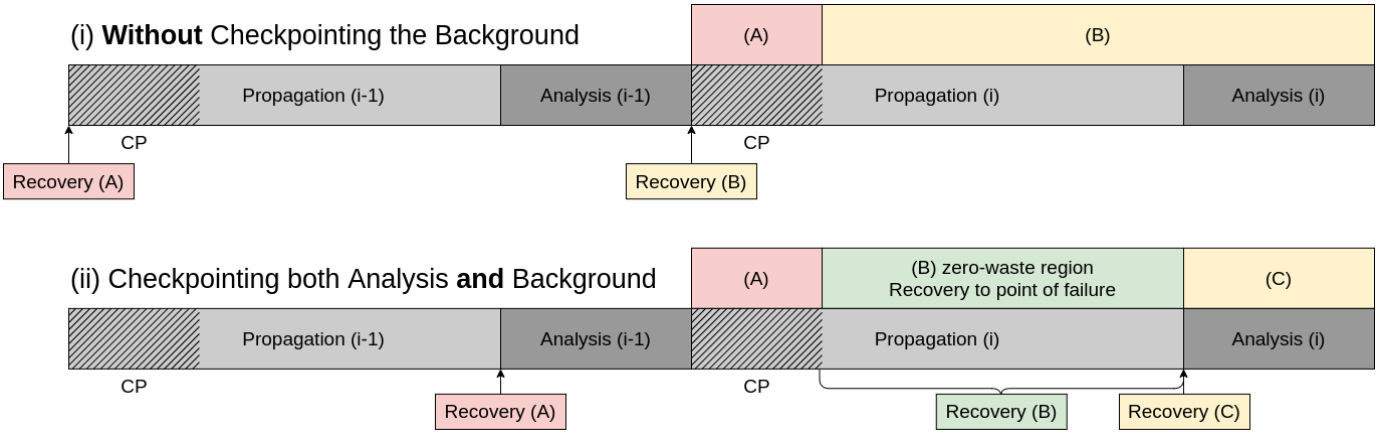


Fig. 5: On the top, we see the characteristic failure regimes when checkpointing only the analysis ensemble. Below the regimes, when checkpointing both background and analysis ensembles. **(ii)** Failures in region A result in a rollback to the end of the previous propagation. For failures in B we recover to the point where the failure occurred (zero-waste recovery), failures in C result in a rollback to the end of the propagation from the current iteration. The graphic illustrates, that failures in **(ii)** lead to significantly fewer recomputations than failures in **(i)**.

V. Methodology

This section discusses the dependency between recovery time and point of failure, describes the experiments we have performed and presents the methodology.

A. Failure Regions

The performance of a checkpoint/restart based protection is characterized by (a) the time for the checkpoint creation, T_{cp}, and (b) the revival time, T_{rev}. Here, T_{rev} comprises the time to recover, T_{rec}, and the time for recomputations, T_{com}. The total cost for a failure additionally includes the downtime and initialization. However, since those are subject to cluster and application type and independent of the type of recovery method, we do not consider them in our model. The revival time varies, depending on if the failure happens during the

A propagation step, *before* checkpoint completion,
B propagation step, *after* checkpoint completion,
C analysis step.

The zero-waste scenario is B. In this case, we restart where we have left off, recovering from the analysis and part of the background ensemble checkpoint. The worst case scenario is A, where we need to rollback to the end of the propagation step from the previous iteration, repeating the analysis step and part of the current propagation. We need to roll back to the end of the current propagation step for scenario C, merely repeating part of the analysis. Figure 5 summarizes the scenarios graphically. The figure also compares to the case when we checkpoint only the analysis ensemble. The revival times for the three regions can be expressed by:

$$T_{rev,A} = 2T_{rec} + T_{com,A} \quad , T_{com,A} = T_{ana} + \alpha_A T_{cp} \tag{3}$$

$$T_{rev,B} = (1 + \alpha_B)T_{rec} \quad , T_{com,B} = 0 \tag{4}$$

$$T_{rev,C} = 2T_{rec} + T_{com,C} \quad , T_{com,C} = \alpha_C T_{ana} \tag{5}$$

α_A, α_B and α_C have values between 0 and 1, indicating that for regions A and C the failure can happen at any point during the checkpoint or analysis step and for region B that we recover from only a fraction of the background state ensemble. Since the dimensions of analysis and background states are the same, we assume that their recovery per state takes the same amount of time [1]. we will see later in Section VI that this assumption is violated in our case, however, to take this into account, we just need to express $T_{rec} = T_{rec,for} + T_{rec,ana}$ explicitly [2]. The probability of failures inside the three regions is given by:

$$p_A = T_{cp} \times T_{iter}^{-1} \tag{6}$$

$$p_B = (T_{for} - T_{cp}) \times T_{iter}^{-1} \tag{7}$$

$$p_C = T_{ana} \times T_{iter}^{-1} \tag{8}$$

Where $T_{iter} = T_{for} + T_{ana}$. Note that we perform the checkpoint completely in the background using threads and dedicated MPI processes. That is why the iteration time only consists of the time for propagation and analysis steps. In Section VI we develop a model, using Equations (6) to (8), to predict the average revival time, $\langle T_{rev} \rangle$, for the continuous operation of the framework.

B. Experiments

We performed experiments where the

(T1) Server uses checkpoint threads and the
(T0) Server does not use checkpoint threads

The server always uses dedicated FTI processes for the asynchronous creation of the shared HDF5 file on the PFS. The modes from above are combined with

(H1) Runners using dedicated FTI processes and
(H0) Runners not using dedicated FTI processes

[1] The factor 2 in front of the recovery times in Equations (3) and (5) enters due to our implementation. Upon the server restart, we first recover the latest analysis ensemble and than check for available background state checkpoints. In future implementation this can be improved, avoiding the recovery of the previous analysis ensemble for cases A and C

[2] for and ana correspond to propagation (a.k.a., forecast) and analysis step respectively

We use the FTI terminology, *heads*, for the dedicated processes to label the experiments (heads → **H**). To provide a baseline, we also performed experiments without FTI (**NOFTI**). To identify the experiments that we refer to, we simply concatenate the labels. For instance, experiments with checkpoint threads on the server and head processes on the runners are labelled **H1T1**. We scaled the framework to ensembles with 64, 128, 256, and 512 members. The most relevant parameters of the experiments are listed in Table I.

C. Data Collection

We instrumented the code with timing events marking the beginning and the end of regions that we want to trace. Leveraging the UNIX system clock allows comparing events from different runners and the server. For the server side we instrumented: **initialization**, **propagation step**, **analysis step**, **checkpoint pre-processing**, **checkpoint post-processing**, **total execution time**, **recovery analysis**, **recovery background** and **total recovery time**. For the runners: **send state**, **receive state**, **model propagation**, **effective checkpoint time** and **idle time** (complement to model propagation). The effective checkpoint time comprises checkpoint pre and post-processing for **H0** (synchronous) and only the pre-processing for **H1** (asynchronous).

D. Failure Injection

To measure the recovery cost for regions A, B and C, we injected failures at the corresponding points into the server. To resemble a realistic situation, we injected the failures from the launcher using signals. In addition to the launcher's knowledge of the framework's status, we added a mechanism into FTI that enables the forwarding of the current checkpoint stage (i.e., idle, pre, or post-processing) to the launcher.

VI. Evaluation

In this section we discuss the benefits of checkpointing both ensembles towards only the analysis ensemble and afterward evaluate the checkpoint and recovery performance of our implementation.

A. Checkpointing Background and Analysis Vs. Only Analysis

This section opposes checkpointing background and analysis to only analysis ensemble, to justify the implementation effort and additional resource utilization. We start applying the considerations from Section V-A to the simpler case, protecting only the analysis ensemble. Note that we always need to roll back to the last available analysis ensemble. The formerly three regions become only two since Regions B and C are now identical (see also Figure 5):

A′ propagation step, *before* checkpoint completion and

B′ propagation or analysis step, *after* checkpoint completion.

The recovery for A′ takes place at the end of the analysis step from *two* iterations before, and for B′ at the beginning of the current propagation step. The respective revival times are:

$$T_{rev,A'} = T_{rec} + T'_{for} + T_{ana} + \alpha_{A'} T_{cp} \tag{9}$$

$$T_{rev,B'} = T_{rec} + T_{cp} + \alpha_{B'} \left(T'_{for} - T_{cp} + T_{ana} \right) \tag{10}$$

The times for the analysis step (T_{ana}), checkpoint (T_{cp}), and recovery (T_{rec}) are identical to the other case. However, the time for the propagation step (T'_{for}) might be shorter as we do not checkpoint on the runner side. We account for this by [3]:

$$\frac{T'_{for}}{T_{for}} = \delta \quad , \quad \text{with} \quad 0 \leq \delta \leq 1. \tag{11}$$

For a continuous operation of the framework, for instance in operational NWP, we can compute average values for the revival times using the probabilities from Equations (6) to (8):

$$\langle T_{rev} \rangle = p_A \langle T_{rev,A} \rangle + p_B \langle T_{rev,B} \rangle + p_C \langle T_{rev,C} \rangle \tag{12}$$

$$\langle T'_{rev} \rangle = p_{A'} \langle T_{rev,A'} \rangle + p_{B'} \langle T_{rev,B'} \rangle \tag{13}$$

The derivation of probabilities $p_{A'}$ and $p_{B'}$ is similar to p_A, p_B and p_C and omitted for brevity. With the following set of dimensionless parameters:

$$\mathsf{AF} = \frac{\langle T_{ana} \rangle}{\langle T_{for} \rangle} \quad , \quad \mathsf{RC} = \frac{\langle T_{rec} \rangle}{\langle T_{cp} \rangle} \quad , \quad \mathsf{CF} = \frac{\langle T_{cp} \rangle}{\langle T_{for} \rangle} \tag{14}$$

the dimensionless forms of Equations (12) and (13) become:

$$\tau_{rev} = \frac{\mathsf{AF}^2 + 2(2\mathsf{RC}+1)\mathsf{AF}\,\mathsf{CF} + (\mathsf{RC}+1)\mathsf{CF}^2 + 3\mathsf{RC}\,\mathsf{CF}}{2(\mathsf{AF}+1)} \tag{15}$$

$$\tau'_{rev} = (\mathsf{RC}+1)\mathsf{CF} + \frac{1}{2}(\mathsf{AF}+\delta) \tag{16}$$

Where $\tau_{rev} = {}^{\langle T_{rev} \rangle}/_{\langle T_{for} \rangle}$ and $\tau'_{rev} = {}^{\langle T'_{rev} \rangle}/_{\langle T_{for} \rangle}$ [4]. We chose $\alpha_A = \alpha_B = \alpha_C = \alpha_{A'} = \alpha_{B'} = 0.5$, assuming an unbiased point of failure. We can now compare the revival times using relations between the characteristic quantities (Equation (14)) rather than absolute values. The plots in Figure 6 show the relative reduction of the revival time for the three dimensionless parameters, depending on δ. We observe a reduction between 0% - 300% and a strong dependency on the parameter CF. This parameter determines the width of the zero waste region. In the limit, when the checkpoint takes as long as the propagation step (i.e., $\mathsf{CF} = 1$), the zero-waste region vanishes.

B. Climate Model

We adapted a parallel version of the Lorenz-96 model [19] for our experiments. The Lorenz models represent toy models to test the efficiency of data assimilation techniques. We use a 4-th order Runge-Kutta numerical solver for the systems evolution. The model differential equation reads:

$$\frac{dx_i}{dt} = (x_{i+1} - x_{i-2})x_{i-1} - x_i + F \tag{17}$$

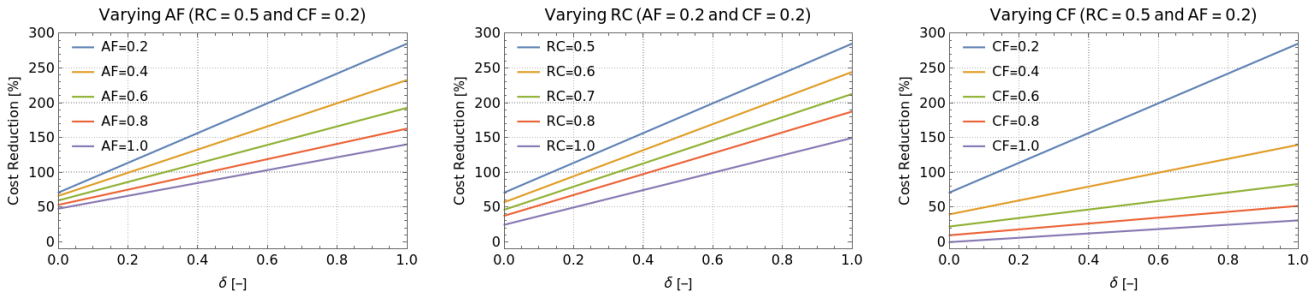
The linear term describes internal dissipation, the quadratic terms advection and the constant term an external forcing as parts of an atmospheric model.

[3] We will see later that the checkpoint creation for most cases can be hidden, yet, for very large state sizes overhead might still be imposed.

[4] Note that this model breaks down when the analysis state's checkpoint or the background state's recovery takes longer than the propagation step.

Parameter	Value	Members / Runners	Checkpoint (For+Ana)	Server Nodes	Server (FTI) Processes	Runner (FTI) Processes	Total Nodes	Total (FTI) Processes
State dimension	1024*1024*1024	64 / 16	1 TB	32	256 (128)	752 (16)	48	1008 (144)
Number of observations	21474 (0.002%)	128 / 32	2 TB	64	512 (256)	1504 (32)	96	2016 (288)
State size	8 GB	256 / 64	4 TB	128	1024 (512)	3008 (64)	192	4032 (576)
Observation size (Kb)	168 Kb	512 / 128	8 TB	256	2048 (1024)	6016 (128)	384	8064 (1152)

TABLE I: Parameters for the experiments (left) and scale of the experiments (right). The number of processes dedicated to FTI, in parenthesis, are a subset of the processes preceding the parenthesis.



(a) **(b)** **(c)**

Fig. 6: Speedup for the relative revival time $((\tau_{rev}-\tau'_{rev})/\tau_{rev})$ protecting both analysis *and* background, towards *only* protecting the analysis ensemble.

C. Experimental Setup

All experiments are performed on Marenostrum4 [1], the supercomputer at the Barcelona Supercomputing Center (BSC). The compute nodes are equipped with 48 cores/node (2 × Intel Xeon Platinum 8160), 96 GB of main memory and a 200 GB SSD.

The server uses at all scales 4 application and 4 FTI processes (i.e. heads) per node. Each MPI process is assigned 2 cores. In that way, the checkpoint thread can be executed on a separate core. The pre-processing on the server side takes place on the local SSDs. The server is memory bound and we cannot afford to perform the first checkpoint stage on the RAM disk. The checkpoints on the runner side never use checkpoint threads. The model is CPU bound and we can perform the checkpoint pre-processing in-memory. We use 1 FTI head per runner node.

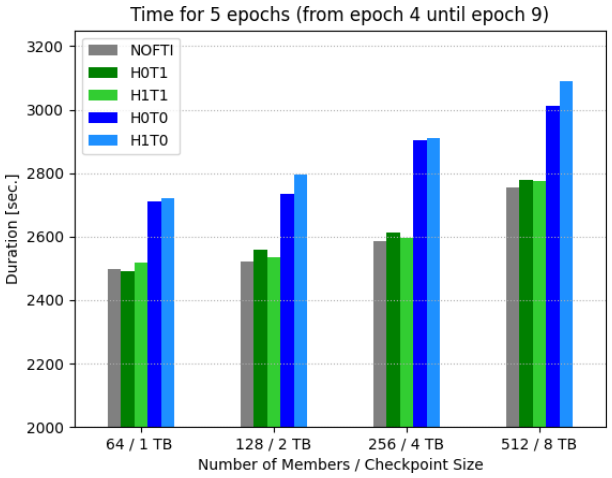


Fig. 7: Time for 5 epochs (from epoch 4 to 9). Bars in green show experiments with dedicated checkpoint threads and in bars in blue, without. Bars in gray show times for the experiments without protection (i.e., baseline).

D. Performance Evaluation during Runtime

Figure 7 gives a high-level view on the performance of the different setups, showing the execution times for five iterations. The execution times for the experiments are very similar at all scales, except for executions without server threads (H0T0 and H1T0), where a considerable amount of overhead is imposed. Note that this is a weak scaling scenario, i.e., the problem size per node remains the same, hence, the workload for the checkpoint pre-processing remains the same. The overhead for the cases without server threads, is consistent with the results shown in Figure 8. The time for the checkpoint pre-processing is at all scales about 45 seconds (i.e., 225 seconds for 5 iterations). Thus, utilizing checkpoint threads successfully hides this overhead.

The checkpoint of the background states on the runner, is placed after the send of background and before receiving the analysis state. This allows overlapping the checkpoint with the runner's idle period. At the time the runner sends the background state, the server might be busy with other runner requests and, in any case, the state needs to arrive on the server nodes before the server can react. Figure 9 visualizes this concept. Figure 10 shows histograms comparing the effective checkpoint time towards the runner's idle time. The upper plots show experiments with asynchronous checkpointing (H1) and the lower plots with checkpointing inline (H0). The lower plots show that some of the longer checkpoint times widen the runners idle period. We expect that for executions at very large scale this leads to a significant overhead. On the other hand, the pre-processing time in the asynchronous case is expected to be independent of the scale (compare Figure 8).

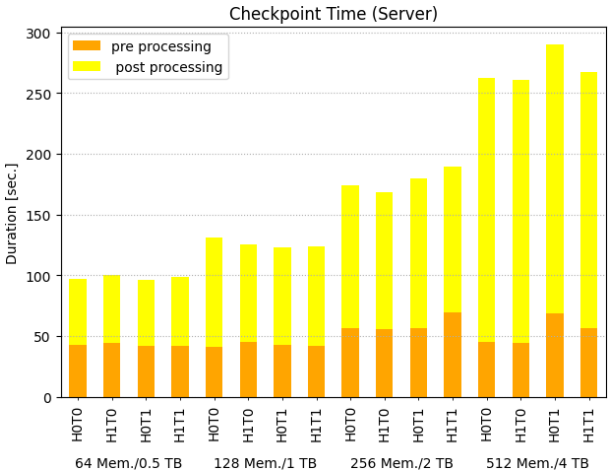


Fig. 8: Time for the pre (i.e., node SSD) and post-processing (i.e., asynchronous shared HDF5 file creation) for checkpoints on the server.

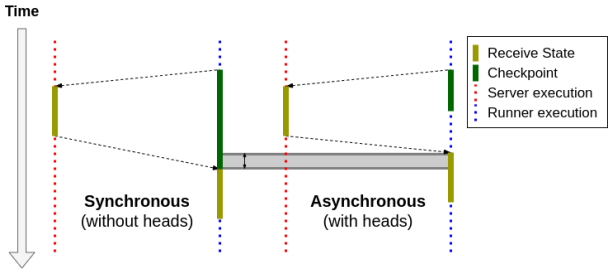


Fig. 9: Communication graph for state circulation between runner and server. The right showing the case with dedicated FTI processes (i.e., asynchronous checkpointing) and the left, without (i.e., synchronous checkpointing).

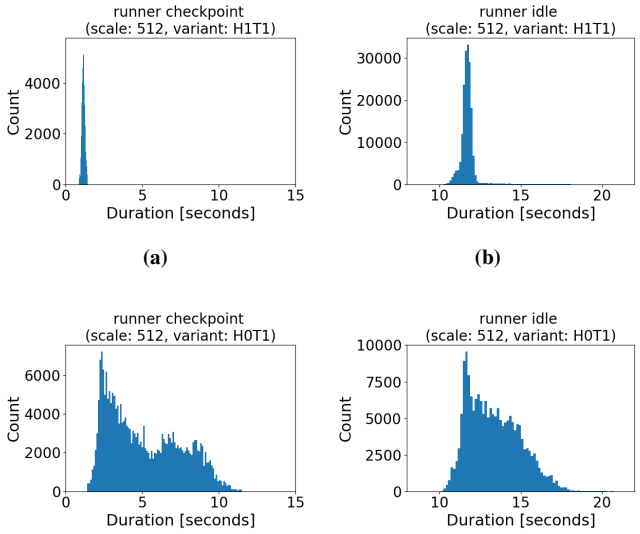


Fig. 10: Histograms of the runner idle and checkpoint times. The runner idle period is the time between two model propagations. The checkpoint time is part of the idle time. The upper plots show executions with FTI heads and the lower, without. We observe that synchronous checkpointing broadens the runners idle time.

E. Performance Evaluation Recovery

The results of our experiments show that we successfully overlap checkpointing with the propagation (server) and the state exchange (runner). Hence, the framework is protected without a palpable penalty on execution time. However, the checkpoint duration affects the average cost for failures. In Section V-A we derived revival times for failures, depending on the region they take place in. Equations (6) to (8) give the associated probabilities for failures happening in those regions. Failures in region A are the most costly ones, followed by region C and B (zero-waste region). p_A and p_B both depend on the checkpoint duration (i.e., the full checkpoint duration, including pre and post-processing), p_C only depends on the assimilation and propagation time and is independent of the fault tolerance implementation.

It is easy to see that by minimizing the checkpoint time, we minimize p_A and maximize p_B and therefore, minimizing the probability for failures in the most costly region and maximizing the probability in the zero-waste region. Hence, to provide minimal revival times, it is important to ensure both good checkpoint and recovery performance. Figure 11 shows traces of six executions including a failure and the recovery. The first row shows executions with checkpointing both the background and analysis state ensemble. The second row shows executions with checkpointing only the analysis ensemble. The server and runner traces are separated by a dotted blue line. The vertical dotted lines indicate the failure and revival points. We also marked the passive recovery time (i.e., the time from the crash until the restart of the framework) and the active recovery time (i.e., the time from server initialization until the revival point). The passive recovery time consists mostly of the time the runners wait for the server reply, i.e., the time until the runners notice the server crash. Note that this time can be reduced by adjusting the timeout. With a little more effort the waiting time can be eliminated entirely by implementing a server polling and by relying on the launcher to notify the runners of the server crash. Hence, to make a fair comparison, we compare the active recovery time. We can see that indeed the revival times are faster when checkpointing both ensembles. The trace showing the failure in region B (top row, second trace) demonstrates that indeed the propagation is resumed from the point of failure. We can see that especially for region A and C, the revival times without protecting the background are much longer, as they include the repetition of the entire or most of the propagation step.

We can determine the probabilities for failures in the various regions using Equations (6) to (8) (recall, the regions are shown in Figure 5). Using Equations (15) and (16), we can also compute the average revival times for both cases (i.e., checkpointing both ensembles and only the analysis ensemble). Table II lists the values for 64, 128, 256 and 512 members. As the recovery times for 256 members already have been rather long, we measured 390 seconds for the recovery of one ensemble to only 190 seconds for the checkpoint, we projected the times for executions with 512, to save computing

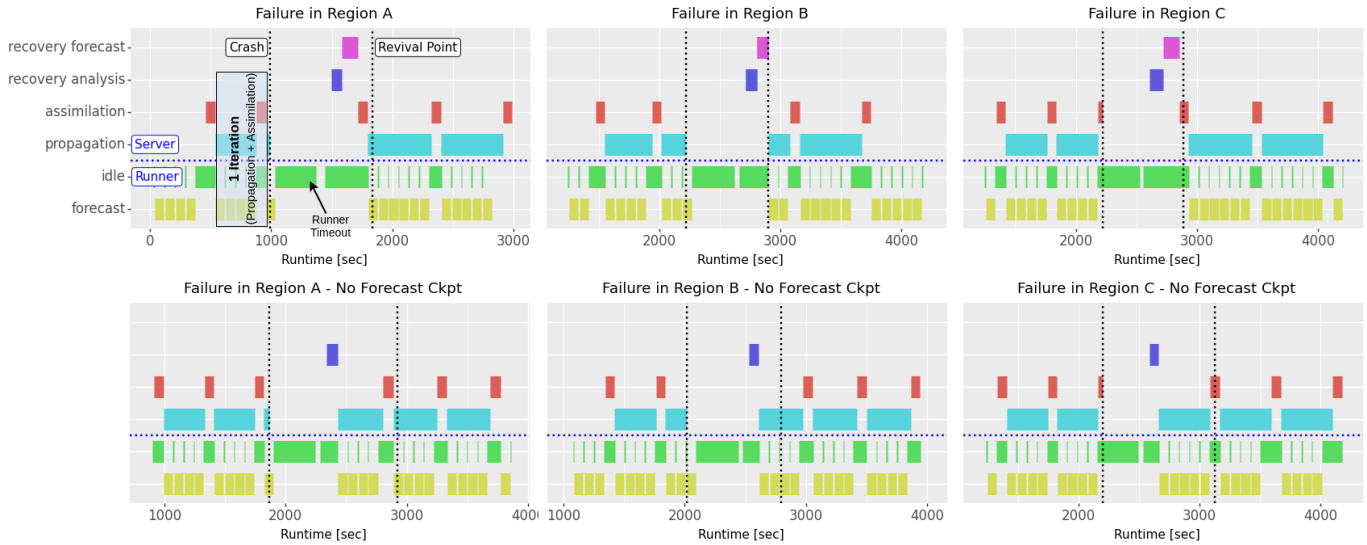


Fig. 11: Gantt charts showing the server and runner execution (i.e., one runner instance). The charts show execution, failures in region A, B and C, and the recovery. The upper row showing the cases when protecting both background *and* analysis ensembles and the lower, *only* protecting the analysis ensemble.

	64 (H1T1)	**128 (H1T1)**	**256 (H1T1)**	**512 (H1T1)**
P_A [%]	19.47	24.35	36.48	48.17
P_B [%]	66.93	61.80	49.53	37.21
P_C [%]	13.60	13.85	13.99	14.62
$P_{B'}$ [%]	80.53	75.65	63.52	51.83
AF	0.16	0.16	0.16	0.17
RC	1.01	1.20	2.05	2.89
CF	0.23	0.28	0.42	0.56
$\langle T_{rev} \rangle$ [sec]	192.15	287.02	747.98	1514.98
$\langle T'_{rev} \rangle$ [sec]	449.16	525.00	838.56	1320.12
speedup [%]	133.76	82.91	12.11	-12.86

TABLE II: Probabilities (Equations (6) to (8)), revival times (avg.), $T_{for}\tau_{rev}$ (Equations (12) and (13)), and speedup ($(\tau_{rev}-\tau'_{rev})/\tau_{rev}$). The text in blue color indicates that we estimated the recovery time for 512 members.

resources. However, even for the measured recovery times we observe a speedup for executions below 512 members, when checkpointing both ensembles.

VII. Discussion

Our results show that we protect the Melissa-DA framework from failures with a minimum in recomputations at large scale. Moreover, our protections do not affect the runtime, as they are performed completely hidden behind the framework's normal execution. As we stated in the introduction, Melissa-DA takes an intermediate approach between an online and offline setup. The offline setup uses different binaries for propagation and analysis and circulates the states through the PFS using restart files. In an online setup, propagation and analysis run inside the same binary and circulate the states through MPI. Melissa-DA performs propagation and analysis on different binaries, hence, reduces the size of dependent failure domains, however, circulates the states directly between propagation and analysis through the network, avoiding the staging through the PFS. We added fault tolerance to the framework in form of global checkpoint files using HDF5 to make the data available for reanalyses and data-processing. With our additions, Melissa-DA possesses the best from both setups. Fast state circulation as in the online setup and global checkpoint files as in the offline setup, without additional overhead.

We also identified starting points for further improvements. For instance, the ensemble states are decomposed among all server ranks. Consequently, the parts of the states that reside on the ranks become smaller with increasing server size. This leads to a high level of fragmentation inside the global checkpoint files, which results into an unfavorable scaling behavior of the checkpoint post-processing (see Figure 8). This also affects the recovery. In Figure 5, we can see that by minimizing the total checkpoint cost, we maximize the zero-waste region and minimize the worst-case region. Thus, by solving this and other issues, we further improve the operational performance of the framework, minimizing the average revival time.

Shorter checkpoint times can already be achieved in the current implementation, by changing to a faster checkpoint method. For instance, the standard FTI checkpoint format (i.e., POSIX IO and one file per process), or differential checkpointing, both will reduce the total time for the checkpoint completion. This becomes especially interesting for cases when the checkpoint data is not needed for analyses. To improve the total checkpoint performance, without changing to a faster checkpoint method, one could leverage burst buffer storage [17], [23], [22], if available on the cluster.

VIII. Conclusion

We presented a novel checkpoint-restart implementation for Melissa-DA, a distributed framework for ensemble based data assimilation. Melissa-DA keeps the ensemble states in memory and upon failures the simulation state is lost. Our implementation protects the framework from this. We showed that our implementation manages checkpointing the full ensembles of

background and analysis states, without imposing palpable overhead. Moreover, we showed that by checkpointing both ensembles, we manage to recover without recomputations when the failure takes place in the zero-waste region. Since we checkpoint every epoch, the recomputation will be always less then one epoch. In fact, the maximum recomputation results from failures in region A (compare Figure 5) and is bound by the time for one assimilation step plus the time to complete the checkpoint. The checkpoints are stored leveraging the HDF5 IO interface of FTI. The checkpoint cost is hidden behind the frameworks execution, although, there is still some work to be done to improve both checkpoint and recovery performance (see Section VII). We performed experiments with a state dimension of 10^9 and 2×10^4 observations. We scaled the experiments to 512 ensemble members with a total checkpoint size of 8 TB. Runners and server together executed on 8064 processes and the assimilation step reached 52 teraFLOPS. We estimated the average revival time (including recomputation and recovery) per failure for 512 member executions to be 1514 seconds. Considering a MTBF of 24 hours, this corresponds to less than 2% additional time, compared to a failure free execution.

IX. Acknowledgements

Part of the research presented here has received funding from the Horizon 2020 (H2020) funding framework under grant/award number: 824158; Energy oriented Centre of Excellence II (EoCoE-II). The present publication reflects only the authors views. The European Commission is not liable for any use that might be made of the information contained therein.

References

[1] Support Knowledge Center @ BSC-CNS - https://www.bsc.es/user-support/mn4.php.

[2] ZeroMQ - https://zeromq.org/.

[3] A fault-tolerant HPC scheduler extension for large and operational ensemble data assimilation: Application to the Red Sea. *Journal of Computational Science*, 27:46–56, July 2018.

[4] Ulfm 2.0, fault tolerance research hub, 2019.

[5] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2011.

[6] Tommaso Benacchio, Luca Bonaventura, Mirco Altenbernd, Chris D Cantwell, Peter D Düben, Mike Gillard, Luc Giraud, Dominik Göddeke, Erwan Raffin, Keita Teranishi, et al. Resilience and fault-tolerance in high-performance computing for numerical weather and climate prediction. *International Journal of High Performance Computing Applications*, 2020.

[7] A Benedetti, J Morcrette, O Boucher, A Dethof, R Engelen, M Fisher, H Flentjes, N Huneeus, L Jones, J Kaiser, et al. *Aerosol analysis and forecast in the ECMWF integrated forecast system: Data assimilation*. ECMWF, 2008.

[8] L Bertino and K A Lister. The topaz monitoring and prediction system for the atlantic and arctic oceans. *Journal of Operational Oceanography*, 1(2):15–18, 2008.

[9] Geir Evensen. Sequential data assimilation with a nonlinear quasi-geostrophic model using monte carlo methods to forecast error statistics. *Journal of Geophysical Research: Oceans*, 99(C5):10143–10162, 1994.

[10] Geir Evensen. *Data Assimilation: The Ensemble Kalman Filter*. Springer-Verlag, Berlin, Heidelberg, 2006.

[11] Sebastian Friedemann and Bruno Raffin. An elastic framework for ensemble-based large-scale data assimilation, 2020.

[12] Wilco Hazeleger, Camiel Severijns, Tido Semmler, Simona Ştefănescu, Shuting Yang, Xueli Wang, Klaus Wyser, Emanuel Dutra, José M Baldasano, Richard Bintanja, et al. Ec-earth: a seamless earth-system prediction approach in action. *Bulletin of the American Meteorological Society*, 91(10):1357–1364, 2010.

[13] Ibrahim Hoteit, Tim Hoar, Ganesh Gopalakrishnan, Nancy Collins, Jeffrey Anderson, Bruce Cornuelle, Armin Khl, and Patrick Heimbach. A MITgcm/DART ensemble analysis and prediction system with application to the Gulf of Mexico. *Dynamics of Atmospheres and Oceans*, 63:1–23, September 2013.

[14] Alicia R. Karspeck, Gokhan Danabasoglu, Jeffrey Anderson, Svetlana Karol, Nancy Collins, Mariana Vertenstein, Kevin Raeder, Tim Hoar, Richard Neale, Jim Edwards, and Anthony Craig. A global coupled ensemble data assimilation system using the Community Earth System Model and the Data Assimilation Research Testbed. *Quarterly Journal of the Royal Meteorological Society*, 144(717):2404–2430, 2018. _eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.3308.

[15] Matthias Katzfuss, Jonathan R. Stroud, and Christopher K. Wikle. Understanding the ensemble kalman filter. *The American Statistician*, 70(4):350–357, 2016.

[16] Kai Keller, Konstantinos Parasyris, and Leonardo Bautista-Gomez. Design and Study of Elastic Recovery in HPC Applications, 2020.

[17] Donghun Koo, Jaehwan Lee, Jialin Liu, Eun-Kyu Byun, Jae-Hyuck Kwak, Glenn K. Lockwood, Soonwook Hwang, Katie Antypas, Kesheng Wu, and Hyeonsang Eom. An empirical study of I/O separation for burst buffers in HPC systems. *Journal of Parallel and Distributed Computing*, 148:96–108, February 2021.

[18] Samuel Kortas. Welcome to decimate's documentation!, 2018.

[19] Edward N Lorenz. Predictability: A problem partly solved. In *Proc. Seminar on predictability*, volume 1, 1996.

[20] Takemasa Miyoshi, Keiichi Kondo, and Koji Terasaki. Big Ensemble Data Assimilation in Numerical Weather Prediction. *Computer*, 48(11):15–21, November 2015. Conference Name: Computer.

[21] Philipp Neumann, Peter Dben, Panagiotis Adamidis, Peter Bauer, Matthias Brck, Luis Kornblueh, Daniel Klocke, Bjorn Stevens, Nils Wedi, and Joachim Biercamp. Assessing the scales in numerical weather and climate predictions: will exascale be the rescue? *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 377(2142):20180148, April 2019. Publisher: Royal Society.

[22] L. Pottier, R. F. da Silva, H. Casanova, and E. Deelman. Modeling the Performance of Scientific Workflow Executions on HPC Platforms with Burst Buffers. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 92–103, September 2020. ISSN: 2168-9253.

[23] Wolfram Schenck, Salem El Sayed, Maciej Foszczynski, Wilhelm Homberg, and Dirk Pleiter. Evaluation and Performance Modeling of a Burst Buffer Solution. *ACM SIGOPS Operating Systems Review*, 50(2):12–26, January 2017.

[24] Keita Teranishi, Marc Gamell, Rob Van der Wijingarrt, and Manish Parashar. Fenix a portable flexible fault tolerance programming framework for mpi applications. 3 2018.

[25] Koji Terasaki, Masahiro Sawada, and Takemasa Miyoshi. Local Ensemble Transform Kalman Filter Experiments with the Nonhydrostatic Icosahedral Atmospheric Model NICAM. *Sola*, 11:23–26, 2015.

[26] Théophile Terraz, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin. Melissa: Large Scale In Transit Sensitivity Analysis Avoiding Intermediate Files. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, pages 1 – 14, Denver, United States, November 2017.

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

[28] H. Yashiro, K. Terasaki, Y. Kawai, S. Kudo, T. Miyoshi, T. Imamura, K. Minami, H. Inoue, T. Nishiki, T. Saji, M. Satoh, and H. Tomita. A 1024-member ensemble data assimilation with 3.5-km mesh global weather simulations. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, Los Alamitos, CA, USA, nov 2020. IEEE Computer Society.