Resilience for Large Ensemble Computations

Kai Rasmus Keller

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Supervisor: Dr. Leonardo Bautista Gomez
Tutor: Prof. Dr. Adrian Cristal Kestelman

May 26, 2022

To my beloved father Edgar Keller, who introduced me into the hidden mysteries of our universe, and who still never wanted to believe in gravitons…

Resum

Amb l'increment de les capacitats de còmput dels supercomputadors, es poden simular models de sistemes físics encara més detallats, i es poden resoldre problemes de més grandària en qualsevol tipus de sistema numèric. Durant els últims vint anys, el rendiment dels clústers més ràpids ha passat del domini dels teraFLOPS (ASCI RED: 2.3 teraFLOPS) al domini dels pre-exaFLOPS (Fugaku: 442 petaFLOPS), i aviat tindrem el primer supercomputador amb un rendiment màxim que sobrepassa els exaFLOPS (El Capitan: 1.5 exaFLOPS). Les tècniques d'ensemble experimenten un renaixement amb la disponibilitat d'aquestes escales tan extremes. Especialment les tècniques més noves, com els filtres de partícules, se'n beneficiaran. Els mètodes d'ensemble actuals en climatologia, com els filtres d'ensemble de Kalman, exhibeixen una dependència lineal entre la mida del problema i la mida de l'ensemble, mentre que els filtres de partícules mostren una dependència exponencial. No obstant, juntament amb les oportunitats de poder computar massivament, apareixen desafiaments com l'alt consum energètic i la necessitat de tolerància a errors. El temps de mitjana entre errors es redueix amb el nombre de components del sistema, i s'espera que els errors s'esdevinguin cada poques hores a exaescala. En aquesta tesis, explorem i desenvolupem tècniques per protegir grans càlculs d'ensemble d'errors. Presentem noves tècniques en punts de control diferencials, recuperació elàstica, punts de control totalment asincrònics i compressió de punts de control. A més, dissenyem i implementem un filtre de partícules tolerant a errors amb captació i emmagatzematge en caché de partícules de manera preventiva. I finalment, dissenyem i implementem un marc per la validació automàtica i l'aplicació de compressió amb pèrdua en l'assimilació de dades d'ensemble. En total, en aquesta tesis presentem cinc contribucions, les dues primeres de les quals milloren les tècniques de punts de control més avançades, mentre que les tres restants aborden la resiliència dels càlculs d'ensemble. Les contribucions representen tècniques independents de tolerància a errors; no obstant, també es poden utilitzar per a millorar les propietats de cadascuna. Per exemple, utilitzem la recuperació elàstica (segona contribució) per a mitigar la resiliència en un marc d'assimilació de dades d'ensemble en línia (tercera contribució), i construïm el nostre marc de validació (cinquena contribució) sobre la nostra implementació del filtre de partícules (quarta contribució). A més, demostrem que les nostres contribucions milloren la resiliència i el rendiment amb experiments en diverses arquitectures, com processadors Intel, IBM i ARM.

Abstract

With the increasing power of supercomputers, ever more detailed models of physical systems can be simulated, and ever larger problem sizes can be considered for any kind of numerical system. During the last twenty years the performance of the fastest clusters went from the teraFLOPS domain (ASCI RED: 2.3 teraFLOPS) to the pre-exaFLOPS domain (Fugaku: 442 petaFLOPS), and we will soon have the first supercomputer with a peak performance cracking the exaFLOPS (El Capitan: 1.5 exaFLOPS). Ensemble techniques experience a renaissance with the availability of those extreme scales. Especially recent techniques, such as particle filters, will benefit from it. Current ensemble methods in climate science, such as ensemble Kalman filters, exhibit a linear dependency between the problem size and the ensemble size, while particle filters show an exponential dependency. Nevertheless, with the prospect of massive computing power come challenges such as power consumption and fault-tolerance. The mean-time-between-failures shrinks with the number of components in the system, and it is expected to have failures every few hours at exascale. In this thesis, we explore and develop techniques to protect large ensemble computations from failures. We present novel approaches in differential checkpointing, elastic recovery, fully asynchronous checkpointing, and checkpoint compression. Furthermore, we design and implement a fault-tolerant particle filter with pre-emptive particle prefetching and caching. And finally, we design and implement a framework for the automatic validation and application of lossy compression in ensemble data assimilation. Altogether, we present five contributions in this thesis, where the first two improve state-of-the-art checkpointing techniques, and the last three address the resilience of ensemble computations. The contributions represent stand-alone fault-tolerance techniques, however, they can also be used to improve the properties of each other. For instance, we utilize elastic recovery (2nd contribution) for mitigating resiliency in an online ensemble data assimilation framework (3rd contribution), and we built our validation framework (5th contribution) on top of our particle filter implementation (4th contribution). We further demonstrate that our contributions improve resilience and performance with experiments on various architectures such as Intel, IBM, and ARM processors.

Acknowledgments

I want to thank Leonardo Bautista Gomez and Adrian Cristal Kestelmann for supervising the thesis and giving me priceless advice. I want to thank especially Leonardo, for giving me the opportunity of this doctorate, for always having an open ear, and always treating me respectful. I also want to thank my wife, Maialen Lehane Lezaun, who helped me to endure difficult and frustrating moments. I want to thank my sister Maren Keller, who especially in the beginning, provided me with advice and assistance, while I was writing my first articles. I want to thank my mother Gerda Keller, and my sister Solveigh Keller for always having an open door for me. I want to thank Konstantinos Parasyris (Dinos), teaching me many lessons. I want to thank the EoCoE-II project for funding the doctorate, and Sebastian Lührs for his outstanding work as work-package leader. I want to thank Sebastian Friedemann, Bruno Raffin, and Yen-Sen Lu, for the exciting collaboration on working with MelissaDA. I want to thank EuroLab-4-HPC funding a one-month stay at the Jülich Supercomputing Centre, and I want to thank the NASDAC project for funding a two-month stay at the Argonne National Laboratory in Chicago. I want to thank my colleges from my time at argonne, Shintaro Iwasaki, and Rohit Zambre. I want to thank RIKEN for the seven-month internship and access to Fugaku. I want to thank Mohamed Wahib, Balazs Gerofi, and Hisashi Yashiro, for supervising the internship and giving me advice. I want to thank PRACE Summer Of HPC for giving me the opportunity to supervise two students. I want to thank the two students that participated, Kevser Ildes and Athanasios Kastoras. I want to thank Gemma Pidelaserra Martí for translating the abstract into catalan. And finally, I want to thank my co-workers and colleges from other centers for their advice and good collaboration. Especially (in more or less random order): Albert Njoroge Kahira, Julian Pavon, Saber Nabavi, Mikel Cortes Goicoechea, Tarun Harjani Daryanani, Sawsane Ouchtal, Alexandre de Limas Santana, Sanem Arslan Yılmaz, and many others.

List of Figures

2.1 Example showing 4 levels of different reliability and speed 6

2.2 Asynchronous checkpoint creation. The first step (pre-processing) is performed inline, and the second step (post-processing) is performed by dedicated processes asynchronously to the application. 7

2.3 (*Left*) Lorenz Attractor $L(\sigma = 28, \rho = 10, \beta = 8/3)$ with initial state at $\mathbf{x} = (0, 1, 1.05)$ in blue and slightly perturbed with initial state at $\mathbf{x} = (0.00001, 1, 1.05)$ in orange. (*Right*) The Root Mean Square Error (RMSE) between the first and second Lorenz system. 8

3.1 Exponential increase in peak performance (TOP500 lists). 13

3.2 Exponential increase in the average number of cores (TOP500 lists). 14

5.1 The FTI-FF structure can be resolved into three layers of abstraction: The first layer comprises global metadata (for instance, filesize and datasize) and checkpoint data (protected variables), the latter is build from M chunks C_i of which each contains N_i variable blocks. The actual application data are stored in the variable blocks. Each chunk also contains metadata describing the layout of the variable blocks. 23

5.2 The streaming DCP file structure contains M layers L_i; one layer for each checkpoint update. The layers L_i contain the N_i variable blocks that contain the differential variable update. each layer also contains metadata holding information such as the layer size, block size, and checkpoint ID. 24

5.3 DCP detection and update scheme. Processes left to the blue circle happen before and processes to the right after the DCP update. The circle indicates the dirty region request loop 25

5.4 The bars show the estimated Differential Checkpointing (DCP) threshold, i.e. the ratio of dirty to total data we need, to make the DCP operation beneficial. The left axis shows the dirty data ratio, η, the right axis shows the corresponding value of ρ (ratio between the hash time, $t_{b,h}$, and I/O time, $t_{b,w}$, for block size b). The experiment has been performed with 768 and 2400 processes and 1GB per rank. 29

5.5 Data differences per rank in checkpoints at different timesteps in LULESH. The x-axis shows the ranks (512 in total), and the y-axis shows the percentages written in percentage. 31

5.6 Data differences per rank in checkpoints at different timesteps in xPic. The x-axis shows the ranks (192 in total), and the y-axis shows the percentages written in percentage. 32

5.7 Data differences per rank in checkpoints at different timesteps in Heat2D. The x-axis shows the ranks (768 in total), and the y-axis shows the percentages written in percentage. 33

5.8 Measured and estimated speedup/overhead of DCP updates. The green background indicates the region where we have speedup and the red region indicate overhead. $\tau/t_w = 0$ corresponds to the threshold (i.e., the full Checkpointing (CP) baseline) The datasets with the label *corrected*, refer to measurements that used a buffer to collect small chunks in order to avoid small chunck writes. 36

5.9 Cumulative distribution function (CDF) for chunk sizes of contiguous dirty regions during DCP updates. 37

6.1 The entire grid of the data used by the application. The different gray scales indicate the domains of the MPI processes. (a) shows the initial execution with 4 MPI processes and (b) the execution with 3 processes after the elastic recovery. 43

6.2 Top: Relative checkpoint overhead. Bottom: Relative recovery overhead. For xPic we separated the recovery overhead into read and re-distribution of the particles.The difference in latency between online and offline recovery depending on the number of nodes. 49

6.3 Relative checkpoint overhead for our new extensions for varying checkpoint data per process (strong scaling). 51

6.4 Relative checkpoint overhead for our new extensions for varying number of nodes and constant checkpoint data per process (Weak scaling). 52

6.5 The difference in latency between online and offline recovery depending on the number of nodes. 53

6.6 Relative additional overhead for executions on fewer nodes with constant total load. The x-axis shows the percentage of nodes lost upon failure. 55

6.7 Comparison of the 2 techniques (PP and PC) to organize the particle data in the checkpoint file. Recovery on 16 nodes means recovery on the same number of nodes, whereas recovery on 15 nodes means elastic recovery on a reduced number of nodes (i.e., simulates the loss of 1 node upon failure). 56

7.1 Server-runner concept of MelissaDA. 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. 64

7.2 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. 65

7.3 Server mainloop showing the mechanism to register new runners, the scheduling and checkpointing. 66

7.4 Virtual cluster, generated by FTI if deployed in local test mode. The number of processes per virtual node is set to 4 and we have M processes per physical node. This leads to M/4 dedicated FTI processes per node. 67

7.5 Flowchart of the MelissaDA runner workflow. 67

7.6 On the top, we see the characteristic failure regimes if checkpointing only the analysis ensemble. Below, the regimes if checkpointing both the background and analysis ensemble. **(i)** Failures in region A lead to a roll back to the beginning of the propagation step from the previous iteration, failures in B to a rollback to the beginning of the propagation of the current iteration. **(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). . 72

7.7 Time for 5 epochs (from epoch 4 to 9). Bars in green show the runtime for experiments with, and bars in blue without dedicated checkpoint threads on the server. Bars in gray show runtimes for the experiments without protection (baseline). The percentages above the bars indicate the overhead compared to the baseline. 74

7.8 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). 74

7.9 Time for the pre (i.e., node SSD) and post-processing (i.e., asynchronous shared HDF5 file creation) for checkpoints on the server. 75

7.10 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. 76

7.11 Gantt charts showing the server and runner execution (i.e., one runner instance). The charts show execution, failures in region A and B, and the recovery. The upper plots showing the cases when protecting both background *and* analysis ensembles and the lower, *only* protecting the analysis ensemble. 77

7.12 Speedup of checkpointing both ensembles towards checkpointing only the analysis ensemble. The speedup is plotted versus the location of the failure in the respective region. For failures in region A, α_A represents the normalized distance from the beginning of the region to the end. For failures in region B the respective normalized distance is given by α_B. 78

7.13 Speedup of checkpointing both ensembles towards checkpointing only the analysis ensemble. The speedup is plotted versus the location of the failure in the respective region. For failures in region A, α_A represents the normalized distance from the beginning of the region to the end. For failures in region B the respective normalized distance is given by α_B. 80

8.1 Initially particles are uniformly sampled. They are propagated to T_1 where they are weighted taking into account observation data. Resampling leads to discard some particles with low weights (top and bottom), while others with high weights become parent of several ones (3 here). 84

8.2 Runners/server architecture. The model processes perform the state propagation, the helper processes send propagated states to the PFS and prefetch next scheduled states to the local cache in the background. Communications with the server combine MPI and ZMQ data exchanges. 86

8.3 Two possible schedules of 24 propagation tasks of equal duration on 4 runners. All particles propagated from the same parent state have the same color (9 parents here). Top schedule is optimal with 9 compulsory loads (one per parent), and one for the dark blue parent that cannot fit in one runner. The bottom schedule, with 2 more sate loads, is a possible one that our on-line scheduling algorithm can produce. This is not optimal but still bellow the general $P + R - 1$ bound as the algorithm ensures that no more than $R - 1$ "color cuts” occur and avoids the same runner loads more than once a given parent state. 88
8.4 The topography of the target domain of Europe for the simulation. 90
8.5 Gantt chart of particle propagations executed by 15 (out of 511) randomly selected runners over 5 assimilation cycles. Tasks are green when the associated parent state was already present in the runner cache and did not require a load from the PFS (red otherwise). 91
8.6 Trace detailing the activity of a runner over the course of an assimilation cycle. Helper processes enable to keep model processes busy with particle propagation, except at the end of assimilation cycles when they wait for the server to finish particle resampling (dark blue). Some activities are so thin that they are not visible here (state copies from cache to model). they can become idle 92
8.7 Server response times on runner requests. 93
8.8 Gantt chart as in Figure 8.5. Two runners crashed (black cross) and 2 restarted (top 2 runners). 94
8.9 Left: strong scaling efficiency using different numbers of particles with 63 runners. One runner sets the reference case. Right: weak scaling performance test: assimilation cycle duration for different numbers of runners, but always 5 particles per runner. 94

9.1 Workflow in validation mode. 102
9.2 Linear correlation between the timely evolution of the NRMSE of compressed to lossy compressed states. We plot the NRMSE of the compressed states for each cycle by the respective values for the lossy compressed state. 107
9.3 Trace of randomly selected validator for one validation cycle 109
9.4 Comparison of the time for one assimilytion cycle leveraging the dynamic mode of our proposed framework. 110

A.1 Z-Value deviation, $\Delta\text{RMSZ}^p_{X_c}$ (Equation 9.11) for FPZIP. The colors indicate values at different cycles. 123
A.2 Z-Value deviation, $\Delta\text{RMSZ}^p_{X_c}$ (Equation 9.11) for ZFP in precision mode. The colors indicate values at different cycles. 123
A.3 Z-Value deviation, $\Delta\text{RMSZ}^p_{X_c}$ (Equation 9.11) for ZFP in accuracy mode. The colors indicate values at different cycles. 124
A.4 Z-Value deviation, $\Delta\text{RMSZ}^p_{X_c}$ (Equation 9.11) for half and single precision. The colors indicate values at different cycles. 124
A.5 Normalized maximum pointwise error and normalized root mean square error for (a) FPZIP, half and single precision, and (b) ZFP in accuracy and precision modes. The colors indicate values at different cycles. 125

List of Tables

5.1 Collision rates (i.e. the probability of collision per iteration) achieved by application of algorithm 1. We did not detect any collision for CRC32 or MD5 and the collision rates for Fletcher32 mod(65535) were almost identical to Fletcher mod(65536). Thus, we do not list the results here. For all cases, the number of iterations have been within 160-180 million. . . 27

5.2 Impact of the block size b on the DCP update time for xPic using MD5. Negative values of τ correspond to a speedup and positive values to overhead. *HASH SIZE* lists the respective memory sizes that the hash tables occupy in memory. The problem size was 1568MB per rank. 31

5.3 Relative overhead ($\Delta T/T_0$ [%]) of the checkpoint creation in LULESH with FTI-FF leveraging DCP, compared to classic CP with the original FTI file format. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead. . 33

5.4 Dataset sizes for the various xPic configurations. 34

5.5 Relative overhead ($\Delta T/T_0$ [%]) of the checkpoint creation in xPic with FTI-FF leveraging DCP, compared to classic CP with the original FTI file format. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead. . . 34

5.6 Relative overhead ($\Delta T/T_0$ [%]) of the checkpoint creation in Heat2D with FTI-FF leveraging DCP, compared to classic CP with the original FTI file format. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead. . . 35

6.1 Different C/R scenarios tested in the evaluation section with respect to the file format, the checkpoint method and the recovery method. 47

6.2 Configuration and scale for the benchmark experiments. 48

6.3 Results for the relative CR overheads. 50

6.4 The resulting values for the relative CR overheads for N-1 (shared HDF5 file on PFS). Comparison between our proposal and ADIOS. 50

6.5 Configuration for the measurements of the 2 different data organization pattern in the checkpoint file for xPic. 56

7.1 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. 70

7.2 Experiments that we have performed with the respective labels. 70

7.3 Probabilities (Equation 7.5 and Equation 7.6), average revival times (Equation 7.10), and speedup ($\langle T'_{rev}\rangle - \langle T_{rev}\rangle / \langle T_{rev}\rangle$). The numbers in green indicate that protecting both ensembles is beneficial, and the red numbers indicate that it is not. 77

8.1 Experimental setting and performance overview at 4 different scales. The times are given as average in all cases. 91

9.1 Summary of the best compression parameters and the exclusion criteria. 108
9.2 Compression rates, CR_c, for selected compression parameters, ordered by the state size. . . . 110
9.3 Speedup for the various compression parameters while storing and loading the states fomr the PFS. 111

A.1 Average values of the statistical qualifiers NRMSE, NPME and ρ_{X_c} for selected compression parameters. The rows show the evolution of the qualifiers by assimilation cycles. 122

Acronyms

3D-Var Three Dimensional Variational Data Assimilation

4D-Var Four Dimensional Variational Data Assimilation

ADIOS Adaptable Input Output System

CESM Community Earth System Model

CP Checkpointing

CKPT Checkpoint

CR Checkpoint-Restart

CRC32 Cyclic Redundancy Check 32-Bit

DA Data Assimilation

DART Data Assimilation Research Testbed

DCP Differential Checkpointing

EC-EARTH European Community Earth-System Model

ECMWF European Centre for Medium-Range Weather Forecasts

EKF Extended Kalman Filter

EnKF Ensemble Kalman Filter

FLOPS Floating Point Operations Per Second

FT Fault Tolerance

FTI Fault Tolerance Interface

GPU Graphics Processing Unit

HDF5 Hierarchical Data Format

HPC High Performance Computing

IFS Integrated Forcasting System

IO Input and Output

KF Kalman Filter

LAPF Localized Adaptive Particle Filter

LETKF Local Ensemble Transform Kalman Filter

MD5 Message-Digest Algorithm 5

MITgcm Massachusetts Institute of Technology Ocean General Circulation Model

MPI Message Passing Interface

MTBF Mean Time Between Failures

NICAM Nonhydrostatic ICosahedral Atmospheric Model

NVMe Non Volatile Memory Express

NWP Numerical Weather Prediction

OS Operating System

PDF Probability Density Function

PF Particle Filter

PFS Parallel File System

RMSE Root Mean Square Error

SA Sensitivity Analysis

SCR Scalable Checkpoint/Restart

SIR Sequential Importance Resampling

SSD Solid State Drive

TCP Transmission Control Protocol

TOPAZ Transient One Dimensional Pipe Flow Analyzer

VeloC Very Low Overhead Checkpointing System

WRF Weather Research and Forecasting model

ZeroMQ Zero Message Queue

Contents

List of Figures iv

List of Tables viii

I Prologue 1

1 Introduction 3

2 Background 5

2.1 Modern Checkpointing 5
2.1.1 Multilevel Checkpointing 5
2.1.2 Asynchronous Checkpointing 6
2.1.3 Approximate Checkpointing 7
2.2 Ensemble Data Assimilation 8
2.2.1 Variational Methods 9
2.2.2 Sequential Monte Carlo Methods 10
2.2.2.1 Ensemble Kalman Filter 10
2.2.2.2 Particle Filter 10

3 Motivation 13

4 State of the Art 17

4.1 Modern Ensemble Data Assimilation Frameworks 17
4.2 Fault Tolerance in Ensemble Data Assimilation 18

II Contributions to Checkpoint Schemes 19

5 Differential Checkpointing 21

5.1 Terminology 22
5.1.1 Definition of Incremental Checkpointing 22
5.1.2 Definition of Differential Checkpointing 22
5.2 Differential Checkpointing Implementation in FTI 22
5.2.1 A Storage Space Efficient Differential Checkpointing Implementation for Dynamic Dataset Sizes 23
5.2.2 The Dynamic File Structure in FTI-FF 23

5.2.3 A Safe Update of the FTI-FF Differential Checkpoint Files . . . 24
5.2.4 A Streaming Implementation of Differential Checkpointing . . . 24
5.2.5 Tracking the differences . . . 24
5.3 Choice of the Hash Algorithm . . . 26
5.4 When is Differential Checkpointing Beneficial? . . . 27
5.5 Evaluation . . . 28
5.5.1 HPC Applications . . . 29
5.5.1.1 LULESH 2.0 . . . 29
5.5.1.2 xPic . . . 30
5.5.1.3 Heat2D . . . 30
5.5.2 Variation of the Block Size b . . . 30
5.5.3 Spatial and Temporal Differences . . . 32
5.5.4 Overhead reduction on HPC Applications . . . 33
5.6 Discussion . . . 34
5.7 Related Work . . . 36
5.8 Conclusion . . . 38

6 Elastic Recovery **39**
6.1 Background . . . 40
6.1.1 MPI Layer Fault Tolerance . . . 40
6.1.2 General Purpose IO . . . 40
6.1.2.1 HDF5 . . . 40
6.1.2.2 ADIOS . . . 40
6.2 Implementation . . . 41
6.2.1 Design Objectives . . . 41
6.2.2 API Specification . . . 41
6.2.2.1 Complex Data Representation . . . 41
6.2.2.2 Descriptive Data Representation . . . 42
6.2.3 Accessing the Checkpoint Data . . . 43
6.2.4 Elastic recovery . . . 43
6.2.5 Checkpoint Strategies . . . 44
6.2.6 Asynchronous Checkpoint . . . 44
6.3 Methodology . . . 45
6.3.1 Generalized Evaluation Metric . . . 45
6.3.2 Measurements . . . 46
6.3.3 Experiments . . . 46
6.3.4 Applications . . . 47
6.3.4.1 Heat2D (C++) . . . 47
6.3.4.2 xPic . . . 47
6.4 Evaluation . . . 48
6.4.1 HPC Environment . . . 48
6.4.2 Performance Measurements . . . 48
6.4.3 Comparison to ADIOS . . . 49

6.4.4 Scaling . . . 50
6.4.4.1 Strong Scaling . . . 51
6.4.4.2 Weak Scaling . . . 51
6.4.5 Offline vs Online Elastic Recovery . . . 52
6.4.6 Elastic Recovery with Fewer Processes . . . 54
6.4.7 Data Distribution on Irregular Applications . . . 54
6.5 Discussion . . . 56
6.6 Related Work . . . 57
6.7 Conclusion . . . 57

III Contributions to Resiliency in Large Ensemble 59

7 Background Checkpointing in Operational Ensemble Data Assimilation 61
7.1 Background . . . 62
7.1.1 Data Assimilation and the Ensemble Kalman Filter . . . 62
7.1.2 MelissaDA . . . 63
7.1.3 Asynchronous Checkpointing and Elastic Recovery . . . 64
7.2 Implementation . . . 64
7.2.1 Launcher . . . 65
7.2.2 Server . . . 65
7.2.3 Runner . . . 67
7.2.4 Recovery . . . 68
7.3 Related Work . . . 69
7.3.1 Fault Tolerance for DART-MITgcm with Decimate . . . 69
7.3.2 Fault Tolerance Methods for Numerical Climate Models . . . 69
7.4 Methodology . . . 69
7.4.1 Experiments . . . 70
7.4.2 Data Collection . . . 71
7.4.3 Failure Regions . . . 71
7.4.4 Failure Injection . . . 73
7.5 Evaluation . . . 73
7.5.1 Climate Model . . . 73
7.5.2 Experimental Setup . . . 73
7.5.3 Performance Evaluation during Runtime . . . 73
7.5.4 Performance Evaluation Recovery . . . 75
7.5.5 Checkpointing Background and Analysis Vs. Only Analysis . . . 78
7.6 Discussion . . . 79
7.7 Conclusion . . . 80

8 Resilient Online Particle Filter using a Local Particle Cache 83
8.1 Particle Filters . . . 84
8.2 Architecture . . . 85
8.2.1 Runner and Cache Interaction . . . 85

8.2.2 Cache Eviction Strategy . . . 87
8.2.3 Fault Tolerance and Elasticity . . . 87
8.2.4 Scheduling . . . 88
8.2.5 Implementation Details . . . 89
8.3 Evaluation . . . 89
8.3.1 Runner activity . . . 90
8.3.2 Server activity . . . 92
8.3.3 State transfers to/from PFS . . . 92
8.3.4 Fault tolerance, elasticity and load balancing . . . 93
8.3.5 Scaling . . . 93
8.4 Related Work . . . 95
8.5 Conclusion . . . 95

9 A Framework for Automatic Validation and Application of Lossy Data Compression in Ensemble Data Assimilation 97
9.1 Background . . . 98
9.1.1 Ensemble Data Assimilation . . . 98
9.1.2 Terminology . . . 99
9.2 Design and Implementation . . . 99
9.2.1 MelissaDA Particle Filter . . . 99
9.2.2 High-Level View on the Validation Framework . . . 100
9.2.3 Validation Mode . . . 101
9.2.4 Dynamic Mode . . . 104
9.3 Evaluation . . . 104
9.3.1 Experimental Setup . . . 105
9.3.2 Methodology . . . 105
9.3.3 Statistical Evaluation . . . 105
9.3.3.1 Z-Value Deviation . . . 106
9.3.3.2 Pearson Correlation Coefficient . . . 106
9.3.3.3 Normalized Error Statistic . . . 106
9.3.3.4 Summary of the Validation Study . . . 108
9.3.4 Performance . . . 108
9.3.4.1 Validation Mode . . . 108
9.3.4.2 Dynamic mode . . . 109
9.3.5 Discussion . . . 111
9.4 Related Work . . . 111
9.5 Conclusion . . . 112

IV Epilogue 113

10 Thesis Conclusion 115

11 List of Publications 117

V Appendix 119

A Validation Framework - Figures and Tables 121

Bibliography 127

Part I

Prologue

Chapter 1

Introduction

> *When I have clarified and exhausted a subject, then I turn away from it, in order to go into darkness again.*
>
> **— Carl Friedrich Gauss**

High Performance Computing (HPC) is becoming ever more important for research and development in both public and industry sectors. Supercomputers have observed an exponential increase in size and performance over the last couple of decades. Exascale computing (i.e., 10^{18} floating point operations per second) is the next frontier, and it promises to bring orders of magnitude more computing power into the hands of scientists. However, the increase in computational power also comes with a certain number of challenges. Power consumption and resilience are among the most pressing issues that need to be addressed [1]. Indeed, the increasing number of components in large-scale systems makes the machine more prone to failures, reducing the Mean Time Between Failures (MTBF). It is expected that the next generation of HPC systems experiences failures every few hours [2, 3]. Consequently, most long-running extreme scale HPC applications will experience multiple failures during their execution [4, 5].

A sample of systems that encloses common information in the statistical moments of the sample is commonly known as a computational ensemble. The individual systems are each Monte Carlo realizations of the system we want to resolve. The individual systems can be simple or complex systems. Ensemble methods for complex systems gain interest with the increasing availability of computing resources. Computational ensembles find application in various fields of HPC. For instance, in solving linear systems [6], ensemble sensitivity analysis [7–9], ensemble learning [10, 11], and Numerical Weather Prediction (NWP) [12, 13]. What makes ensembles different from common parallel computing techniques is, that typically the ensemble members are processing independently of each other. We understand a computational ensemble as a group of numerical systems that generate dependent output, and that can be executed independently of each other. This definition applies for a number of (though not for all) ensemble methods among different fields; for instance, Bootstrap Aggregating (Bagging) in ensemble learning, and particle and ensemble Kalman filtering in ensemble data assimilation. Ensemble methods, being based on Monte Carlo principles, need sample sizes of statistical significance. Depending on the problem size and complexity of the individual ensemble members, this often translates into extensive resource requirements. Consequently, long-running ensemble systems need to perform fault-tolerant processing.

Thankfully, ensemble methods yield promising starting points for introducing resiliency. Being independent, failures of one ensemble member does not directly affect others, so we might consider a task based fault tolerance approach. Or we may want to exploit the stochastic origin of the ensemble method and simply discard failed member executions. To find and evaluate appropriate fault-tolerance mechanisms for ensemble methods has been the mission of this doctorate, and in this thesis, we present the fruits of our efforts in accomplishing this.

Before we present our contributions to improving resiliency of large ensembles, we will present our contributions to state-of-the-art checkpoint features. Checkpointing still represents the most common resiliency measure in HPC, and we will use the techniques developed by us and other modern features in our efforts to protect ensemble runs. The thesis is devided into three parts. The first part (Part I - Prologue) introduces the basic concepts (chapter 2), and the motivation behind this work (chapter 3). Further, we will present state-of-the-art ensemble techniques, as well as existing Fault Tolerance (FT) mechanisms for ensemble methods, and general FT techniques (chapter 4). The second part (Part II - Contributions to Checkpoint Schemes) contains our contributions to improving and extending state-of-the-art checkpoint methods. In chapter 5 we present our **first contribution:** a proposal for Differential Checkpointing (DCP), addressing issues with state-of-the-art DCP methods. we implemented our proposal in the Fault Tolerance Interface (FTI) library, and evaluated it with three different HPC applications at large scale. In chapter 6, we present our **second contribution:** our proposal for an API and runtime for multilevel Checkpoint-Restart (CR) libraries, that mitigates the elastic recovery (recovery with an arbitrary number of processes), and provides access to the checkpoint data for scientific post-processing. The third part (Part III - Contributions to Resiliency in Large Ensemble) contains our contributions to resiliency in ensemble computations. In chapter 7 we present our **third contribution:** where we utilize asynchronous CR techniques and threads to entirely hide the checkpoint cost, and minimize the recovery cost for the state-of-the-art ensemble Data Assimilation (DA) framework MelissaDA. In chapter 8, we present our **fourth contribution:** a Particle Filter (PF) implementation that uses local storage layers for implementing a distributed particle-state cache. The cache allows one to perform the state transfer from and to the global storage layer entirely in the background, and with this, it achieves a very high parallel efficiency. In chapter 9, we present our **fifth contribution:** a validation framework for ensemble systems, using robust statistical qualifiers to evaluate state and ensemble consistency when applying lossy compression to the individual state variables.

Chapter 2

Background

The most general and common technique to introduce Fault Tolerance (FT) into High Performance Computing (HPC) applications is rollback-and-recovery (a.k.a., checkpoint and restart). For this, the application state is periodically stored in dedicated buffers on some storage device. If the application is interrupted unexpectedly, it can be recovered using the application state from the latest *checkpoint* available on the dedicated buffers. In this chapter, we will briefly introduce the most common and efficient Checkpoint-Restart (CR) techniques. Afterwards, we will introduce briefly the ensemble methods that we have used in our studies.

2.1 Modern Checkpointing

In CR, application data is periodically stored into checkpoints, and upon failure, used to recover the application state at which the checkpoint was taken. CR techniques differ in: (1) the data contained in the checkpoints; (2) the way the checkpoint is performed; and (3) the way the data is recovered from the checkpoint. For instance, the checkpoint data may comprise an informational core from which the runtime data can be computed. Upon checkpoint creation, the runtime data is reduced to this informational core, and upon recovery it is reconstructed. This is beneficial if the time to create and reconstruct the data is less than the time we save while storing fewer data. This is often the case, because computing is typically faster compared to the throughput of the Input and Output (IO) layer. A similar argument holds for compressing the data before writing it to the file system, where the compression can be lossless, i.e., without loss of information, or lossy, i.e., with a certain loss of information. Further, the checkpoint creation can be coordinated, i.e., performed at a global synchronization point, or uncoordinated, i.e., independent of the other system components (typically other ranks). CR libraries mitigate the implementation of modern FT techniques in HPC applications. Some libraries provide transparent checkpoint integration, i.e., without modifying the application code, while others are based on a user-level API, or provide both options. Libraries that provide an API are typically more efficient, as the developer can explicitly define which data to checkpoint, and when. Note, however, that library approaches typically require the application to wait until the end of the current iteration in order to checkpoint less state. On the other hand, transparent checkpointing holds promise in providing effortless checkpoint protection. In the following sections, we introduce CR techniques that we have used in our work, and that need to be introduced for better understanding.

2.1.1 Multilevel Checkpointing

Multilevel checkpointing represents checkpointing at various levels of reliability. Levels of lower reliability are typically faster than levels of higher reliability. Multilevel checkpointing represents a tradeoff between reliability and speed. Common levels are:

1. local checkpoints,
2. partner checkpoints (a.k.a., buddy checkpoints),
3. encoded checkpoints, and
4. global checkpoints.

Where the reliability increases and speed decreases from 1 to 4 (Figure 2.1). The first three levels can only be applied if local storage is available, where local storage can comprise node memory or local persistent storage (Solid State Drive (SSD), Non Volatile Memory Express (NVMe), etc.). For the first level, the checkpoint is created on local storage. Consequently, if the checkpoint remains unavailable after the failure (e.g., node failure), the checkpoint is lost and the application cannot be recovered. The second level, provides a better protection, as local checkpoints are duplicated and sent to partner nodes. In this case, as long as one version of the checkpoint remains available, numerous node failures can be tolerated. The third level increases the reliability again, grouping several checkpoints using erasure codes (e.g., Reed-Solomon codes). Reed-Solomon codes have the property that after adding K blocks to M blocks of data, the loss of any of up to K blocks can be tolerated, since the missing blocks can be reconstructed with the information from the remaining blocks. Thus, arranging the checkpoint files into groups of M members, and encoding them by adding M additional files, the loss of any of up to M files from the group can be tolerated. The forth level provides the highest reliability, as it can tolerate failures of the entire system, assuming an intact global storage after the failure. Some prominent multilevel checkpoint libraries are Fault Tolerance Interface (FTI) [14], Scalable Checkpoint/Restart (SCR) [15] and VeloC [16].

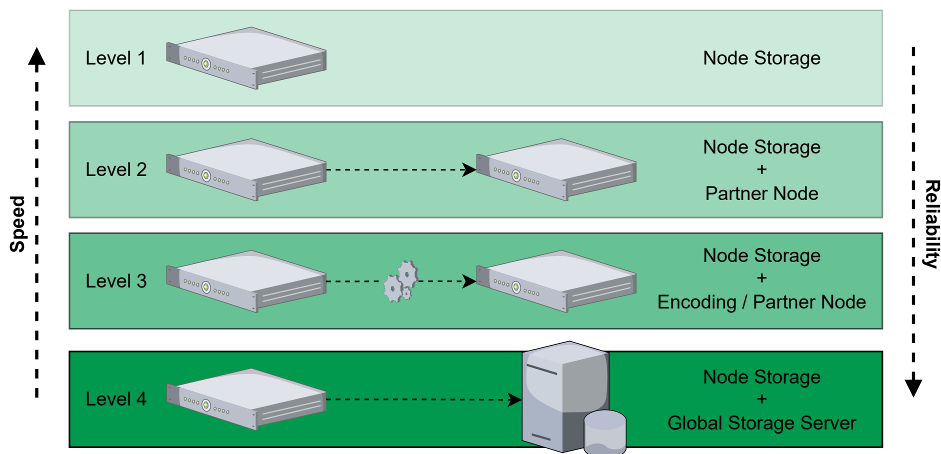


Figure 2.1: Example showing 4 levels of different reliability and speed

2.1.2 Asynchronous Checkpointing

Sometimes, the type of the checkpoint or the application workflow allow creating the checkpoint entirely or partially asynchronously to the applications' execution. The levels introduced in subsection 2.1.1 can be performed partially in the background. The second, third, and fourth level can be divided into 2 stages, the first stage being a first level (local) checkpoint. While the first stage must be performed by the application inline, the second stage can be performed in the background (Figure 2.2). For instance, the partner copy can be sent to the partner node by dedicated worker processes, while the application continues its

execution. Similarly, we can perform the encoding and the copy to global storage in the background. Sometimes, the application itself allows asynchronous checkpointing, for instance through pipelines. This is the case when the checkpoint can be performed intermittently, and if some checkpoint data is not required for a long computational period involving different data. While the application performs the computation, the data not required can be added to the checkpoint in the background. In some cases, it is even possible to hide the entire checkpoint creation, for instance, by leveraging intermediate buffers, or when the entire checkpoint creation can be arranged in pipelines.

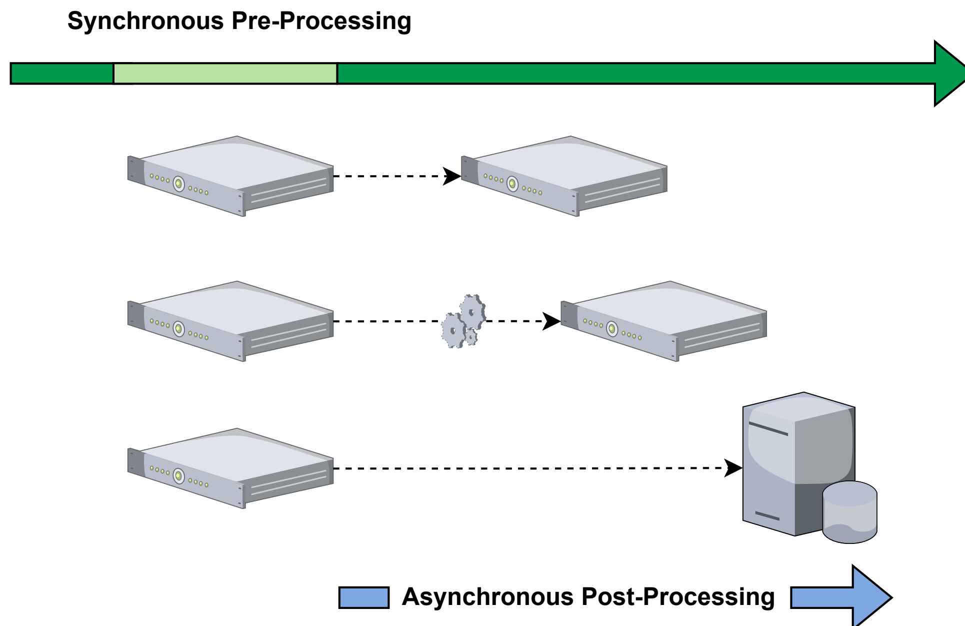


Figure 2.2: Asynchronous checkpoint creation. The first step (pre-processing) is performed inline, and the second step (post-processing) is performed by dedicated processes asynchronously to the application.

2.1.3 Approximate Checkpointing

If the application can tolerate a certain loss in accuracy, the checkpoint can be created using an approximate data representation to reduce the amount of data stored. This can be beneficial due the IO bottleneck arising for checkpoints containing very large amounts of data. Lossless data compression is often not enough, due to relatively low compression rates. The reduction can be improved with lossy data compression. The HPC landscape provides various algorithms for lossy compression, for instance ZFP [17], FPZIP [18], ISABELA [19], SZ [20], MGARD [21] and MGARD+ [22], to name a few. The algorithms follow different approaches and might reveal their strength only for certain types of applications. Typically, the algorithm performance can further be tuned by parameters, such as bit precision, tolerance, and fixed compression rate. However, data compression is not the only method for data reduction, the data can also be converted into lower precision data formats, for instance, from double precision into single or half precision floating point representations. Furthermore, data reduction can be achieved using interpolation techniques, spectral data decomposition, and other data specific approaches.

2.2 Ensemble Data Assimilation

Numerical climate models are non-linear systems and very sensitive to initial conditions (i.e., chaotic). A famous example of a chaotic system is the Lorenz attractor [23], also called lorenz butterfly. Changing the initial conditions by only a fraction of a percent in one of the three parameters of the model, leads to an entirely different system evolution (see Figure 2.3).

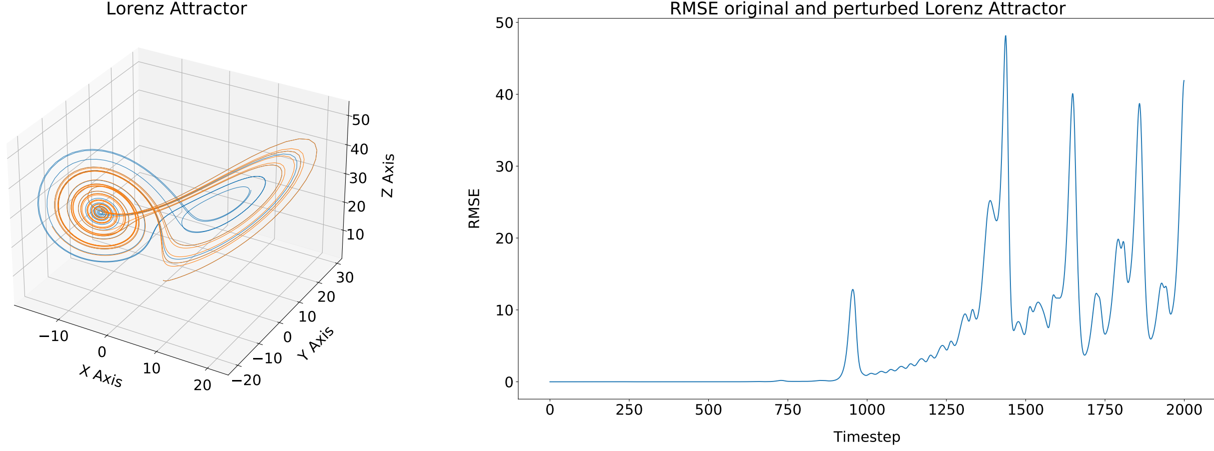


Figure 2.3: (*Left*) Lorenz Attractor $L(\sigma = 28, \rho = 10, \beta = 8/3)$ with initial state at $\mathbf{x} = (0, 1, 1.05)$ in blue and slightly perturbed with initial state at $\mathbf{x} = (0.00001, 1, 1.05)$ in orange. (*Right*) The RMSE between the first and second Lorenz system.

Clearly, real climate models exhibit chaotic behavior as well, and the vivid example from above demonstrates that meaningful predictions of weather and climate systems require an accurate knowledge of the initial state. However, the numerical models cannot exactly describe the dynamics of a climate system. For instance, processes that take place at a smaller scale than the model resolution need to be described by parametrizations that only qualitatively capture their influence. Besides, current (and likely also future) climate models cannot take into account all the processes involved, due to the complexity of the climate system. This leads to model-induced errors in the systems' evolution. As a consequence, even if it was possible to provide the real state of the weather system without uncertainty as initial state, after a certain amount of time, the error introduced by the imperfect model-would inevitably lead to entirely wrong predictions. Only the frequent *assimilation* of observations can account for this, and ensure the correct evolution of the numerical model [24]. Thankfully, there are methods to include real world observations that guide the model trajectory into the right direction. Those methods are known under the generic term of *Data Assimilation*. Most of the Data Assimilation (DA) techniques deployed, are based on Bayesian inference:

$$P(A|B) \propto P(B|A) \cdot P(A) \tag{2.1}$$

resulting from Bayes theorem with constant Probability Density Function (PDF) $P(B)$ for the evidence B available. Applied to our problem we have:

$$P_{analysis} \propto P_{observation} \cdot P_{model} \tag{2.2}$$

where $P_{observation}$ is the conditional PDF, or *Likelihood*, of the observations, conditioned on the model state. P_{model} is the PDF of the model state, or *Prior distribution*, before taking the observations into account, and $P_{analysis}$ is the PDF, or *Posterior distribution*, of the so-called *analysis* state. The latter encodes the information of both model and observation uncertainty. DA aims to minimize the uncertainty

of the posterior distribution. There are different techniques to achieve this. The most prominent techniques are Three Dimensional Variational Data Assimilation (3D-Var), Four Dimensional Variational Data Assimilation (4D-Var), and ensemble methods such as the Ensemble Kalman Filter (EnKF) and Particle Filter (PF).

Ensemble DA techniques provide an explicit description of the flow-dependent covariance matrix. The covariance matrix encodes the uncertainty of the climate state, which is often as important as the best state estimate itself, as it provides a range of likely realizations of the system's trajectory. Determining the possible trajectories of a tropical cyclone apart from the most likely one, for instance, is important to assess the likelihood of an impact in other regions. Whereas 4D-Var often leads to more accurate predictions, it does not provide a simple way to access the prediction uncertainty [25]. Therefore, it is frequently combined with an ensemble method. On the other hand, ensemble methods are often preferred from the beginning; for instance, due to the typically smaller implementation effort or because they constitute the better fit in certain cases [26–28]. In the following, we will give a short introduction to both variational and ensemble-based DA techniques.

2.2.1 Variational Methods

Variational DA methods are based on iterative minimization of a cost function, which is derived from Bayes' theorem and contains distributions from model and observation uncertainties. 3D-Var, being the simplest of the variational methods, tries to minimize the following cost function:

$$J_{3D}(\mathbf{x}) = (\mathbf{x} - \mathbf{x}_b)^T \mathbf{B}^{-1} (\mathbf{x} - \mathbf{x}_b) + (\mathbf{y}_0 - \mathbf{H}\mathbf{x})^T \mathbf{R}^{-1} (\mathbf{y}_0 - \mathbf{H}\mathbf{x}) \tag{2.3}$$

where $\mathbf{x}$ is the object of the minimization, i.e., the *minimum variance* state estimate [29]. $\mathbf{x}_b$ is the *background* state, sometimes also called *first guess*, and B is the approximated error covariance matrix of the background state, where we use the conventional notation suggested in [30]. The background state, in variational methods, is the result of the propagation of $\mathbf{x}$ from $t \rightarrow t+1$ with the numerical model (a.k.a., model operator):

$$\mathbf{x}_t = \mathbf{M}\mathbf{x}_{t-1} \tag{2.4}$$

with $\mathbf{M}$ being the *linearized* model operator. Further, denote $\mathbf{y}_0$ as the real-world observations, and $\mathbf{H}$ as the *linear* observation operator defined by:

$$\mathbf{y} = \mathbf{H}\mathbf{x} \tag{2.5}$$

yielding the transformation from model space to observation space. All variational methods require both the model and observation operator to be linear. Thus, for the application of variational methods, the model operator needs to be linearized. The contributions of model and observation state to the cost function are weighted by their respective error covariance matrices, $\mathbf{B}$ and $\mathbf{R}$. All variational methods assume a Gaussian model and observation errors with zero-mean.

4D-Var differs from the 3D-Var method by the additional time dimension. Whereas in 3D-Var the observations are assimilated at the beginning of the assimilation window (indicated by the zero index of $\mathbf{y}_0$), in 4D-Var, the observations are assimilated at the actual time they are measured. This is achieved by adding respective terms in the cost function that compare the states and observations at different times:

$$J_{4D}(\mathbf{x}) = (\mathbf{x} - \mathbf{x}_b)^T \mathbf{B}^{-1} (\mathbf{x} - \mathbf{x}_b) + \sum_i (\mathbf{y}_i - \mathbf{H}\mathbf{x}_i)^T \mathbf{R}^{-1} (\mathbf{y}_i - \mathbf{H}\mathbf{x}_i) \tag{2.6}$$

2.2.2 Sequential Monte Carlo Methods

A different approach is taken by the ensemble methods in DA. Instead of minimizing a cost function, the ensemble methods follow a Monte Carlo approach and approximate the best state estimate by averaging over several realizations of the climate state. The two available techniques for ensemble DA are the EnKF and the PF. In the following, we will give a brief introduction to both methods.

2.2.2.1 Ensemble Kalman Filter

The Ensemble Kalman Filter (EnKF) was first introduced by Evensen in 1994 [31], during his efforts toward leveraging the Extended Kalman Filter (EKF) for DA of quasi-geostrophic models [32, 33]. Problems with the error covariance evolution equation in the EKF, and its intense computational cost, have motivated him to explore other strategies of finding approximations for the evolution of the error covariance matrix [31]. Emerging from these efforts, Evensen introduced Monte Carlo based ensemble statistics into the Kalman filter and established the EnKF formalism.

The Kalman filter, as well as the variational methods, assume Gaussian error covariances for model and real-world observations. However, in contrast to the variational methods, a Kalman filter does not require the linearization of the model and observation operators. The state space equations for the EnKF can thus be expressed as:

$$\mathbf{x}_t = \mathcal{M}\mathbf{x}_{t-1} + \eta_t, \quad \eta_t \sim \mathcal{N}(0, Q_t) \tag{2.7}$$

$$\mathbf{y}_t = \mathbf{H}\mathbf{x}_t + \varepsilon_t, \qquad \varepsilon_t \sim \mathcal{N}(0, R_t) \tag{2.8}$$

where $\mathcal{M}$ and $\mathbf{H}$ are the full (i.e., not linearized) model and *linear* observation operator. The EnKF also takes the imperfection of the model into account, indicated by the noise process η_t. However, whereas in general, non-linear models cause non-Gaussian errors, the error here is assumed to have a Gaussian form. Both the model and observation errors are assumed to be Gaussian with mean zero, generated by the respective error covariance matrices Q_t and R_t.

An important advantage of the EnKF is the statistics encoded in the ensemble. The statistical moments (e.g., mean and variance) can directly be computed from the ensemble. Thus, in contrast to the variational methods, ensemble methods provide a measure of prediction uncertainty. Furthermore, the EnKF formulation is more general, as it doesn't require model linearization. The update step of the EnKF in [31] is:

$$\mathbf{x}_t^a = \mathbf{x}_t^b + \mathbf{K}(\mathbf{y}_t - \mathbf{H}\mathbf{x}_t^b) \tag{2.9}$$

$$\mathbf{K} = \mathbf{P}_t^f \mathbf{H}_t^T \left[\mathbf{H}_t \mathbf{P}_t^f \mathbf{H}_t^T + \mathbf{R}_t\right]^{-1} \tag{2.10}$$

where $\mathbf{x}_t^b$ results from $\mathbf{x}_t^b = \mathcal{M}\mathbf{x}_{t-1}^a$, i.e., the application of the model operator to the *analysis* state, $\mathbf{x}_{t-1}^a$, from the filter step at $t-1$. $\mathbf{K}$ is the Kalman gain and $\mathbf{P}_t^b$ the background error covariance matrix, which is computed from the background state ensemble.

2.2.2.2 Particle Filter

Another Monte Carlo approach for climate data assimilation is the Particle Filter (PF) [34]. We have variational methods, which require a linear model operator, and the EnKF, which relaxes this condition,

while keeping the assumption of Gaussian error for the covariance. On the other hand, the PF has no restrictions on the model operator and also no assumptions on the distributions. Instead, the distribution is generated by weighting the Monte Carlo realizations (**particles**) of the model state. The formulation of the PF is again based on Bayes' theorem:

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})} \tag{2.11}$$

with $p(\mathbf{x}|\mathbf{y})$ being the posterior, $p(\mathbf{x})$ the prior, and $p(\mathbf{y}|\mathbf{x})$ the likelihood. $\mathbf{x}$ and $\mathbf{y}$ denote the model state and real-world observations respectively. In the so-called *bootstrap* particle filter [35], the prior is approximated by:

$$p(\mathbf{x}) \approx \sum_{p=1}^{P} \frac{1}{P}\delta(\mathbf{x} - \mathbf{x}_p) \tag{2.12}$$

with P being the number of particles in the sample, and $\delta(\mathbf{x})$ the Dirac delta function. After replacing the PDF of the observations (which is unknown) by:

$$p(\mathbf{y}) = \int d\mathbf{x}\ p(\mathbf{y}|\mathbf{x})p(\mathbf{x}) \tag{2.13}$$

$$\overset{(2.12)}{\approx} \frac{1}{P}\sum_{p=1}^{P} p(\mathbf{y}|\mathbf{x}_p) \tag{2.14}$$

After substituting $p(\mathbf{x})$ (Equation 2.12) and $p(\mathbf{y})$ (Equation 2.14) in Equation 2.11, we can write the approximated posterior distribution as:

$$p(\mathbf{x}|\mathbf{y}) \approx \sum_{p=1}^{P} \frac{\hat{w}_p}{\sum_{q=1}^{P} p(\mathbf{y}|\mathbf{x}_q)}\ \delta(\mathbf{x} - \mathbf{x}_p) \tag{2.15}$$

$$= \sum_{p=1}^{P} w_p \delta(\mathbf{x} - \mathbf{x}_p) \tag{2.16}$$

where $\hat{w}_p$ is the **unnormalized** particle weight:

$$\hat{w}_p = p(\mathbf{y}|\mathbf{x}_p) \tag{2.17}$$

and w_p the **normalized** particle weight:

$$w_p = \frac{p(\mathbf{y}|\mathbf{x}_p)}{\sum_{q=1}^{P} p(\mathbf{y}|\mathbf{x}_q)} \tag{2.18}$$

Equation 2.16 gives us an explicit description of the posterior distribution at hand, and with that, access to all statistical moments of the system. Since we have not made any assumptions on the shape of the distribution, PF provides an elegant way of accounting for non-linearities in the climate model.

Chapter 3

Motivation

In section 2.2 we introduced the main concepts employed for data assimilation in climate models. Four Dimensional Variational Data Assimilation (4D-Var) and Ensemble Kalman Filter (EnKF) are well established in data assimilation. Both are frequently used and do not compete for deployment. It has been observed that each method reveals certain advantages over the other, depending on the problem at hand. On the other hand, Particle Filter (PF) is rather new in Data Assimilation (DA) for climate science. However, recent work by Leeuwen et al. [35–37] shows promise for using PF where model non-linearities become more important [35]. Whether usinng a variational or Monte Carlo method, increasing model resolution and simulation volume motivate an increasing demand in computing resources. While those needs are being answered by the rapid advances in scale and performance of supercomputers, the Mean Time Between Failures (MTBF) of large clusters is steadily falling, and operation at very large scale will become impractical without appropriate measures for resiliency. We have witnessed an exponential increase in computing power during the last decades, with thee current most powerful supercomputer, Fugaku, reaching over 0.5 EFLOPS (Figure 3.1). El Capitan is planned to reach more than 1.5 EFLOPS in 2023 [38].

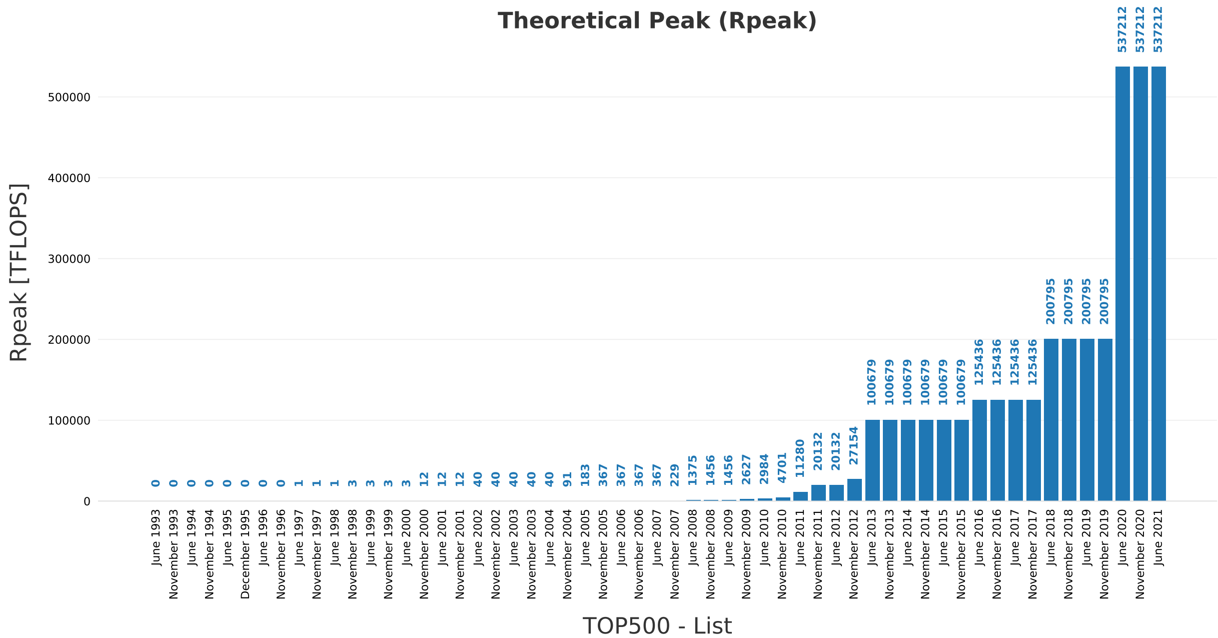


Figure 3.1: Exponential increase in peak performance (TOP500 lists).

While this will give ever more computing power in the hand of scientists, the increase in computing power comes along with an increasing number of components (Figure 3.2). The MTBF of a system made up from N components, is given by the MTBF of the individual components divided by N [39]. Therefore, we will encounter more failures utilizing more components, reducing the systems MTBF. It is

expected that the MTBF reduces to a few hours at exa-scale [2, 3]. Consequently, applications running at that scale need protection to successfully terminate their execution.

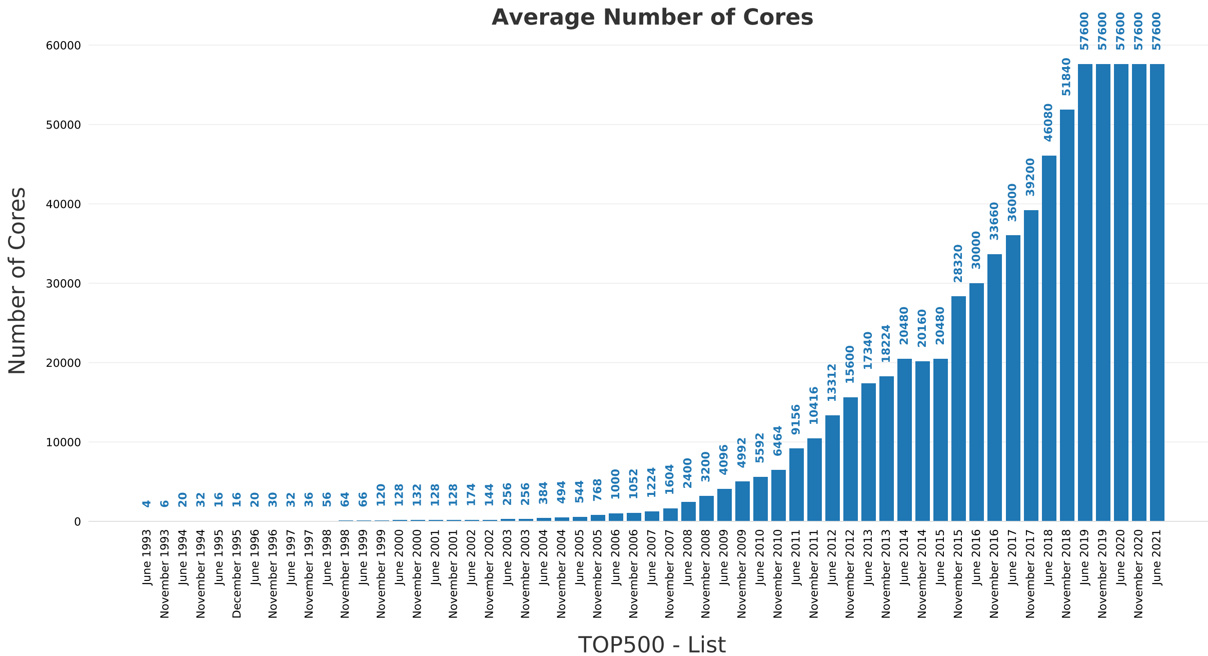


Figure 3.2: Exponential increase in the average number of cores (TOP500 lists).

Due to the high dimensionality of climate states (on the order of 10^9), ensemble methods require large ensemble sizes to ensure statistical significance. Recent studies suggest ensemble sizes between 1,000 and 10,000 for reliable predictions in models that exhibit strong non-linearities [40]. In addition, pushing towards higher model resolution leads to an increasing non-linearity in the model, due to the fact that parametrizations need to be replaced by more realistic numerical models. A general statement about ensemble sizes shows that the required size grows linearly with the dimension of the climate state for EnKF and exponentially for PF [13, 41]. Though this requirement can be relaxed by introducing techniques such as localization and ensemble inflation, the trend of ensemble sizes is pointing towards larger scales, and with this comes the requirement of fault tolerance.

A computational ensemble is a group of numerical systems that generate dependent output, and that can be executed independently of each other. This definition of ensembles applies to both EnKF and PF. Ensemble states are propagated through the numerical model independently of each other, until they are connected during the filter step. This holds certain advantages from the Fault Tolerance (FT) perspective. For instance, the propagations of the climate states can be considered as tasks, which only have output dependencies. We can thus leverage existing techniques from scheduling theory to achieve resiliency. Furthermore, the separation of the filter step and the propagation allows for a decoupling of both steps into different application domains, which can be protected independently. This reduces the complexity of the FT measures, and reduces the size of the individual systems that need to be protected.

There are two main execution workflows for ensemble methods: the **online workflow**, where the entire system is executed as a single executable with communication over the network layer; and the **offline workflow**, executing the ensemble members and the filter system on separate executables, while communicating results through the storage layer. While the latter approach allows for convenient protection by leveraging the files as restart files, the former approach is very difficult to protect. On the other hand,

the online approach is typically much faster, delivering high throughput in high-bandwidth network layers.

Not only do the ensemble methods require extensive scaling, but they are often also deployed in continuous operation. Consequently, their operation requires robust resiliency with low overheads. Ensemble methods with large application states, can not just rely on offline approaches, due to scale constraints. Global storage systems can simply not deliver the bandwidth for reasonable performance, leaving online approaches as the only option. However, online approaches currently rely on MPI for communication. Though there has been much effort at improving the FT properties of MPI (ULFM, Fenix, etc.), dealing with faults in MPI without the use of global rollback and recovery strategies is still immature. Hence, due to the lack of alternatives, most ensemble methods rely on offline workflows, while accepting the accompanying large overhead.

With this, the motivation of this work is clear. The challenge is protecting very large computational systems with reasonable overhead. Simple Checkpoint-Restart (CR) approaches are not enough to meet the requirements for timely availability of results needed, for instance, for high-frequency predictions in Numerical Weather Prediction (NWP). Therefore, we will approach the task from two different viewpoints: (i) we will develop and enhance state-of-the-art checkpoint methods to improve their performance and broaden their applicability under strict performance requirements; and (ii) we will develop new architectures and frameworks that mitigate FT by exploiting the properties of ensembles that we have referred to earlier.

Chapter 4

State of the Art

In this chapter we will present state-of-the-art ensemble data assimilation frameworks and studies about the scale and the current size of the ensembles being deployed. Later, we will review state-of-the-art fault tolerance techniques in ensemble data assimilation systems.

4.1 Modern Ensemble Data Assimilation Frameworks

A selection of modern large-scale ensemble data assimilation frameworks in operation are Transient One Dimensional Pipe Flow Analyzer (TOPAZ) [42], employed, for instance, by the Arctic Marine Forecasting Center providing a 10-day forecast of the ocean currents and sea ice on a daily basis [43]. The Community Earth System Model (CESM) [44, 45] is a framework of coupled land, atmospheric, ocean and sea ice models using an Ensemble Kalman Filter (EnKF). The model is used in a number of recent studies and developments [46–48]. The Integrated Forcasting System (IFS) [49, 50], a Numerical Weather Prediction (NWP) system in operation at the European Centre for Medium-Range Weather Forecasts (ECMWF). Further, there is the Japanese global cloud resolving Nonhydrostatic ICosahedral Atmospheric Model (NICAM) model [51], which has been integrated with the Local Ensemble Transform Kalman Filter (LETKF) [52] into a high resolution global NWP framework.

Typical ensemble sizes of state-of-the-art operational ensemble Data Assimilation (DA) systems range from about 10 to 50 ensemble members. For instance there is the CESM large ensemble with 30 members [53], and the IFS operational system with 50 members and a number of other operational centers world wide [54]. A recent study on the ensemble size using the IFS system suggests that from a user perspective ensembles, with more than 50 members are desirable. Miyoshi et al. [40] used the Japanese K-Computer to explore ensemble sizes of 10,240 members to study the dependency of DA on the ensemble size. They observes that non-linearities are better resolved if using ensemble sizes above 1000 members. In 2016, Miyoshi et al. [55] suggest Big Data Assimilation (BDA), assimilating very large observational datasets using 100 ensemble members. A hypothetical BDA system with a DA frequency of 10 seconds would require about 180,000 nodes of the K-Computer for the 100 members. Studies that explore the scale required by future ensemble data assimilation systems include two works by Terasaki et al. Irom 2015, they used about 5,700 nodes of the K-Computer at RIKEN reaching 720 TFloating Point Operations Per Second (FLOPS) [56], and in 2020 on the Fugaku supercomputer, they used more than 130,000 nodes, reaching 79 PFLOPS [57]. In the future, high-resolution weather and climate prediction that will support resolutions of less than 10km is expected to run at full scale on exascale systems [58].

4.2 Fault Tolerance in Ensemble Data Assimilation

Until today, a widely deployed workflow in ensemble data assimilation was to perform the climate simulation and the data assimilation on separate executables [59]. The ensemble members (i.e., the climate simulations) store the climate states to the file system, and after all simulations have finished, the data assimilation system reads the files, assimilates the observations, and writes the improved states back to storage. The ensemble members then reread them to perform the next assimilation cycle. It is straightforward to protect such systems against failures; only the bookkeeping of running and finished state propagations is needed, along with a detection mechanism to acknowledge failed propagations. The actual implementation of such a system, however, can be cumbersome. Toye et al. [60, 61] presented a submission framework based on the scheduler extension Decimate [62], which is capable of precisely this: bookkeeping of submission, and resubmission of failed jobs.

A different approach to introduce resiliency in climate systems is taken by Düben et al. [63]. It is based on a shadowing backup model, which is executed at the same time as the actual model, but uses a coarser resolution. The backup model is used to identify corrupting errors resulting from faulty hardware and for the replacement of corrupted data structures by the coarser representation. The original data is thus replaced by an approximation. The advantage of such a system is that the simulation can continue after a failure, and thus reduce the impact of failures on the time for completion.

Friedemann et al. [64] presented the framework MelissaDA, which is based on Melissa [65], a framework to perform large-scale sensitivity analysis. MelissaDA uses the same architecture as Melissa, providing a launcher, server and multiple workers. The framework can be used to perform ensemble data assimilation without intermediate files. The simulation and data assimilation systems are still seperated into different executables. However, the transfer of the states between the two systems is realized through the network layer. The workers propagate the model states and the server performs the update step. The states are transferred between the server and runners via Transmission Control Protocol (TCP). The framework provides resiliency through the launcher. Whenever a runner fails, it requests new resources to replace the failed runner. The server acknowledges the failure and reschedules the failed propagation.

Part II

Contributions to Checkpoint Schemes

Chapter 5

Differential Checkpointing

Publications

- International Symposium on Cluster, Cloud and Grid Computing (CCGRID) 2019:
  - **Keller, K.**, Bautista-Gomez, L. (2019, May). **Application-level differential checkpointing for HPC applications with dynamic datasets.** In 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (pp. 52-61). IEEE.
- International Symposium on Cluster, Cloud and Grid Computing (CCGRID) 2020:
  - Parasyris, **K., Keller**, K., Bautista-Gomez, L., Unsal, O. (2020, May). **Checkpoint restart support for heterogeneous hpc applications.** In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID) (pp. 242-251). IEEE.

Main Contributions

- Hash-based Differential Checkpointing (DCP) implementation detecting changes in the *variables* (not only in memory pages). We provide two file formats that (1) remains of stable size and (2) grows in size, but performs well also for very small block-sizes and data updates.
- Testing the hash algorithms Adler32, Fletcher32, CRC32, and MD5 regarding their applicability for hash-based DCP.

DCP has been proposed in order to avoid re-writing checkpoint data that is identical between two consecutive checkpoints (i.e., no change of data). Previous research works have attempted to implement such a technique by tracking dirty memory pages in the system and only updating those within the checkpoint files [66]. While this method works, it is not always efficient as many applications do re-write the exact same content (e.g., zero) into the same memory cells. From the OS perspective, these memory pages have changed as they are dirty, but in reality the content has not changed. Direct comparison between stored and dirty pages and compression of the differences [67], and hashing the memory pages to detect real changes has also been proposed [68]. While these techniques solve the problem of the redundant data in checkpoint files, they further increase the runtime overhead. Furthermore, since the methods rely on the memory pages, we are restricted to block sizes corresponding to the memory page size that the application uses.

We have implemented a hash-based strategy in which we partition the application datasets (not the memory pages) in blocks and keep track of the changes of each block by comparing the corresponding hashes. The block size for the partitioning can be customized independently of the memory page size. We further introduce a new file format that is capable of incorporating the changes without increasing the file size, and to dynamically adapt to changing variable sizes. We further evaluate the collision robustness of multiple hash algorithms and show that Message-Digest Algorithm 5 (MD5) and Cyclic Redundancy Check 32-Bit (CRC32) are viable solutions for differential checkpointing. We integrate our implementation into the multilevel checkpointing library Fault Tolerance Interface (FTI) and evaluate it with three High Performance Computing (HPC) applications. In our measurements, we obtain up to 62%

reduction in checkpointing time in comparison to traditional checkpointing. Furthermore, we propose a theoretical model that predicts performance gains that can be obtained with our DCP technique.

The rest of this chapter is organized as follows. Section 5.1 introduces the terminology that we use. Section 5.2 introduces our DCP implementation. Section 5.3 explores the robustness of different hashing algorithms. Section 5.4 presents our analytical model to predict performance gains. The results of our large-scale evaluation are presented in Section 5.5. Section 5.6 discusses the strong points and limitations of the proposed technique. Section 5.7 discusses related work, and finally Section 5.8 concludes our work.

5.1 Terminology

The term *incremental checkpointing* is used in the literature to denote two different processes. To avoid confusion we would like to clarify what we refer to when we use the terms incremental and differential checkpointing.

5.1.1 Definition of Incremental Checkpointing

We refer to incremental checkpointing as to be the *incremental completion of a checkpoint file*. This technique serves primarily to avoid overhead caused by oversaturated network channels. It may be used within applications that provide datasets, that define the current program state, at different times. Thus, instead of writing the whole checkpoint data at once, it is incrementally written during some period of time, which reduces the stress on the network.

5.1.2 Definition of Differential Checkpointing

We refer to differential checkpointing as to be the *differential update of a checkpoint file*. That is, the data blocks in the previous checkpoint file that by the time of a subsequent checkpoint differ to the corresponding data block of the current application state, will be replaced by the up-to-date data block. The rest of the blocks (i.e., those that did not change) will not be updated.

5.2 Differential Checkpointing Implementation in FTI

We implement our proposed DCP mechanism into FTI, an application-level checkpointing library with an API that provides flexibility and allows user to flag datasets that need to be protected. FTI is a multi-level Checkpointing (CP) library that offers 4 levels of increasing reliability and FTI implements a dedicated process that performs post-processing work for the more reliable CP levels asynchronously to the application processes. The mechanisms used to track data differences in applications can be divided into two categories: tracking dirty pages (i.e. pages that were accessed by a store operation), or tracking actual changes of data by direct comparison of hash representations of the data. In our implementation, the address range of the datasets will be partitioned into blocks of size b. We create hashes of these blocks and keep them in memory. The hashes are created from the dataset representation in memory immediately after a successful Checkpoint (CKPT) and *before* the application continues its normal execution so that the hashes in memory represent the current state of the data in the checkpoint file.

Instead of creating one file for each update, we provide two techniques that integrate the data into the existing checkpoint file. The first method updates the data inside the file directly, the second method

appends the updates at the end of the file. Both methods include the meta-data inside the file, allowing the reconstruction of the dataset upon recovery. The first method keeps the file size constant, however, imposes the risk of losing the checkpoint upon failures during the checkpoint creation. The second method leads to increasing file sizes, however, is designed to be safe, even when failures occur during the checkpoint creation.

5.2.1 A Storage Space Efficient Differential Checkpointing Implementation for Dynamic Dataset Sizes

To implement a storage efficient DCP mechanism, the FTI-protected datasets need to be arranged into immutable blocks in the CKPT file. For protected datasets with steady sizes, this is accomplished naturally. However, FTI supports datasets with dynamic sizes. Thus, to maintain immutable positions inside the checkpoint file we need to allow fragmentation of the datasets, and we need to implement a bookkeeping of the fragmentation. FTI stores the checkpoint metadata in separate files. We could follow this practice in order to keep track of the fragmentations. However, the meta data that FTI usually writes is very little and the files use the INI format, which is an ASCII text format. For heavily fragmented datasets, this approach is not efficient. Consequently, we decided to develop a file format for FTI that includes the metadata for both the fragmentation and the checkpoint metadata inside the file structure. The proposed file format avoids additional files, and with this reduces the stress on the metadata server. We call the new format simply FTI File Format (FTI-FF). A diagram of the file structure is shown in figure 5.1. For a comprehensive description please consult the FTI online documentation [69].

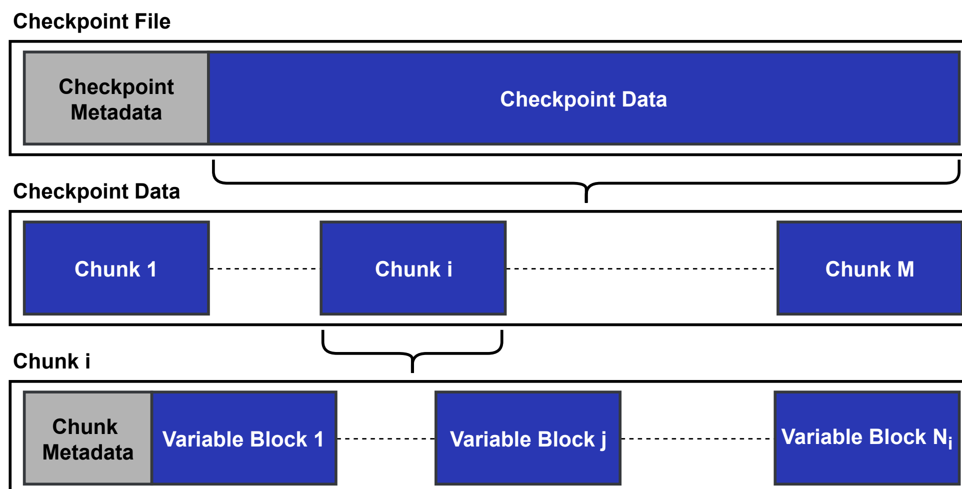


Figure 5.1: The FTI-FF structure can be resolved into three layers of abstraction: The first layer comprises global metadata (for instance, filesize and datasize) and checkpoint data (protected variables), the latter is build from M chunks C_i of which each contains N_i variable blocks. The actual application data are stored in the variable blocks. Each chunk also contains metadata describing the layout of the variable blocks.

5.2.2 The Dynamic File Structure in FTI-FF

When first created, each dataset is stored inside a variable block with the current size of the dataset. All variable blocks added to the file are appended to the chunk metadata located at the beginning of the chunk. The chunk metadata contains the variable-ids, the offsets of the variables' data inside the file, the size of the variable block, etc. Once created, variable blocks never change in size or in their position inside the

file. If a dataset increases its size, and exceeds the total size of all its variable blocks, another variable block is created with the excess as its size. The new block is appended at the end of the file, tailing the corresponding metadata block. When a variable decreases in size, the metadata for the affected variable blocks is updated and the respective updates of the variable are written to the appropriate location in the file. Since the variable blocks never change after they have been created, this can lead to unused blocks in the checkpoint file.

5.2.3 A Safe Update of the FTI-FF Differential Checkpoint Files

The *prime directive* of any checkpointing library is that we must not update the data in the CKPT file directly. This is due to the danger of corrupting the file if an error occurs during the update. However, if we enable DCP in FTI-FF we violate this principle for the advantage of reducing the required storage space. To ensure the safe operation while using DCP and FTI-FF, it is necessary to create a temporary copy of the checkpoint file, and only write directly into the copy. After the successfull update, we can delete the temporary file again and release the storage space. This becomes more efficient, if we can create the copy in the background, for instance leveraging FTI dedicated background processes.

5.2.4 A Streaming Implementation of Differential Checkpointing

We have implemented another file format specifically for DCP in FTI. The format is designed for high performance for any block size, and number of updates. In contrast to FTI-FF, we simply append the updates to the file. Additionally, we include metadata that holds information to reconstruct the dataset upon recovery. Figure 5.2 shows a diagram of the file structure in the streaming implementation. The name comes from the usage of the C standard streaming IO library. We take advantage of the buffered IO for dealing with many very small updates, to minimize the number of IO calls.

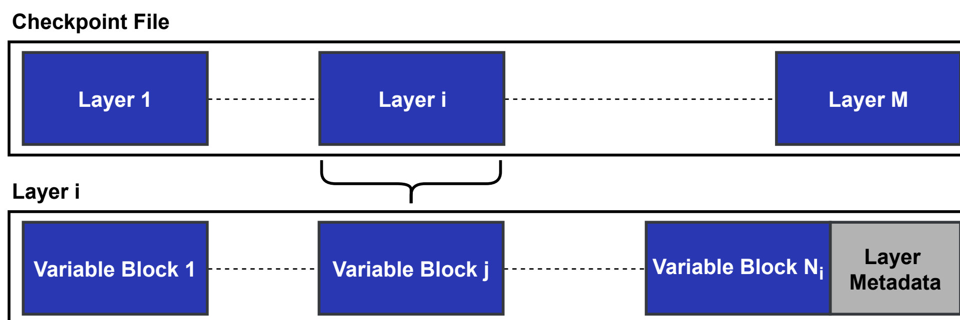


Figure 5.2: The streaming DCP file structure contains M layers L_i; one layer for each checkpoint update. The layers L_i contain the N_i variable blocks that contain the differential variable update. each layer also contains metadata holding information such as the layer size, block size, and checkpoint ID.

5.2.5 Tracking the differences

In our DCP implementation, we partition the datasets in blocks of a custom size. The blocks can be in two different states. A block can be *dirty* or *clean*, where we keep the terminology from the DCP implementations that use the memory page protection mechanism. Dirty blocks are added to the checkpoint file, and clean blocks are not. In addition to that, we differ between *valid* blocks, for blocks of data that is present in the checkpoint file, and *invalid* blocks, for blocks that are not (e.g., if the data size has increased,

or the dataset is written for the first time). Invalid blocks do not need to be compared, as they have no counterpart in the checkpoint file and can be added directly. Our algorithm for updating the checkpoint file is:

1. mark new blocks as invalid.
2. create hashes for valid blocks, and compare to the old ones.
3. update the dirty and invalid blocks in the checkpoint file.
4. update the hashes for dirty blocks.
5. create hashes for invalid blocks.

The process is visualized in figure 5.3. The figure is divided in three sections separated by a dashed line. The left section corresponds to (1) and is performed during the call to `FTI_Protect`. The function exposes and updates datasets to instruct FTI including them into the CKPT files. FTI creates metadata related to the dataset within this function, and we intercept here to flag blocks of new datasets of new blocks of datasets as invalid. The middle section of the figure corresponds to (2) and (3) and is performed during the checkpoint creation. Finally, the third section corresponds to (4) and (5) and is performed after the successful creation of the checkpoint. The total amount of additional memory $\Delta\mathrm{MEM}_{\mathrm{hash}}$ of our implementation consists of the hashes of the data blocks and 2 boolean values for each block (valid/invalid, dirty/clean):

$$\Delta\mathrm{MEM}_{\mathrm{hash}} = \frac{\text{data size}}{\text{block size}} \times (\text{digest size} + 2) \tag{5.1}$$

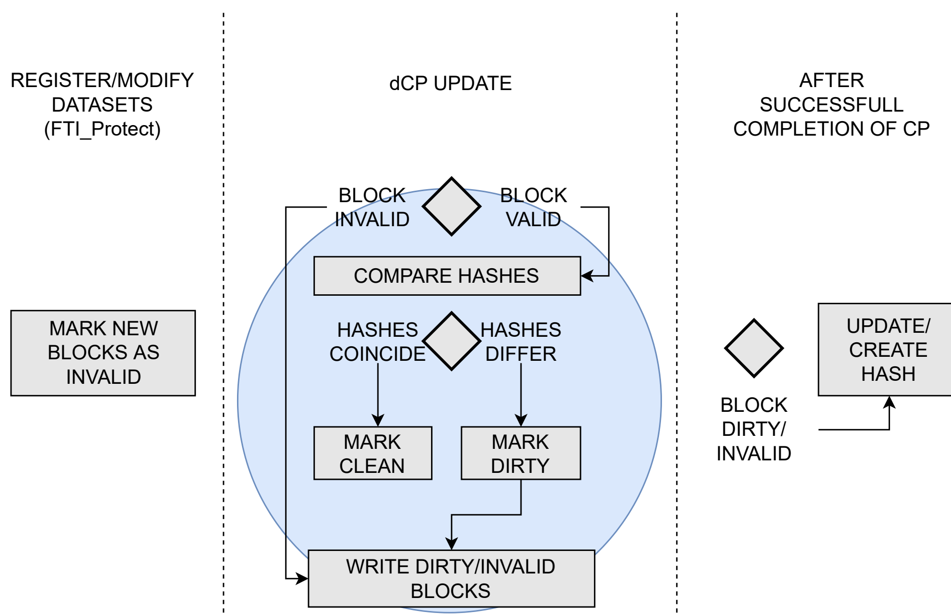


Figure 5.3: DCP detection and update scheme. Processes left to the blue circle happen before and processes to the right after the DCP update. The circle indicates the dirty region request loop

5.3 Choice of the Hash Algorithm

Depending on the size of the protected datasets, the hash arrays might get significantly large. For instance, the MD5 digest length is 128 bits (16 bytes). Assuming a hash-block size of 128 bytes and 1GB of protected data per rank, we have to reserve 144MB of RAM for the hash metadata (see Equation 5.1). To reduce this size, we can either increase the block size or decrease the digest size. With increasing block sizes, we capture less of the real data changes, and with smaller digest sizes we risk higher collision rates, and with this a higher probability of data inconsistency (when a collision occurs, two different blocks have the same digest, and we cannot detect data changes correctly). In order to provide a small digest size, we tested three hash algorithms with 32 bits digest size on their performance and reliability: Adler32, Fletcher32, and CRC32. We also included the widely used MD5 algorithm with a digest size of 128 bits for comparison. The Adler32 and CRC32 checksums were calculated using the zlib data compression library [70], Fletcher32 was implemented using the recommendations in [71] and for MD5 we used the OpenSSL library [72]. To obtain a statement about the reliability of the checksums we performed a simple collision test. We focussed on the so-called *avalanche effect* [73], which is the property that the hash changes significantly for small changes of the data. This is an important property, as some applications are very sensitive to small perturbations. The algorithm is shown in 1. Where C_b and D_b are buffers that contain random integers, $b = \{2^i \mid 7 \leqslant i \leqslant 15\}$ denotes the hash block sizes and p denotes the patterns that are used to modify the elements of C_b. For N_b we have $b \bmod (N_b \times 64) == 0$. The elements of p correspond to bit flips of the last 1 ($p_0 = \texttt{0x1}$), 2 ($p_1 = \texttt{0x3}$), 4 ($p_2 = \texttt{0xff}$), 8 ($p_3 = \texttt{0xfff}$) and 16 ($p_4 = \texttt{0xffff}$) bits, and an arbitrary patterin in p_5 to simulate a random change.

Algorithm 1 Count hash collisions of modified buffers

```
repeat
    for all b do
        populate C_b with N_b random u64 integers;
        create hashes h_{C_b} of C_b;
        for all p do
            for i=1, N_b do
                D_{b,i} = C_{b,i} ⊕ p;
                Create hash h_{D_{b,i}} of D_{b,i};
                if h_{D_{b,i}} == h_{C_{b,i}} then
                    c_{b,p} ++;                          ▷ c_{b,p} := Collision Counter
                end if
            end for
        end for
    end for
until N iterations
```

The results of the collision test are listed in table 5.1. Fletcher32 is commonly implemented with $M = 2^n$ or $M = 2^n - 1$ (M is the modulo value for the checksum [71]). The case $M = 2^n - 1$ leads to identical checksums for buffers that differ only in one or more groups of two consecutive bytes that are all `0x00` in one and all `0xff` in the other buffer. In our opinion, this is already reason enough to disqualify the algorithm for its usage in DCP. Fletcher32 and Adler32 are both significantly faster than CRC32 in our tests. However, both also have poor collision resistant characteristics for block sizes between 128 and 32768 Bytes. Most of the collisions for Adler32 occurred for 1-bit or 2-bit flips and decrease for

increasing block sizes. The collisions for Fletcher32 are homogeneously distributed for all modifications and block sizes. In addition to the reliability issue of Fletcher32, that we have mentioned earlier, the collision rate of both algorithms, Adler32 and Fletcher32, is too high to provide a sufficient level of reliability. MD5 and CRC32, on the other hand, did not show any collisions. While the test does not ensure reliability of CRC32 and MD5, it allows us to disqualify Adler32 and Fletcher32 for our purpose. Based on the literature about CRC32 and MD5 (CRC32 is used in zlib and other cases to provide data integrity [70, 74, 75]) and based on our results we think that it is safe to use both algorithms in our DCP implementation.

	$\mathbf{p}_0$	$\mathbf{p}_1$	$\mathbf{p}_2$	$\mathbf{p}_3$	$\mathbf{p}_4$	$\mathbf{p}_5$
b			**ADLER32**			
128	6.84e-3	1.42e-3	8.56e-5	3.68e-7	6.13e-9	1.23e-8
256	1.70e-3	3.46e-4	2.12e-5	8.59e-8	1.23e-8	1.23e-8
512	4.24e-4	8.69e-5	5.39e-6	1.84e-8	0	6.13e-9
1024	1.06e-4	2.21e-5	5.39e-6	1.84e-8	0	6.13e-9
2048	2.56e-5	5.21e-6	2.58e-7	0	6.13e-9	0
4096	6.23e-6	1.37e-6	9.20e-8	0	6.13e-9	0
8192	1.56e-6	2.70e-7	1.84e-8	6.13e-9	0	0
16384	3.56e-7	4.91e-8	4.29e-8	6.13e-9	0	0
32768	1.41e-7	7.98e-8	1.84e-8	0	0	0
			FLETCHER32 - MOD(65536)			
128	1.54e-5	1.47e-5	1.52e-5	1.52e-5	1.50e-5	1.54e-5
256	1.53e-5	1.55e-5	1.53e-5	1.54e-5	1.56e-5	1.54e-5
512	1.48e-5	1.56e-5	1.53e-5	1.52e-5	1.53e-5	1.52e-5
1024	1.48e-5	1.55e-5	1.51e-5	1.53e-5	1.58e-5	1.56e-5
2048	1.49e-5	1.51e-5	1.49e-5	1.49e-5	1.50e-5	1.56e-5
4096	1.57e-5	1.53e-5	1.57e-5	1.53e-5	1.51e-5	1.50e-5
8192	1.55e-5	1.51e-5	1.49e-5	1.54e-5	1.47e-5	1.54e-5
16384	1.51e-5	1.55e-5	1.52e-5	1.54e-5	1.55e-5	1.52e-5
32768	1.56e-5	1.59e-5	1.48e-5	1.57e-5	1.53e-5	1.52e-5

Table 5.1: Collision rates (i.e. the probability of collision per iteration) achieved by application of algorithm 1. We did not detect any collision for CRC32 or MD5 and the collision rates for Fletcher32 mod(65535) were almost identical to Fletcher mod(65536). Thus, we do not list the results here. For all cases, the number of iterations have been within 160-180 million.

5.4 When is Differential Checkpointing Beneficial?

In order to estimate the threshold at which differential checkpointing becomes beneficial, we will derive a cost function from the saving, $\Delta T^{(-)}$, for writing fewer data, and the cost, $\Delta T^{(+)}$, for creating and comparing the hashes. They are defined by:

$$\Delta T_b^{(-)} = (N_{b,t} - N_{b,d})\, t_{b,w} \tag{5.2}$$

$$\Delta T_b^{(+)} = (N_{b,t} + N_{b,d})\, t_{b,h} \tag{5.3}$$

Where $t_{b,w}$ is the duration to write a block of data with block-size b, $t_{b,h}$ the duration of hashing the block, $N_{b,d}$ is the number of dirty blocks, and $N_{b,t}$ is the total number of blocks. Equation 5.3 involves both values,

$N_{b,t}$ and $N_{b,d}$, since, we cannot commit the new hashes for data-blocks that differ prior to the successful completion of the CKPT, hence we need to compute these twice[1]. After subtracting Equation 5.2 from Equation 5.3, and normalizing the result by the total number of blocks $N_{b,t}$ we get:

$$\Delta T_b / N_{b,t} = (T_b^{(+)} - T_b^{(-)})/N_{b,t} =: \tau_b$$
$$\tau_b = (t_{b,h} - t_{b,w}) + n_{b,d}\,(t_{b,w} + t_{b,h}), \quad n_{b,d} = N_{b,d}/N_{b,t}. \tag{5.4}$$

Equation 5.4 is our cost function that turns into an overhead reduction (i.e., speedup) for $\tau_b < 0$ and to additional overhead for $\tau_b > 0$. We can infer, that the maximal overhead accounts to $2N_{b,t}t_{b,h}$ when $n_{b,d} = 1$ (i.e., all blocks are dirty). This corresponds to a maximal relative overhead of $2t_{b,h}/t_{b,w}$ (i.e., relative to the time without DCP). The threshold, η_b, at $\tau_b = 0$, i.e., when the application of our DCP implementation becomes beneficial is:

$$\eta_b := n_{b,d}\Big|_{b,\tau_b=0} = \frac{t_{b,w} - t_{b,h}}{t_{b,w} + t_{b,h}} = \frac{1-\rho}{1+\rho}, \quad \rho = \frac{t_{b,h}}{t_{b,w}}. \tag{5.5}$$

η_b corresponds to the ratio of dirty to total number of blocks, below which we can expect a speedup. The lower the value for ρ (i.e., hashing faster than writing) the closer η gets to one, which means that for already very small changes DCP gets beneficial.

According to Equation 5.5, we can compute the threshold by measuring the times $t_{b,h}$ and $t_{b,w}$. To measure the average time, $t_{b,w}$, to write a block of size b to disk, we measure the total time, $T_{b,w}$, to write n buffers of size b with p processes. The average is then given by $t_{b,w} = T_{b,w}/n$. We require that at time $T_{b,w}$ all processes have finished writing, to simulate the synchronisation point at the end of the checkpoint. Similarly, we measure the average time $t_{b,h}$ to compute the hash for a block of size b. However, this time independently of the other processes, since the hash creation is local to the ranks and computed asynchronously. We expect a perfect scaling behavior for $t_{b,h}$. For $t_{b,w}$ instead (i.e., for writes to global storage), we have to consider network congestion and file-system bandwidth saturation, resulting into a deterioration of the I/O throughput at large scale. Thus, we expect an increasing speedup for increasing total problem sizes. Figure 5.4 shows the results for the measurements we performed for 768 and 2400 processes. In both cases, the total buffer size was 1GB per process which leads to the total problem sizes of 0.75 TB and 2.3 TB respectively. We can see that the threshold indeed increases for a growing problem size. We observe a better performance of MD5 towards CRC32 in all cases. The performance of MD5 depends slightly on the hash-block size. This dependency is less strong at a larger scale. This also applies for the performance difference between CRC32 and MD5. The results show that for $b = 32$KB and MD5, the threshold is at about 95% (i.e. only 5% less to write).

5.5 Evaluation

In section 5.4 we have seen that even when applications update 95% of the checkpoint data (i.e. we save only about 5% of I/O) DCP can already be beneficial for HPC applications. In order to support this result with empirical evidence, we analyze the behavior of DCP in FTI while checkpointing three HPC applications at large scale. We conduct representative experiments that analyze performance and overhead. All experiments were performed on MareNostrum4, where each node is composed by [76, 77]:

[1] We may avoid the redundancy here if we store the hashes for the *dirty* blocks in a separate array, which would lead to a higher memory footprint.

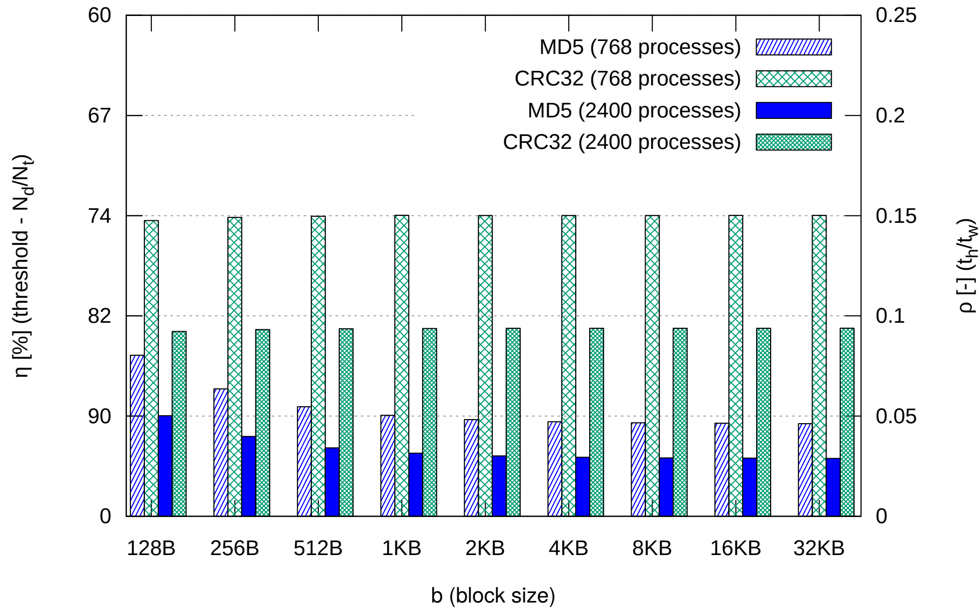


Figure 5.4: The bars show the estimated DCP threshold, i.e. the ratio of dirty to total data we need, to make the DCP operation beneficial. The left axis shows the dirty data ratio, η, the right axis shows the corresponding value of ρ (ratio between the hash time, $t_{b,h}$, and I/O time, $t_{b,w}$, for block size b). The experiment has been performed with 768 and 2400 processes and 1GB per rank.

- 2 Intel Xeon Platinum 8160 CPU (24 cores at 2.10GHz)
- 12 $\times$ 8 GB DDR4-2667 DIMMS (96GB/node)
- 100 Gbit/s Intel Omni-Path HFI Silicon 100 Series PCI-E
- 10Gbit Ethernet
- 200 GB SSD local to the nodes
- SUSE Linux Enterprise Server 12 SP2

5.5.1 HPC Applications

In this section we introduce Lulesh2.0, xPic and Heat2D, three HPC applications that we used for our evaluation. We selected applications, that have different memory access patterns, to compare the efficiency of DCP for a range of applications.

5.5.1.1 LULESH 2.0

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [78] is part of the Advanced Simulation and Computing (ASC) program at the Lawrence Livermore National Laboratory (LLNL). It simulates a Sedov blast wave propagation within a material in three dimensions [79]. The modeling space is discretized into an unstructured hex mesh. The system state is updated using stencil operations. The purpose of LULESH is to provide a proxy application that possesses the characteristics of an HPC application from this field in order to analyze performance on various platforms and various programming models. That makes it suitable for our purpose as well, since it represents a broad field of applications. To maximize the checkpoint load, we conducted measurements that determined the

maximum problem size we can apply, without the risk of a memory overflow on the node. The checkpoint data is serialized, which increases the memory footprint of the application. With a CKPT size of 430MB per rank, we use about 80GB of the node memory (96GB available) and achieve an aggregate CKPT size of 725GB.

5.5.1.2 xPic

xPic is an alternative implementation of iPic3D [80], a particle-in-cell application. iPic3D and xPic are part of the application co-design in the DEEP-EST project [81]. The application models space plasma simulations. The modeling space is discretized by a rigid mesh. The simulation is always initialized to the equilibrium state. During each time step, the particle states and electromagnetic fields are advanced using the Vlasov equation, which couples the equation of motion to the Maxwell equations. xPic takes its runtime parameters from a configuration file. In order to scale the problem size, we used a combination of the parameters `ntcx` (number of cells in x-direction), `ntcy` (number of cells in y-direction) and `nppc` (number of particles per cell). To control the number of contiguous datasets, we used the parameters `nblockx` (number of blocks in x-direction), `nblocky` (number of blocks in y-direction) and `nspec` (number of species). We implemented two distinct mechanisms in order to expose datasets to FTI. In the first implementation, xPic-c (c for common), we expose every memory contiguous dataset individually to FTI. Depending on the configuration of xPic, this may lead to a large number of protected variables. In the second implementation, xPic-s (s for serialized), we use Boost [82] for serializing the variables.

5.5.1.3 Heat2D

Heat2D is a 2D heat distribution simulation using a 1D domain decomposition. It simulates the transition from a non-equilibrium heat distribution to the equilibrium state. In each time step, the cells of the temperature grid are updated via a 4-point stencil operation that stores the average of the 4 neighbor cells temperatures into the center cell. The ranks exchange adjacent rows of the temperature grids. The simulation runs until the total value of the temperature differences reaches a pre-defined minimal value or exceeds a certain number of iterations. The large majority of memory used by Heat2D is checkpointed which enabled us to perform large scale executions with large checkpoitn sizes, for instance a run with a total problem size of about 2.8TB with 2304 processes on 48 nodes.

5.5.2 Variation of the Block Size b

We start by analyzing the impact of the block size over the effectiveness of DCP. We measured the time of a DCP update for various block sizes b and compared the results to ordinary CP (DCP disabled). All CKPTs were performed at the same application state. We performed experiments with both MD5 and CRC32, the results were very similar for both hashing algorithms, and we only list the MD5 results for the sake of brevity. By decreasing the block size, we increase the granularity. That means that we have a better chance to get close to the actual percentage of data that did change.

Table 5.2 shows the results for the experiment we performed with the xPic application (see 5.5.1.2 for details). The first column of the table shows the block size and the third column shows the percentage of data written compared to the original checkpoint size. We notice that as the block size increases, the amount of data to write increases as well, due to the lower granularity. However, the overhead (shown

b	τ	DCP RATE	SHARE HASH	SHARE WRITE	HASH SIZE [MB]
128B	1333%	52.25%	1.51%	97.67%	196
256B	1106%	53.84%	1.53%	97.39%	98
512B	666%	56.25%	2.10%	96.13%	49
1KB	231%	59.15%	4.40%	91.40%	25
2KB	15%	61.42%	12.82%	73.93%	12
4KB	-32%	62.25%	21.77%	55.07%	6
8KB	-35%	62.41%	22.69%	52.47%	3
16KB	-36%	62.48%	22.66%	52.52%	1.5
32KB	-36%	62.50%	22.67%	52.07%	0.76

Table 5.2: Impact of the block size b on the DCP update time for xPic using MD5. Negative values of τ correspond to a speedup and positive values to overhead. *HASH SIZE* lists the respective memory sizes that the hash tables occupy in memory. The problem size was 1568MB per rank.

in the second column) is incredibly large for high block granularities (i.e., small blocks). To understand this phenomena, we measured the time spent hashing and the time spent writing data for each case. We observe that the large majority of checkpointing time is spent in writing and not hashing. This is caused by the fragmentation of the updates into small chunks. It has been shown in the past (e.g. [83–85]), that PFSs have poor performance when small chunk sizes need to be written. For xPic, block sizes of less than 4KB degrade performance and block sizes greater than 4KB improve performance up to 36%. In addition, we measured the amount of memory consumed to store the hash arrays. Most of the block sizes have hash arrays that represent less than 1% of the memory used by the process. For block sizes of 16KB the hash arrays take only 0.1% of the memory used by the application. Based on this analysis, we decided to use block sizes of 16KB during the following measurements.

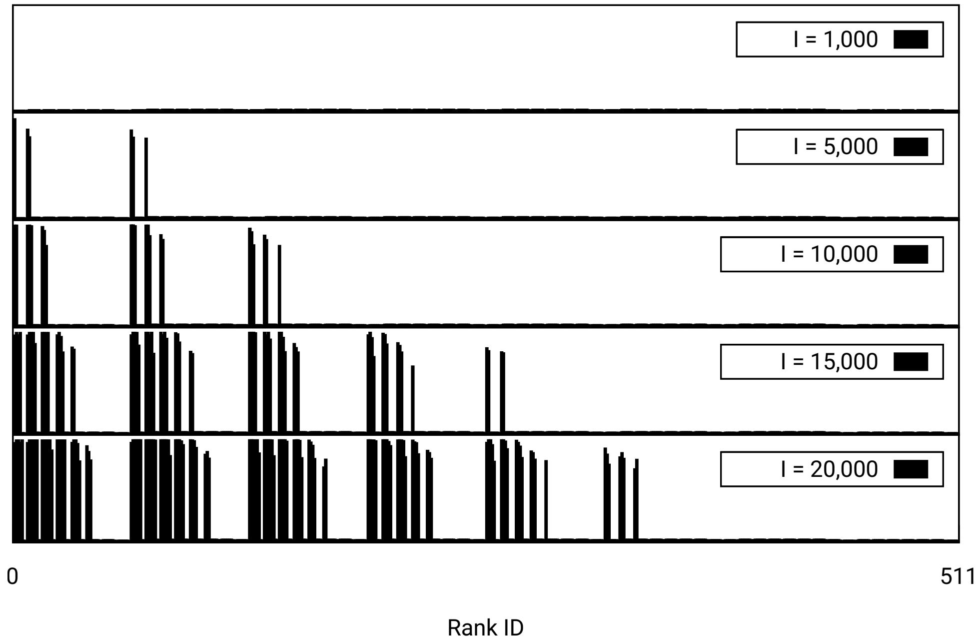


Figure 5.5: Data differences per rank in checkpoints at different timesteps in LULESH. The x-axis shows the ranks (512 in total), and the y-axis shows the percentages written in percentage.

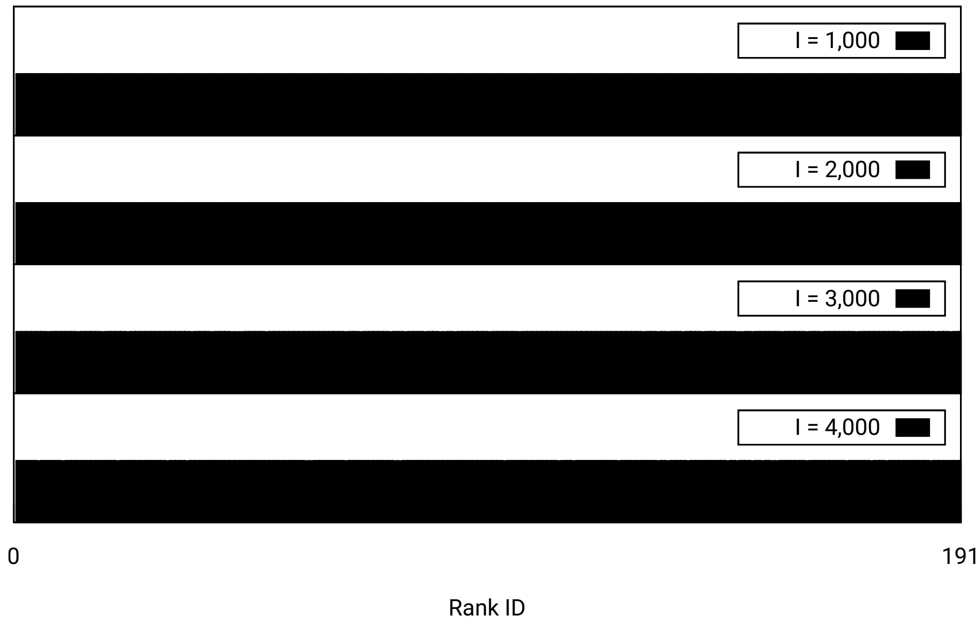


Figure 5.6: Data differences per rank in checkpoints at different timesteps in xPic. The x-axis shows the ranks (192 in total), and the y-axis shows the percentages written in percentage.

5.5.3 Spatial and Temporal Differences

After finding the right block size to avoid too coarse hashes as well as to fine I/O writes, we investigate the amount of data that is actually being updated between two consecutive checkpoints for the applications presented in Section 5.5.1. The results are depicted in Figure 5.5, 5.6, and 5.7. The three figures are divided into several temporal regions following the y axis (i.e., DCP taken at iteration 1000, 5000, etc.) and spatial regions following the x axis (i.e., the process rank which is representative of a slice of the domain). First, we observe that LULESH does not change too much data during the first iterations; and as the time passes (up to iteration 20000) the number of ranks where data is actually modified increases. This reflects the shock wave that is simulated by LULESH. This demonstrates that for applications like LULESH, the benefits of DCP might vary depending on time and space.

xPic on the other hand, shows a completely different behavior, the amount of data updated is consistently the same across all the ranks and regardless of the time in the execution. This is explained by the fact that xPic is a plasma simulation in which particles are constantly in movement, even in those changes are minimal, they are enough to trigger updates as they will produce a different block hash. There are a few variables of the application that are read-only and that do not change through out the simulation, which is why not a 100% of the data is updated at every checkpoint.

Looking into Heat2D, we observe a middle ground between LULESH and xPic. Indeed, Heat2D also increases the data differences as time evolves, but at a much lower pace than LULESH, giving it a less dynamic look. We observe that the most affected ranks are organized in strides, which is consistent with the 1D partitioning mentioned in Section 5.5.1.3. However, other initial conditions could translate into a more homogeneous updates across ranks.

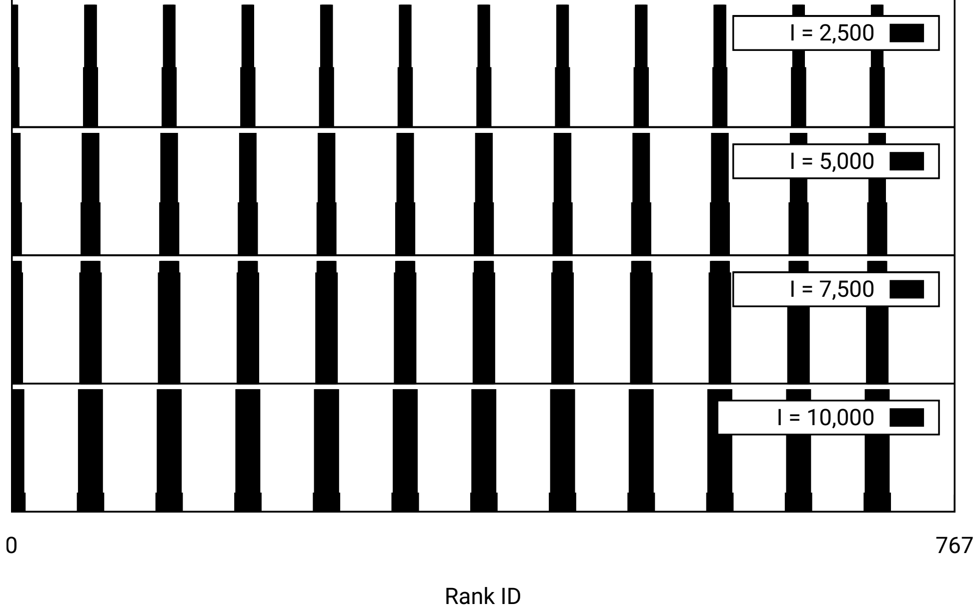


Figure 5.7: Data differences per rank in checkpoints at different timesteps in Heat2D. The x-axis shows the ranks (768 in total), and the y-axis shows the percentages written in percentage.

5.5.4 Overhead reduction on HPC Applications

In this section we evaluate the overhead of DCP in comparison with classic CP for the three applications. Table 5.3 lists the results of our measurements performed with LULESH leveraging FTI-FF with CRC32, MD5, and DCP disabled. The latter represents the performance difference between the checkpoint creation leveraging FTI-FF and the original FTI file format. The first row shows the time of the first checkpoint (everything dirty). ordinary CP and the second a checkpointing with DCP in which only 3% of the data is updated. We have only two rows since we never had updates significantly different to 3%. This result indicates that the propagation of the wave is slow as shown previously. This great reduction in checkpoint size with DCP in LULESH translates into a 62% reduction in the CP time.

Relative overhead of FTI-FF with DCP (CRC32, MD5, disabled) compared to ordinary CP			
Data diff. (n_d)	**MD5**	**CRC32**	**DCP disabled**
100%	-9$\pm$12	-5 $\pm$ 13	-5 $\pm$ 13
3%	-62 $\pm$ 10	-60 $\pm$ 8	-

Table 5.3: Relative overhead ($\Delta T/T_0$ [%]) of the checkpoint creation in LULESH with FTI-FF leveraging DCP, compared to classic CP with the original FTI file format. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead.

For xPic, we evaluate the non-serialized as well as the serialized implementations (xPic-c and xPic-s, see 5.5.1.2) were each implementation is tested against two distinct configurations (A and B). For configuration A, the FTI protected memory consist of many relatively small contiguous datasets. Configuration B instead has few but rather large contiguous datasets. Table 5.4 summarizes the relevant runtime parameters for both configurations. The results of our evaluation is shown in Table 5.5. We

observe that the reduction on checkpoint size is the same for executions with and without serialization. Further, the application of DCP for configuration A does not reduce the checkpoint overhead. The reason for this is that configuration A produces a large number of small chunks to be written. A detailed analysis of this phenomena is presented in section 5.6. In contrast to configuration A, we do observe a significant overhead reduction for configuration B. The best performance we measured for xPic-s (serialized) using MD5 and using configuration B, is a 35% speedup while writing only 62% of the original checkpoint size.

	CONFIG. A		CONFIG. B	
	xPic-c	**xPic-s**	**xPic-c**	**xPic-s**
SIZE OF DATASETS [MB]	4.22	1360	168	1344.25
# OF DATASETS	320	1	8	1
CP SIZE / RANK [MB]	1350.32	1360.38	1344.55	1344.80
CP SIZE TOTAL [GB]	760	765	882	883

Table 5.4: Dataset sizes for the various xPic configurations.

Relative overhead of FTI-FF with DCP (CRC32, MD5, disabled) compared to ordinary CP				
	Data diff. (n_d)	**MD5**	**CRC32**	**DCP disabled**
xPic-c (A)	100%	7 ± 11	6 ± 12	0 ± 9
	50%	9 ± 12	11 ± 9	-
xPic-c (B)	100%	9 ± 16	14 ± 11	-3 ± 9
	62%	-33 ± 6	-28 ± 6	-
xPic-s (A)	100%	7 ± 17	14 ± 9	0 ± 7
	50%	-4 ± 6	0 ± 6	-
xPic-s (B)	100%	5 ± 5	11 ± 7	-2 ± 6
	62%	-35 ± 7	-29 ± 6	-

Table 5.5: Relative overhead ($\Delta T/T_0$ [%]) of the checkpoint creation in xPic with FTI-FF leveraging DCP, compared to classic CP with the original FTI file format. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead.

As mentioned in Section 5.5.3, the data difference in Heat2D depend significantly on the initial conditions. Heat2D shows a good reduction of checkpoint size, in the regime of 40% to 100%. Table 5.6 lists the results. We can see that MD5 has clearly performance benefits in comparison to CRC32. We notice that almost all of the experiments show an significant reduction on the checkpoint overhead. We observe important speedups of up to 49% for a 40% DCP update using MD5.

Overall, the three applications (although with different behaviours) show substantial improvements thanks to DCP. The reduction in checkpointing overhead goes up to 62%, 35% and 49% for LULESH, xPic and Heat2D respectively.

5.6 Discussion

In section 5.4 we developed a model (Equation 5.4) that can be used to estimate the speedup we may achieve using DCP. In this section, we want to check whether the predictions from the model coincide with the measurements or not. We first write down the relative time difference, S, of a DCP update

Relative overhead of FTI-FF with DCP (CRC32, MD5, disabled) compared to ordinary CP			
Data diff. (n_d)	**MD5**	**CRC32**	**DCP disabled**
100%	-2 ± 9	1 ± 6	-4 ± 11
99%	-5 ± 7	-2 ± 7	-
95%	-8 ± 6	-7 ± 7	-
87%	-14 ± 6	-12 ± 6	-
79%	-19 ± 8	-17 ± 6	-
71%	-26 ± 6	-22 ± 6	-
63%	-35 ± 5	-30 ± 5	-
56%	-40 ± 5	-37 ± 4	-
40%	-49 ± 5	-46 ± 7	-

Table 5.6: Relative overhead ($\Delta T/T_0$ [%]) of the checkpoint creation in Heat2D with FTI-FF leveraging DCP, compared to classic CP with the original FTI file format. Negative values correspond to a reduction of the overhead (speedup) and positive values to an increase in the overhead.

towards a conventional CP:

$$S(n_d) = \Delta T(n_d)/T_0 := \begin{cases} < 0 : \text{overhead reduction} \\ > 0 : \text{overhead increase} \end{cases} . \tag{5.6}$$

T_0 denotes the time for a full CP and DCP disabled. Using Equation 5.4, we can write this as:

$$S(n_d) = \frac{\tau}{t_w} = \rho - 1 + n_d(\rho + 1) \quad , \quad \rho = \frac{t_h}{t_w} \tag{5.7}$$

Where we used $T_0 = t_w N_t$. We determined t_w and t_h for b = 16KB:

$$b = 16\text{KB} \tag{5.8}$$
$$t_w = 1.35 \times 10^{-3} s \tag{5.9}$$
$$t_h = 3.92 \times 10^{-5} s \quad [\text{MD5}] \tag{5.10}$$
$$\rightarrow \rho = 0.029 \tag{5.11}$$

Again, for the sake of brevity, we will only present the results for MD5. Figure 5.8 shows the measured relative speedups depending on the dirty data ratio. We also added the estimation according to our model from Equation 5.7 and the fit to the data. The fit leads to $\rho = 0.013 \pm 0.008$, which excludes our measured value of $\rho = 0.029$. However, the fit shows a p-value of 0.8, which represents a confidence interval of only about 20%. Nevertheless, the values for xPic-B and for Heat2D above 50% dirty data ration are in good agreement with the fit. LULESH shows the highest speedup with 62%, however, using equation 5.7 we would expect a speedup of about 94%. For xPic-s with configuration A, we measured a 4% speedup but expected about 46%. Given the disagreement between theoretical prediction and experimental results for LULESH and xPic with configuration A, we performed a more detailed analysis. Figure 5.9 shows the cumulative density function (CDF) of chunk sizes written contiguously during a DCP update for all four scenarios. The figure reveals a correlation between the size of the chunks and the performance. xPic-s A and LULESH show both less performance than expected and both write mostly chunks of relatively small sizes (4MB - 12MB). On the other hand, we have good performance in xPic B and Heat2D where we observe relatively large chunk sizes (mostly over 200MB).

If the small writes are the explanation for the inaccurate model predictions, one should be able to meet the estimated performance by avoiding IO operations with small chunks. This can be accomplished by using the second file format that we developed (subsection 5.2.4). We indeed observe a significant improvement in the experiments. The results for LULESH and xPic leveraging the streaming file format are denoted as *Streaming* in figure 5.9. We can see that now xPic-A indeed is in very good agreement with the model prediction. LULESH also improved, but is still not as good as the model predicts. We continue the detailed analysis of LULESH, and we noticed that LULESH has about 2-3% updates in all ranks *except* in rank 0. Rank 0 has a DCP share of 80%. A large amount of the data is thus written by only one rank. We also observe a high anisotropy in the distribution of the dirty data in Heat2D (Figure 5.7), and a completely homogeneous distribution in xPic (Figure 5.6). Indeed, Heat2D matches the model prediction worse than xPic, that is in almost perfect agreement with the model. Thus, the anisotropy in the dirty data ratio among the ranks seems to explain the discrepancy between our model estimation and the fit; the model is too simple, as it does not take into account these differences. However, considering the simplicity of the model, we achieve a satisfactory matching between model prediction and experimental results.

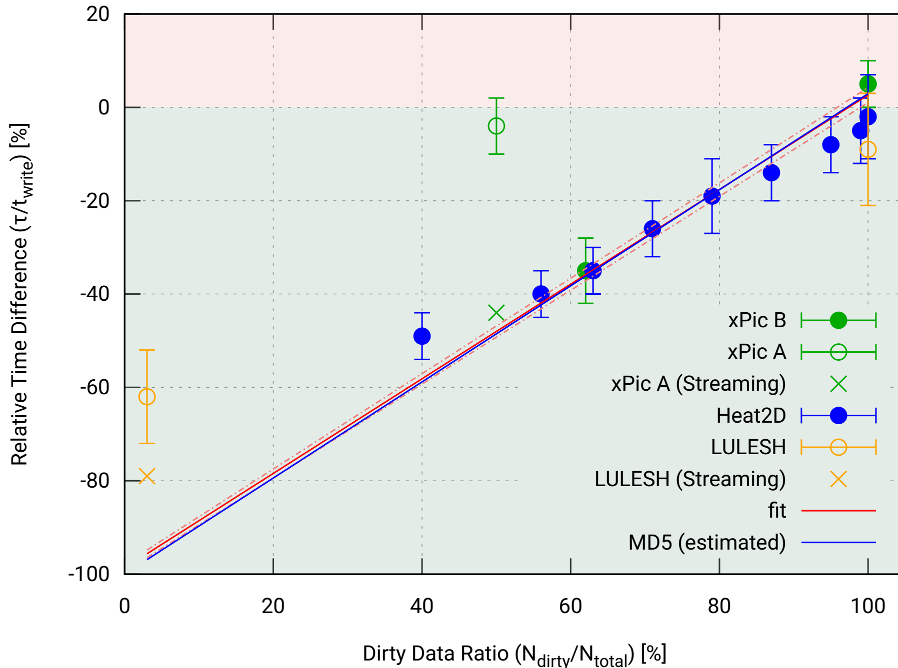


Figure 5.8: Measured and estimated speedup/overhead of DCP updates. The green background indicates the region where we have speedup and the red region indicate overhead. $\tau/t_w = 0$ corresponds to the threshold (i.e., the full CP baseline) The datasets with the label *corrected*, refer to measurements that used a buffer to collect small chunks in order to avoid small chunck writes.

5.7 Related Work

The library *libckpt* [66] can be operated almost[2] transparently (i.e. without modifying the application code). Without instrumentation, the library will checkpoint the full address space of the application.

[2]The routine `main` in C has to be renamed into `ckpt_target`, and the main `PROGRAM` module in Fortran has to be changed into `SUBROUTINE ckpt_target`. Other than that, the library can be operated transparently.

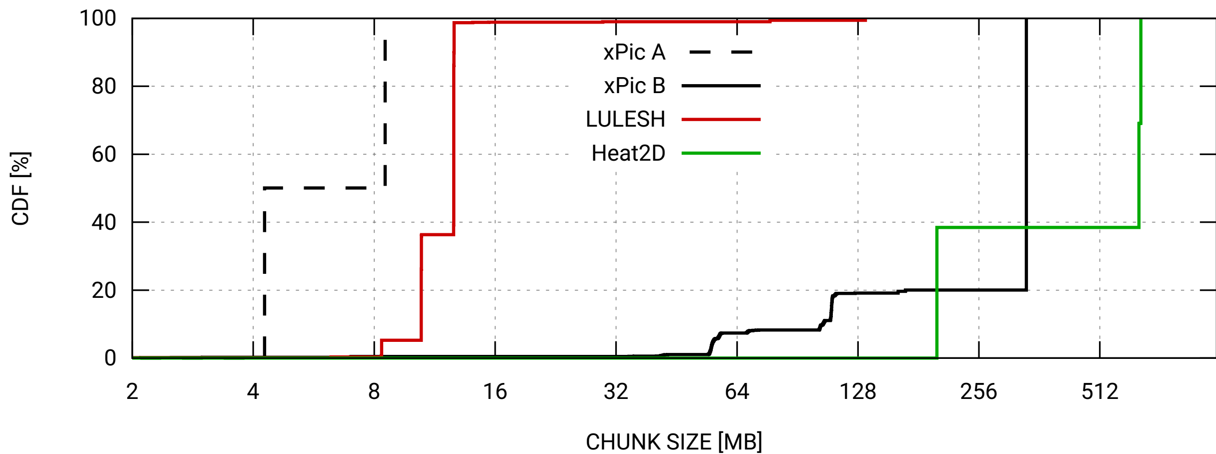


Figure 5.9: Cumulative distribution function (CDF) for chunk sizes of contiguous dirty regions during DCP updates.

However, by protecting the whole address space, one incorporates data that is not necessarily needed for a successful restart. For this reason, libckpt provides an API to explicitly define certain regions to be included/excluded from the checkpoint, and furthermore when to checkpoint, and where to checkpoint (i.e., checkpoint path). In order to detect data updates between consecutive checkpoints, libckpt employs the UNIX page protection mechanism; all memory pages that correspond to the process address space are set to read-only after each checkpoint. Every store operation to one of the protected pages will rise a segmentation fault signal (`SIGSEGV`) which invokes the libckpt signal handler. Inside the handler, the address of the page is marked dirty and will be written to disk during the next checkpoint.

This approach has several limitations. First, applications may update continuously all the datasets, which does not imply that the value after the update differs from the one before (e.g., zeros in a domain). Applications with dynamic variable sizes sometimes need to reallocate buffers, which may lead to a relocation of the variable to a different memory address without changing the data contained in the old buffer. In these cases, the page protection mechanism will not lead to a significant reduction of data. Second, the signal raised by the application may be miss-interpreted by the library, and may interfere with a signal handler of the application itself. Third, each call to the signal handler interrupts the application and imposes direct overhead to the execution. Fourth, the number of files steadily increases on the file system, since all the updates (i .e., files) are needed for the reconstruction, and further each open/close operation adds overhead for the reconstruction upon recovery.

Some of the limitations above are addressed by the library itself, for instance, a threshold for the number of files can be given, after which the files are consolidated into one file. If performed in the background, this can reduce the overhead on the reconstruction and reduces the stress on the meta-data server. Further, Ferreira et al . [68, 86], and Plank [67], proposed methods to account for variables that access, but not change contents of the memory regions. Ferreira et al. creates hashes of the memory pages that are accessed while being protected. By the time of the checkpoint, the hashes can be compared to the hashes of the memory page at the time of the checkpoint. While this can reduce the amount of redundant data further, it does not account for relocations of the data, and imposes additional overhead within the signal handler. Plank suggests another elegant way accounting for the real differences. During the signal handler, the memory page is copied entirely to a buffer, and by the time of the checkpoint, the current data and the copy are compared with XOR and the result is compressed and afterwards added to the checkpoint. The advantage here is that the XOR operation produces many zeros, if the data has

not changed much. And zeros are easy to compress. While these methods consider the actual changes in variables, they further increase the overhead during runtime.

Our implementation addresses all those issues, making this proposal the only general purpose multi-level checkpointing library that implements a version of differential checkpointing that adapts to datasets with dynamic sizes, and which is not intrusive, avoiding the invocation of a signal handler. Furthermore, our implementation avoids the generation of secondary files, as it incorporates the meta-data in the checkpoint files, reducing the stress on the meta-data server and minimizing the IO operations upon recovery.

5.8 Conclusion

In this chapter, we presented a per variable based DCP mechanism. Former DCP implementations are based on the UNIX page protection mechanism to determine data differences, but this mechanism is not capable of recognizing all the differences correctly, as we have pointed out in section 5.7. We tested four hash algorithms: Adler32, Fletcher32, CRC32 and MD5 upon performance and reliability and our conclusion is that from those four, only CRC32 and MD5 are safe choices for DCP.

Our proposed DCP implementation accounts for relocation of data in memory and for dynamic variable sizes. For this, we developed two new file formats, FTI-FF (subsection 5.2.1) for optimized storage utilization, and a streaming file format (subsection 5.2.4) for alleviating the impact of many very small updates. Our results demonstrate, that the application of DCP is beneficial in most cases. We further presented a model (Equation 5.4), which can be used to estimate the benefit of DCP. For this, we need to measure the time to write and the time to hash a block of data. The model can then predict the theoretical speedup, depending on the dirty data ratio. We also identified an issue with our first implementation of FTI-FF, when it comes to very small DCP updates. However, after using a buffering mechanism, this issue has been resolved. Our experiments with three different HPC applications show speedups of up to 49% in Heat2D, 35% in xPic and 62% in LULESH.

Chapter 6

Elastic Recovery

Publications

- International Conference on High Performance Computing, Data, and Analytics (HiPC) 2020:
  - **Keller, K.**, Parasyris, K., Bautista-Gomez, L. (2020, December). **Design and Study of Elastic Recovery in HPC Applications.** In 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC) (pp. 261-270). IEEE.

Main Contributions

- We propose API extensions for checkpoint-and-restart libraries that target both objectives resiliency and scientific IO, and enable the automatic recovery with an arbitrary number of processes (elastic recovery).
- We design and implement a runtime allowing the asynchronous creation of HDF5 checkpoint files.
- We showcase online recovery using a resilient MPI implementation (ULFM), and demonstrate automatic elastic recovery on fewer processes.

During the last decade, Checkpoint-Restart (CR) has been highly optimized by multilevel checkpoint libraries like Fault Tolerance Interface (FTI) [14], Scalable Checkpoint/Restart (SCR) [15] and Very Low Overhead Checkpointing System (VeloC) [87]. These libraries offer numerous settings to improve the checkpoint performance by taking advantage of multiple storage levels, and the utilization of modern checkpointing techniques such as Differential Checkpointing (DCP), asynchronous checkpointing, incremental checkpointing, Graphics Processing Unit (GPU) checkpointing, and in-memory checkpointing. Despite their high performance, multilevel CR libraries still suffer from some shortcomings. They often use an opaque file format, and the data layout in the checkpoint files is agnostic to the developer. Although High Performance Computing (HPC) applications typically run with a variable number of processes, CR libraries do not support recovery from a checkpoint with an adjustable number of processes (a.k.a. elastic restart). Consequently, the recovery from a failure needs to take place with the exact number of processes as the application was formerly executed with. On the other hand, the developer cannot extract the data from the checkpoint files manually and restart on a changed decomposition of the application without knowledge of the underlying file format. The same applies for utilizing the checkpointed data for visualization or data analyses, as the developer cannot extract the data from the checkpoint files.

We propose a new API and runtime, for Checkpointing (CP) libraries, that empowers developers to expose additional information for the datasets (e.g., decomposition, variable names, etc.), and to control the data layout in the checkpoint files. As the checkpointed data is often also the object of the analysis, we make two friends with one gift; the application is protected, using all the features of the CP library, and stores the data into a format that contains all information to perform further analyses. Moreover, once the application has been instrumented with our API, it can be restarted with an arbitrary number of processes.

The remainder of the chapter is structured as follows: Section 6.1 explains the foundation of our work. Section 6.2 focuses on the design and implementation of our proposed API. Section 6.3 describes

the methodology for our detailed analysis, and section 6.4 presents and discusses the results of our experiments. Section 6.6 explores related work, and Section 6.7 concludes the chapter.

6.1 Background

In this section, we introduce the main libraries and concepts necessary to understand the contributions presented in this chapter.

6.1.1 MPI Layer Fault Tolerance

The most common technique used to protect applications towards failures is to periodically create checkpoints and, in case of a failure, roll back to the last checkpoint taken. Typically, the failure leads to the termination of the application, hence, all data stored in dynamic memory is lost and needs to be restored upon recovery. But often, the failure affects only a small region of the application domain (e.g., single node failures). Hence, most of the application data is still valid, and a replacement and recovery of only the failed processes would be sufficent. But Message Passing Interface (MPI) does not support such scenarios innately. Fault-tolerance inside the MPI layer has been an object of investigation for some years now [88–90]. Some popular proposals for fault tolerant MPI implementations are ULFM [91], FT-MPI [90] and MPI_Reinit [92, 93]. The common goal of these frameworks is to provide a mechanism for the developers to mitigate local failures in MPI. ULFM, for instance, proposes extended semantics to the MPI specification that allows one to exclude failed processes from the MPI communicator, or to invalidate communicators [94]. The goal of all of the fault tolerant MPI implementations is to empower the developer of detecting process failures and to reconfigure the application accordingly during runtime.

6.1.2 General Purpose IO

6.1.2.1 HDF5

Hierarchical Data Format (HDF5) [95] is a file format that allows the storage of complex datasets embedded inside a hierarchical folder-like structure [96]. Inter alia, HDF5 allows the creation of named groups (similar to folders in file systems) and named datasets (continuing the analogy, the files in a file system). The HDF5 file format represents the standard format for scientific datasets. The API is very complete and actively maintained. Furthermore, there exist bindings to a great variety of programming languages and applications such as C/C++, Fortran, Python, MatLab and R, and it has been optimized for multiple file systems [97, 98].

6.1.2.2 ADIOS

Adaptable Input Output System (ADIOS) [99] is a state-of-the-art IO library for HPC applications. ADIOS provides a rich API to define variables with additional information for shape, type, etc. The library allows one to write the data in different formats, and allows conversion of the own file format into various others. Under the term Sustainable Staging Transport (SST) [100], ADIOS also provides asynchronous file Input and Output (IO). In combination with HDF5 as output format, we can stage the single file HDF5 data locally before it is consolidated asynchronously on the Parallel File System (PFS). For this, the data is

first buffered locally, and then accessed via Remote Direct Memory Access (RDMA), and consolidated to a single file on the PFS. ADIOS provides bindings to C/C++, Fortran and Python.

6.2 Implementation

In this section we outline the design objectives of our extensions and present the basic concepts and mechanisms. The proposed interface gives developers a tool at hand that solves the disagreement mentioned at the introduction to this chapter. In short, it consolidates the IO work for resiliency with the IO work for data processing and it also allows for elastic restart from the checkpoint files using a variable number of processes. Deploying our extensions does not restrict the other features of the CR library in any way. It remains possible to perform checkpoints in all other reliability levels. We implemented our extensions on top of FTI (without loss of generality).

6.2.1 Design Objectives

The design objectives for our API extensions are:

Complex Data Representation: The core of the problems that HPC applications try to solve is formulated using complex data structures. These structures are usually organized either as structure of arrays or as an array of structures. Our interface should provide an intuitive mechanism to define both.

Accessibility to the Data: Checkpoint libraries in HPC do typically not intend to provide access to the Checkpoint (CKPT) data from outside of the library. Our objective is to remove this restriction and to give the developer the capacity of structuring the data inside the checkpoint in a way, that it can conveniently be used for data processing, independently of the checkpoint library.

Intuitive API: Developers prefer using libraries and APIs which they already know before investing their time to learn a new specification. Therefore, we apply the terminology of common file formats (i.e., HDF5 and NetCDF [101]) to our API, providing descriptive function names.

6.2.2 API Specification

In this section we showcase the proposed API extensions. We use a simple example to demonstrate how the extensions can be used to structure the data inside the checkpoint file. This will clarify i) how to organize the data for scientific post-processing and visualization and ii) how to prepare the application buffers to allow the elastic recovery.

6.2.2.1 Complex Data Representation

We exemplify our API using a simple example that simulates the movement of particles exposed to a force in 3-dimensional space. Each particle state is represented by its position and velocity (see Listing 6.1).

```
1 typedef struct coord_t {
2     double x,y,z;
3 } coord_t;
4
5 typedef struct particle_t{
```

```
6     coord_t position;
7     coord_t velocity;
8 } particle_t;
```

Listing 6.1: The particles data type is represented by a structure with two members, which are represented by the respective data-structures. This is an example for a nested composite data type. We explain how to expose this type with our API in Listing 6.2.

Our interface provides mechanisms to describe complex datasets of the application and store those to the checkpoint file with the corresponding information about shape, type and relationship. The first step is to expose the data types of the datasets to the checkpoint library. Besides the standard types (integer, floating-point, char, etc.), which are predefined, the user can define derived data types that correspond to structures or classes. Composite data types, for instance the `particle_t` datatype from Listing 6.1, are created by calls to `FTI_InitCompositeType`. The members of the composite data type can be added one after the other by calling `FTI_AddScalarField` for scalar members and `FTI_AddVectorField` for array members. Listing 6.2 shows this in detail for the composite types from Listing 6.1. Note that, because the particle type has members that are of a composite type itself, we first need to create this type.

```
1 fti_id_t FTI_COORD = FTI_InitCompositeType("COORD", sizeof(coord_t), NULL);
2 FTI_AddScalarField(FTI_COORD, "X", FTI_DBLE, offsetof(coord_t, x));
3 FTI_AddScalarField(FTI_COORD, "Y", FTI_DBLE, offsetof(coord_t, y));
4 FTI_AddScalarField(FTI_COORD, "Z", FTI_DBLE, offsetof(coord_t, z));
5
6 fti_id_t FTI_PARTICLE = FTI_InitCompositeType("PARTICLE", sizeof(particle_t), NULL);
7 FTI_AddScalarField(FTI_PARTICLE, "POSITION", FTI_COORD, offsetof(particle_t, position));
8 FTI_AddScalarField(FTI_PARTICLE, "VELOCITY", FTI_COORD, offsetof(particle_t, velocity));
```

Listing 6.2: To expose the nested particle data type from Listing 6.1, we need to expose the coordinate type first. Composite data types are defined with `FTI_InitCompositeType`. Their members are added with `FTI_AddScalarField` (or `FTI_AddVectorField` for array members).

6.2.2.2 Descriptive Data Representation

Now that we learned how to expose composite data types, we will explain the definition of global datasets. By global dataset, we are referring to the full dataset, before its decomposition to the application ranks. The global properties of the dataset are its description (name), the location in the file (parent group) and the dimensions. They are exposed by calling `FTI_DefineGlobalDataset`. Local properties, such as offset and count of the rank's data share, are exposed by calls to `FTI_AddSubset`. Finally, the actual data buffer is exposed via `FTI_Protect`. Listing 6.3 exemplifies the required steps exposing a global particle dataset. The dimensions of the global dataset in our example are represented by the three variables `gX`, `gY` and `gZ`, whereas the dimension of the subsets on each process are represented by the variables `lX`, `lY` and `lZ`. Scalar variables, such as the iteration counter or communicator size, that take the same value on all ranks, can be defined in the same way. In that case, the values for count and offset are simply one and zero on all ranks.

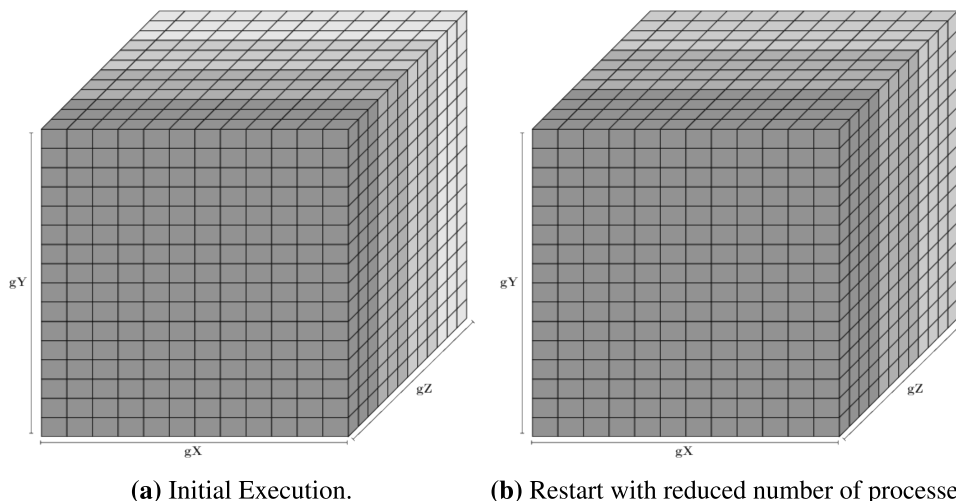
```
1 lX = gX;
2 lY = gY;
3 lZ = gZ/nbProcs;
4
5 size_t MEMSIZE = lx*ly*lz*sizeof(particle_t);
6 particle_t *particles = (particle_t*)malloc( MEMSIZE );
```

```
7
8  int globalDim = {gZ, gY, gX};
9  size_t offset[3] = {0, 0, rank*lZ};
10 size_t count[3] = {lZ, lY, lX};
11
12 FTIT_H5Group GRID;
13 FTI_InitGroup(&GRID, "GRID", NULL);
14
15 int VAR_ID_PARTICLES = 0;
16 int DATASET_ID_PARTICLES = 0;
17
18 FTI_DefineGlobalDataset(DATASET_ID_PARTICLES, 3, globalDim, "PARTICLES", &GRID, FTI_PARTICLE);
19 FTI_Protect(VAR_ID_PARTICLES, particles, lX*lY*lZ, FTI_PARTICLE);
20 FTI_AddSubset(VAR_ID_PARTICLES, 3, offset, count, DATASET_ID_PARTICLES);
```

Listing 6.3: In this example, we create an HDF5 group `GRID` to show how to create a hierarchy in the checkpoint file. The particle dataset is then added to this group. Further, we define the global dimensions of the dataset (line 15) and specify the process local region inside the dataset (line 17) according to the domain decomposition (here we merely partition along the z-axis.)

6.2.3 Accessing the Checkpoint Data

For the current implementation, the underlying file format is HDF5. However, the API is not restricted to a specific format. Hence, the implementation can be extended, adding other formats like netCDF [102] or ADIOS. The proposed API can be used to create a hierarchical structure (groups) for the datasets inside the checkpoint file. Therefore, accessing the data in the files is straightforward, as the developer has defined the structure by himself. The files are created with the chosen file format and can be accessed with any binding that might be available for it. With this, checkpointing is not anymore just overhead for the purpose of resiliency, but also serves as an interface for the scientific application IO.



(a) Initial Execution. **(b)** Restart with reduced number of processes.

Figure 6.1: The entire grid of the data used by the application. The different gray scales indicate the domains of the MPI processes. (a) shows the initial execution with 4 MPI processes and (b) the execution with 3 processes after the elastic recovery.

6.2.4 Elastic recovery

The information that is provided through the two functions `FTI_DefineGlobalDataset` and `FTI_AddSubset` is used by FTI to recover elastically with any number of application processes.

Providing the information is simple in most cases, since it coincides with the information about the domain decomposition for the parallelization with MPI. However, the developer needs to explicitly instruct FTI that an elastic recovery is requested. For this, the `failure` tag in the FTI configuration files has to be set to `3`. Otherwise, FTI assumes a restart with the same number of processes. To request global checkpoints with HDF5, the developer further needs to pass the checkpoint level `FTI_L4_H5_SINGLE` to `FTI_Checkpoint`. All the other levels are still available and can be set passing the respective flag (e.g., `FTI_L1`, `FTI_L2`, etc.).

The decomposition in our example is simple (Listing 6.3). Only the z-axis is decomposed between the ranks. In figure 6.1, we show the global view of this domain decomposition. The left cube shows the initial decomposition into 4 processes and the right cube the decomposition for the recovery into 3 processes. The offset and count in listing 6.3 are defined in such a way that the correct values are computed automatically for the current decomposition, i.e. according to the number of processes that participate in the current execution. Thus, for the elastic recovery with a different decomposition, the appropriate information about offset and count for the subsets is automatically exposed to FTI.

6.2.5 Checkpoint Strategies

We want to showcase with a simple example how we can use different resilience measures, including the descriptive file creation as a checkpoint level on its own. The resiliency strategy should comprise a combination of all reliability levels available. Most of the CR libraries provide very fast and highly scalable checkpoint levels (node-local checkpointing, buddy checkpointing, etc.). Those levels are suitable to perform checkpoints at a high frequency. Those checkpoints do not require to be in a file format suitable for extracting the scientific application data. They target minimizing the time for recomputation upon failures. Therefore, the best checkpointing strategy is typically a mixture of all levels. Listing 6.4 shows an example of a simple checkpoint strategy that uses 4 different levels of reliability.

```
1  int iter;
2  for (iter=1; iter<=MAX_ITER; iter++){
3      if( FTI_Status() != 0 ) FTI_Recover();
4      // structured and globally shared HDF5 file (global file system)
5      else if(iter%8000 == 0) FTI_Checkpoint(iter, FTI_L4_H5_SINGLE);
6      // encoded checkpoint files (node-level)
7      else if(iter%4000 == 0) FTI_Checkpoint(iter, FTI_L3);
8      // partner checkpointing (node-level)
9      else if(iter%2000 == 0) FTI_Checkpoint(iter, FTI_L2);
10     // single checkpoints (node-level)
11     else if(iter%1000 == 0) FTI_Checkpoint(iter, FTI_L1);
12     simulateSystem(grid);
13 }
```

Listing 6.4: Example of a simple checkpoint strategy that combines checkpointing into a shared HDF5 file with other levels of reliability. Higher levels are preferred. The recovery is performed automatically upon restart.

6.2.6 Asynchronous Checkpoint

The checkpoint creation into the global self-descriptive checkpoint file can optionally be performed in two stages. Initially, the application processes store the data to fast local storage on the nodes (stage 1). Afterwards, the local data are united to a global file on the PFS (stage 2). The first stage is performed by the application processes, and the second stage by dedicated FTI processes. The application will

only experience the overhead of the first stage, since the second stage is performed in the background (*asynchronous checkpoint*). To leverage the asynchronous feature, the application has to allocate one extra process per node. The dedicated FTI processes are removed from the main communicator during the call to `FTI_Init`. Therefore, the application needs to use the FTI communicator (`FTI_COMM_WORLD`) after calling `FTI_Init` to function properly.

6.3 Methodology

In this section, we will introduce the experiments that we have performed and the application we used. We will further introduce the metrics that we apply to evaluate the performance of the proposed runtime.

6.3.1 Generalized Evaluation Metric

To evaluate the performance of our implementation, we propose a metric that provides a clear definition of the checkpoint and recovery overheads. The metric is suitable for comparing results among experiments performed at different times and architectures. The metric depends on, t_{ckpt}, the effective time to checkpoint (i.e., experienced by the application), the effective time to recover (i.e. without downtime and recomputations), and the Mean Time Between Failures (MTBF). The metric is define as:

$$\delta_{\text{ckpt}} = \sqrt{\frac{t_{\text{ckpt}}}{2\,\text{MTBF}}} = \frac{t_{\text{ckpt}}}{t_{\text{opt}}}, \quad \text{relative checkpointing overhead} \tag{6.1}$$

$$\delta_{\text{reco}} = \frac{t_{\text{reco}}}{\text{MTBF}}, \quad \text{relative recovery overhead} \tag{6.2}$$

where $t_{\text{opt}} = \sqrt{2\,t_{\text{ckpt}}\,\text{MTBF}}$ is Young's optimal checkpoint interval [103]. The relative overheads defined in equations 6.1 and 6.2 can further be used to estimate the respective absolute overheads of our proposed runtime. In the following we will demonstrate how to do that. The total overhead imposed by the protections and recoveries from failures is:

$$\Delta T_{\text{tot}} = \Delta T_{\text{ckpt}} + \Delta T_{\text{reco}} + \Delta T_{\text{corr}}. \tag{6.3}$$

Where ΔT_{ckpt} is the total checkpoint time for all checkpoints, ΔT_{reco}, the total recovery time for all recoveries (i.e., failures), and ΔT_{corr} is the correction, accounting for the recomputations and downtime between failure and recovery. For the total checkpoint and recovery overhead we can write:

$$\Delta T_{\text{ckpt}} = N_{\text{ckpt}} t_{\text{ckpt}} = \frac{T_{\text{exp}}}{t_{\text{opt}}} t_{\text{ckpt}} = \delta_{\text{ckpt}} T_{\text{exp}} \tag{6.4}$$

$$\Delta T_{\text{reco}} = N_{\text{fail}} t_{\text{reco}} = \frac{T_{\text{exp}}}{\text{MTBF}} t_{\text{reco}} = \delta_{\text{reco}} T_{\text{exp}} \tag{6.5}$$

With T_{exp} being the expected runtime time of the application without protection and failures, N_{ckpt} the number of checkpoints, and N_{reco} the number of recoveries after a failure. In Equation 6.4, we assume that the checkpoint interval is t_{opt}, applying the Young's optimal checkpoint interval. Equations 6.3, 6.4, and 6.5 can now be used to express the total overhead by:

$$\Delta T_{\text{tot}} = (\delta_{\text{ckpt}} + \delta_{\text{reco}}) T_{\text{exp}} + \Delta T_{\text{corr}} \tag{6.6}$$

We can further give an estimation for the correction term by:

$$\begin{aligned}\Delta T_{\text{corr}} &\approx N_{\text{fail}} \left(E[t_{\text{recomputation}}] + E[t_{\text{downtime}}] \right) \\ &\approx N_{\text{fail}} \left({}^{\text{MTBF}}\!/\!_{2} + E[t_{\text{downtime}}] \right) \end{aligned} \tag{6.7}$$

where $E[\cdot]$ denotes the expected value. Because otherwise the model would need to take into account the times for additional checkpoints and recoveries, we further require:

$$\Delta T_{\text{corr}} + (\delta_{\text{ckpt}} + \delta_{\text{reco}}) T_{\text{exp}} < \text{MTBF} \tag{6.8}$$

6.3.2 Measurements

To determine the checkpoint and recovery overhead, we performed three different types of experiments:

E0 Execution without protections (i.e., T_{exp}).

E1 Execution without failures, performing N_{ckpt} checkpoints.

E2 Execution interrupted by one failure + recovery, and performing N_{ckpt} checkpoints.

The respective times for recovery and checkpointing are then given by:

$$\Delta T_{ckpt} = T(\text{E1}) - T(\text{E0}) \tag{6.9}$$

$$\Delta T_{reco} = T(\text{E2}) - T(\text{E1}) \tag{6.10}$$

From this, we can compute the quantities of our proposed metric by:

$$\delta_{\text{ckpt}} = \frac{t_{\text{ckpt}}}{t_{\text{opt}}} = \frac{{}^{\Delta T_{\text{ckpt}}}\!/\!_{N_{ckpt}}}{t_{\text{opt}}} \tag{6.11}$$

$$\delta_{\text{reco}} = \frac{t_{\text{reco}}}{\text{MTBF}} = \frac{\Delta T_{\text{reco}} - T_{\text{recompute}}}{\text{MTBF}} \tag{6.12}$$

6.3.3 Experiments

To assess the performance of the proposed runtime, we need to compare it against a similar approach of an existing state-of-the-art mechanism. First, we test our runtime against the fourth reliability level of the traditional interface of FTI. FTI creates one checkpoint file per process on the PFS at this level. Experiments that use the traditional interface are labeled with **Trad** and experiments leveraging our proposed runtime are labeled with **Novel**. We further measured both interfaces in asynchronous mode labeled **Async**, and in synchronous mode labeled **Sync**. Furthermore, we tested the novel interface performing the recovery *online*, that is without terminating the application upon a failure, and *offline*, terminating the application upon failures. The recovery configurations are labeled **On** and **Off** respectively. For the online recovery, we leverage ULFM (See Section 6.1.1). We listed the various configurations and their labels in Table 6.1.

File Format		Checkpoint Methodology		Recovery methodology	
Trad	Traditional interface, binary file format (N-N)	**Sync**	Checkpoint post processing by application processes	**Off**	Offline recovery; application terminates upon failure.
Novel	Checkpointing into shared HDF5 file (N-1) for independent data post processing and elastic restarts	**Async**	Checkpoint post processing by the dedicated processes	**On**	Online recovery; application stays alive upon failure, reinitialization and restart with remaining processes.

Table 6.1: Different C/R scenarios tested in the evaluation section with respect to the file format, the checkpoint method and the recovery method.

6.3.4 Applications

We want to study how the proposed runtime behaves with different application types. The HPC ecosystem has a wide spectrum of scientific codes, some of them are based on regular static grids that are easy to work with, while others are more irregular and significantly harder to work with. This complexity has an impact on both checkpoint and recovery. We selected two applications representing the boundaries of both sides of the spectrum. Heat2D, an application with a very simple domain decomposition and regular data distribution, and xPic, an application with a complex decomposition and irregular data distribution (particles move between domains). In the following, we briefly discuss the peculiarities of both applications.

6.3.4.1 Heat2D (C++)

As benchmark application we use Heat2D that was introduced in subsubsection 5.5.1.3. To use it for automatic online recovery, we ported it to C++. In the new implementation, the body of the application is arranged in three parts: (a) initialization, (b) mainloop, and (c) finalization. This structure is very common for HPC applications. In spite of its simplicity, Heat2D is a good representative of many HPC codes (i.e., stencils) and provides a simple example for show-casing the elastic recovery with fewer processes within an online scenario (i.e., without termination upon failures).

6.3.4.2 xPic

As an example for an application that shows irregular IO patterns, we use xPic, a Particle In Cell (PIC) application for space plasma simulations. We introduced xPic in subsubsection 5.5.1.2, however, we want to emphasize certain properties that are important to understand the evaluation section. The simulation space in xPic is decomposed into a 3-dimensional grid of cells and each cell is initialized with a certain amount of particles at the beginning. The number of cells for each direction is customizable as is the number of particles per cell. The electric and magnetic fields are discretized and defined at fixed grid points (nodes). On the contrary, the particle positions are continuous and the particles can move between the cells of the grid. As a consequence, the number of elements per cell is constant for the field arrays, but not for the particle arrays, which is a challenging scenario for the elastic recovery. Each rank operates on a sub-volume of the grid, dictated by the MPI decomposition. The global field data in xPic is organized in contiguous arrays, one for each component (i.e., X,Y, and Z components of the magnetic and electric

fields). We organize the fields in the global checkpoint file into a row-major data alignment (3D to 1D mapping). Since the grid in xPic is static, each rank is able to compute offset and count for its share of the global field data according to the domain decomposition. The same applies for the recovery (elastic and common recovery). Thus, the application of elastic recovery is straight forward for the fields. This does not apply to the particles. Given that particles move around the grid during the execution, the number of particles per cell varies, thus, at checkpoint time the ranks cannot know the offset in the shared file without communication among the other ranks. Even more challenging is the recovery in that case. Because of that, xPic is a good example of an irregular application to demostrate the generality of the proposed API and runtime. We will explain in more detail how we approach elastic recovery for the particles in section 6.4.7, where we will present two different solutions for the problem.

6.4 Evaluation

In this section, we evaluate the performance of our implementation, compare it to a state-of-the-art IO library (ADIOS), and demonstrate its scalability. We investigate the benefits of online recovery, and evaluate the overhead that is imposed due to the increased execution time when restarting with fewer processes. Finally, we analyse some of the challenges of elastic restart on irregular applications and we present two solutions that achieve good performance for both checkpoint and recovery.

6.4.1 HPC Environment

All experiments are performed on MareNostrum4, the supercomputer at the Barcelona Supercomputing Center (BSC). Each compute node is equipped with 2 Intel Xeon Platinum CPUs (24 cores each), 12x8 GB DDR4 main memory, a 100Gbit network and 10Gbit ethernet to the PFS [104]. All local checkpoints are performed in-memory, using the node's RAM disk (`/dev/shm`).

Evaluation Parameters	xPic	Heat2D
Nodes	16	16
MPI-processes per Node	12	47
Threads Per Rank	3	1
CP size per Node (GB)	11	18
Num of MPI-ranks	192	752
Total CP size (GB)	176	288

Table 6.2: Configuration and scale for the benchmark experiments.

6.4.2 Performance Measurements

Recent studies show that modern HPC systems have several failures per day [105, 106], we will use a benchmark value of 6 hours for the MTBF for our experiments. This choice is based on an estimated MTBF for executions on 15K cores [107]. The methodology for the experiments has been presented in section 6.3 and the scale is listed in Table 6.2. In this section, we will present the results for the checkpoint and recovery overhead for the novel and traditional interfaces. We performed experiments in synchronous and asynchronous mode. For the particle recovery in xPic from the global file, we load an even amount of particles on each rank, and afterwards redistribute particles that belong to different

domains. This strategy is optimized for fast checkpoint creation. We observe generally low overheads for recovery and checkpoint for both applications ($\delta_{\text{ckpt}} < 3\%$ and $\delta_{\text{reco}} < 1\%$). The results are listed in Table 6.3. We further observe, that most of the recovery overhead in xPic is due to the particle redistribution (see Figure 6.2). We will compare the approach presented here to an approach without particle redistributions in subsection 6.4.7.

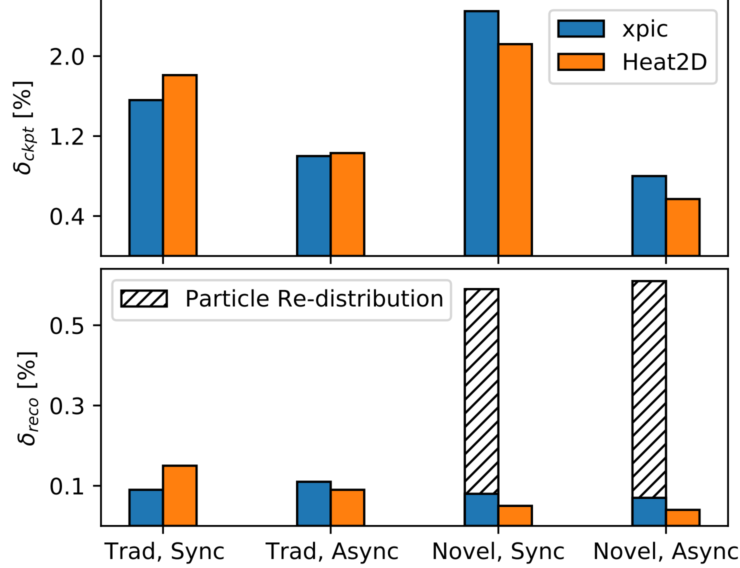


Figure 6.2: Top: Relative checkpoint overhead. Bottom: Relative recovery overhead. For xPic we separated the recovery overhead into read and re-distribution of the particles.The difference in latency between online and offline recovery depending on the number of nodes.

Compared to the traditional interface, our proposed runtime achieves slightly higher values for synchronous checkpointing and slightly lower values for asynchronous checkpointing. The latter results from the fact that we do neither create metadata files nor integrity checksums for the checkpoint files. With the help of equation 6.6 we can further estimate the minimum and maximum overhead for an estimated execution time of 12 hours (i.e., $T_{exp} = 12$ h):

$$\Delta T_{tot} \approx \begin{cases} 22\,(16)\ \text{[min] xPic (Heat2D)} & \textbf{Synchronous} \\ 10\,(4)\quad \text{[min] xPic (Heat2D)} & \textbf{Asynchronous} \end{cases}$$

Overall, these results demonstrate that our proposed technique performs comparably well to the state-of-the-art multilevel checkpoint techniques that leverage the local storage of current deep-memory architectures. The overhead imposed on both applications is extremely low ($< 1.5\%$) when applying asynchronous post-processing, while having the benefit of global data files in a self-descriptive data format.

6.4.3 Comparison to ADIOS

After comparing the performance of the proposed runtime to the traditional runtime of FTI, we compared the asynchronous file creation of our proposal to the asynchronous staging feature (SST) of ADIOS. The feature leverages fast memory buffers and RDMA to stage the files though the network. The shared HDF5 file is created in the background by several dedicated processes, after receiving the checkpoint data from the application processes. This feature serve the same purpose as our asynchronous checkpoint

		δ_{ckpt} [%]		δ_{reco} [%]	
		xPic	**Heat2D**	**xPic**	**Heat2D**
Trad	Sync	1.56	1.81	0.09	0.15
	Async	0.95	1.03	0.11	0.09
Novel	Sync	2.40	2.12	0.59	0.05
	Async	0.80	0.57	0.61	0.04

Table 6.3: Results for the relative CR overheads.

implementation (subsection 6.2.6). ADIOS can also stage the files through node storage. However, the consolidation of the local data needs manual action and is not performed from within the application itself. Therefore, this approach differs substantially from the functionality of our implementation, and we omit a comparison. ADIOS allows allocating an arbitrary number of dedicated processes, which can as well be located on a separate set of nodes. Due to the limited (and not customizable) buffer size per dedicated process, we needed to allocate as many staging processes as application processes. This can be a disadvantage, as it increases considerably the number of processes required in the allocation. Our implementation does not impose such a limitation, however, is less flexible in the number of workers that perform the consolidation of the checkpoint file. The number of dedicated staging processes in FTI is always one dedicated process per node. Therefore, in ADIOS, the consolidation of the checkpoint file can be scaled up with additional worker processes, whereas in FTI it can not.

		t_{ckpt} **(sec)**	t_{reco} **(sec)**	**Bandwidth (GB/s)**
Novel	Sync	19.7	22.6	14.6
	Async	0.92	6.9	157.9
ADIOS	Sync	18.9	21.4	15.2
	Async	5.14	10.4	28.2

Table 6.4: The resulting values for the relative CR overheads for N-1 (shared HDF5 file on PFS). Comparison between our proposal and ADIOS.

Overall, the flexibility of ADIOS' staging feature did not match the performance of our asynchronous checkpoint implementation. Table 6.4 summarizes the results of our measurements. Writing directly to the PFS (Sync) does not show any significant difference in performance between the two libraries. When staging the data though (Async), we achieve a significant performance advantage. Our implementation is over five times faster, writing the checkpoint files asynchronously, which could be considered as a 5x increase in terms of bandwidth. This difference is mostly because ADIOS stages the data over the network, hence, with more effort on developers side, as mentioned earlier, this gap could be closed. An important takeaway from this experiment is that with our interface, we achieve very high performance with little effort for the developer, even in comparison to an established state-of-the-art IO library.

6.4.4 Scaling

We demonstrated the overall low overhead of our runtime and that it performs very well, even in comparison to a state-of-the-art IO library. Now we want to study its scalability. We performed experiments

that vary the checkpoint load per process (strong scaling) and experiments that vary the total checkpoint size, but keep the size per process constant (weak scaling). In the next sections we will discuss our results.

6.4.4.1 Strong Scaling

Figure 6.3 shows the outcome of the strong scaling experiments. The plot shows the scaling for an increased checkpoint load per process. This is strictly speaking not identical to a strong scaling, since the total problem size varies. However, it is very similar, as the number of processes operating on the largest checkpoint size per process (0.4 GB) is doubled going from right to left in the plot. It is more meaningful in our case, however, to show the increase in overhead with increasing problem size per process. This metric allows us to estimate the overhead for higher checkpoint loads. A fit with a weighted linear regression model on the data shows that, for the synchronous checkpoint creation, the relative overhead increases by about 3.3%/GB (4.1% at 1 GB) and for the asynchronous checkpoint creation by about 1.2%/GB (1.3% at 1 GB).

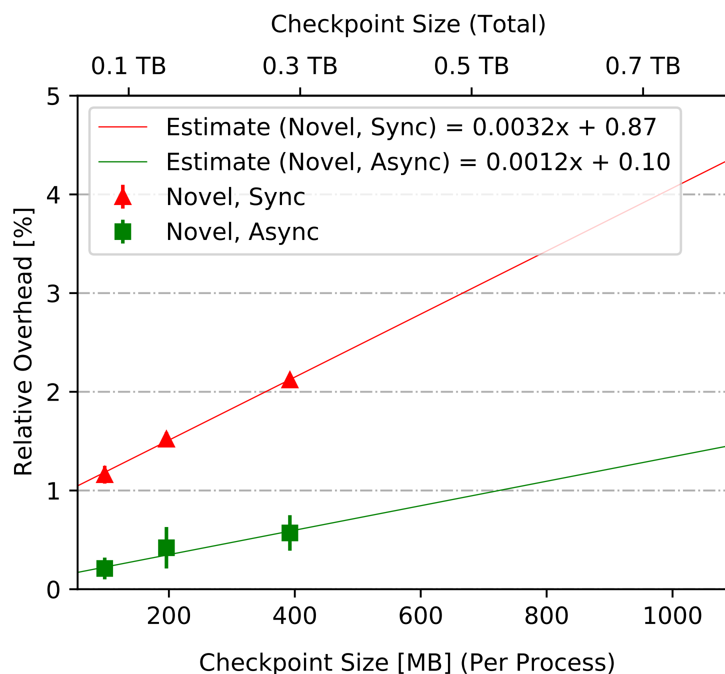


Figure 6.3: Relative checkpoint overhead for our new extensions for varying checkpoint data per process (strong scaling).

6.4.4.2 Weak Scaling

For the weak scaling we performed experiments on 16, 32, 64 and 128 nodes with 47 processes per node. The checkpoint size per process is kept constant at about 400 MB in all experiments. The results of the measurements are shown in Figure 6.4. We observe that the overhead for the synchronous checkpointing increases linearly with the number of processes. On the other hand, the overhead for the asynchronous checkpointing remains practically constant. The latter is expected, since the effective checkpoint overhead in that case, arises from the first stage of the checkpoint creation. The first stage, however, is the checkpoint creation on node storage, and with this it uses resources that are not shared globally. We can thus expect a continued constant weak scaling behavior for the asynchronous mode. Using again a weighted linear regression model, we can estimate the overheads for higher scaled runs for the synchronous mode. The fit

estimates an increase in overhead of about 0.7%/10K processes (2.8% at 10K processes). This result is only valid until we reach the maximum bandwidth of the PFS though. Furthermore, since the synchronous mode uses globally shared resources, it has to take into account other users on the cluster and potential congestions resulting from this.

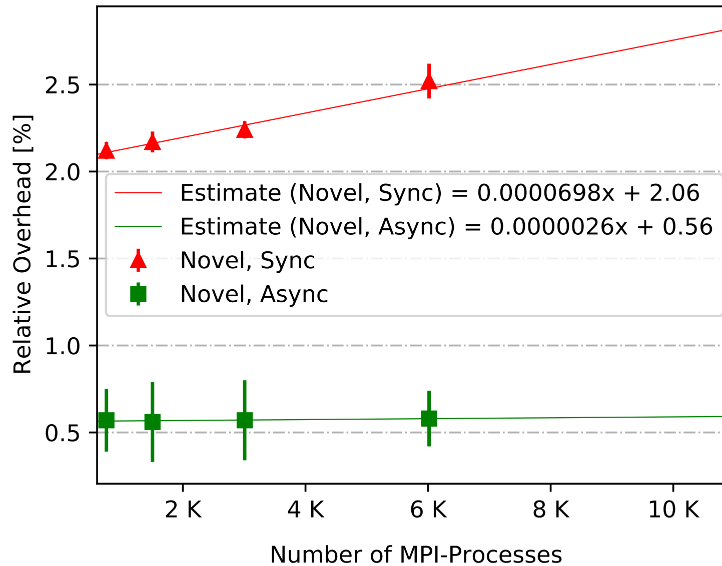


Figure 6.4: Relative checkpoint overhead for our new extensions for varying number of nodes and constant checkpoint data per process (Weak scaling).

6.4.5 Offline vs Online Elastic Recovery

Our proposed runtime can be used to perform a recovery online. For instance, after the loss of a number of processes that still allows the continuation of the application. The online recovery needs to take place on a reduced number of processes, and the recovery can be performed elastically using our proposed API. This can be achieved using our interface in combination with any fault tolerant MPI implementation. We implemented an elastic online-recovery capability into the Heat2D application leveraging ULFM. In the following we explain briefly the key facts of our implementation. First, to get notified upon a process failure, we set the standard MPI error handler to `MPI_ERRORS_THROW`. Second, we wrapped the computation block (99.99% of the execution time) of the main loop inside a try-catch statement, and further we inject failures within that region to test the implementation. After the failure injection, the affected processes terminate execution and the surviving processes throw an exception and call the error handler. Inside the handler, the processes invoke the ULFM function `MPIX_Comm_shrink` which excludes the failed processes from the MPI communicator. The remaining processes are then rolled back to the last checkpoint and continue execution. Listing 6.5 shows the handler, we have implemented in order to react to the failure.

The experiments use the same scale as the benchmarking experiments from section 6.4.2 (Table 6.2). We simulated the loss of 1 node and then restarted on the remaining 15 nodes using either online recovery (On) or offline recovery (Off) (Table 6.1). Figure 6.5 shows the time difference between the both methods. As expected, online recovery is faster than offline recovery (even if assuming an immediate restart of the job). We can see that the effect is noticeable already at a small scale. Towards large executions, the effect becomes ever more important. After applying a linear model, we estimate an increase of 57 seconds

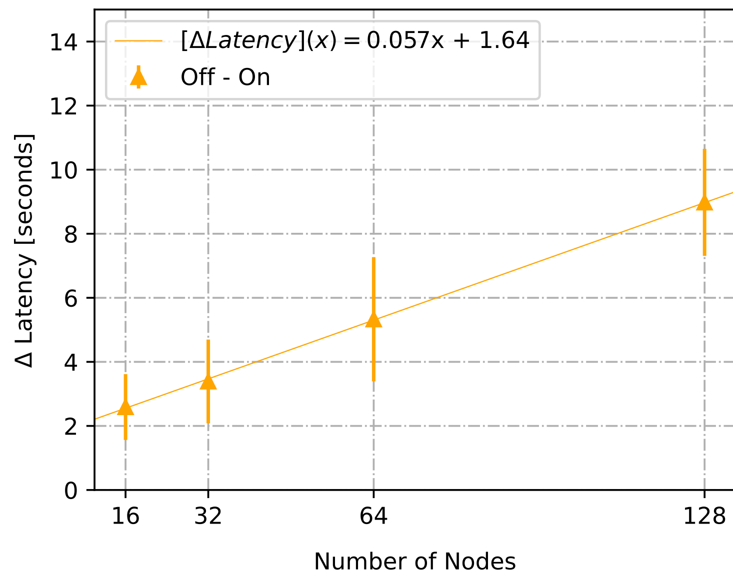


Figure 6.5: The difference in latency between online and offline recovery depending on the number of nodes.

each 1,000 nodes. However, the difference between online and offline recovery is likely to depend on the application complexity and the initial memory allocation. That is to say, the effect could be even more drastic for applications with complex initialization processes. Further, due to scaling effects, the initialization time could increase faster than linear, as we assumed in our model. With our model, we estimate a difference of about 10 minutes at a scale of 10K nodes. considering multiple failures and recoveries, which is expected to happen at such scale, this is a significant overhead which can be avoided applying online recovery.

```
1  void TDist::handle_error()
2  {
3  // shrink communicator to surviving processes
4  MPI_Comm new_application_comm;
5  MPIX_Comm_shrink( application_comm, &new_application_comm );
6  application_comm = new_application_comm;
7
8  // reinitalize FTI
9  FTI_Init( fti_config_file, application_comm );
10 application_comm = FTI_COMM_WORLD;
11
12 /* reinitialize application. This method also defines the new dimensions of the global
       datasets according to the new decomposition. */
13 this->reinit();
14
15 // update protections
16 this->protect();
17
18 // recover from failure
19 this->recover();
20 }
```

Listing 6.5: Error handling for the online recovery (R1) in Heat2D

6.4.6 Elastic Recovery with Fewer Processes

The API supports both the elastic recovery with more and fewer processes. In fact, the recovery can take place with an arbitrary number of processes. However, restarting with fewer processes can lead to load imbalances and to higher workloads per processor. Which can result into longer execution times. In this section we present a simple experiment with Heat2D, showing the tradeoff between fewer computing resources and the immediate continuation of the execution. For this, we first execute Heat2D on 50 nodes with 47 processes per node and about 400 MB checkpoint size per process. We denote the iteration time for this experiment with T_0. Furthermore, we measured the iteration time, T_i, for executions on $50 - i$ nodes operating on the same workload, for $1 \leqslant i \leqslant 10$. We define the imposed relative slowdown resulting from the execution on fewer nodes by:

$$\delta_i = \frac{T_i - T_0}{T_0}. \tag{6.13}$$

The definition of δ_i allows us to relate the iteration time after each failure that results into fewer nodes to the original iteration time. With this, we can give a formula to model the total execution time after multiple such failures. The increase in time per iteration, ΔT_i for executing on i fewer nodes is given by $\Delta T_i = \delta_i \, T_0$. With this we can compute the total additional execution time by:

$$\Delta T_{\text{tot}} = \sum_{k=1}^{F} N_k \, \delta_{i_k} \cdot T_0 \tag{6.14}$$

where F corresponds to the total number of failures during the execution, and N_k is the number of iterations between two failures. Figure 6.6 shows the result of our measurements. The y-axis corresponds to δ_i and the x-axis to the percentage of failed nodes w.r.t. the initial number of nodes ($i = 2 \mathrel{\widehat{=}} 1\%$, etc.). Note that a loss of 20% of the nodes is very unrealistic. In practice, we will be confronted with failures of less than 5%. A linear regression up to 5% shows an almost direct proportional slope, i.e. 1% loss corresponds to 1% overhead. Assuming an execution that is interrupted by failures with a loss of 2% of the nodes each time, we can give an estimation of the total time imposed by the longer iteration times using Equation 6.14. Considering an execution time of 24h and a MTBF of 6h (i.e.,$N_k = {}^{\text{MTBF}}/_{(\delta_{i_k} T_0 + T_0)}$), we run 6h with 100% of the nodes, 6h with 98%, 6h with 2% of 98%, and so on. With Equation 6.14 we get:

$$\Delta T_{\text{tot}} = 6\text{h} \left(\frac{0.02}{1 + 0.02} + \frac{0.0396}{1 + 0.0396} + \frac{0.0592}{1 + 0.0592} \right) \approx 40 \, \text{minutes} \tag{6.15}$$

Hence, in the case from above, which is a rather pessimistic example, we will have about 40 minutes of extra time, due to the additional workload per processor. Even this example might prove beneficial, considering the sometimes significant amount of time an application might wait for the new allocation between two failures. Clearly, the model we presented is rather simple, and load balance issues could increase the additional overhead. Nevertheless, the extra overhead for losses within a certain range, poses a good tradeoff to the time spent in a job scheduler queue waiting for a new allocation.

6.4.7 Data Distribution on Irregular Applications

In section 6.3.4.2, we mentioned the challenge that xPic poses for the recovery from a shared file. The particles in xPic move freely between the grid cells, and consequently the number of particles per processor changes repeatedly. Because of this, the elastic recovery becomes non-trivial, we cannot arrange the

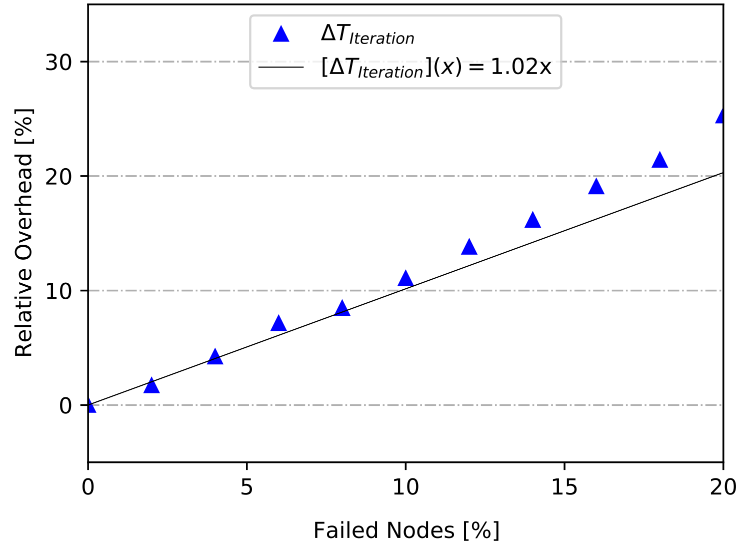


Figure 6.6: Relative additional overhead for executions on fewer nodes with constant total load. The x-axis shows the percentage of nodes lost upon failure.

particles contiguously inside the checkpoint file without knowledge of the particle distribution on the other processes. In this section, we propose and study the benefits of two different data layouts, (1) per process (*PP*) and (2) per cell (*PC*).

The first layout is based on the number of particles per process. This approach is oriented towards the best performance while checkpointing. Before writing the particles, the ranks communicate the number of particles they will write. With this information, each rank computes its offset within the global particle dataset and writes the particles accordingly. No additional information is kept. Thus, upon recovery with a different number of ranks, we have not enough information to directly read those particles, that indeed belong to the respective ranks. Instead, upon the recovery, we divide the total number of particles by the number of ranks and read from the dataset in equal parts. Afterward, we re-distribute the particles that have been read by the wrong rank.

The second layout is based on the number of particles per cell. This approach is oriented towards the best performance upon recovery by eliminating the necessity of particle re-distributions after the restart. The global file contains one dataset per grid cell to store the particles. Hence, upon recovery, each rank can directly read the contents of the cells belonging to the domain it operates on. This approach tends to the creation of many small datasets, and fragmentation inside the checkpoint file, which, in turn, can lead to the deterioration of the IO performance.

To have a certain degree of complexity for the experiments, we compared the two methods operating on a cubic grid (i.e., the number of cells is equal in each coordinate direction). Table 6.5 lists the key parameters, and figure 6.7 shows the results of the experiments. We observe that the second mode (PC) leads to significantly more overhead than the first mode (PP), when operated in synchronous mode. However, if operated in asynchronous mode, the overhead of both techniques becomes almost identical (2 and 3 seconds for PP and PC respectively). The recovery, on the other hand, is much faster using the first approach (PC). We can see that when recovering on the same number of nodes, the time for the particle redistribution for the first mode, is in the order of the time for the entire recovery with the second mode.

Evaluation Parameters	xPic
Nodes	16
MPI-processes per Node	32
Grid Dimensions	120x120x120
Number of Cells	$2{\cdot}10^6$
Number of Particles	$3{\cdot}10^9$
Accumulated CP size	280GB

Table 6.5: Configuration for the measurements of the 2 different data organization pattern in the checkpoint file for xPic.

When restarting on fewer nodes, the time for the redistribution even becomes significantly longer. A good solution for normal and elastic recovery poses the second mode (PC) operated asynchronously. Hence, with our implementation for the asynchronous write of the shared file, we can efficiently perform elastic restarts in particle-in-cell applications as an example for an irregular application.

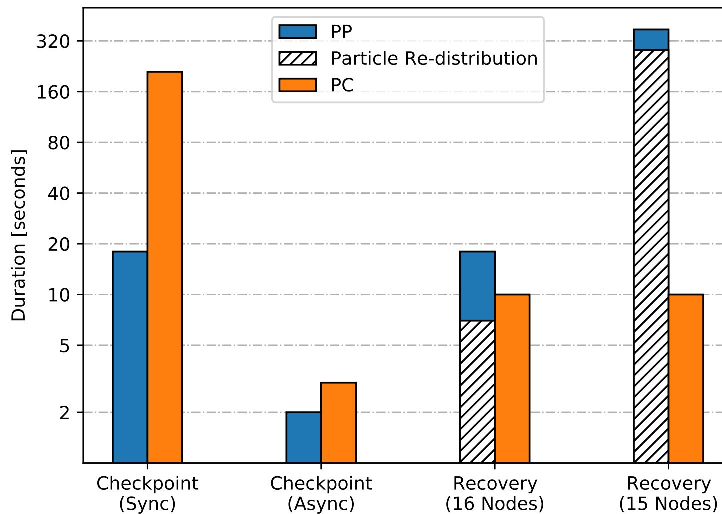


Figure 6.7: Comparison of the 2 techniques (PP and PC) to organize the particle data in the checkpoint file. Recovery on 16 nodes means recovery on the same number of nodes, whereas recovery on 15 nodes means elastic recovery on a reduced number of nodes (i.e., simulates the loss of 1 node upon failure).

6.5 Discussion

In this chapter we presented our proposed API extensions and a runtime to allow multilevel CR libraries to consolidate IO for resilience and general IO, and the ability to perform an elastic recovery through a simple interface while guaranteeing high performance. Nonetheless, it would be naive to think that the proposed API could cover all the corner cases that the entire HDF5 interface allow. For instance, we did not implement HDF5 attributes and the API currently does not offer the possibility to create strided datasets. There is currently also no simple way of exposing linked lists to the API. However, we expect that the limitations of the proposed API will become apparent with time and practical application. We further demonstrated that elastic restart is viable at large scale, and that we can reduce the overhead further using online recovery. We evaluated the overhead when executing on a reduced number of nodes and arrived to an acceptable value compared to the alternative, which is, spending time in the scheduler queue. It is important to highlight that elastic restart is useful for other purposes different to resilience

(e.g., malleability of ensemble runs), which increases the scope of our proposal. Finally, we analyzed the generality of our approach by testing with a highly irregular application. This raised several challenges and highlighted the trade-off between checkpoint performance and restart performance. The first approach (PP) was far more general and did not involve any application-specific measure other than particle redistribution upon restart. The second one (PC) is more application-specific but can leverage the advantages proposed in this work to achieve both fast checkpoint and fast recovery.

6.6 Related Work

Our proposal focuses on two aspects. The first addresses the consolidation of multilevel checkpointing and scientific IO, the second addresses the elastic restart, i.e., the restart with a different number of processes than the checkpoint has been created with. Regarding the first aspect, besides application-specific solutions, there are several IO-libraries that can be used for resiliency as well as for scientific IO. However, as we emphasized, there is no library combining the virtues of both families optimized for both purposes. Among the most common IO libraries are ADIOS, netCDF and HDF5. From those libraries, ADIOS is most similar to our proposal. It allows asynchronous staging to the PFS and IO to node local storage and with this, it can be used for integrating basic checkpointing in HPC applications. However, multilevel checkpoint techniques as for instance, partner checkpointing or encoded checkpoint files are not available. On the other hand, the most important multilevel libraries such as FTI, SCR and VeloC do not provide access to their checkpoint files, as they typically use an opaque file format and do not offer interfaces to extract the data from the files.

For the other aspect of our proposal, elastic recovery, several ways to continue the execution without the missing processes have been proposed. Redundancy schemes [108], context migration from checkpointed migratable objects [109] or process migration using failure prediction [110] are some of the examples. However, these techniques often impose additional requirements, as for instance, the allocation of shadow nodes for proactive process migration and redundancy (or performance degradation when using over-subscription). Migratable objects, such as used in Charm++ [111], need to be considered at program creation as it requires the application to use a specific programming language and code layout. Certainly, there are other ways to allow for an elastic restart and resilient scientific IO inside an application using an application-specific solution. However, the proposed interface and the methods presented in this chapter can be applied to a wide range of applications, with reasonably low effort from the developer.

6.7 Conclusion

In this chapter we proposed novel API extensions and a runtime for multilevel checkpoint libraries, that allow the elastic recovery to an arbitrary number of processes, and the creation of global self-descriptive checkpoint files. We implemented the runtime and API into FTI, a state-of-the-art multilevel CR library. We demonstrated that the checkpoint overhead of our implementation is comparable to other multilevel checkpoint techniques. We have shown that we can further reduce the overhead with an asynchronous implementation of the global checkpoint creation, showing a negligible overhead of only 0.6% (at scale). We compared our asynchronous mechanism to a similar feature of the state-of-the-art IO library ADIOS, and demonstrated that our method is five times faster. We accomplished online recovery using ULFM together with elastic recovery and showed that it reduces significantly the total recovery time compared to

traditional offline recovery. Our analysis of the performance degradation after an elastic recovery, due to the higher workload per processor, shows that for the loss of 1% of the nodes, we can expect an additional overhead of about 1%, and that it represents a viable solution compared to rescheduling the application. We further analyzed the challenges raised by irregular applications and their trade-offs regarding the performance for checkpointing and recovery, and we have proposed methods to solve those challenges.

Part III

Contributions to Resiliency in Large Ensemble

Chapter 7

Background Checkpointing in Operational Ensemble Data Assimilation

Publications

- International Conference on High Performance Computing, Data, and Analytics (HiPC) 2021
  - **Keller, K.**, Kestelman, A. C., Bautista-Gomez, L. (2021, December). **Towards Zero-Waste Recovery and Zero-Overhead Checkpointing in Ensemble Data Assimilation.** In 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) (pp. 131-140). IEEE.

Main Contributions

- We introduced checkpoint and recovery to the server of the online Data Assimilation (DA) framework MelissaDA. We use the contribution presented in chapter 6 to store the checkpoint data into shared HDF5 files, and allow the elastic recovery.
- We move the checkpoint creation entirely in the background leveraging threads and exploiting idle times.
- Our checkpoint implementation allows checkpointing at a high frequency (every cycle), and partial checkpointing during the propagation step to minimize recomputation.

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 TFloating Point Operations Per Second (FLOPS) [56], and in 2020 on the Fugaki supercomputer on more than 130,000 nodes reaching 79 PFLOPS [57]. 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 [58].

An important part of Numerical Weather Prediction (NWP) is DA [52]. Some popular models using ensemble based DA are Transient One Dimensional Pipe Flow Analyzer (TOPAZ) [112], Nonhydrostatic ICosahedral Atmospheric Model (NICAM)-Local Ensemble Transform Kalman Filter (LETKF) [52], Community Earth System Model (CESM) [47], European Community Earth-System Model (EC-EARTH) [113] and Integrated Forcasting System (IFS) [49]. 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) the **propagation** and (2) the **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 current best estimate of the true state. The two steps are repeated until either the desired accuracy is reached or all the available observations are consumed.

The majority of research in DA is applied to numerical weather or climate forecasting models.

However, it finds application in other fields too. For instance, in robot localization [114], and digital twins [115]. A field where DA is a crucial ingredient is operational NWP, a 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.

MelissaDA [116] is a recent general-purpose online ensemble DA framework, with a simple interface to attach numerical climate models. Traditionally, online ensemble DA implements the state circulation with MPI. In MelissaDA, the states are circulated via Transmission Control Protocol (TCP) leveraging the Zero Message Queue (ZeroMQ) [117] library. The framework is based on a server-runner architecture. Several runners and the server are each executed on different resources (i.e., separate cluster jobs or MPI domains). 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 ZeroMQ 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 chapter, we present the effortless framework protection leveraging Checkpoint-Restart (CR). The checkpoints fulfill the purpose of resiliency *and* scientific Input and Output (IO), providing the climate data in the checkpoint files via Hierarchical Data Format (HDF5). The checkpoints contain the ensembles of 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). Our implementation manages further to recover from failures with none or few recomputations (zero-waste). 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 chapter in section 7.1, present our implementation in section 7.2, acknowledge related work to better understand the topic and introduce different concepts in section 7.3, present our experimental methods and goals in section 7.4, the results of our experiments in section 7.5, discuss the results in section 7.6, and eventually conclude the chapter in section 7.7.

7.1 Background

7.1.1 Data Assimilation and the Ensemble Kalman Filter

An essential part of climate research is making predictions and reanalysis 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 means to reduce uncertainty is data assimilation. DA combines the error probability distributions for observation and model states to decrease the uncertainty. Kalman Filter (KF) is among the most common techniques

for DA. The foundation of the formalism is represented by the state space equations:

$$x_t = \mathcal{M}x_{t-1} + q_t, \quad q_t \sim \mathcal{N}(0, Q_t) \tag{7.1}$$

$$y_t = \mathbf{H}x_t + r_t, \qquad r_t \sim \mathcal{N}(0, R_t) \tag{7.2}$$

Here, x_t represents the true state, $\mathcal{M}$ the model operator, y_t the observed state and $\mathbf{H}$ 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 within 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. A brief introduction of these and other techniques is provided in section 2.2. Detailed introductions to the individual methods can be found in many textbooks and articles [12, 118–120].

7.1.2 MelissaDA

MelissaDA is a spin-off from Melissa [121], an online framework for large-scale Sensitivity Analysis (SA). 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 billion (10^9) variables, comprises at least 7.5 TB of memory, just for representing the ensemble states (in double precision). 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 storage layer (**offline mode**). In that case, the propagation and analysis are performed on separate executables and the ensemble is circulated through files. Another possibility is using one executable for both propagation and analysis while circulating the states through MPI (**online mode**). Both methods have certain advantages and drawbacks. The first method is intrinsically fault tolerant, since the state ensemble from the last completed step (analysis or propagation) is available inside restart files on the file system. However, the storage layer introduces a bottleneck. The second method provides better performance but misses the intrinsic fault tolerance. On the other hand, introducing resiliency for online models at large scale is costly, diminishing the advantage towards offline models.

MelissaDA takes an intermediate approach. As in the first method, the components performing the propagation and analysis run on separate executables, however, running concurrently instead of sequentially. The ensemble of states is circulated through the network using ZeroMQ instead of MPI. Hence, we have separate executables as in an offline approach, but, circulate the states through the network as in an online approach. MelissaDA is based on a server-client architecture. The clients (**runners**) compute the background states and the server gathers the states and performs the analysis step. Leveraging ZeroMQ 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

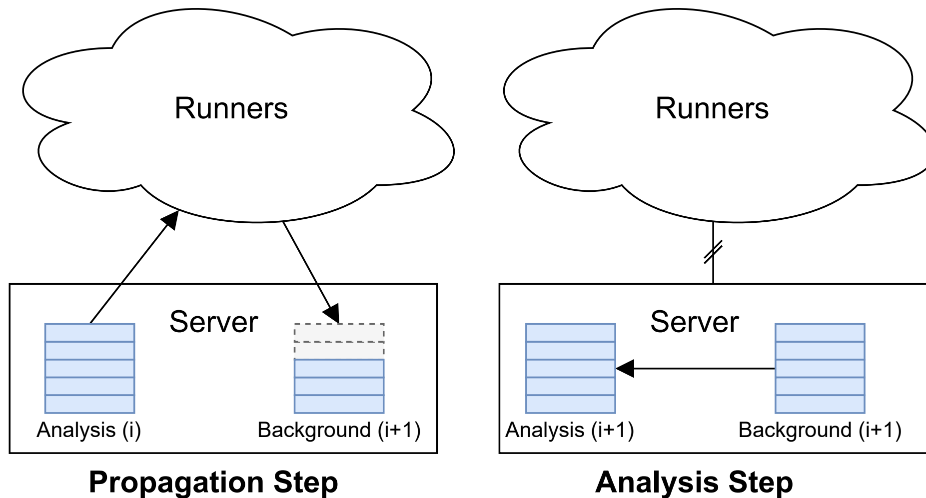


Figure 7.1: Server-runner concept of MelissaDA. Each iteration, the server distributes the analysis ensemble to the runners, which in turn compute the background state as input for the next analysis step.

runtime. The state ensemble is exchanged between the server and runners directly, without intermediate storage layer. 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 7.1 visualizes the concept.

7.1.3 Asynchronous Checkpointing and Elastic Recovery

In modern multilevel checkpoint libraries, the various levels are typically performed in two steps. The first step (**pre-processing**) is performed inline, that is, performed by the application processes while halting the application flow. During this step, the checkpoint data is stored to the local storage layer (Solid State Drive (SSD), Non Volatile Memory Express (NVMe), node memory, etc.). The second step (**post-processing**), is performed asynchronously, that is, in the background of the application flow. During this step, the remaining action for completing the checkpoint level is performed. For instance, a copy is sent to the partner node for partner checkpointing. Fault Tolerance Interface (FTI) [14] provides the asynchronous staging of globally shared HDF5 files. Furthermore, the FTI checkpoints leveraging this feature, can recover with a different number of processes as the checkpoint has been created with (**elastic recovery**). The second stage is performed with extra MPI processes on the nodes. Hence, we have to dedicate one process per node to FTI if we want to use this feature (see chapter 6).

7.2 Implementation

Our implementation involves modifications in all modules of the Melissa framework (launcher, server and several runner). In this chapter, we will gradually introduce the three modules along with our modifications.

7.2.1 Launcher

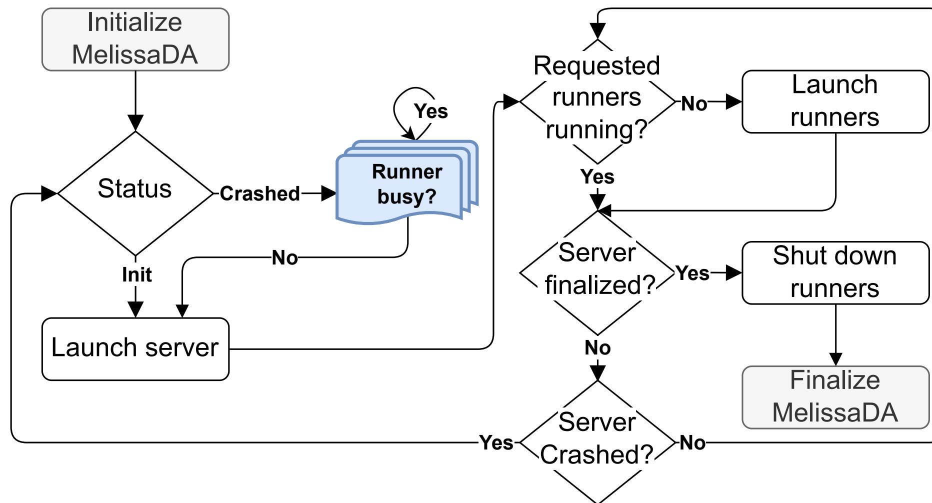


Figure 7.2: Launcher workflow. Upon a runner failure, the launcher starts a new runner instance. Upon server failures, the launcher waits until the runners completed their computations and checkpoints and restarts the framework.

The launcher is the monitoring unit of the MelissaDA 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**). After initialization, the launcher starts the server. Afterwards, it launches all requested runner instances. With this, the launcher has arrived to the main loop, where it monitors the runner and server instances. If runners fail, the launcher simply launches new runner instances. If the server fails, the launcher waits for the runners to complete the current state propagation and the associated checkpoint, before restarting the workflow. Figure 7.2 shows the restart mechanism in detail.

7.2.2 Server

The server workflow is different during the propagation and the analysis step. During the propagation step, the runners generate the background state ensemble, and the server maintains the bookkeeping of which state has yet to be propagated. The server keeps copies of all states and serves propagations to the runners. Each time a runner requests a new state, the server transfers a state to the runner for its propagation. After all states have been propagated, the server performs the analysis on the background ensemble to generate the analysis ensemble, and starts the next cycle. MelissaDA intrinsically provides basic fault tolerance to the framework, as the launcher detects and restarts failed runners (Figure 7.2, and Figure 7.3). The server, however, is not protected by default. As the server has to be of sufficient scale for holding twice

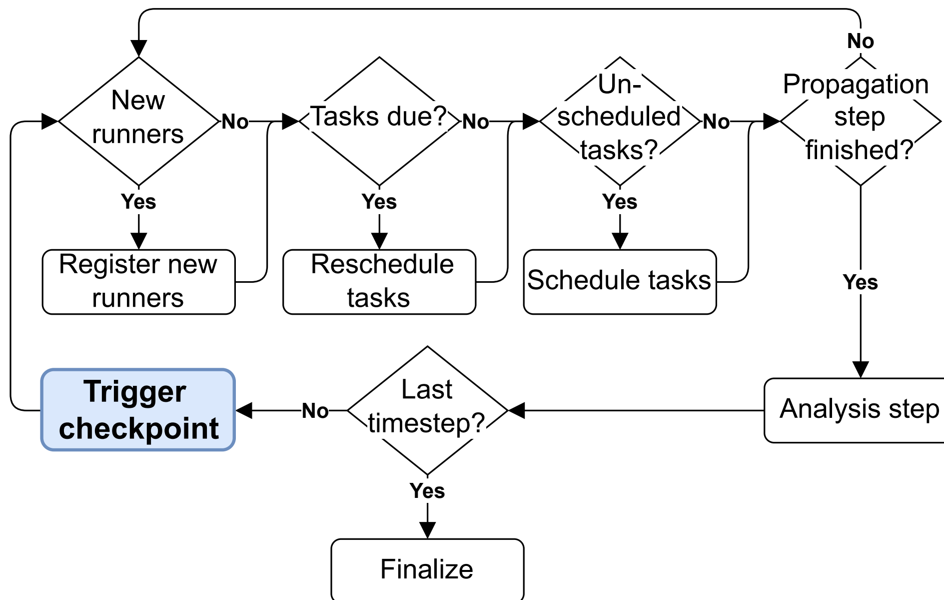


Figure 7.3: Server mainloop showing the mechanism to register new runners, the scheduling and checkpointing.

the ensemble states (background and analysis ensemble), it is operated at a large scale for large ensemble sizes. Hence, fault protection is necessary. To provide Fault Tolerance (FT) for the server, we create checkpoints containing the analysis state ensemble, leveraging the asynchronous global HDF5 feature presented in subsection 7.1.3. FTI provides only one process per node for the post-processing. However, we exploited the local test setting of FTI to enable several dedicated processes per node. The test mode is intended for running FTI on a desktop computer simulating an underlying cluster structure. Essentially, if applied on a real cluster, FTI creates a virtual cluster distributed on the nodes, and we can decide how many virtual nodes (i.e. dedicated processes) we want to deploy on each physical node. Exploiting this mode on a cluster works only if (1) the Message Passing Interface (MPI) processes are mapped equally among all nodes, and (2) consecutive ranks are placed in increasing order filling up node by node. Now, by setting the number of processes per node in the FTI configuration file, we can control the number of virtual nodes per physical node. Figure 7.4 shows the distribution, if we have M processes per physical node, and 4 processes per virtual node.

The server is typically memory bound; it has to provide enough node memory for the states of both ensembles. The filter update during the analysis operates on a decomposition of the climate state. In numerical weather prediction, the state size is typically in the order of 10^9 (less than 10 GB), and the servers' MPI decomposition is limited by this value. Hence, for very large ensembles, the server allocation is large due to the memory requirement, but typically not all processors of the allocated nodes will be used. Thus, exploiting the test mode in FTI, lets us utilize the idle processors, speeding up the completion of the checkpoint in the background. Furthermore, we are able to move the pre-processing stage in the background, by deploying the checkpoint pre-processing on threads. As the server is not changing the analysis states during the propagation step, we can safely copy the states directly from the servers' memory to the local file system, while the server is carrying out the scheduling for the propagation step.

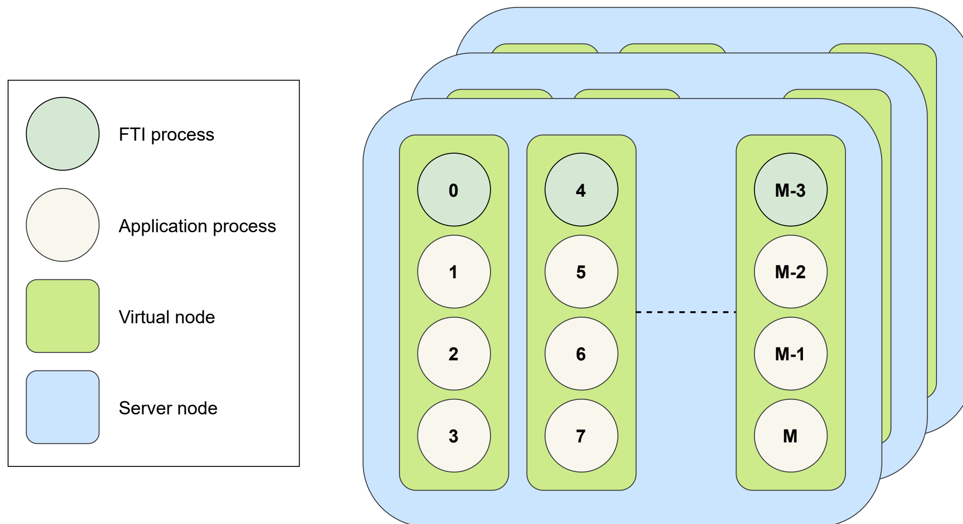


Figure 7.4: Virtual cluster, generated by FTI if deployed in local test mode. The number of processes per virtual node is set to 4 and we have M processes per physical node. This leads to M/4 dedicated FTI processes per node.

7.2.3 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 [57]). In order to interface with the MelissaDA framework, the simulation model needs to implement merely the two API functions: (1) `melissa_init` and (2) `melissa_expose`. The state exchange between runner and server takes place during `melissa_expose`. The function essentially involves a send and receive operation. Runners receive analysis states from the server and return background states. In subsection 7.2.2, we described how the server creates the checkpoint of the analysis ensemble. In the following, we will describe how the runners create the checkpoints for the background ensemble.

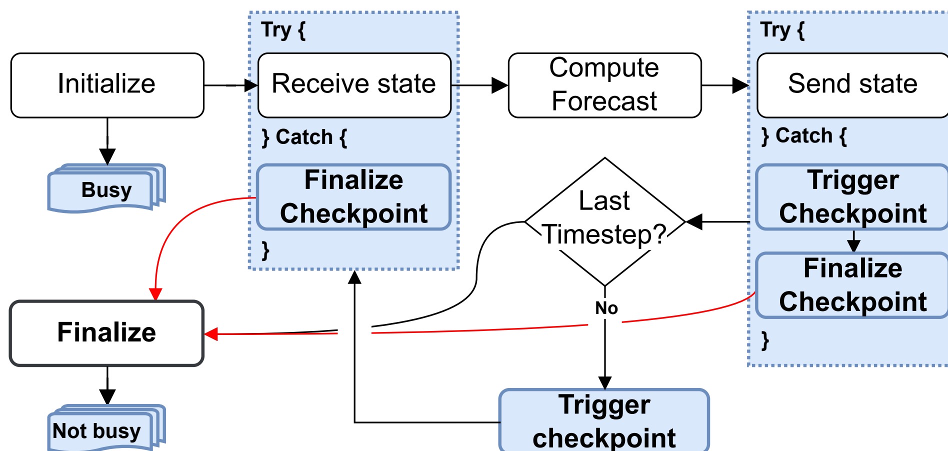


Figure 7.5: Flowchart of the MelissaDA runner workflow.

Each runner propagates only one state at a time, and only keeps the data of that state in memory. Furthermore, each runner is executed separately. On the one hand, we can not create one shared file containing the whole background ensemble, on the other hand, creating the checkpoint of the background

states on the runners at the end of its propagation, integrates smoothly into the load balancing of the state propagations. To maximize the checkpoint performance on the runners, the exact timing of the checkpoint is essential. The runners are idle between the send and the receive operation for some time, because they are 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 checkpoint overhead entirely in that way, even without staging through the local storage layer. However, for scalability (global storage congestion or limited throughput), it is good practice applying asynchronous checkpointing for the runners as well. We observe in our experiments that there is no need to move the pre-processing in the background with threads. The implementation effort for this is not justified (allocate extra buffers to copy the background state, implement multi-threading). Nevertheless, we performed experiments leveraging threads on the runners, demonstrating that we indeed observe no speedup in that case.

Checkpointing the analysis ensemble is enough to ensure FT for the framework. However, if we additionally checkpoint the background ensemble, we can minimize the necessary recomputation when recovering. Since we create one checkpoint file for each background state, we can ensure the consistent recovery to the point of failure, as long as the analysis ensemble has been successfully checkpointed on the server side. In case of a server failure, the runners are guaranteed to complete the current propagation and checkpoint of the associated background state. Figure 7.5 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 server failures (ZeroMQ error or timeout), the runners shut down gracefully after completing the propagation and the subsequent checkpoint.

7.2.4 Recovery

With the presented checkpointing scheme on runners and server, we entirely hide the checkpoint overhead behind the framework's execution, and we recover, if the checkpoint creation on the server side was successful, back to the point where the failure has occurred. Both server and runners, checkpoint into globally shared files, using the FTI API. The API enable the elastic recovery from the checkpoint files. This has several benefits. First, 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. Second, the server can be restarted elastically. For instance, when running on heterogeneous hardware.

Takeaway section 7.2

- The server creates checkpoints of the analysis state ensemble. The entire ensemble is stored into a single HDF5 file on global storage. The checkpoint is moved entirely to the background. The pre-processing is performed by threads, and the post-processing by dedicated MPI processes.
- The runners checkpoint the background state ensemble. each background state is stored into one shared HDF5 file on global storage. The pre-processing is perfomed inline (no threads), and the post-processing by dedicated MPI processes. The checkpoint cost on the runners is effectively hidden behind the runner/server communication.
- Both the background and analysis ensemble can be recovered elastically.

7.3 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 [122]. The second work comprises a comprehensive overview of available FT mechanisms to protect numerical climate models [123].

7.3.1 Fault Tolerance for DART-MITgcm with Decimate

Decimate [124] 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 [125], the Massachusetts Institute of Technology Ocean General Circulation Model (MITgcm) [126] on the Data Assimilation Research Testbed (DART) [127]. 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, and 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.

7.3.2 Fault Tolerance Methods for Numerical Climate Models

Benacchio et al. [123] published 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. Three approximate data recovery 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 [20] compression. Other methods discussed in the article (referred to as *system resiliency*) rely on fault-tolerant MPI implementations such as ULFM [91] and Fenix [128]. Again other rely on replication or message logging.

7.4 Methodology

In this section, we will introduce the experiments that we have performed to evaluate our implementation. In addition, we will discuss the different cases of failures that can occur. Further, we will develop the

foundation for the model allowing us to estimate the time for checkpoint and recovery in continuous operation.

7.4.1 Experiments

We designed our experiments to reveal the benefits of the asynchronous checkpoint creation. For this we want to compare server executions that leverage threads for the local staging (pre-processing) of the checkpoint files to executions that perform the staging inline (no threads). In all experiments, the server uses dedicated FTI processes for the checkpoint post-processing. Furthermore, we want to compare the checkpoint creation inline (no threads, not dedicated FTI processes), to executions leveraging dedicated processes for the post-processes, on the runners. To reveal possible correlations between the different settings, we perform experiments, combining the different server and runner checkpoint modes. The experiments are labelled as:

T0 pre-processing inline (server)

T1 pre-processing leveraging threads (server)

H0 post-processing inline (runner)

H1 post-processing leveraging dedicated FTI processes (runner)

We use the FTI terminology to label the experiments, where the dedicated processes are called *heads* (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 7.1, and the labels are listed in Table 7.2.

Parameter	**Value**	**Members / Runners**	**CP (For+Ana)**	**Server (FTI) Proc.**	**Runner (FTI) Proc.**	**Total (FTI) Proc.**
dim. state	$O(10^9)$	64 / 16	1 TB	256 (128)	752 (16)	1008 (144)
dim. obs.	$O(10^4)$	128 / 32	2 TB	512 (256)	1504 (32)	2016 (288)
size state	8 GB	256 / 64	4 TB	1024 (512)	3008 (64)	4032 (576)
size obs.	168 Kb	512 / 128	8 TB	2048 (1024)	6016 (128)	8064 (1152)

Table 7.1: 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.

↓ Checkpoint Setting / Label →	**H0T0**		**H1T0**		**H0T1**		**H1T1**	
	server	**runner**	**server**	**runner**	**server**	**runner**	**server**	**runner**
pre-processing sync. (no threads)	✗		✗	✗				✗
pre-processing async. (threads)					✗		✗	
post-processing async. (MPI)	✗		✗	✗	✗		✗	✗
entire checkpoint inline		✗				✗		

Table 7.2: Experiments that we have performed with the respective labels.

7.4.2 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).

7.4.3 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_{rep}. 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 during failures in the following regions:

(A) *before* completion of the checkpoint for the analysis ensemble

(B) *after* completion of the checkpoint for the analysis ensemble

The zero-waste scenario is (B). In this case, we restart where we have left off, recovering the entire analysis ensemble and the background ensemble states that have been completed before the failure. The worst case scenario is (A), where we need to repeat the analysis step and part of the propagation (until the point of failure). Figure 7.6 summarizes the scenarios graphically. The figure also compares to the case if checkpointing only the analysis ensemble. We can see that for both regions, the recomputations are much less. The revival times for the two regions can be expressed by:

$$\begin{aligned} T_{rev,A} &= T_{rec,ana} + T_{rec,for} + T_{rep,A} \\ &= T_{rec,ana} + T_{rec,for} + \alpha_A \left(T_{ana} + T_{cp}\right) \end{aligned} \tag{7.3}$$

$$\begin{aligned} T_{rev,B} &= T_{rec,ana} + \alpha_R \, T_{rec,for} + \underbrace{T_{rep,B}}_{=0} \\ &= T_{rec,ana} + \alpha_R \, T_{rec,for} \end{aligned} \tag{7.4}$$

α_A, α_B, and α_R have values between 0 and 1, indicating that for region A the failure can happen at any point during the analysis or checkpoint creation and that for region B, we only need to recover those states that have been successfully propagated and checkpointed before the server failure. The recovery of both the analysis and background states (a.k.a., forecast states) in Equation 7.3 enters due to our implementation. Upon the server restart, we first recover the latest analysis ensemble and then check for available background states. In a future implementation this can be improved, avoiding the recovery of the previous analysis ensemble, since for case A, we do not need to recover the analysis ensemble from

the iteration before. The probability of failures inside the three regions is given by:

$$p_A = (T_{ana} + T_{cp}) \times T_{iter}^{-1} \tag{7.5}$$

$$p_B = (T_{for} - T_{cp}) \times T_{iter}^{-1} \tag{7.6}$$

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. We can further compute the average revival time by:

$$\langle T_{rev} \rangle = p_A \langle T_{rev,A} \rangle + p_B \langle T_{rev,B} \rangle \tag{7.7}$$

Where $\langle \cdot \rangle$ indicates the average value. We also average over the revival times in the individual regions, since their value depends on the time when the failure occurs:

$$\begin{aligned} \langle T_{rev,A} \rangle &= \int_0^1 T_{rec,ana} + T_{rec,for} + \alpha (T_{ana} + T_{cp})\, d\alpha \\ &= T_{rec,ana} + T_{rec,for} + \frac{1}{2} (T_{ana} + T_{cp}) \end{aligned} \tag{7.8}$$

$$\begin{aligned} \langle T_{rev,B} \rangle &= \int_0^1 T_{rec,ana} + \frac{T_{cp} + \alpha (T_{for} - T_{cp})}{T_{for}} T_{rec,for}\, d\alpha \\ &= T_{rec,ana} + \frac{1}{2} \left(\frac{T_{cp}}{T_{for}} + 1 \right) T_{rec,for} \end{aligned} \tag{7.9}$$

Where we used: $\alpha_R \triangleq (T_{cp} + \alpha_B\,(T_{for} - T_{cp}))/T_{for}$. Now we can write the explicit form of Equation 7.7:

$$\begin{aligned} \langle T_{rev} \rangle = \Bigg\{ &(T_{ana} + T_{cp}) \left(T_{rec,ana} + T_{rec,for} + \frac{1}{2} (T_{ana} + T_{cp}) \right) \\ &+ (T_{for} - T_{cp}) \left(T_{rec,ana} + \frac{1}{2} \left(\frac{T_{cp}}{T_{for}} + 1 \right) T_{rec,for} \right) \Bigg\} \Bigg/ (T_{ana} + T_{for}) \end{aligned} \tag{7.10}$$

After measuring the respective times for the recovery, checkpointing, propagation and analysis, we can estimate the average revival times if the framework runs in operational mode.

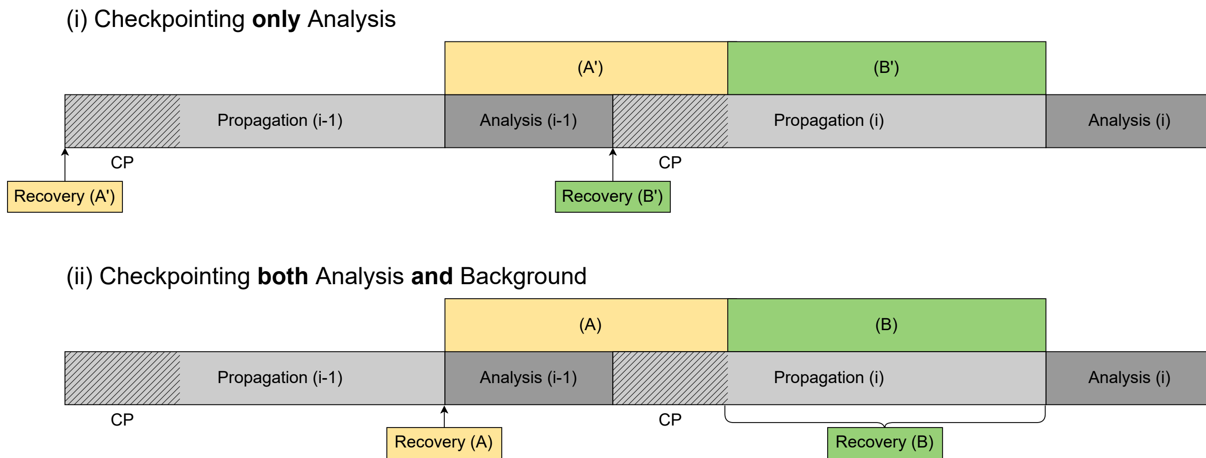


Figure 7.6: On the top, we see the characteristic failure regimes if checkpointing only the analysis ensemble. Below, the regimes if checkpointing both the background and analysis ensemble. **(i)** Failures in region A lead to a roll back to the beginning of the propagation step from the previous iteration, failures in B to a rollback to the beginning of the propagation of the current iteration. **(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).

7.4.4 Failure Injection

To measure the recovery cost for regions A and B, we injected failures at the corresponding locations in the server execution. To resemble a realistic situation, we injected the failures from the launcher using signals (external failure injection). In addition to the launchers' knowledge of the frameworks' status, we added a mechanism into FTI that exposes the current checkpoint stage (idle, pre, or post-processing) to the launcher. The launcher can then inject failures into the server while checkpointing the analysis ensemble.

7.5 Evaluation

In this section we first present the evaluation of the checkpoint and recovery performance of our implementation. Afterward, we discuss the benefits of checkpointing both ensembles towards checkpointing only the analysis ensemble.

7.5.1 Climate Model

We adapted a parallel version of the Lorenz-96 model [129] 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:

$$\frac{dx_i}{dt} = (x_{i+1} - x_{i-2})x_{i-1} - x_i + F \tag{7.11}$$

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

7.5.2 Experimental Setup

All experiments are performed on Marenostrum4 [130], the supercomputer at the Barcelona Supercomputing Center (BSC). The compute nodes are equipped with 48 cores/node (2 $\times$ Intel Xeon Platinum 8160), 96 GB of main memory and a 200 GB SSD. In all experiments, the server uses 8 MPI processes per node, while 4 from those are dedicated to FTI for the checkpoint post-processing. Each MPI process is mapped on 2 cores. In that way, the checkpoint thread can execute on a separate core, and does not compete with the server execution. The pre-processing on the server side leverages the local SSDs for staging the checkpoint data. The runners execute on 47 MPI processes per node, if the post-processing is performed on dedicated FTI processes (1 FTI process per node), and on 46 MPI processes, if it is performed inline. The checkpoints on the runner side never use checkpoint threads, consequently we map the MPI processes each to 1 core. The pre-processing on the runner side leverages the node memory (RAM disk) for staging the checkpoint data.

7.5.3 Performance Evaluation during Runtime

Figure 7.7 shows the duration of completing 5 assimilation cycles (cycles 4 to 9) for each of the experiments. The green bars show the results for experiments with checkpoint threads on the server side, and the blue bars the experiments without threads. Light blue and light green show the experiments leveraging dedicated FTI processes on the runners, and dark blue and dark green the experiments for the checkpoint creation inline. Finally, the gray bars show the duration for the execution without protection

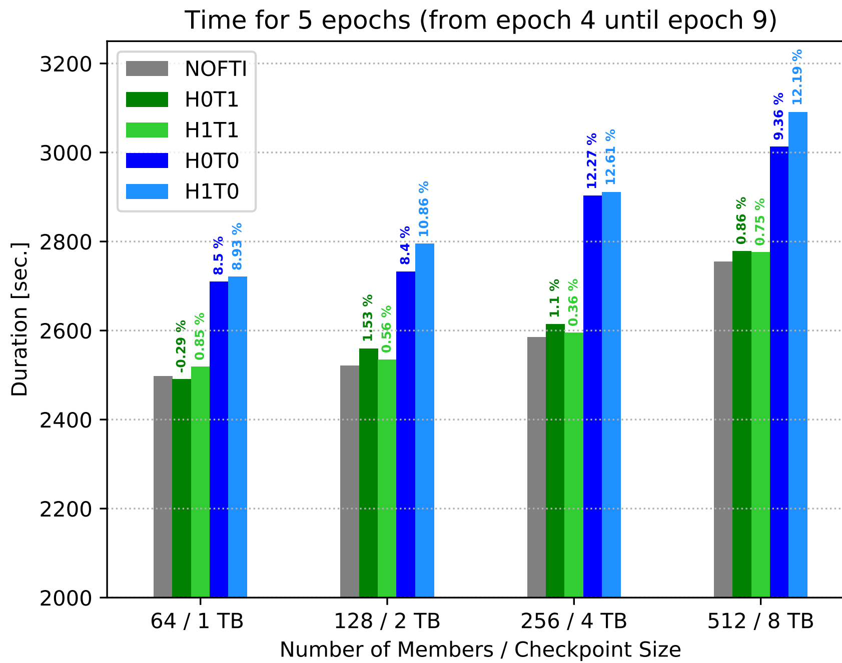


Figure 7.7: Time for 5 epochs (from epoch 4 to 9). Bars in green show the runtime for experiments with, and bars in blue without dedicated checkpoint threads on the server. Bars in gray show runtimes for the experiments without protection (baseline). The percentages above the bars indicate the overhead compared to the baseline.

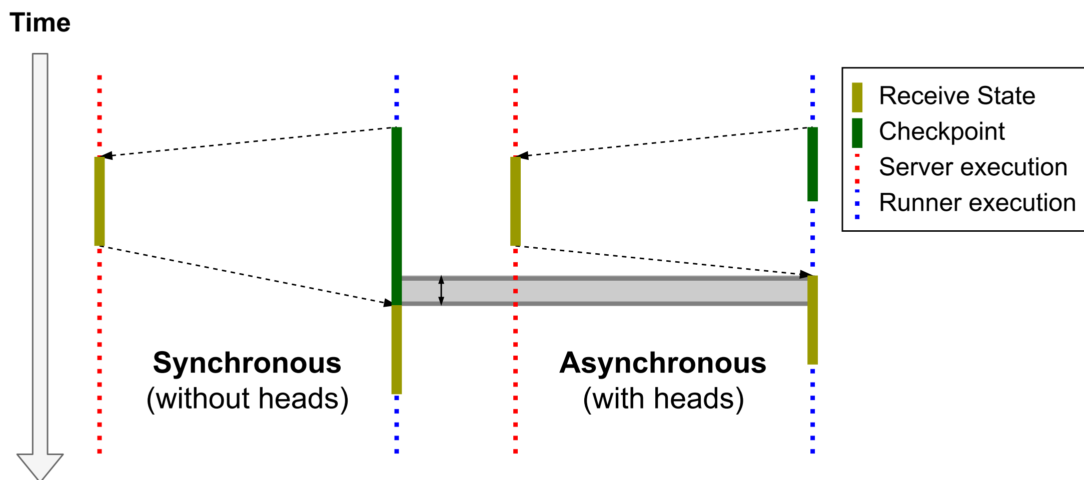


Figure 7.8: Communication graph for state circulation between runner and server. The right showing the case with dedicated FTI processes (i.e., asynchronous checkpointing) and the left, without (i.e., synchronous checkpointing).

(FTI disabled). We observe that the execution times for the gray and green bars are almost identical at all scales. Averaging over differences between the green and the blue bars (threads/no threads) and dividing by 5 (5 cycles) gives 50 seconds per cycle. Which is precisely the average of the time for the checkpoint pre-processing (Figure 7.9). Hence, we successfully hide the pre-processing, leveraging threads on the server side. We further see, that whether we use heads or not on runner side, the execution time does not significantly change.

The checkpoint of the background states on the runner, is performed after the background state has been sent, and before the analysis state has been received. This allows overlapping the checkpoint creation with the runners' idle period. Because, at the time the runner sends the background state, the server might be busy with other runner requests. Even if the server is free, the state takes a certain amount of

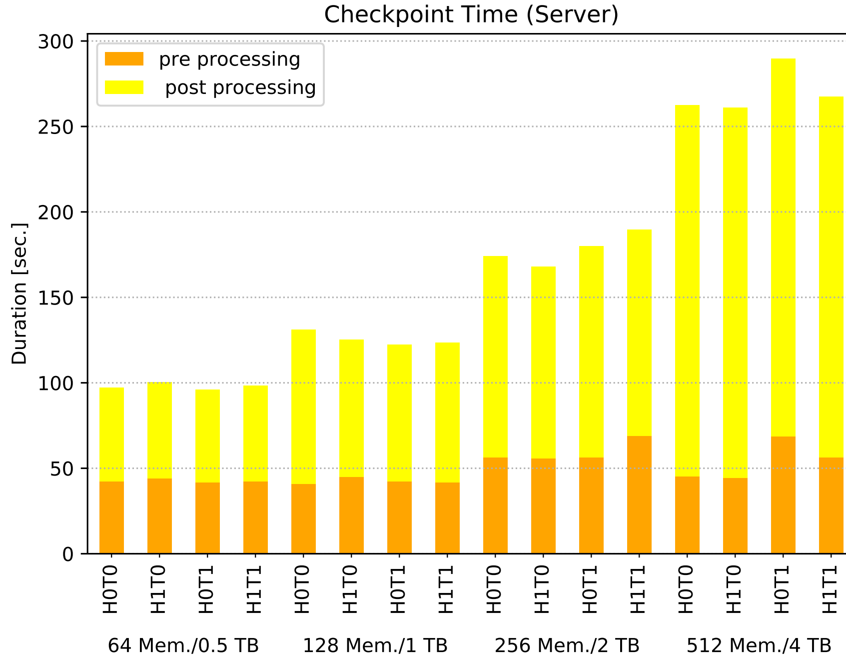


Figure 7.9: Time for the pre (i.e., node SSD) and post-processing (i.e., asynchronous shared HDF5 file creation) for checkpoints on the server.

time to arrive at the server node. Furthermore, the the analysis state need to arrive on the runner node after it has been sent by the server, which also takes a certain amount of time. Figure 7.8 visualizes this concept. Hence, in our experiments, the idle time is enough to hide the checkpoint cost in either case if we checkpoint inline or asynchronous. However, we expect that at a larger scale, checkpointing asynchronously will be beneficial. This is supported by the histograms in Figure 7.10. The histograms compare the effective checkpoint time towards the runners' idle time. The upper plots show experiments with asynchronous checkpointing (H1) and the lower plots with checkpointing inline (H0). The lower plots show that occasionally 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 asynchronous pre-processing does not affect the runners' idle period, as we can see in the histograms, and further it is expected to be independent of the scale, since in asynchronous mode local devices are used (not shared with other users), and with this we do not see congestion or saturation effects as on the global storage.

7.5.4 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 subsection 7.4.3 we derived revival times for failures, depending on the region they take place in. Equation 7.5 and Equation 7.6 give the associated probabilities. Failures in region A are the most costly ones, followed by 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). Our implementation does not influence the times for the analysis or the propagation. These are defined by the climate model and the filter that is deployed. Our implementation can only be improved by decreasing the times for checkpoint and recovery. From Equation 7.5 and Equation 7.6, we can infer that by minimizing the checkpoint time, we minimize p_A and maximize p_B, and

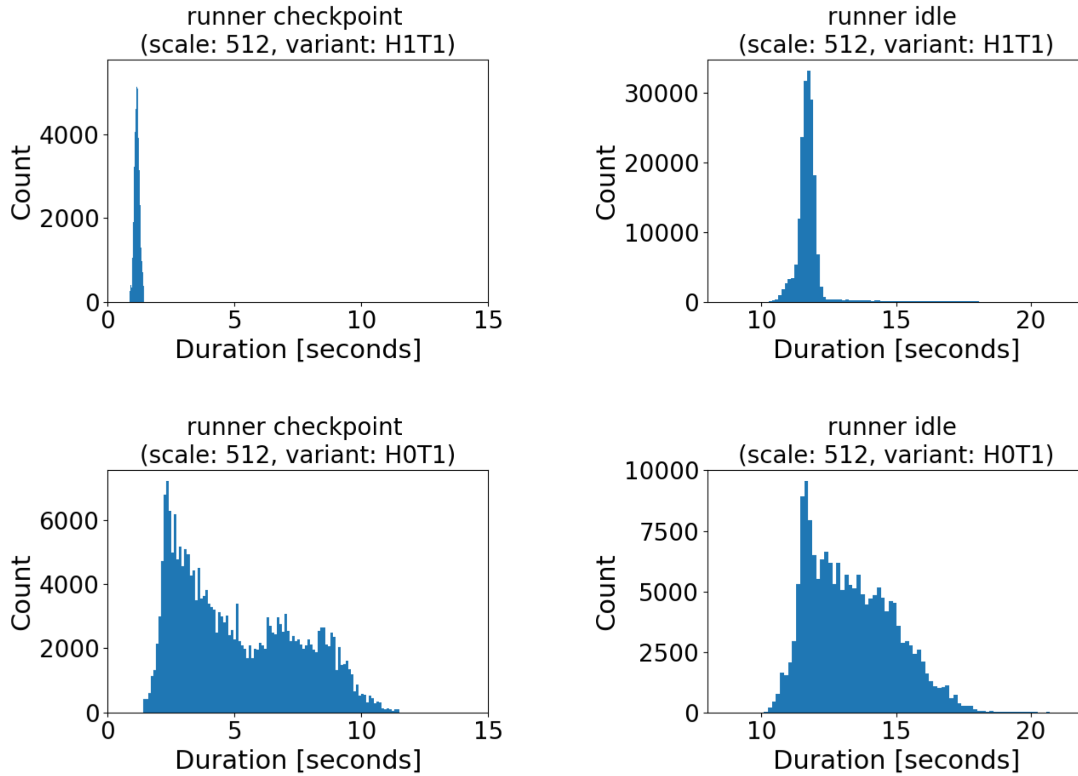


Figure 7.10: Histograms of the runner idle and checkpoint times. The runner idle period is the time between two model propagations. The checkpoint time is part of the idle time. The upper plots show executions with FTI heads and the lower, without. We observe that synchronous checkpointing broadens the runners idle time.

therefore minimizing the probability for failures in the most costly region and maximizing the probability for failures within the zero-waste region. Furthermore, the revival times (Equation 7.3 and Equation 7.4) depend on the times for the recoveries of analysis and background ensembles. Hence, to provide minimal revival times, it is important to ensure both good checkpoint and recovery performance.

Figure 7.11 shows traces of executions which have been interrupted by a failure. The upper plots show executions where we protect both ensembles, and the plots below show executions where we only protect the analysis ensemble. In both cases we interrupt the execution once in region A, and once in region B. The traces show the server execution, and additionally the execution of one runner instance. 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. Consequently, to make a fair comparison, we should only compare the active recovery time. We can see that indeed the revival times are faster when checkpointing both ensembles. The trace of executions with the failure in region B (top row, second trace) demonstrates that we indeed resume from the point of failure. We can see that the revival times without protecting the background are significantly longer, especially for failures in region A, as these include the repetition of the entire or most of the propagation step.

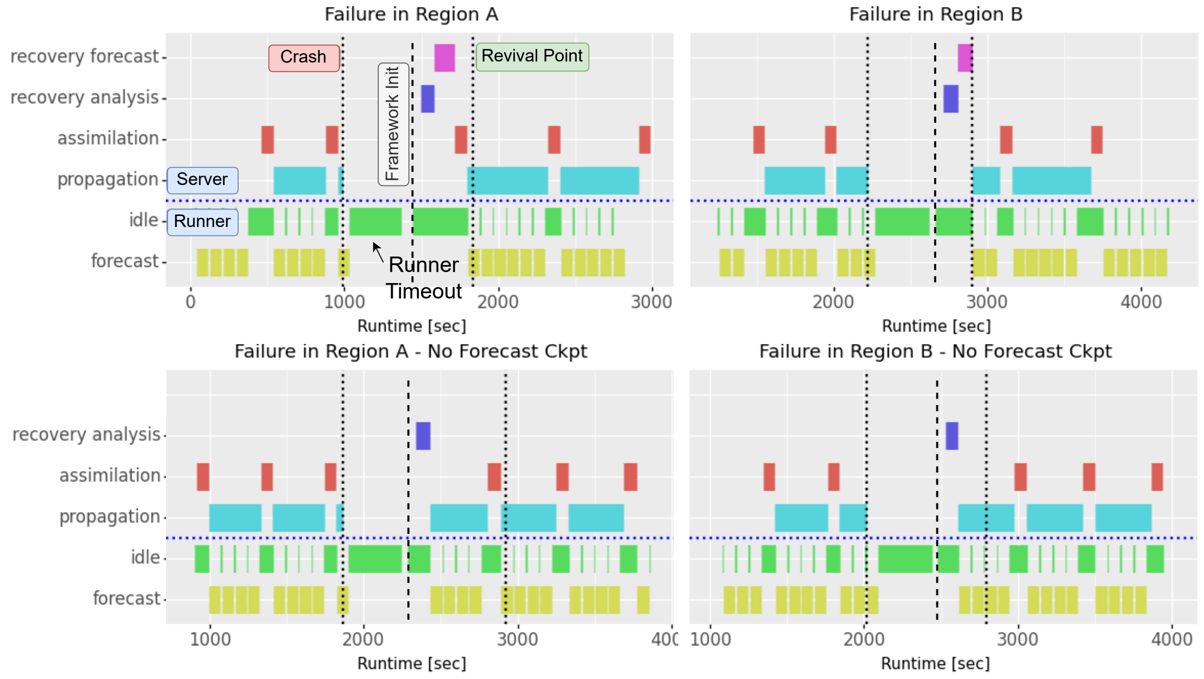


Figure 7.11: Gantt charts showing the server and runner execution (i.e., one runner instance). The charts show execution, failures in region A and B, and the recovery. The upper plots showing the cases when protecting both background *and* analysis ensembles and the lower, *only* protecting the analysis ensemble.

	32 (H1T1)	**64 (H1T1)**	**128 (H1T1)**	**256 (H1T1)**	**512 (H1T1)**
P_A [%]	31.7	32.6	38.1	48.9	64.1
P_B [%]	68.3	67.4	61.9	51.1	36.9
$\langle T_{rev} \rangle$ [sec]	82	191	293	752	1574
$\langle T'_{rev} \rangle$ [sec]	371	418	492	637	929
speedup [%]	353	119	68	-15	-41

Table 7.3: Probabilities (Equation 7.5 and Equation 7.6), average revival times (Equation 7.10), and speedup ($\langle T'_{rev} \rangle - \langle T_{rev} \rangle / \langle T_{rev} \rangle$). The numbers in green indicate that protecting both ensembles is beneficial, and the red numbers indicate that it is not.

We can determine the probabilities for failures in the two regions (Figure 7.6) using Equation 7.5 and Equation 7.6. Using Equation 7.10, we can further compute the average revival time for checkpointing both ensembles. A formula for the revival time if checkpointing only the analysis ensemble, can be derived in the same way. We have computed the values for the probabilities and revival times. The values are listed in Table 7.3. The table shows the results for 32, 64, 128, 256 and 512 members. The speedup is negative for executions with more than 128 members. This means that it is more beneficial to checkpoint only the analysis ensemble. However, we observed that the recovery times for the background states and analysis states differ considerably. For executions with 256 members, the recovery of the forecast ensemble takes more than twice as long as the recovery of the analysis ensemble (analysis: 197 seconds, background: 581 seconds). For 512 members it takes even about three times as long (analysis: 381 seconds, background: 1168 seconds). Furthermore, the recovery is significantly longer than the checkpoint creation. Both can likely be improved by tuning the file-creation parameters and block size for reading and writing. We will discuss this issue further in section 7.6.

7.5.5 Checkpointing Background and Analysis Vs. Only Analysis

The aim of this section is to support the advantages of checkpointing background and analysis ensembles, and to justify the additional implementation effort and storage utilization. For this, we will compare the two methods analytically towards each other. We start applying the considerations from subsection 7.4.3 to the simpler case, protecting only the analysis ensemble. Note that we always need to roll back to the

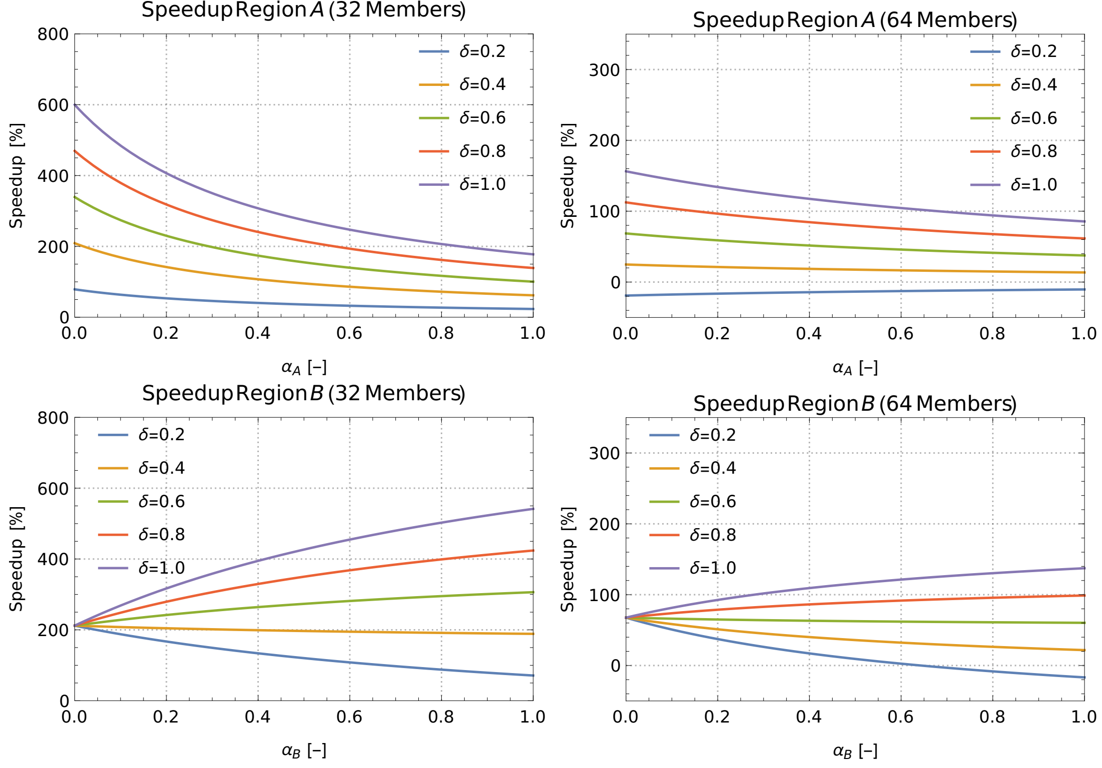


Figure 7.12: Speedup of checkpointing both ensembles towards checkpointing only the analysis ensemble. The speedup is plotted versus the location of the failure in the respective region. For failures in region A, α_A represents the normalized distance from the beginning of the region to the end. For failures in region B the respective normalized distance is given by α_B.

last available analysis ensemble. The two regions with different revival costs are the same as before (see also Figure 7.6):

(A′) *before* completion of the checkpoint for the analysis ensemble

(B′) *after* completion of the checkpoint for the analysis ensemble

However, the recovery takes place at different locations. 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:

$$T_{rev,A'} = T_{rec,ana} + T'_{for} + \alpha_A \left(T_{ana} + T_{cp}\right) \tag{7.12}$$

$$T_{rev,B'} = T_{rec,ana} + T_{cp} + \alpha_B \left(T'_{for} - T_{cp}\right) \tag{7.13}$$

The meaning of the α coefficients is the same as before. Further, the times for the analysis step (T_{ana}), checkpoint (T_{cp}), and recovery (T_{rec}) are identical to the other case[1]. However, the time for the propagation

[1]Note that we do not have terms for $T_{rec,for}$, since we do not checkpoint the background ensemble.

step (T'_{for}) might be shorter as we do not checkpoint on the runner side. We account for this by:

$$\frac{T'_{for}}{T_{for}} = \delta \quad , \quad \text{with} \quad 0 \leqslant \delta \leqslant 1. \tag{7.14}$$

For a more general form of Equation 7.3, Equation 7.4, Equation 7.12, and Equation 7.13, we divide by T_{for}. With this, we can estimate the speedup giving the relations between the various quantities rather than their absolute value. This leads to:

$$\tau_{rev,A'} = \tau_{rec,ana} + \delta + \alpha_A \left(\tau_{ana} + \tau_{cp}\right) \tag{7.15}$$

$$\tau_{rev,A} = \tau_{rec,ana} + \tau_{rec,for} + \alpha_A \left(\tau_{ana} + \tau_{cp}\right) \tag{7.16}$$

$$\tau_{rev,B'} = \tau_{rec,ana} + \tau_{cp} + \alpha_B \; \left(\delta - \tau_{cp}\right) \tag{7.17}$$

$$\tau_{rev,B} = \tau_{rec,ana} + \alpha_R \, \tau_{rec,for} \tag{7.18}$$

Where τ denotes the relative time T divided by T_{for}. We can now determine the differences for the respective regions by:

$$\Delta\tau_A \coloneqq \tau_{rev,A'} - \tau_{rev,A} = \delta - \tau_{rec,for} \tag{7.19}$$

$$\begin{aligned} \Delta\tau_B \coloneqq \tau_{rev,B'} - \tau_{rev,B} &= \tau_{cp} + \alpha_B \; \left(\delta - \tau_{cp}\right) - \left(\tau_{cp} + \alpha_B \left(1 - \tau_{cp}\right)\right) \tau_{rec,for} \\ &= \left(1 - \alpha_B\right) \left(1 - \tau_{rec,for}\right) \tau_{cp} + \alpha_B \; \left(\delta - \tau_{rec,for}\right) \end{aligned} \tag{7.20}$$

Where we again used $\alpha_R \triangleq {}^{(T_{cp} + \alpha_B \, (T_{for} - T_{cp}))}/_{T_{for}} = \tau_{cp} + \alpha_B \left(1 - \tau_{cp}\right)$ (subsection 7.4.3). We can now compute the speedup using Equation 7.19 and Equation 7.19 by:

$$\text{Speedup}_A = {}^{\Delta\tau_A}/_{\tau_{rev,A}} \tag{7.21}$$

$$\text{Speedup}_B = {}^{\Delta\tau_B}/_{\tau_{rev,B}} \tag{7.22}$$

We plotted the speedup for both regions, using the results from our experiments. We observe, that for large ensemble sizes (more than 256 members), it becomes more beneficial to omit the checkpoint for the background state ensemble (Figure 7.13). However, for ensemble sizes between 32 and 128 members, we achieve considerable speedups with checkpointing both ensembles (Figure 7.12).

7.6 Discussion

Our results show that we protect the MelissaDA 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, MelissaDA 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 a traditional online setup, propagation and analysis run inside the same executable and circulate the states through MPI. MelissaDA performs propagation and analysis on different executables, however, circulates the states between propagation and analysis through the network, avoiding the staging through the file system. We added fault tolerance to the framework in form of global checkpoint files using HDF5. This makes the data available for reanalyses and data-processing. With our additions, MelissaDA incorporates 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.

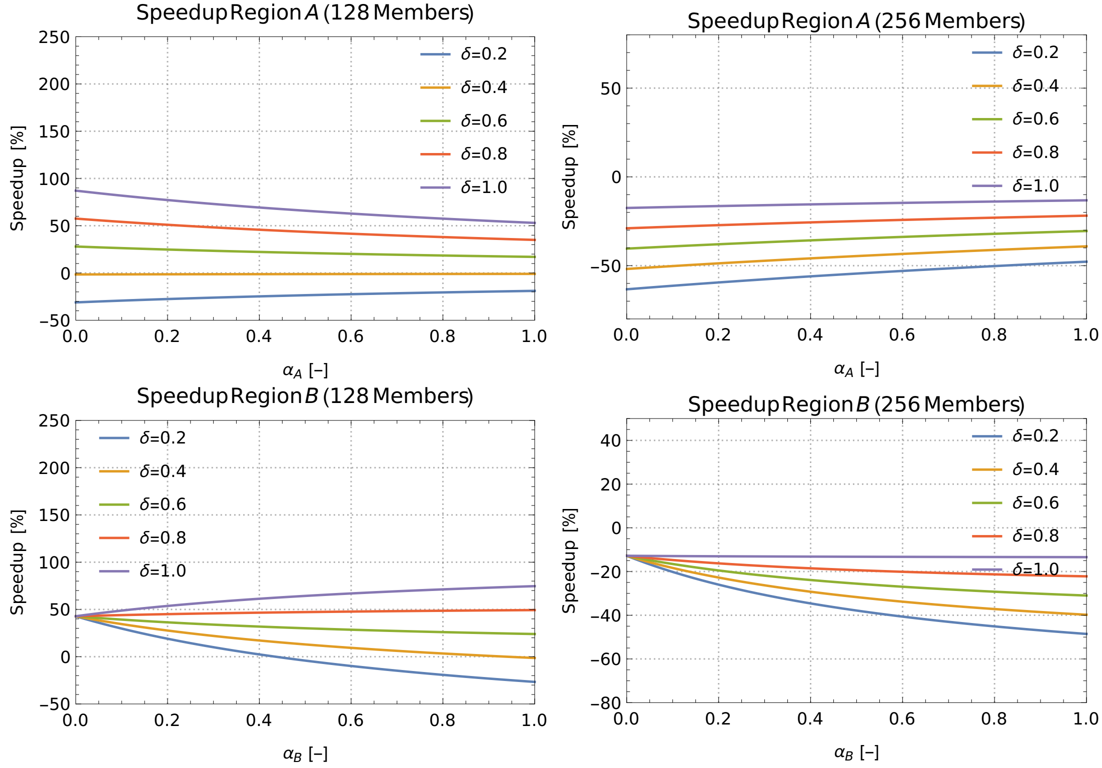


Figure 7.13: Speedup of checkpointing both ensembles towards checkpointing only the analysis ensemble. The speedup is plotted versus the location of the failure in the respective region. For failures in region A, α_A represents the normalized distance from the beginning of the region to the end. For failures in region B the respective normalized distance is given by α_B.

We also identified starting points for 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 increases the data fragmentation inside the global checkpoint files, which can lead to a poor scaling performance of checkpoint post-processing and recovery. From Figure 7.6, Equation 7.5, and Equation 7.6, we can see that by minimizing the total checkpoint cost, we maximize the zero-waste region and minimize the worst-case region. Thus, changing the decomposition into a decomposition by state, likely improves the checkpoint and recovery performance, and therefore decreases the average revival times. We can also minimize the checkpoint and recovery time by changing to a faster checkpoint method. For instance, the standard FTI checkpoint format (one binary file per process). Doing so, we can use other techniques to speed up the checkpoint performance even further, leveraging differential checkpointing or checkpoint compression. 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 checkpoint performance for HDF5 files, we need to experiment with the file creation parameters available in HDF5. It is also likely that leveraging burst buffer technologies [131–133] will improve the checkpoint performance.

7.7 Conclusion

We presented a novel checkpoint-restart implementation for MelissaDA, a distributed framework for ensemble based data assimilation. MelissaDA 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 than one epoch. In fact, the maximum time for the recomputation is bound by the time for one assimilation step plus the time to complete the checkpoint (Equation 7.3). 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 7.6). 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 a 512 member executions to be 1574 seconds. Considering a MTBF of 24 hours, this corresponds to less than 2% of the time of a failure free execution.

Chapter 8

Resilient Online Particle Filter using a Local Particle Cache

Main Contributions

- We designed and implemented a fault-tolerant large-scale online Particle Filter (PF). The PF leverages a node-local distributed and asynchronous particle cache to achieve high parallel efficiency.
- We designed an implemented a scheduling algorithm with particle virtualization for an efficient load balancing and minimal transfers between local and global storage layers.
- We evaluated the PF with the Weather Research and Forecasting model (WRF), a state-of-the-art Numerical Weather Prediction (NWP) system, on a european domain with high resolution, utilizing 2555 particles.

Data assimilation aims to reduce uncertainties and correct the trajectories of numerical model states by integration of observation data available through measuring devices like satellites, or networks of IoT sensors. Data Assimilation (DA) is today used routinely for geoscience applications like weather forecast [134]. However, DA methods are likely to gain importance in many other domains as the growing availability of data motivates the needs for methods capable of hybridizing data and numerical models, such as for digital twins [135]. Departing from the Gaussian assumptions of other popular large scale DA techniques like Ensemble Kalman Filter (EnKF) or Four Dimensional Variational Data Assimilation (4D-Var), particle filters are of growing importance, as no assumptions have to be made on the shape of the underlying Probability Density Function (PDF). Particles (a.k.a., realizations, samples or members) correspond to states of the numerical model, initially drawn from a proposal probability distribution. The particles are each individually propagated forward in time through the numerical model to the next timestep when observation data is available. The PDF at each timestep is generated by weighting the particles depending on the observation likelihood for the particle state. The particles for the next assimilation cycle are then drawn by filter algorithms like Sequential Importance Resampling (SIR). The goal of resampling is to keep a representative sample of particles, discarding particles that took trajectories too unlikely (low weight), while generating new ones with high weights.

In this chapter, we present our proposal for an efficient and fault-tolerant architecture for large scale particle filters. As the number of particles depends quadratic on the dimensionality of the model state [13, 41], Particle Filter (PF) need to handle a very large number of particles efficiently. We exploit certain properties of PFs to modularize the components of the architecture, and we implement particle virtualization to conceptualize particle propagations into tasks. Particle filter updates can be decoupled from the high dimensional states, involving merely the weights instead. The component updating the PDF and resampling new particles, can therefore be implemented as a lightweight component of the system, and the particle states do not need to be gathered in a central point. The propagations takes place on so-called runners, which form part of the particle virtualization. Runners can be considered as a worker pool advancing the scheduled particle propagations. We equip the runners with a distributed particle cache that leverage node local storage hierarchies, where we cache and prefetch particles for a seamless bridging of consecutive propagations. The cache is maintained by dedicated processes (one per runner node),

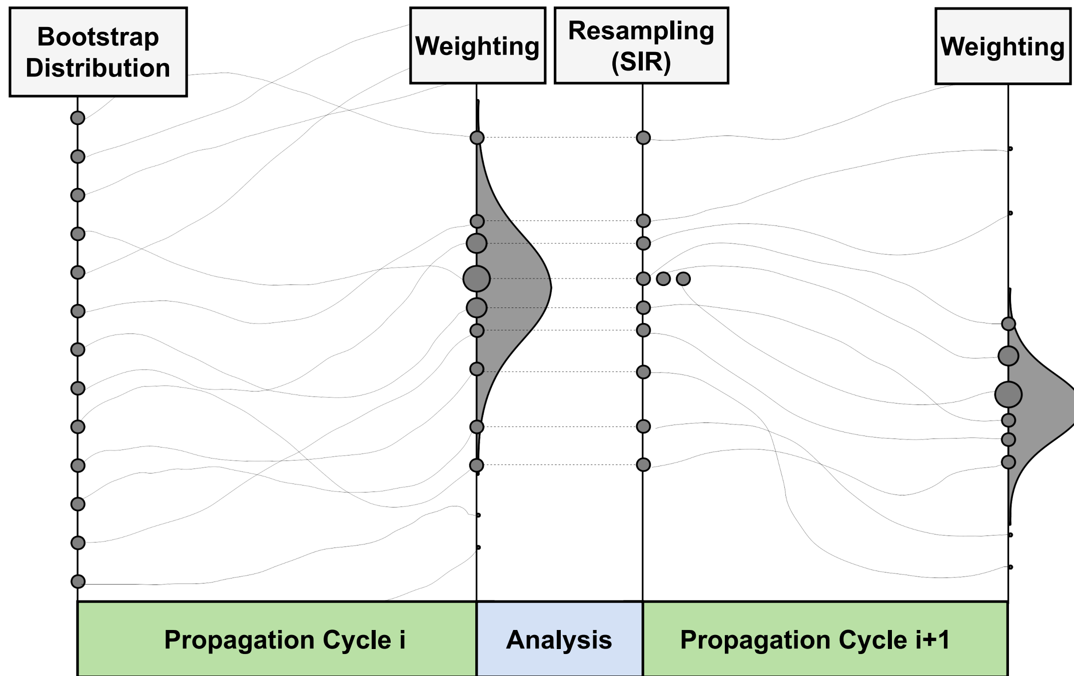


Figure 8.1: Initially particles are uniformly sampled. They are propagated to T_1 where they are weighted taking into account observation data. Resampling leads to discard some particles with low weights (top and bottom), while others with high weights become parent of several ones (3 here).

and we ensure that the cache content is pushed to global storage to make it available for other runners and ensure system resiliency. We evaluate our proposal with a realistic use-case based on the Weather Research and Forecasting (Weather Research and Forecasting model (WRF), version 3.7.1) model [136]. WRF is a popular weather model for both operational forecasting and research. We deployed our proposed PF with 2,555 particles using 20,442 compute cores for simulations on the european domain (15 km horizontal resolution) based on the ERA5 data set, while achieving 87% parallel efficiency. Within the next sections, we review the principles of particle filters and the associated workflowsection 8.1, present the architecture of our approach in section 8.2, evaluate our proposal in section 8.3, discusses the related work in section 8.4, and conclude the chapter in section 8.5.

8.1 Particle Filters

We have given an introduction to particle filtering in subsubsection 2.2.2.2. Now we want to point out some properties of particle filters that we exploit in our proposal. It is important to understand that in contrast to other DA techiques, particle states remain unchanged during the PF update step. Particles that have departed too much from the observations are discarded, and the sample set is reduced without any further corrections to the remaining particles. However, especially for high dimensional problems, particle filters tend to suffer from weight degeneration, i.e., one normalized weight is close to 1 and all the others close to 0. A common approach addressing degeneration consists in resampling the particles based on their importance (high weights). Such a method is SIR. After each filter update, a new sample of P particles is drawn from the generated PDF. Doing so, particles of low weights are automatically discarded, while particles with high weights are drawn several times, leading to multiple propagations starting from the corresponding states (Figure 8.1). It is clear that this method only works if the model is

randomized in an appropriate way. If the model itself does not include randomization, the particles need to be perturbed in a controlled fashion before their propagation in the next cycle.

Takeaway

The following properties are important for our implementation:

(a) Unnormalized weights depend only on the associated particle and observations (Equation 2.17)

(b) The Filter update only depends on the weights (Equation 2.16)

(c) The particle states remain unchanged by the filter update

These properties enable greater resiliency, and we can exploit them to improve the parallel efficiency of our PF implementation.

8.2 Architecture

Other ensemble data assimilation techniques like the various flavors of the EnKF, correct the trajectory of the climate states during the filter update. As we have emphasized before, this is not the case in PF, the states remain unchanged. Moreover, the majority of the filter update can be performed in a local operation; the only operation that involves the particle state, the weight calculation. Beside the corresponding particle state, we need the observation to compute the weight, but not the other particle states. The remaining steps of the update are the normalization of the weights and the resampling. Thus, in contrast to the EnKF, we only need to gather an array of scalar weights, but not the high dimensional climate states for the update. This allows a lightweight central server implementation, where the server performs the relatively cheap filter update, resampling, and scheduling, while the runners advance the particle states with the numerical model, and generate the particle weight at the end of each propagation. The architecture comprises three components the launcher, the server, and the runners. In a nutshell, the launcher submits jobs for the server and runners, the server schedules the particle propagations to the runner pool, and the runners execute the particle propagations. Figure 8.2 visualizes the concept of the server-runner architecture. We can see the runners, implementing a local cache which is maintained by one dedicated process per node. The connection between runners and server is accomplished using TCP with Zero Message Queue (ZeroMQ). In the next sections, we will gradually introduce the components and explain their operation in more detail.

8.2.1 Runner and Cache Interaction

The runners execute the numerical model simulation, which are often itself advanced parallel codes of considerable scale (several nodes). To serve as a runner, the model code implements the MelissaDA API, consisting of two functions for initializing the framework and for updating the model states. From model perspective, the update function initializes the model to a new state, which is then propagated until the next update step, where the model receives the next new state. This continues as often the runner is requested propagations by the server. The complexity of the runtime is hidden by the update function. Four things are happening during the function call:

1. the particle state is stored to local storage
2. the particle weight is computed and send to the cache controller

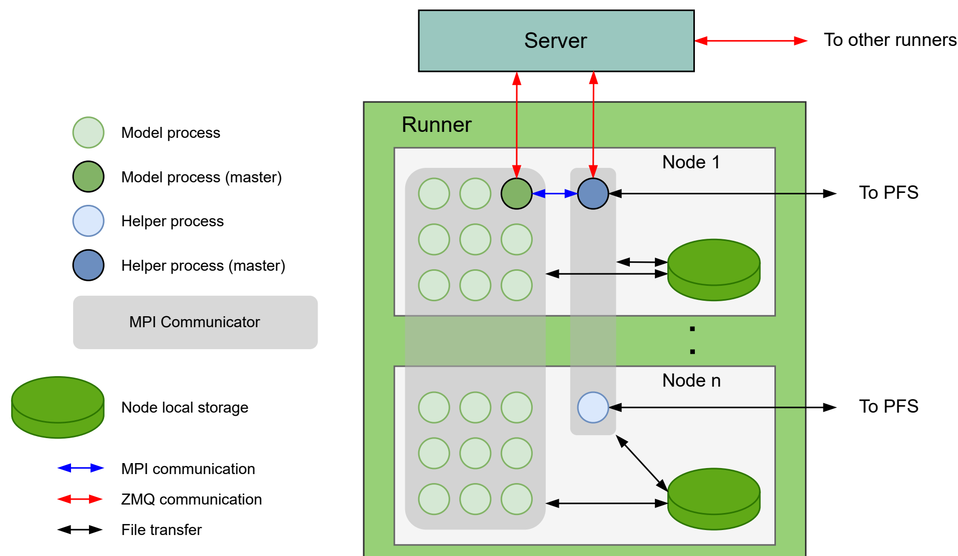


Figure 8.2: Runners/server architecture. The model processes perform the state propagation, the helper processes send propagated states to the PFS and prefetch next scheduled states to the local cache in the background. Communications with the server combine MPI and ZMQ data exchanges.

3. a new particle is requested from the server

4. the new particle is copied from local storage into the state buffer

Those are performed inline by the model processes. The cache controller, on the other hand, ensures that the simulation finds the states on the local storage. Now that the model code is equipped with a new particle for the propagation, two things happen on the cache controller asynchronously:

1. the particle state is copied from local to global storage

2. the weight is sent to the server

The cache controller is loop based and event triggered. Besides the event from above, it receives prefetch requests from the server. Particle eviction as well is maintained by the server. Every time the cache needs to evict a particle due to storage limitations, it submits an eviction request to the server. Knowing the contents of the runner caches empowers the server to distribute the propagations more efficiently. The server can assign propagations of particles prefered to the runners that have them in their cache. On the other hand the server can request runners to prefetch states which are not available in their cache and assign the propagation afterwards.

A typical DA run relies on several runners to propagate states. An important concept in MelissaDA is that each component (launcher, server, several runners) execute on separate allocations. This is an essential part of the Fault Tolerance (FT) concept of the framework, as runners can dynamically be added or removed without affecting the other components. For instance when runner fail, they can simply be replaced. At the same time it is important to make the particle state globally available for the purpose of particle virtualization, each runner must be able to propagate any particle. At the same time, storing the particles on global storage ensures fault tolerance for the case of server failures (even full system failures). The implementation of the cache controller serves the purpose of fault tolerance and particle virtualization, as it stores the particles on global storage. Moreover, by moving the interactions with global storage

in the background, we increase parallel efficiency as we reduce the holding time between two particle propagations. At the same time it improves scalability, since the particle states are provided to the model on exclusive storage devices (no Parallel File System (PFS) saturation, network congestion, etc.).

8.2.2 Cache Eviction Strategy

The number of particle states that can be kept in the cache is limited by the local storage capacity (SSD, NVMe, RAM disk, etc.). To avoid evicting particles that are needed for future propagations, the cache controller is in contact with the server. For the correct operation, the cache needs to provide at least 2 slots, one for the particle state currently propagating, and one for the particle propagating next. The cache controller needs to guarantee that at least that many spots are free. For this it is necessary to evict particles from the cache from time to time. States that are available on global storage are potentially safe for eviction, however, they might be needed in future propagations. The cache controller needs to consults the server as it has knowledge of the cache content of the other runners, and about the remaining particle propagations. When the server is requested it selects the particle that first meets one of following properties in the given order:

1. Particle was discarded during the previous filter update.
2. All propagations for the particle have been completed.
3. Particle with the lowest weight.
4. A randomly selected particle.

Particles that meet properties 1 and 2 can safely be removed from the cache, since those states will not be needed anymore. Particles with property 3 might still be needed for propagations during the current cycle, but may not be selected for the following cycle.

8.2.3 Fault Tolerance and Elasticity

Besides storing the particle states on global storage, MelissaDA implements a second mechanism to ensure fault tolerance. Runners that experience hard failures can be replaced by the launcher. The framework detects dysfunctional runners in two different ways. Terminated runners are recognized by the launcher monitoring the runner's states using system calls or the cluster scheduler. Unresponsive runners, on the other hand, are detected by the server if runners miss due dates for particle propagations. In the first case, the launcher immediately launches another runner instance for compensating the loss. In the second case, the server notifies the launcher which in turn replaces the unresponsive runner. In any case, everytime the launcher starts a new runner, the runner contacts the server and requests the incorporation into the runner pool. This mechanism allows also to increase the runners externally, when new resources become available. With this, the framework becomes elastic and at the same time ensures resiliency towards runner failures. We further protect the server by checkpointing the particle queues and weights. Server failures are detected by the launcher again with timeouts or system calls. Server failures are handled by cleaning up the execution of all runners and resubmission of the framework. This is necessary to reestablish the server runner connections (changed server ip). Finally launcher failures require the resubmission of the entire framework. However, thanks to checkpointing on the server and the globally stored particles, we can recover to the last consistent state.

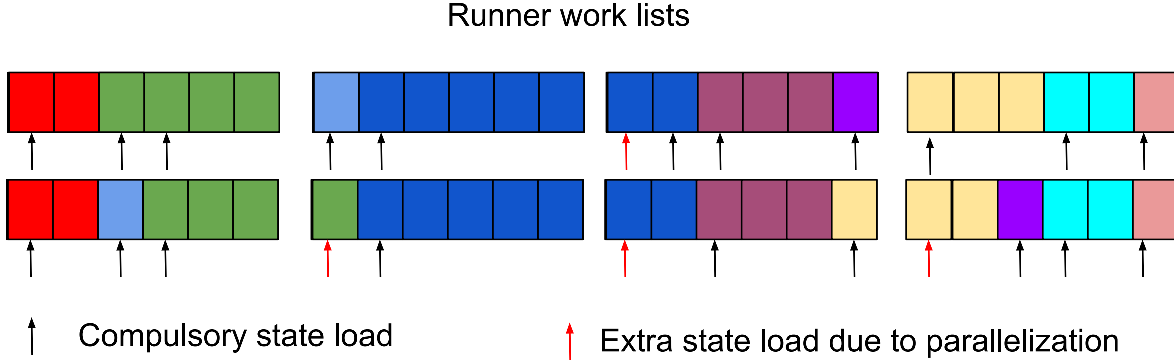


Figure 8.3: Two possible schedules of 24 propagation tasks of equal duration on 4 runners. All particles propagated from the same parent state have the same color (9 parents here). Top schedule is optimal with 9 compulsory loads (one per parent), and one for the dark blue parent that cannot fit in one runner. The bottom schedule, with 2 more sate loads, is a possible one that our on-line scheduling algorithm can produce. This is not optimal but still bellow the general $P + R - 1$ bound as the algorithm ensures that no more than $R - 1$ "color cuts" occur and avoids the same runner loads more than once a given parent state.

8.2.4 Scheduling

In this section we present the scheduling algorithm used by the server to distribute the particle propagations to the runners. Let R be the number of runners, and $p_i, i = 0, 1, \ldots, P$ the P particle states, selected for the next assimilation cycle. The total number of particles to be propagated is $M = \sum_{i=0}^{P} \alpha_i$, where α_i is the number of times the particle p_i was drawn during the resampling. We will first derive a lower and upper bound for the minimum number of particle loads (from global storage) per assimilation cycle c^* assuming that (i) runners do not cache states, (ii) the number of runners is constant and (iii) all particle propagations take the same amount of time. under these conditions, each runner will propagate $\frac{M}{R}$ particles. Because each particle state needs to be loaded at least once, the number of compulsory state loads is P. If $\alpha_i = 1$ for all $0 < i \leqslant P$, i.e., every particle has only one realization, then $c^* = P$. Otherwise, parallelizing the propagations may require to assigne some particles to more than one runner, resulting into additional particle loads. The numer of runners, s_i, particle p_i is assigne to is given by:

$$s_i = \left\lceil \frac{\alpha_i}{\frac{M}{R}} \right\rceil. \tag{8.1}$$

As we have R runners, the list of M particles is cut at most $R - 1$ times, hence, the extra particle loads are at most $R - 1$ (Figure 8.3). This upper limit occurs if only one particle was drawn M times (i.e., $\alpha_0 = M$ and $P = 1$): $c^* = R$. Thus, in the general case the minimum number of state loads c^* is tightly bound by:

$$P \leqslant c^* \leqslant P + R - 1. \tag{8.2}$$

Implementing a static scheduling is not efficient in our case, as the number of runners is dynamical, and the propagation time is not homogeneous. Hence, a static scheduling woud lead to load imbalance. Our extension to a dynamic case relies on dynamic list scheduling to ensure an efficient load balancing [137, 138]: when idle, a runner requests work from the server that returns a particle to propagate. Our scheduling policy is based on the split factor s_i defined in Equation 8.1, however, with dynamic values for M, R, and α_i. The split factor provides the maximum number of runners that can propagate particle p_i. To support this algorithm, the server needs to know the number of successful propagations for particle p_i, and the number of associated current propagations. The scheduling algorithm is:

1. If $\alpha_i > 0$ for particle p_i currently propagated by the runner, decrement α_i and assign p_i again.
2. Otherwise, select another particle p_j and compute the associated split factor s_j. By default any particle could be selected, but priority goes to particles already in the runner cache (See subsection 8.2.2).
3. If s_j runners are already scheduled to propagate particle p_j go back to step 2).
4. Otherwise, assign p_j, decrement α_j and register p_j as being propagated by this runner.

When runners fail, the server needs to update the bookkeeping accordingly. Assuming conditions (i), (ii), and (iii) from above, this algorithm behaves like the static schedule and respects the bound of Equation 8.2.

8.2.5 Implementation Details

The cache is deployed on the runner nodes sharing the global MPI domain with the model processes. The caches form a distributed cache, serving as the cache layer for the global particle propagation. The content of the caches (eviction and prefetching) is regulated by the server, as it has a global view on the assimilation workflow (particles propagated, not yet propagated, etc.). The distributed cache can be considered as level 1 cache of the runners before the main storage (PFS). The local caches leverage dedicated FTI processes allowing the operation in the background of the propagations. During the initialization, FTI splits the main application communicator into one for the model and another one for the helper processes. To modify FTI for our purpose, we extended the event mechanism of the FTI processes (helpers), and we decoupled the staging to allow a better controlling of state transfers between local and global storage. In that way we can control when to upload the local data to global storage, and when to download the data in order to prefetch particles. The communication between model and helper processes relies on asynchronous MPI messages. Communication with the server is established leveraging ZeroMQ.

8.3 Evaluation

To evaluate our PF implementation we rely on the popular WRF model [136]. The core of WRF is based on solving fully compressible non-hydrostatic equations with complete Coriolis and curvature terms, and a large set of physics options. The simulation domain covers most of Europe (Figure 8.4) resolved into 220 by 220 grid cells with horizontal resolution of 15 km and 49 vertical levels with uneven thickness to perform short–range weather forecasting. We randomly choose the date 2018-07-19 to simulate 48 hours in time steps of 100 seconds. The model employs the WSM6 microphysics, MYNN2 boundary layer physics, Grell-3 cumulus parameterization, Eta Monin-Obukhov similarity surface layer processes, and RUC land surface model. We also employ non-hydrostatics increasing the detail in simulated cloud and precipitation. The input initial and boundary condition is based on the reanalyzed ERA5 dataset from the European Center for Medium-Range Weather Forecasts (ECMWF). Data assimilation is performed using the cloud cover fraction (CFRACT). For each particle, the cloud cover fraction is compared with its pendant obtained from the EUMETSAT CMSAF satellite data [139]. The observation data is available hourly, so we perform assimilation cycles over 36 model time steps ($36 \times 100\,\mathrm{s} \mathrel{\widehat{=}} 1\,\mathrm{h}$) to assimilate all observation data testing our approach under high stress.

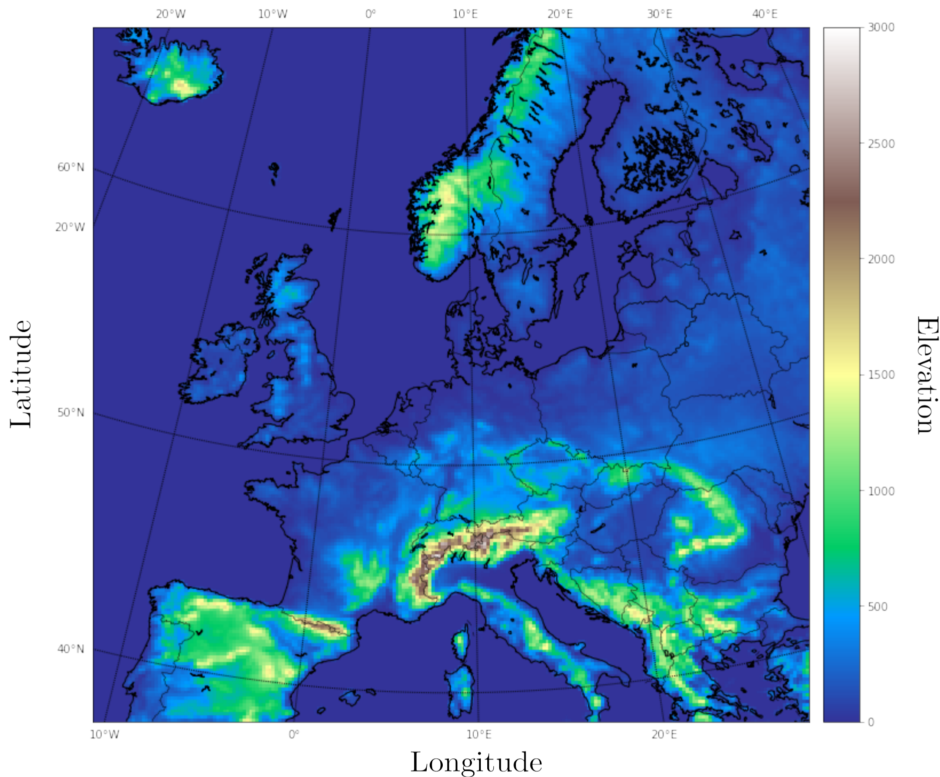


Figure 8.4: The topography of the target domain of Europe for the simulation.

In the following sections we present our experimental results, If not explicitly said, using our PF and WRF on the european domain, with 2555 particles utilizing 20442 compute cores on 512 Nodes of the Jean-Zay supercomputer. Each compute node of Jean-Zay is equipped with 2 Intel Cascade Lake 6248 processors, summing up to 40 cores with 2.5 GHz and 192 GiB RAM per node. Intel Omni-Path (100 GB/s) connects the compute nodes with each other while an IBM Spectrum Scale (ex-GPFS) parallel file system with SSD disks (GridScaler GS18K SSD) is used for persistent file storage. To capture the meteorological state of the European domain, each particle state accounts for 2.5 GiB of data. All experiments that we have performed are listed in Table 8.1.

8.3.1 Runner activity

We want to evaluate elasticity and efficiency of our implementation for the propagation phase. For this, we will first look at selected runner traces from scheduling perspective (Figure 8.5). We observe several things in this trace. First, we can see the elasticity of the framework. During the first two assimilation cycles the individual runners are connecting to the framework, requesting the incorporation into the runner pool. During this phase, as fewer runners are available, each runner has to take on several propagations more than in later cycles. The trace further shows the behavior of the our scheduling mechanism. The red boxes indicate the necessity of transfers from global storage to the runner cache. The green boxes, on the other hand, indicate that no transfer was necessary (up to 69% each cycle).

A closer view on the individual tasks on one runner (Figure 8.6) reveals the times for particle loads (stores) from (to) global storage, and the idle times due to load imbalances. We can further see that the operations on the cache are asynchronously to the particle propagation. We can see that indeed, the propagation occupies the majority of the simulation trace, leading to high parallel efficiencies. A particle

Experimental Setup				
Particles	315	635	1275	2555
Number of runners	63	127	255	511
Number of nodes	64	128	256	512
Model processes	2457	4953	9945	19929
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 cache (s)	2.1	2.1	2.4	4.1
Write state from cache to PFS (s)	1.4	1.6	1.8	2.3
Writes to PFS per cycle (TiB)	0.77	1.55	3.11	6.24
Reads from PFS per cycle (TiB)	0.30-0.4	0.64-0.79	1.27-1.79	2.54-3.82

Table 8.1: Experimental setting and performance overview at 4 different scales. The times are given as average in all cases.

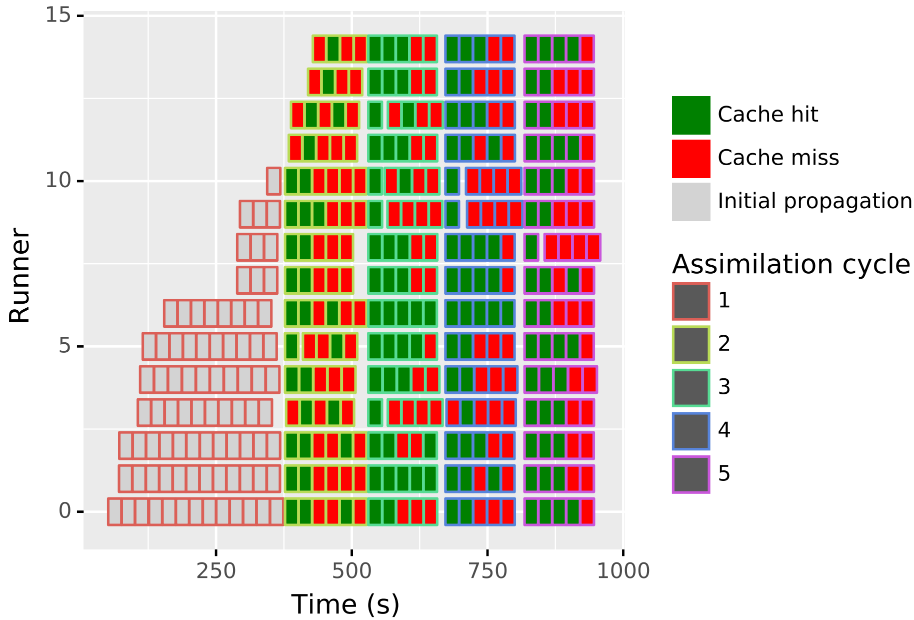


Figure 8.5: Gantt chart of particle propagations executed by 15 (out of 511) randomly selected runners over 5 assimilation cycles. Tasks are green when the associated parent state was already present in the runner cache and did not require a load from the PFS (red otherwise).

propagation takes between 24 and 26.5 seconds, keeping model processes busy 87% of their time. The rest is dedicated to weight calculation (1%), and communication with the server (12%), including waiting times due to load imbalances at the end of the assimilation cycle. Thanks to the asynchronous caching and prefetching, 94% of the particle loads are from local storage reducing the Input and Output (IO) overhead by 14% (compared to using global storage directly), while the deployment of the cache only corresponds to $\widehat{=}$ 2.7% of the total compute resources.

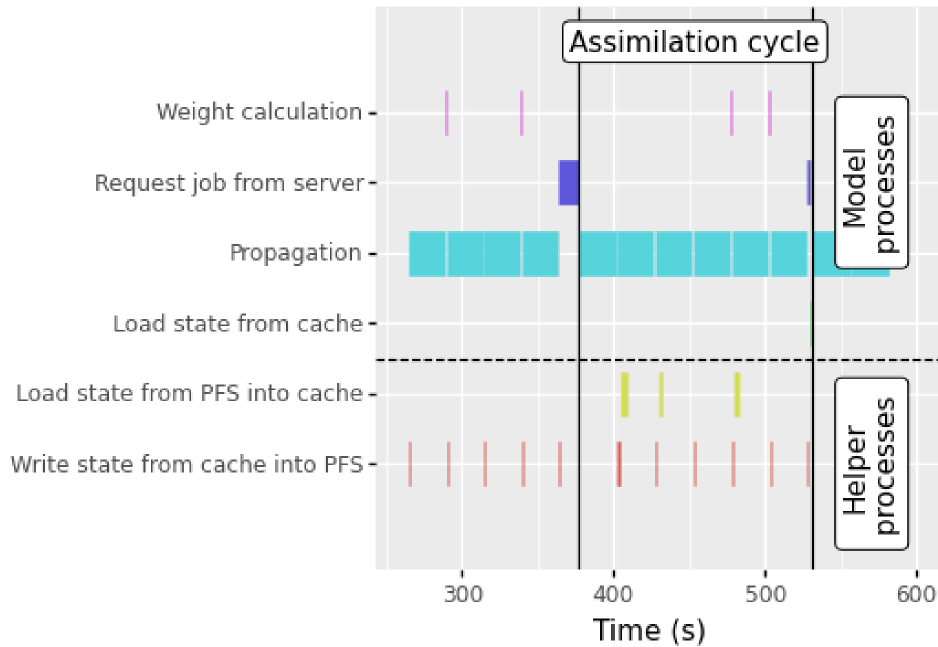


Figure 8.6: Trace detailing the activity of a runner over the course of an assimilation cycle. Helper processes enable to keep model processes busy with particle propagation, except at the end of assimilation cycles when they wait for the server to finish particle resampling (dark blue). Some activities are so thin that they are not visible here (state copies from cache to model). they can become idle

8.3.2 Server activity

The server response time is always in the order of a few hundred microseconds, except for job requests at the end of assimilation cycles if no particle remains for propagation (Figure 8.7). Therefore, these longer times (in the order of seconds) are due load imbalances, and not due to server saturation. In executions with 511 runners, we observed a maximum of 676 requests per second. Assuming average response times of about 500 microseconds, this translates to less than 500 ms of busy time per seconds. Thus, with 511 runners, the server is not even at 50% of its capacity, even though it is deployed leveraging a sequential python code. Simple optimizations are at reach if the server needs to be accelerated (e.g., server parallelization).

8.3.3 State transfers to/from PFS

Each particle state has a size of 2.5 GiB, leading to a transfer of 6.2 TiB to global storage during each assimilation cycle (2,555 particles). Maintaining a cache size of 5 particles, the cache controllers provides between 1024 and 1563 particle states locally (through prefetching), that otherwise would have needed to be loaded from global storage. The P value (number of distinct particles) per cycle is between 1594 and 1629, with up to 5 realizations for some particles. The scheduling algorithm, without caching, is expected to achieve less than $P + R - 1$ loads (compare Equation 8.2). Leveraging the distributed cache, even the minimal number P of state loads is undercut (See subsection 8.2.4). The time to load or to store a state from the PFS can vary significantly and increases with the number of runners, experiencing congestion of the global storage layer. This effect is expected to gain importance at larger scales. However, the distributed cache effectively hides most of the IO cost writing to global storage, as the model processes only write to local storage. The transfer between local and global storage takes place on the helper cores

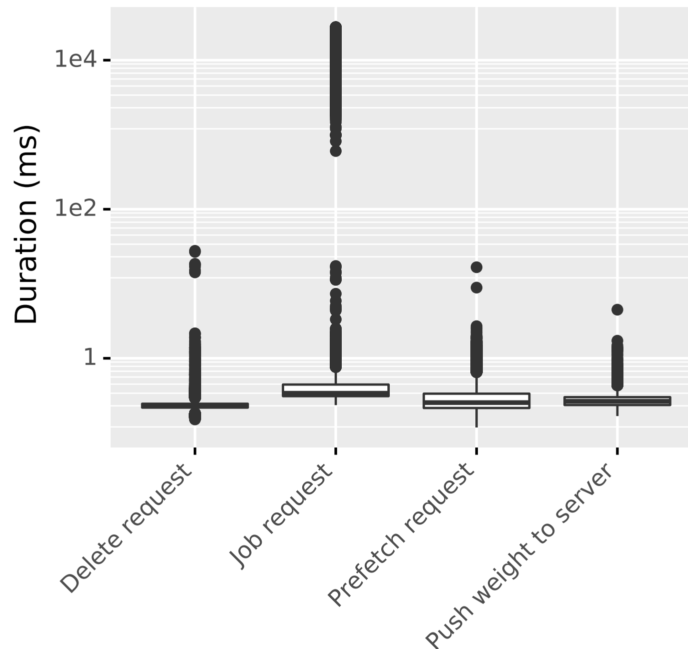


Figure 8.7: Server response times on runner requests.

asynchronously.

8.3.4 Fault tolerance, elasticity and load balancing

We tested the fault tolerance and elasticity of our PF implementation with 63 runners crashing 2 runners (Figure 8.8). We observe that the fault tolerance algorithm reacts as intended, restarting new runners after each crash. The first crash (runner 53) occurs in the worst situation: just after propagating the last particle of the current cycle, which leads to a significant idle period. This is due to the server only detects the runner crash (runner is unresponsive) after the timeout of 60 seconds. Only after that, the particle is redistributed (to runner 44). As the particle that runner 53 was propagating, was the last remaining particle for the cycle, all runners are in idle state until the propagation has finished. As we can see for the second crash (runner 48), crashing at the beginning of the cycle, the idle period due to the failure recognition stays out. We merely observe a gap larger to the other gaps between the cycles, due to the resulting load imbalance. However, we observe a generally well balanced execution with small gaps between two consecutive cycles. The load balancing becomes more important for models that show more variability in propagation times. In WRF, the propagations show rather small variability of about 10%. with other simulation codes, or if executing on heterogeneous resources, propagations might show a much stronger variability.

8.3.5 Scaling

We performed strong scaling experiments with a constant number of runners (63 runners) while increasing the number of particles. We observe a parallel efficiency above 90% (Figure **??**) for executions that allow to distribute the propagations among the runners. The parallel efficiency increases with the number of particles per runner. This is expected, as the load balancing becomes more efficient with increasing possibilities to distribute the propagations. Furthermore, as prefetching enables to overlap IO and

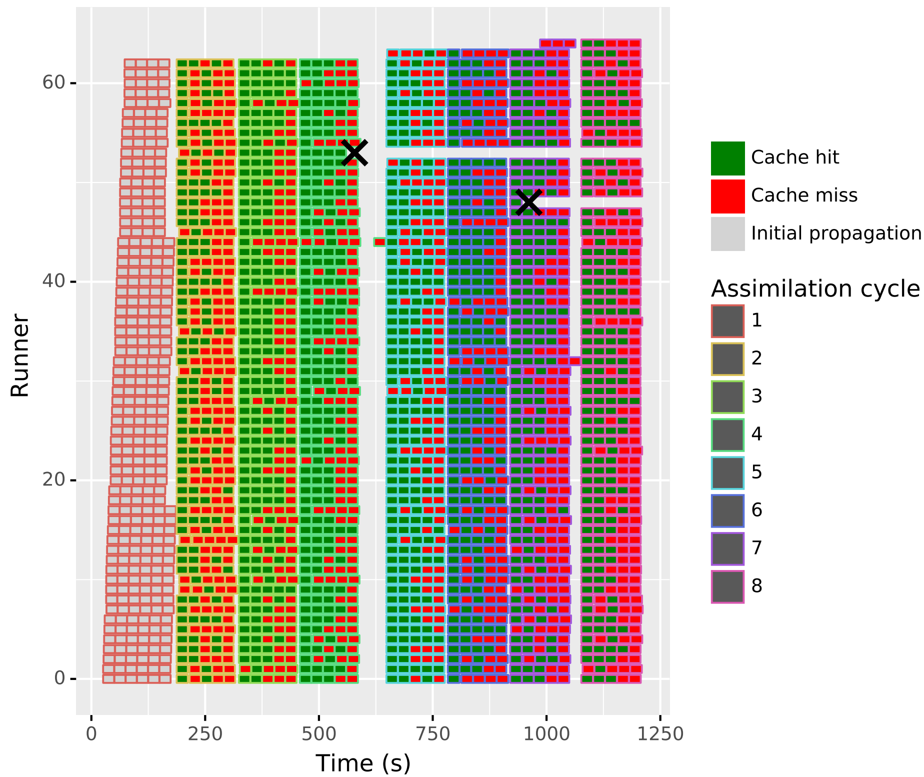


Figure 8.8: Gantt chart as in Figure 8.5. Two runners crashed (black cross) and 2 restarted (top 2 runners).

propagations, increasing the number of particles per runner better amortize the cost of the synchronization associated with resampling. For evaluating weak scaling of the framework, we performed experiments with a constant number of particles per runner, while increasing the number of runners. We observe an increase of 8% for the time of one assimilation cycle from 63 to 511. This translates to about 1.8% for every additional 100 runners and 500 particles.

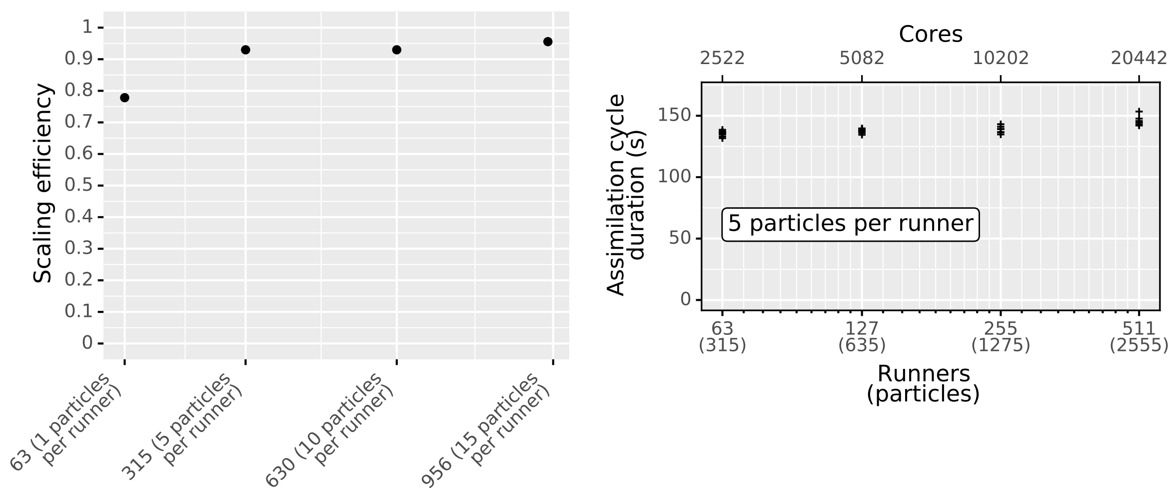


Figure 8.9: Left: strong scaling efficiency using different numbers of particles with 63 runners. One runner sets the reference case. Right: weak scaling performance test: assimilation cycle duration for different numbers of runners, but always 5 particles per runner.

8.4 Related Work

The DA domain features a large variety of techniques and algorithms, for instance, nudging [140], kriging [141], Kalmann filtering [142], ensemble maximum likelihood filtering [143], and particle filtering [144]. We refer to [145, 146] for a comprehensive overview. We focus on Monte Carlo DA methods relying on ensemble runs models to compute a statistical estimator (co-variance matrix for EnKF, PDF for particle filters). To aggregate the data produced by all members (i.e., particles) two main groups of approaches are used. Either the data is stored to files and then processed in a second step (a.k.a., offline mode), or the data is processed online, typically within large MPI codes enclosing the particle propagations, and filter updates. Frameworks relying on the offline mode include EnTK [147], with the largest published runs involving 4,096 members for a molecular dynamics application with EnKF filtering [148]. Similarly OpenDA, using NetCDF for data exchange with the NEMO code [149]. DART supports both online and offline modes [127], with reported large scale runs in offline mode using about 1,000 members with an oceanic code [60], and using 1,024 members with the Local Ensemble Transform Kalman Filter (LETKF) DA utilizing 6 M cores at the Fugaku supercomputer [150, 151].

File based approaches have the benefit of their simplicity, providing fault tolerance and elasticity. But these solutions do not support member virtualization, state caching and prefetching. Hence, starting or restarting a member requires to request new resource allocations, and launching a new instance of the model code with all the associated start-up costs.

The online mode, on the other hand, avoids the I/O bottleneck. PDAF [152], which supports both modes, has been used for online data assimilation with the EnKF in the regional earth system model TerrSysMP using 256 ensemble members [153]. ESIAS uses online DA with particle filters incorporating up to 4,096 particles in a wind power simulation on a european domain [154]. Notice that we work with the same WRF component of ESIAS in this paper, using a configuration an a similar domain, but at higher spatial resolution, inuding with more advanced and more time consuming physics. We can also find ad hoc MPI codes for online DA using an atmospheric model with 10,240 members, using a localized EnKF utilizing 4,608 compute nodes [40].

All these MPI approaches lead to monolithic code without intrinsic support for fault tolerance, elasticity or load balancing. Here, we propose an alternative architecture for particle filters, relying on distributed caching and checkpointing to suppress the server bottleneck and significant reduction of data movements. Recall that this approach is possible, because the filter update does not impose state corrections.

8.5 Conclusion

In this chapter, we proposed an architecture for handling very large ensembles for particle filters. The architecture was designed to address the challenge of exascale computing that will allow massive ensemble runs [155]. The architecture is based on a server/runner model where runners support a distributed cache and virtualization of particle propagation, while the server aggregates the weights computed by the runners and ensures the dynamic balancing of the workload. With the addition of a global checkpointing mechanism for particles, the architecture supports dynamic changes in the number of runners during execution for fault tolerance and elasticity. Experiments with the WRF weather simulation code show that our framework can run at least 2555 particles on 20442 cores with 87% scaling efficiency.

Chapter 9

A Framework for Automatic Validation and Application of Lossy Data Compression in Ensemble Data Assimilation

Main Contributions

- Design and implementation of a framework for validation and application of lossless/lossy compression in ensemble Data Assimilation (DA). The framework provides two operational modes: (1) validation mode: identify viable compression parameters using robust statistical qualifiers that test state and ensemble consistency (2) dynamic mode: apply compression parameters. The compression parameters can vary among all state variables.

Ensemble data assimilation with large ensembles and large models requires high performance I/O [156–158]. This is due to the large amount of data that needs to be circulated between different constituents of the assimilation system. A still widely used workflow in ensemble data assimilation is to perform the climate simulation and the data assimilation on separate executables [59]. The ensemble members (i.e., the climate simulations) store the climate states to the file system, and after all simulations have finished, the data assimilation system reads the files, assimilates the observations and writes back the improved states to storage. The ensemble members then reread them to perform the next assimilation cycle. The amount of data transferred between the two steps often leads to an I/O bottleneck, where the storage subsystem cannot deliver the throughput that is needed to keep up with the computing power. In some cases, I/O can be overlapped with computation and be performed in the background, which alleviates the I/O overhead itself. The states can also be transferred through the network, bypassing the I/O layer. However, this approach is limited by the memory available and raises fault tolerance issues, if it involves large monolithic MPI allocations.

Reducing the data to minimize storage requirements and storage space availability for other users is beneficial in either case. A recent example of storage based state circulation between simulation and data assimilation system has been published by Yashiro et al . [57]. The article presents the execution of the NICAM-LETKF system on Fugaku [159], using 82% of the entire system. During each cycle, the system circulates more than 400 TB of data through the parallel file system. Yashiro et al. compares double to mixed precision executions, where the computing times with mixed precision show a 1.6x speedup compared to double precision, while reducing the data size to the half. This demonstrates the prospect of departing from the double precision doctrine in climate science. There are a number of works studying explicitly the impact and advantage of mixed precision in data assimilation [160–162]

Besides using mixed precision, data can be reduced by compression. Data can be compressed in a lossless fashion, without the loss of accuracy, or in a lossy fashion, with a certain loss of accuracy. Since climate systems are strongly non-linear and typically chaotic, it is essential not to introduce significant perturbations when applying lossy compression. On the other hand, the models and the observations are both imperfect and inevitably introduce errors to the states. This means that the intrinsic precision of the states can be lower than the precision of the data type used in the application. Indeed, multiple previous

studies have shown that climate simulations can tolerate certain loss in data precision [163–166].

We propose a framework that: (1) explores the impact of lossy data compression on the climate states and ensemble consistency through time (error propagation), and (2) dynamically selects the best compression parameters during runtime. The former constitutes the **validation mode** of the framework and the latter the **dynamic mode**. In validation mode, the framework generates and stores validation data for each assimilation cycle. The collected data provides a means to evaluate the impact of repeatedly compressing the states through time. We provide a JSON configuration file to conveniently set the desired compression parameters. In dynamic mode, the compression parameters are applied to the states, reducing their size before writing them to the file system. Optionally, the states can be tested by a validation function before storing them. Currently we provide compression with ZFP [17], FPZIP [18], single-precision and half-precision. The framework allows easy interfacing and is build on top of the ensemble data assimilation architecture of the MelissaDA Particle Filter (PF) that we have developed (chapter 8). After exposing the simulation variables to the PF, and a few more steps, the climate model can be operated with the two modes from above.

The rest of the chapter is organized as follows: Section 9.1 provides background information on ensemble data assimilation techniques. Section 9.2 presents the design and implementation of the proposed framework and explains its usage, and Section 9.3 provides the experimental evaluation. Related work is surveyed in Section 9.4, and finally, Section 9.5 concludes the chapter.

9.1 Background

In this section, we outline the basic concepts behind our work and clarify the terminology that we use.

9.1.1 Ensemble Data Assimilation

Data assimilation is based on Bayes' theorem, allowing us to combine the information from both real world observation and numerical model states. Combining the information from both sources leads to an improved accuracy of the state [167]. Ensemble data assimilation follows a Monte Carlo approach, approximating the error probability density function by a statistically significant sample of states. The most common ensemble methods for data assimilation are the Ensemble Kalman Filter (EnKF) [31] and the Particle Filter (PF) [34]. To address the issues that arise from relatively small ensemble sizes given the high dimensionality of the climate states, modified flavors of the original versions are often used, for instance, the Local Ensemble Transform Kalman Filter (LETKF) [168] or Localized Adaptive Particle Filter (LAPF) [169], and others.

One important difference of PFs compared to Ensemble Kalman Filter (EnKF)s is, that the particle states do not change during the filter update. Instead, weights are assigned to the particles, and particles with small weights are discarded, while particles with high weights are propagated in the next cycle. The particle filter implementation in MelissaDA uses Sequential Importance Resampling (SIR), where the particles of the next cycle are drawn from the Probability Density Function (PDF) which is generated by the weights (see also subsubsection 2.2.2.2). The number of particles in each cycle is constant, and typically particles with high weights are propagated several times, while other particles are not propagated at all (discarded) due to their low weight. Due to the multiple propagation of identical particle states, this technique relies on a randomization of the model. This is achieved either by introducing stochastic

dynamics into the model evolution, or by adding small perturbations to the input particle states before the propagation [13, 34].

9.1.2 Terminology

In spite of the same origin, being both Monte Carlo methods, the terminology used for ensemble Kalman filters and particle filters is quite different. To avoid confusion, we will introduce the terminology that we use in this work. We implemented our validation framework into the particle filter of MelissaDA [1]. For this reason, we will use the particle filter terminology. We refer to a simulation state as particle state, or sometimes just particle or state. The workflow of ensemble data assimilation is divided into assimilation cycles. Each cycle comprises the propagation step and the update step. During the propagation step, the climate states are advanced by the numerical model (propagated). During the update step, the model states are improved by the filter update, assimilating the observations. In particle filtering, the update step is called sampling, or resampling.

9.2 Design and Implementation

In this section, we first introduce the particle filter implementation of MelissaDA, which serves as a testbed for our work, and then explain the implementation and operation of our proposed framework in detail.

9.2.1 MelissaDA Particle Filter

MelissaDA is designed for ensemble data assimilation at large scale. The framework comprises three modules; a launcher, a server, and multiple runners. In comparison to the common practice in ensemble data assimilation leveraging a bash script for the workflow, the launcher replaces the script and orchestrates the submission of the ensemble simulations and the data assimilation system. The launcher has plugins for the most common cluster schedulers to submit and monitor the jobs for the server and runners. In this architecture, the runners constitute a worker pool for propagating the particle states. The server requests available runners to propagate unscheduled particles and performs the resampling after all particles have been propagated. To ensure optimal fault tolerance, the server and each runner is allocated on separate jobs and are restarted by the launcher upon failures. Besides the PF, MelissaDA provides several other ensemble DA filters, for instance, various flavors of the EnKF, and allows the creation of custom filter plugins. MelissaDA implements a particle filter that uses a fast distributed cache on the runner nodes, where particles can be asynchronously prefetched and cached for future propagations (see also chapter 8). The cache is maintained by dedicated MPI processes, asynchronously, and utilizes one process per runner node. Generally, for fault tolerance, the states need to be stored on global storage. Since this is typically slower than using the local storage, the simulation processes only write and load from local storage. The cache controller is responsible for providing the states locally and sending the states to global storage after they have been propagated.

In most particle filters, in contrast to Kalman filters, the states are not changed during the filter step. Instead, states that carry high weights (i.e., that are consistent with the observation data), are selected for

[1]The validation methods that we use, however, are perfectly suitable for other ensemble methods as well.

the next assimilation cycle and states with low weights are discarded. This is precisely why the local cache on the runners helps to overcome the Input and Output (IO) bottleneck; states that have been selected for the next assimilation are still locally available on runners that have propagated them. Since the states remain unchanged during filtering, they are immediately ready for propagation. The local cache leverages Fault Tolerance Interface (FTI) to store and load the states. FTI is a multi-level checkpoint/restart library that is aware of the node local storage. We implemented several compression techniques into FTI to enable the state compression while storing the state and decompression while loading it.

9.2.2 High-Level View on the Validation Framework

Our proposed framework comprises two modes of operation. The first mode allows us to explore the impact of data compression on the integrity of the ensemble and of the states in particular. The second mode allows us to apply the best compression parameters during operation to increase performance, while respecting data consistency. We refer to the first mode as **validation mode** and to the second as **dynamic mode**. The respective modes and corresponding parameters are selected by providing a JSON configuration file (see Listing 9.1).

```
{
  "variables" : [ "state1", "state2" ],
  "compression" : {
    "method" : "validation",
    "validation" : [
      {
        "mode" : "fpzip",
        "parameter" : [16,24,32]
      },
      {
        "mode" : "zfp",
        "type" : "precision",
        "parameter" : [32,40]
      }
    ],
    "dynamic" : [
      {
        "name" : "state1",
        "sigma" : 10e-7,
        "mode" : "zfp",
        "type" : "accuracy",
        "parameter" : [0,6,8,10]
      },
      {
        "name" : "state2",
        "sigma" : 10e-5,
        "mode" : "fpzip",
        "parameter" : [24,28,32,36,40]
      }
    ]
  }
}
```

Listing 9.1: Example of a configuration file. We set the compression parameters for two variables named *state1* and *state2*. The framework will operate in validation mode, since the method key is set to *validation*. To apply the dynamic mode, the method must be set to *dynamic*.

Our framework operates in conjunction with the particle filter of MelissaDA. During the particle filter update, the particle weights are normalized and P particles are drawn from the resulting distribution function. P remains constant during all cycles and it can be set in the MelissaDA configuration. Hence, during each cycle we propagate the same number of particles. However, the SIR algorithm leads to a sample of only M distinct particles with typically $M < P$. Therefore, some particles are multiplied. To account for this, the model needs to provide some randomness to ensure that two propagations of the same particle lead to distinct output states

During the validation mode of our framework, we now propagate $(C + 1) \cdot P$ particles, with C being the number of parameters that are specified in the configuration file. The configuration shown in Listing 9.1, leads to the propagation of 6 ensembles with P particles each. One ensemble for the original states and 5 ensembles that use data compression with the specified parameters. This allows us to compare the ensembles that leverage compression to the original ensemble. Furthermore, we can observe the evolution of each particle ensemble over time for the number of cycles specified in the MelissaDA configuration.

The dynamic mode aims improving the performance of the data assimilation system. Thus, only one ensemble with P particles is propagated, and MelissaDA operates as it would without our additions. However, this time compressing the particles with the parameters that have been specified in the JSON file. During the first assimilation cycle, the compression parameters are tested using a validation function. Per default, we check the point-wise maximum error between the compressed and uncompressed state. A custom function can be provided by the user, or the validation can also be deactivated entirely. If using the default validation function, the compression parameters that lead to a point-wise error larger than *sigma* (see Listing 9.1) are discarded. The qualified parameters are stored in descending order by the compression rate. In subsequent cycles, the best compression parameters (highest rate) are successively checked by the validation function, and the first parameter that qualifies is used for compressing the state before writing it to the file system. We allow setting the compression parameters for each variable of the climate state independently. The meta-data that is required to recover the variables with the correct compression parameters is maintained by FTI. A speedup is achieved, if the time for compression, decompression, and validation is less than the time that we save due to writing and reading fewer (compressed) data.

9.2.3 Validation Mode

In validation mode, the MelissaDA launcher submits several validator instances, in addition to the server and runner instances. As for the runners, the number of validators can be specified in the MelissaDA configuration. Each validator is executed on one node and is parallelized leveraging all available cores on the node. Figure 9.1 shows the workflow in validation mode. The diagram indicates that the propagation of particles by the runners is overlapped with the validation performed on the validators. The workflow is a follows: during the propagation phase, the server schedules particles to the runners and waits for the particles weights to be returned. As soon as all particles have been propagated, and the associated weights received, the server performs the filter update and communicates particle ids and weights to the validators. While the validators calculate and store the statistical qualifiers, The runners continue the propagation of the particles of the next assimilation cycle.

The statistical qualifiers that we calculate are the same as in the work from Baker et al. [165], where the impact of data compression on climate states is discussed in detail. We implemented the root mean

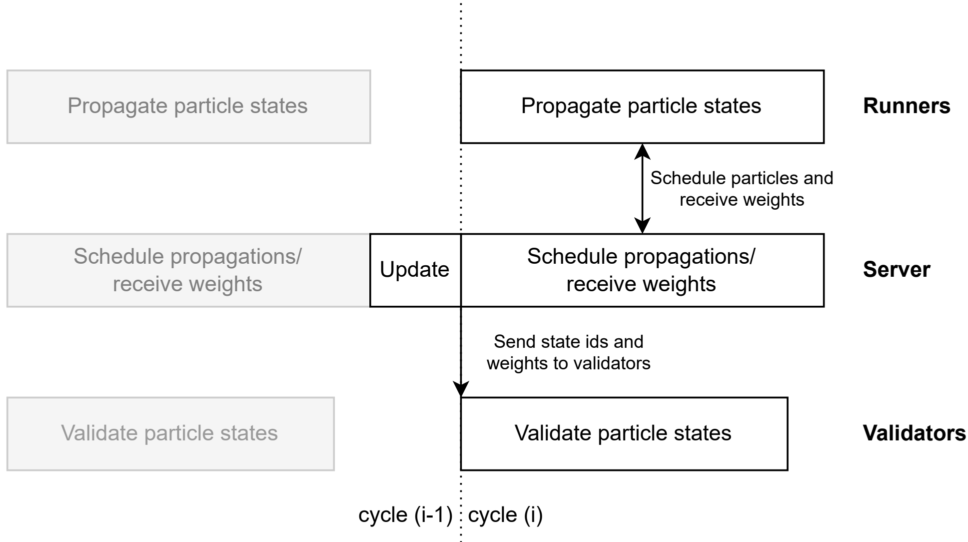


Figure 9.1: Workflow in validation mode.

squared Z-value (RMSZ). Where we calculate 2 different representations of the Z-value. We calculate $Z^{p,0}_{x_{c,i}}$, that encodes information on the ensemble spread, and $Z^{p,+}_{x_{c,i}}$, that can detect a bias on the ensemble spread introduced by the compression. The two Z-values, and the RMSZ are calculated as follows:

$$Z^{p,0}_{x_{c,i}} = \frac{x^p_{c,i} - \bar{x}^{P/p}_{0,i}}{\sigma^{P/p}_{0,i}} \tag{9.1}$$

$$Z^{p,+}_{x_{c,i}} = \frac{x^p_{c,i} - \bar{x}^{P/p}_{c,i}}{\sigma^{P/p}_{c,i}} \tag{9.2}$$

$$\text{RMSZ}^p_{X_c} = \sqrt{\frac{1}{N}\sum_i^N \left(Z^p_{x_{c,i}}\right)^2} \tag{9.3}$$

with:

$$\bar{x}^{P/p}_{c,i} = \frac{\sum_{k!=p}^P w_k x^k_{c,i}}{\sum_{k!=p}^P w_k} \tag{9.4}$$

$$\sigma^{P/p}_{c,i} = \frac{\sum_{k!=p}^P w_k \left(x^k_{c,i} - \bar{x}^{P/p}_{c,i}\right)^2}{\sum_{k!=p}^P w_k} \tag{9.5}$$

$\text{RMSZ}^p_{X_c}$ is the root mean squared Z-value, and constitutes the qualifier that we actually store. This value is computed for both Z-value representations. P is the number of particles, X_c specifies the state variable with c being the compression parameter-id and $c = 0$ indicating the uncompressed state. The index p denotes the particle-id. Consequently, we have one $\text{RMSZ}^p_{X_c}$ value for each particle, state variable, and compression parameter ($P \times C \times$ Number of state variables). Finally w_p denotes the weight of particle p. We further compute the peak signal-to-noise ratio (PSNR):

$$\text{PSNR}_{X_c} = 20\log_{10}\left(\frac{\max\left(|x_{0,i}|\right)}{\text{RMSE}_{X_c}}\right) \tag{9.6}$$

Where $x_{0,i}$ indicate the components of the uncompressed state variable X_0, and RMSE_{X_c} is the root mean squared error between the compressed and uncompressed state variable. Further, we compute the Pearson

correlation coefficient:

$$\rho_{X_c} = \frac{\mathrm{Cov}(X_0, X_c)}{\sigma_{X_0}\sigma_{X_c}} = \frac{\sum_i^N (x_{0,i} - \bar{x}_{0,i})(x_{c,i} - \bar{x}_{c,i})}{\sqrt{\sum_i^N (x_{0,i} - \bar{x}_{0,i})^2 \sum_i^N (x_{c,i} - \bar{x}_{c,i})^2}} \tag{9.7}$$

The pointwise maximum error:

$$\Delta X_c^{\max} = \max\left(|x_{0,i} - x_{c,i}|\right) \tag{9.8}$$

And further the mean, standard deviation, minimum and maximum values for all state variables X_c, and the compression rate:

$$\mathrm{CR}_c = \frac{\text{original size}}{\text{compressed size}} \tag{9.9}$$

The validator is implemented in python and can be further customized by passing custom validation functions to the validator class. The path to the custom validator script is set in the MelissaDA configuration. If no path is set in the configuration, the framework uses the default validator, calculating the introduced qualifiers. Listing 9.2 shows an example of a custom validator script.

```
1  from validator import *
2
3  def custom_write( mean, variance, cycle, nranks, ndims ):
4    '''
5    cycle     - Assimilation cycle
6    nranks    - Number of simulation processes
7    mean      - { "var1" : [mean_rank1, ...], "var2" : [mean_rank1, ...], ... }
8    variance  - { "var1" : [variance_rank1, ...], "var2" : [variance_rank1, ...], ... }
9    ndims     - { "var1" : [ndim_rank1, ...], "var2" : [ndim_rank1, ...], ... }
10   '''
11   ...
12
13 def custom_evaluate( data, rank, name ):
14   '''
15   data - variable data on application rank
16   rank - application rank
17   name - variable name
18   '''
19   ...
20
21 def custom_compare( data, rank, name ):
22   '''
23   data[0] - uncompressed variable data on application rank
24   data[1] - compressed variable data on application rank
25   rank    - application rank
26   name    - variable name
27   '''
28   ...
29
30 validator = Validator(
31   evaluate_function=custom_evaluate,
32   compare_function=custom_compare
33   write_funtion=custom_write
34 )
35 validator.run()
```

Listing 9.2: Example of a custom validator script. The Validator class provides arguments for callback functions. The functions have to comply the function interface. We can pass the functions as scalar variables or arrays. The evaluate and compare functions must return a scalar value. The result will be included in the data output. A function to write the mean particle state into files can also be provided.

The developer can provide three kinds of custom functions. The **evaluation function** is called with the particle state that was compressed with parameter c. The intended use of this function is computing scalar quantities that depend on the state variables X_c. For instance, the total energy, energy budgets, etc. The evaluation function(s) can be used in both the validation and dynamic mode. The **compare function** is called with both the data of the uncompressed state and the compressed counterpart, and provides a means to customize the validation. The developer can provide additional qualifiers testing the consistency of the compressed states. In addition to the two functions, it is also mandatory to pass appropriate reduction functions, since the functions are called with the rank local parts of the data. All values that are calculated will be recorded and written into a comma seperated csv file. Finally, it is possible to provide a custom **write function**. The intended use of this function is to write the mean state and variance into a file. The developer is free to decide the structure and format of the file. The compare and evaluation functions can also be passed as function arrays. It is therefore possible to pass as many functions as desired of those kinds. The compare functions(s) will only be available in validation mode. The evaluation function(s) are available in both modes, and the write function is only available during the dynamic mode.

9.2.4 Dynamic Mode

After identifying the parameters that can safely be applied for data compression leveraging the validation mode, the framework can be operated in dynamic mode. This mode aims providing the best performance for the ensemble data assimilation. The same configuration file as for the validation mode serves here as well for providing the compression parameters (see Listing 9.1). The dynamic mode operates in two phases. During the initial phase we check all compression parameters using a validation function and discard those parameters that fail the validation. The remaining parameters are sorted by the compression rate and kept in a runtime variable. In the following, before writing the particle state to disk, we compress and decompress the state using the best parameters, and check the integrity of the particle state with the validation function. If the validation check passes, the state is stored to the file system. If the test fails, we check the next parameter in the list, and repeat the procedure until we find a parameter that qualifies. This ensures, that we never use a compression parameter that introduces intolerable deterioration to the particle state. However, this procedure is optional. The compression can also be directly applied, without applying the validity check.

Per default, no validator instances are submitted by the launcher during the dynamic mode. However, validators can be submitted, if wanted, to compute certain quantities using custom evaluation functions. Moreover, they can be used to write out the mean particle state with a custom function. This has the advantage, that these tasks are performed asynchronously to the particle propagation. The validators are implemented in Python, hence, we can leverage Python bindings for common IO libraries to write the state data (netCDF4 [170], ADIOS [171] HDF5 [172], i.a.).

9.3 Evaluation

We evaluate our framework based on an exemplary workflow a user would follow applying it to a climate model. First, we use the validation mode to identify viable compression parameters. Afterwards, we perform the data assimilation in dynamic mode, using the qualified parameters from the validation mode.

To measure the performance of the validation tasks, we instrumented the validators with an event logging mechanism. To measure the performance in dynamic mode, we leverage the internal profiler of MelissaDA. We present the analysis of the statistical qualifiers from the validation mode in subsection 9.3.3, the performance of the validators in subsubsection 9.3.4.1, and the performance of the data assimilation in dynamic mode in subsubsection 9.3.4.2.

9.3.1 Experimental Setup

All of our experiments were performed on Fugaku [173], a 488 (double-precision) PFlops supercomputer hosted by RIKEN R-CCS in Japan. Fugaku consists of 158,976 compute nodes that are each equipped with a Fujitsu A64FX CPU. A64FX provides 48 application CPU cores and is integrated with 32 GiB of HBM2 memory. The compute nodes are interconnected through the TofuD network. Each group of 16 compute nodes in Fugaku shares a 1.6 TB SSD storage, and all the nodes can access the global 150 PB Lustre file system. We utilize the SSDs in the so-called *local* mode, where each SSD is divided proportionally among compute nodes in the given group and is exposed as dedicated per-node file system.

9.3.2 Methodology

We performed experiments with different state sizes (N), and different ensemble sizes (P), to examine the scaling behavior of the validation mode. We selected such N, that result in state sizes of 16, 32, 64 and 128 MB (state size in Bytes = $8N$). We further set P to 25, 50 and 100 particles, using 36, 72 and 132 compute nodes respectively. All experiments are performed using the Loren96 [129] model. The model equation reads:

$$\frac{dx_i}{dt} = (x_{i+1} - x_{i-2})x_{i-1} - x_i + F \qquad i = 1, \ldots, N \tag{9.10}$$

For values of $N \geqslant 9$ and $F \geqslant 5$ the model exhibits chaotic behavior [174]. We set the forcing to 6 in all our experiments. The model is initialized adding small perturbations at t_0. Before starting the data assimilation, we run the model for a long enough time ($DT = 10$), so that it exhibits a chaotic state. The time unit in the Lorenz96 model corresponds to about 5 days in an atmospheric model [174]. Hence, running the model for 10 time steps corresponds to about 50 days. We further introduce a generic perturbation to the states at the beginning of each particle propagation. To ensure that we can track the errors that are introduced by the compression method only, we use identical seeds for particles p_c with $c = 0, \ldots, C$, with C being the number of compression parameters. Thus, at the beginning of each particle propagation, we reset the seed to a value $sd = f(\mathrm{pid}, \mathrm{rid}, \mathrm{rank}, \mathrm{t})$, with *pid* the particle-id, *rid* the runner-id, *rank* the mpi rank of the runner and *t* the assimilation cycle.

9.3.3 Statistical Evaluation

In this section, we present the analysis of the statistical qualifiers computed in validation mode. The aim of this analysis is to identify viable compression parameters for production runs. We evaluate the statistical qualifiers presented in subsection 9.2.3, where they have to fulfill certain requirements which we will pose in the next paragraphs. If all the requirements are met, the parameter can safely be applied for compression in the Lorenz96 model.

9.3.3.1 Z-Value Deviation

We start with testing the ensemble consistency. For this, we apply the two Z-tests from Baker et al. [164]. The first test checks if the RMSZ values with $Z^{p,0}_{x_{c,i}}$, and the RMSZ values of the uncompressed ensemble share the same distribution. According to Baker, it is sufficient to measure the Z-value deviations:

$$\Delta\text{RMSZ}^p_{X_c} = \left|\text{RMSZ}^p_{X_c} - \text{RMSZ}^p_{X_0}\right| \tag{9.11}$$

and to require the deviation to be smaller than 0.1:

$$0 \leqslant \Delta\text{RMSZ}^p_{X_c} < 0.1 \tag{9.12}$$

As long as all values stay within this interval, the test is considered passed. Figure A.1, Figure A.2, Figure A.3, and Figure A.4 show the plots resolved by the compression method. The plots show the values for cycles 1 to 18 of the ensemble data assimilation, indicated by the different colors. The variation of the z-value deviation can be read from the error bars of the boxes. Additionally, we added a dotted red line at the value 0.1 to quick verification. The compression methods for which the deviation is in the tolerable range are: FPZIP with bit precision 40 and 48; ZFP with bit precision 32, 40, and 48; and ZFP with a tolerance of 10^{-8} and 10^{-10}. Some methods are disqualified from the first cycle, these are: FPZIP with bit precision 16 and 24; ZFP with bit precision 16; and the conversion into half precision. The conversion into single precision does a good job, only after cycle 13 some of the RMSZ values are outside the allowd interval.

9.3.3.2 Pearson Correlation Coefficient

Second, we look at the Pearson correlation coefficient (Equation 9.7). The coefficient encodes the linear relationship between the compressed and uncompressed state. It can take values between -1 and 1, where 1 corresponds to a perfect linear relationship. Baker et al. [164] requires this value to be at least 0.99999. We list the values of the coefficient per compression method in Table A.1. Strictly speaking, only ZFP with bit precision 48 passes this test. However, if we relax the requirement of Baker et al. slightly, we also may accept ZFP with bit precision 40; ZFP with a tolerance of 10^{-1}; and FPZIP with bit precision 48. All of these are below 0.9999.

9.3.3.3 Normalized Error Statistic

We can evaluate the error probagation looking at the normalized point-wise maximum error (NPME) and normalized root mean squared error (NRMSE):

$$\text{NRMSE}_{X_c} = \frac{\text{RMSE}_{X_c}}{\max\left(x_{c,i}\right) - \min\left(x_{c,i}\right)} \qquad i = 1, 2, \ldots, N \tag{9.13}$$

$$\text{NPME}_{X_c} = \frac{\max\left(\left|x_{c,i} - x_{0,i}\right|\right)}{\max\left(x_{c,i}\right) - \min\left(x_{c,i}\right)} \qquad i = 1, 2, \ldots, N \tag{9.14}$$

We show plots of the two values for all compression parameters in Figure A.5. We can see that the errors increase very quickly in all cases, and that towards the end the curve flattens. Nevertheless, the repeated application of lossy compression leads in all cases to relatively high deviations towards. Besides the plot we also list the values in Table A.1. As we described in subsection 9.3.1, we use the same static seed for the propagation of identical particles. Therefore, the errors that we see in the plots are introduced only by

the compression method. On the other hand, the randomization of the model introduces an error to the states as well, which limits the effective deterioration introduced by the compression. To make the actual impact of the compression method apparent, we repeated the experiments for the 25 particle ensemble, adding a very small additional perturbation of O(10^{-8}) to the states using a random seed. Now, plotting the NRMSE of the various compression methods versus that of the lossless compressed state, gives us a graphical means to decide whether the compression method adds additional error or not. Since the perturbation is indeed random, even the comparison between two identical uncompressed particles differs. However, if the evolution of the NRMSE of a compressed particle is perfectly linear to the one of the lossless compressed states, we can infer that no *additional* error is imposed by the compression method.

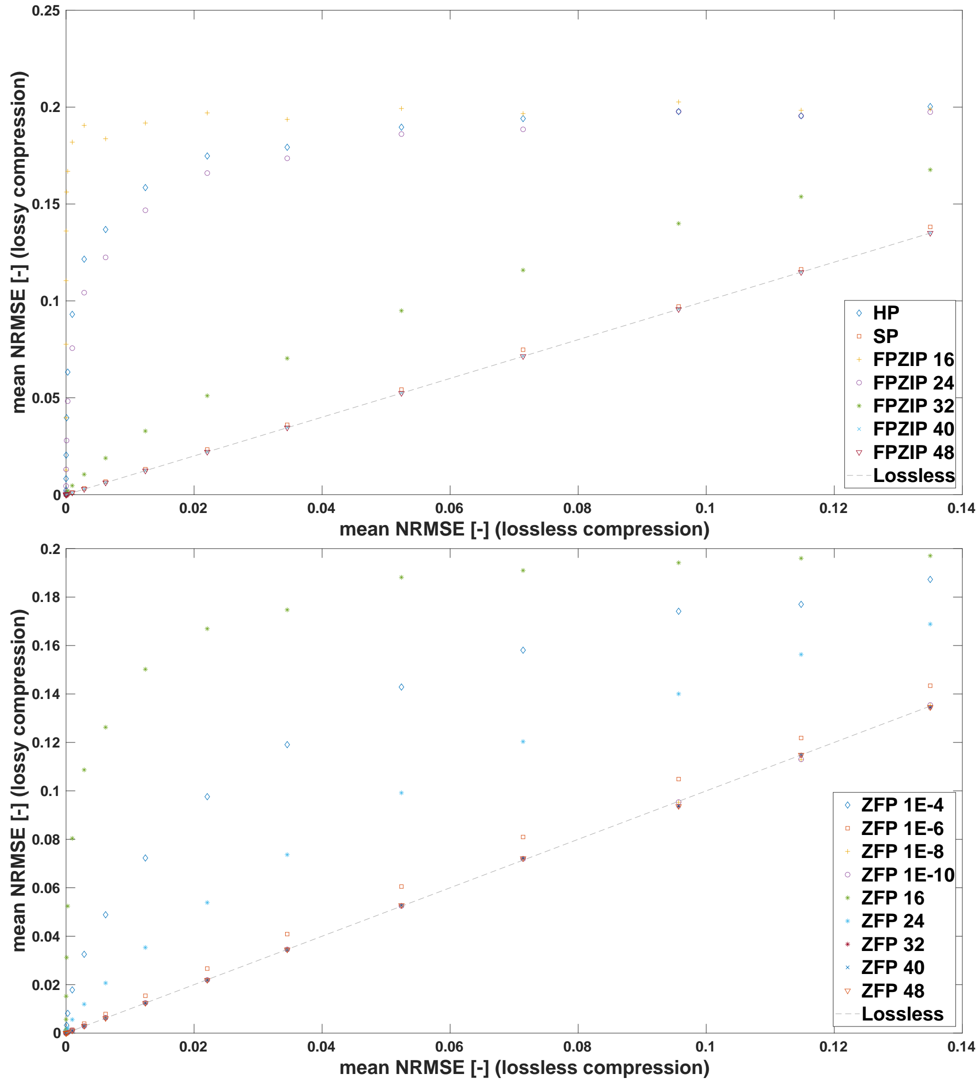


Figure 9.2: Linear correlation between the timely evolution of the NRMSE of compressed to lossy compressed states. We plot the NRMSE of the compressed states for each cycle by the respective values for the lossy compressed state.

Figure 9.2 shows the correlations for all compression methods. We can see that ZFP 32, ZFP 40, ZFP 48, ZFP 1e-8, ZFP 1e-10, FPZIP 40, FPZIP 48 and SP show an almost perfect linear relationship to the uncompressed state. To quantify the correlation and to develop an exclusion criteria, we performed a linear regression of the NRMSE-evolution for all $P(P-1)/2$ combinations of the 25 particles (i.e., $P = 25$), which have been lossless compressed with FPZIP. In that way, we can determine the variation of the linear correlation between identical states. This results into an average correlation of $A = 1.00(01)$,

with A being the slope of the linear regression model $y \sim x$. Thus, we consider the NRMSE test passed, when the value for the linear correlation lies within the error interval of A. On the basis of the second method, the compression parameters that qualify here are: FPZIP with bit precision 40 and 48;ZFP with bit precision 32, 40, and 48; and ZFP with a tolerance of 10^{-8} and 10^{-10} (Table 9.1).

9.3.3.4 Summary of the Validation Study

We can now combine the results of all the qualifiers to identify viable compression parameters for the Lorenz96 model. A detailed view on the values for the NRMSE, NPME, and the Pearsson correlation coefficient is provided in Table A.1. The Z-value deviation is shown in Figure A.1, Figure A.2, Figure A.3, and Figure A.4, resolved by the compression method and assimilation cycle. By matching the results with the requirements, we can extract a subset of viable compression parameters from the initial set. Table 9.1 summarizes the results of the tests for the best parameters after cycle 15. Parameters that pass the tests, indicated by the check-mark, can be safely used for data compression in the Lorenz96 model (in our configuration) within 15 assimilation cycles.

Qualifier	FPZIP 32	FPZIP 40	FPZIP 48	ZFP 32	ZFP 40	ZFP 48
ρ_{X_c}	0.67014	0.98790	0.99995	0.98595	0.99994	1
$\Delta\text{RMSZ}^p_{X_c}$	0.22	0.04	0.0009	0.03	0.001	3.6e-6
A(RMSE)	1.41(05)	1.0007(02)	1.0002(01)	0.9958(22)	0.9964(23)	0.9963(23)
OK	✗	✗	✓	✗	✓	✓
	ZFP 1e-6	**ZFP 1e-8**	**ZFP 1e-10**	**SP**		
ρ_{X_c}	0.85887	0.99574	0.99998	0.91457		
$\Delta\text{RMSZ}^p_{X_c}$	0.21	0.03	0.0005	0.14		
A(RMSE)	1.0855(90)	0.9965(20)	0.9973(20)	1.0232(29)		
OK	✗	✗	✓	✗		

Table 9.1: Summary of the best compression parameters and the exclusion criteria.

9.3.4 Performance

Now that we have outlined how to use the framework and have performed an exemplary analysis to narrow down viable compression parameters, we will take a look at the performance profile of the framework and evaluate the savings that we achieve during the dynamic mode.

9.3.4.1 Validation Mode

The computation of the statistical qualifiers uses a two-way parallelization. The first layer of parallelization is the even distribution of the particles to the available validator instances. In that way each validator computes the qualifiers for about P/V particles, with P being the number of particles, and V the number of validators. The second layer of parallelization is among the cores on each validator node. The climate states are decomposed and the statistical qualifiers are computed in parallel leveraging all available cores. To minimize the number of files that the validators generate, we additionally connect the instances via Transmission Control Protocol (TCP) (Zero Message Queue (ZeroMQ)) and gather all the results on the

master instance. After gathering the qualifiers, the master generates a coma seperated csv file on global storage.

Figure 9.3 shows the profile of one validation cycle on a randomly selected validator instance for the 50 particles and 16 MB experiment. We can see that we spend most of the time to load the particle states (second row in the figure). The calculation of the ensemble mean and standard deviation, require the availability of all the ensemble particle states on the validators. Thus, we need to load in total $(C + 1) \times P$ particles on each validator. In a first implementation, we parallelized the computations for those quantities differently. We computed the ensemble mean and standard deviation partially on each validator, containing only the terms for the particles that have been assigned to the validator. However, The terms have dimension N (see Equation 9.1), and the computation involves the transfer of particle states among the validators, even worse, we need to provide the ensemble mean on all validators (allreduce) for computing the ensemble standard deviation. It turns out that those transfers add a considerable amount of overhead, and the current implementation without state exchanges between validators is considerably faster, despite the overhead of the additional particle loads.

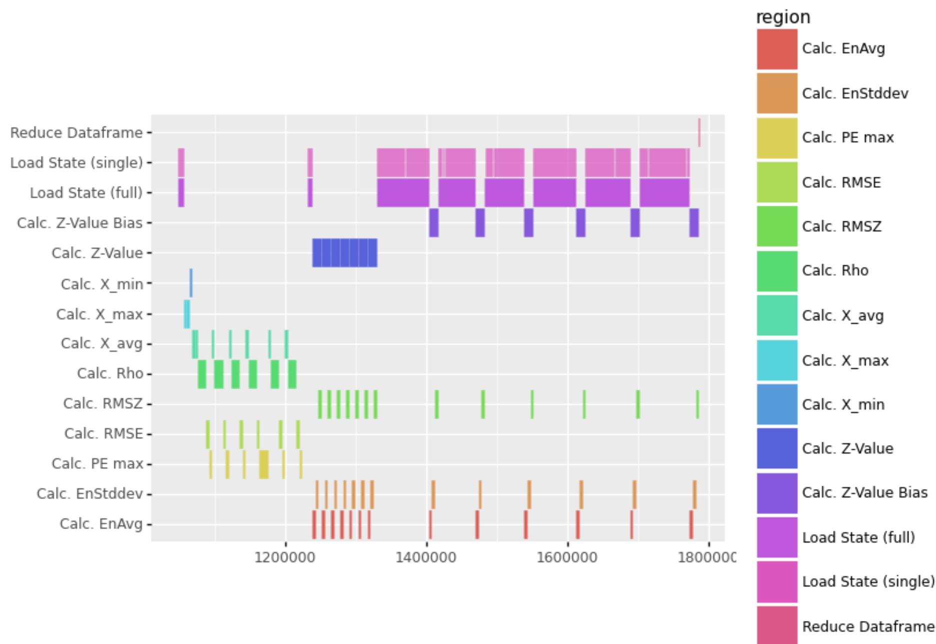


Figure 9.3: Trace of randomly selected validator for one validation cycle

9.3.4.2 Dynamic mode

In subsection 9.2.3, we identified viable compression parameters, studying the results from the validation mode. We executed the climate model in dynamic mode using the selected parameters. The parameter that was selected in dynamic mode was ZFP with bit precision 40 (highest compression rate). We ran the lorenz96 model data assimilation with a state size of 2 GB. We measured a speedup of 19% for loading and 57% for storing the state. The compression rate has been the same as it was for the smaller state sizes (1.534). Therefore, we achieved a reduction of 35% in storage space. We have applied the validation function before we applied the compression to the particle state. We achieved an effective speedup of

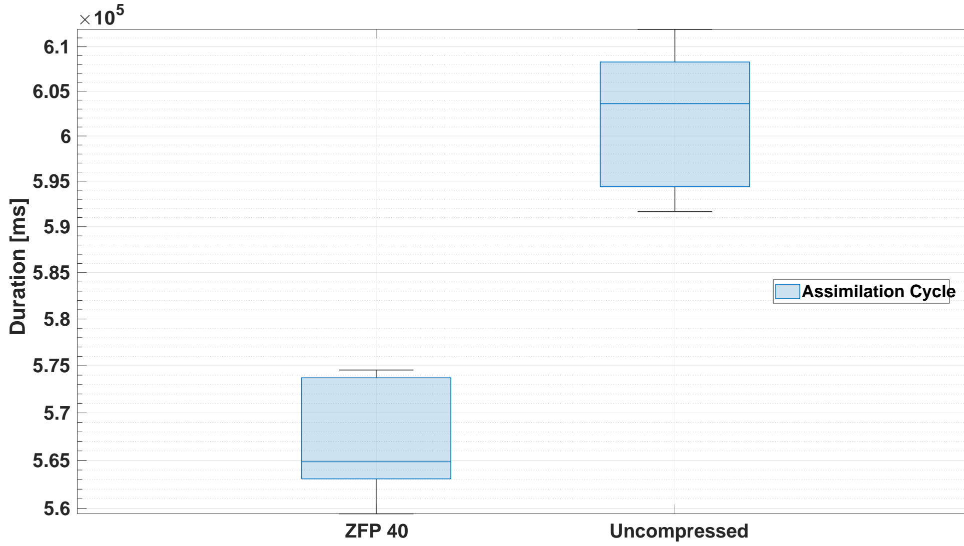


Figure 9.4: Comparison of the time for one assimilytion cycle leveraging the dynamic mode of our proposed framework.

6%, including the time for the compression, decompression, and application of the validation function (Figure 9.4).

In addition to the parameters that we identified during the analysis in the last section, we also measured the speedup and compression rates for other compression parameters. Table 9.3 lists the IO performance that we measured on the runners while executing the climate model in dynamic mode. The times contain the compression and IO operation. We can see that we save most while storing the states, loading the states shows a significantly smaller speedup. We define the speedup is by $\Delta T_{\text{SU}} = \frac{T_c - T_0}{T_0}$. The maximum speedup is achieved with ZFP and FPZIP at 32 bit precision and with ZFP 10^{-6} accuracy. The corresponding speedup is 39%, 43% and 44% respectively for the store, and 19%, 19% and 18% for the load operation. The parameters which have passed all tests also show considerable speedup for storing and loading the states. Even the most accurate methods show a reasonable speedup of 21% and 22% for storing and 15% and 12% for loading the states. We further observe that the speedup increases with larger state sizes. Moreover, we can see in Table 9.2, that the compression rate does not change with an increasing state size. This means that we can expect even better speedups and equal reduction rates for larger state sizes.

Compression Rate								
Cycle	**FPZIP 32**	**FPZIP 40**	**FPZIP 48**	**ZFP 32**	**ZFP 40**	**ZFP 48**	**HP**	**SP**
16	2.332125	1.805610	1.473040	1.897811	1.533923	1.287127	4	2
32	2.332534	1.805876	1.473223	1.897829	1.533934	1.287135	4	2
64	2.332749	1.806008	1.473311	1.897833	1.533936	1.287137	4	2
128	2.332955	1.806137	1.473397	1.897827	1.533933	1.287135	4	2
	ZFP 1e-6	**ZFP 1e-8**	**ZFP 1e-10**					
16	2.240469	1.799499	1.503566					
32	2.240486	1.799508	1.503571					
64	2.240469	1.799498	1.503564					
128	2.240469	1.799497	1.503564					

Table 9.2: Compression rates, CR_c, for selected compression parameters, ordered by the state size.

9.3.5 Discussion

Our analysis is performed on the Lorenz96 model, which has only one state variable. Baker et al. [164] evaluates lossy compression for four different variables of the CESM. The evaluation shows different behavior for all variables in compression rate and variable consistency. For instance, compression with FPZIP-24 leads to compression rates between 2.56 and 5.26 and NRMSE values between 1.8e-5 and 6.5e-7. Further, the ensemble consistency varies significantly among the variables for different compression methods. This demonstrates that we need to allow for different compression parameters per variable and that each variable achieves a different compression rate. That is to say, our evaluation does not give a general statement for the performance of the compression parameters. Moreover, it underlines the importance of studying the impact of lossy compression on consistency for each model and its variables separately.

State Size (MB)	Orig.	FPZIP 32	FPZIP 40	FPZIP 48	ZFP 32	ZFP 40	ZFP 48	ZFP 1e-6	ZFP 1e-8	ZFP 1e-10
Load State from PFS (median) [ms]										
16	877.1	776.8	769.6	779.0	805.3	820.3	824.3	-	-	-
32	933.8	793.7	848.2	860.3	872.2	907.5	926.7	810.1	854.6	871.3
64	989.6	957.5	1011.9	1052.4	845.0	892.4	897.7	877.6	945.7	948.2
128	1185.0	954.5	1030.5	1119.2	957.9	995.7	1011.9	977.3	1045.6	1044.4
Speedup Load [%]										
16	-	11.4	12.3	11.2	8.2	6.5	6.0	-	-	-
32	-	15.0	9.2	7.9	6.6	2.8	0.8	13.2	8.5	6.7
64	-	3.2	2.3	6.3	14.6	9.8	9.3	11.3	4.4	4.2
128	-	19.4	13.0	5.6	19.2	16.0	14.6	17.5	11.8	11.9
Store State to PFS (median) [ms]										
16	524.9	479.6	481.2	491.2	500.0	501.3	516.3	-	-	-
32	651.0	513.7	527.4	567.9	524.7	540.0	593.2	514.3	531.6	588.2
64	761.0	585.4	599.2	659.0	461.1	498.4	534.9	582.6	590.0	653.3
128	1215.5	744.8	966.7	1132.2	698.2	869.9	956.4	676.9	833.3	943.8
Speedup Store [%]										
16	-	8.6	8.3	6.4	4.7	4.5	1.6	-	-	-
32	-	21.1	19.0	12.8	19.4	17.1	8.9	21.0	18.3	9.6
64	-	23.1	21.3	13.4	39.4	34.5	29.7	23.4	22.5	14.2
128	-	38.7	20.5	6.9	42.6	28.4	21.3	44.3	31.4	22.4

Table 9.3: Speedup for the various compression parameters while storing and loading the states fomr the PFS.

9.4 Related Work

Compression of scientific datasets is not only interesting for numerical climate science. Every field in HPC that deals with large datasets benefits from reduced data sizes. Data compression can be applied, for instance, to datasets before visualization and to generate checkpoints. A variety of compression algorithms are used in HPC: ZFP [17], FPZIP [18], ISABELA [19], SZ [20, 175], MGARD [21] and

MGARD+ [22], to name a few. IO libraies such as ADIOS [176], HDF5 [172], and NetCDF4 [177] offer high-level interfaces for compression of datasets in self-descriptive hierarchical files.

The impact and applicability of data compression to scientific datasets has been studied in several works [156, 164–166, 177, 178]. The Community Earth System Model (CESM) [45, 179] includes a Port-Validation tool, originally used to determine the consistency of the results after porting to a different architecture. According to Baker et al. [164], the tool can also be used to validate data compression for the CESM module states. Z-Checker [180] is a framework that can be used to analyze the impact of compression to any scientific dataset. The framework offers offline and online analysis of the datasets. The online mode can be used after instrumenting the code with the Z-Checker API functions. The online mode can be used inside the application to observe the dynamic behavior of data compression.

Our proposed framework is similar to the port validation tool in the CESM, as the tool checks ensemble consistency and can detect issues in the climate model after porting it to a new machine. However, our framework (1) is not constrainted to a certain climate modelling system, (2) is more flexible as it enables definition of custom validation functions and (3) provides automatic and direct comparison between ensembles that use different compression methods. Z-Checker provides several features to evaluate the impact of the compression method on the data, and the online mode can potentially be used to perform an analysis that is similar to ours. However, this would be associated with considerable implementation efforts, and the tool does not provide measures to detect ensemble inconsistencies.

9.5 Conclusion

In this work, we present a novel framework, build on top of the MelissaDA architecture, that provides validation and application of lossy compression in climate models for ensemble data assimilation. We conducted and presented an exemplary study based on the Loren96 model, where we evaluated the applicability of the FPZIP and ZFP 16, 24, 32, 40 and 48 bit precision modes, the ZFP 1e-4, 1e-6, 1e-8, 1e-10 accuracy modes, and single and half precision floating point representations for data compression. Our validation follows the suggestion of Baker et al. [164], requiring the deviation of the Z-Values of compressed and uncompressed states to be less than 0.1 (Chapter 9.12), the pearson correlation coefficient (Chapter 9.7) of the compressed and the uncompressed state to be at least 0.9999 (Baker et al. is more restrictive, requiring at least 0.99999), and the impact on the normalized root mean squared error of compressed and uncompressed state to be negligible (compare Chapter 9.3.3.3). After matching our results with this metric, we remain with FPZIP 48, ZFP 40 and 48 and ZFP 1e-10. According to this, those parameters can safely be applied for the compression of the states of the Lorenz96 model in our configuration for 15 assimilation cycles, without affecting the data assimilation result. The compression rates for those parameters are 1.47, 1.53 and 1.5 respectively, which translates to a saving of 1/3 in storage space. Our measurements during the dynamic mode and a state size of 2GB, show speedups of 19% while loading and 57% while storing the states. We also check the integrity of the states applying the default validation function. These results into an effective speedup of 6% for the full assimilation cycle.

Part IV

Epilogue

Chapter 10

Thesis Conclusion

The contributions that we have made during this doctorate are of two kinds: (i) Advancing state-of-the-art checkpointing techniques, and (ii) Introducing fault-tolerance into ensemble methods. The contributions of the first kind are very general and can be applied to practically every type of HPC application. The contributions of the second kind are applicable to certain kinds of ensemble methods. We have defined the ensembles that we have focused on as numerical systems that are only coupled in their outcome for future iterations (output dependency). Other than that, they can be executed independently of each other. The contributions (ii) can be applied to every ensemble method that follows this definition. Specific examples of those methods can be found in ensemble learning (bagging, random subspace, stacking, etc.) [181], digital twins [182], and numerical weather prediction. In addition to the definition above, systems that benefit from our contributions are typically large scale; many ensemble members and large ensemble states. The contributions are presented in this thesis according to the separation above. Contributions of kind (i) are presented in part II of this thesis and contributions of kind (ii) in part III.

1st Contribution (chapter 5) We designed and implemented a novel Differential Checkpointing (DCP) implementation that relies on hashing the data variables itself, instead of utilizing the page protection mechanism of the Operating System (OS). Our method eliminates the overhead introduced during runtime, resulting from the OS signalling, and removes the restriction to block sizes that must be equal to the memory page size. We observed a reduction of the checkpoint overhead linear to the dirty data ratio of up to almost 80% if writing 3% of the data. We further developed a model that allows to predict the overhead reduction, after measuring the time to write and the time to hash a block of a certain size.

2nd Contribution (chapter 6) We designed and implemented an API and runtime, that allows automatic elastic recovery, and eliminates the separation of IO for resiliency and for scientific data. The runtime provides the asynchronous creation of shared HDF5 files on global storage. We implemented our proposal into the API of FTI to provide both, access to a number of modern checkpointing features (local checkpointing, partner checkpointing, encoded checkpointing, differential checkpointing, etc.), and the output of structured scientific data into a checkpoint file that is accessible with 3rd party tools. Furthermore, the API allows to perform the application restart with an arbitrary number of processes (elastic recovery). We compared the performance of our asynchronous IO to a similar feature provided by Adaptable Input Output System (ADIOS), and we demonstrated that our runtime is 5 times faster.

3rd Contribution (chapter 7) We implemented Checkpoint-Restart (CR) into the MelissaDA framework. MelissaDA is an elastic large-scale ensemble data assimilation framework, offering a number of ensemble methods for data assimilation in climate modelling. The framework shows high parallel efficiency and excellent fault tolerance properties. Our CR implementation manages to (i) hide the checkpoint

overhead entirely, to (ii) recover with only very few or no recomputation, (iii) provide elastic recovery for the server, and (iv) store the scientific output into globally shared HDF5 files. Our implementation uses the API from the 2nd contribution to provide elastic recovery, and to provide the data in HDF5.

4th Contribution (chapter 8) We designed and implemented an efficient and fault-tolerant PF based on the MelissaDA architecture. We exploit certain properties of particle filters that allow us to minimize the transfers between nodes and the global storage layer, and to achieve a high resource utilization. We equip the framework with a distributed particle cache, deployed on node-local storage to further increase parallel efficiency and to mitigate fault-tolerance. The framework shows a resource utilization of 96% for a particle filter with 945 particles and a state size of 2.5 GB.

5th Contribution (chapter 9) We designed and implemented a framework for automatic validation and adaptive application of lossy compression for climate states in ensemble data assimilation. The framework addresses the issues arising from the large amount of data afflicted with ensemble data assimilation. On the other hand, ensemble states in climate science are typically very sensitive to even small perturbations. Therefore, the application of lossy compression needs to be applied very carefully. Our framework provides two modes of operation: the (i) **validation mode**, testing the impact of lossy compression parameters on ensemble and ensemble state, and the (ii) **dynamic mode:**, applying viable parameters during the data assimilation process. The compression parameters can be applied to each variable of the ensemble state independently, as different variables often tolerate a different degree of deterioration. We provide an exemplary study using the Lorenz96 climate model, where we identify viable compression parameters and achieve speedups of up to 28% during the dynamic mode after applying the parameters to the climate state, before storing it to the file system.

Chapter 11

List of Publications

- Keller, K., Kestelman, A. C., Bautista-Gomez, L. (2021, December). Towards Zero-Waste Recovery and Zero-Overhead Checkpointing in Ensemble Data Assimilation. In 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) (pp. 131-140). IEEE.
- Keller, K., Parasyris, K., Bautista-Gomez, L. (2020, December). Design and Study of Elastic Recovery in HPC Applications. In 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC) (pp. 261-270). IEEE.
- Maronas, M., Mateo, S., Keller, K., Bautista-Gomez, L., Ayguadé, E., Beltran, V. (2020). Extending the openchk model with advanced checkpoint features. Future Generation Computer Systems, 112, 738-750.
- Parasyris, K., Keller, K., Bautista-Gomez, L., Unsal, O. (2020, May). Checkpoint restart support for heterogeneous hpc applications. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID) (pp. 242-251). IEEE.
- Keller, K., Bautista-Gomez, L. (2019, May). Application-level differential checkpointing for HPC applications with dynamic datasets. In 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (pp. 52-61). IEEE.
- Losada, N., Bautista-Gomez, L., Keller, K., Unsal, O. (2018, November). Towards Ad Hoc Recovery For Soft Errors. In 2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS) (pp. 1-10). IEEE.
- Gomez, L. B., Keller, K., Unsal, O. (2018, June). Performance study of non-volatile memories on a high-end supercomputer. In International Conference on High Performance Computing (pp. 145-156). Springer, Cham.

Part V

Appendix

Appendix A

Validation Framework - Figures and Tables

NRMSE								
Cycle	**FPZIP 32**	**FPZIP 40**	**FPZIP 48**	**ZFP 32**	**ZFP 40**	**ZFP 48**	**HP**	**SP**
1	2.12e-07	8.32e-10	3.25e-12	1.11e-09	4.34e-12	1.69e-14	1.16e-04	1.42e-08
3	4.88e-06	1.87e-08	6.90e-11	2.34e-08	1.02e-10	3.91e-13	2.47e-03	3.31e-07
5	1.25e-04	5.29e-07	2.23e-09	6.47e-07	3.16e-09	1.35e-11	2.10e-02	8.88e-06
7	1.88e-03	1.17e-05	5.36e-08	1.31e-05	6.32e-08	2.21e-10	6.58e-02	2.09e-04
9	1.04e-02	1.99e-04	8.44e-07	2.39e-04	1.10e-06	5.75e-09	1.20e-01	1.90e-03
11	3.32e-02	1.68e-03	2.20e-05	1.97e-03	4.64e-05	3.38e-07	1.60e-01	8.83e-03
13	7.27e-02	7.45e-03	2.09e-04	8.19e-03	2.26e-04	1.42e-06	1.84e-01	2.73e-02
15	1.18e-01	2.25e-02	1.36e-03	2.44e-02	1.55e-03	2.00e-05	1.94e-01	6.02e-02
	ZFP 1e-6	**ZFP 1e-8**	**ZFP 1e-10**					
1	3.35e-08	2.59e-10	2.03e-12					
3	8.02e-07	5.82e-09	4.39e-11					
5	2.19e-05	1.55e-07	1.21e-09					
7	4.58e-04	3.39e-06	2.57e-08					
9	3.37e-03	6.34e-05	5.36e-07					
11	1.37e-02	7.48e-04	1.04e-05					
13	3.82e-02	3.93e-03	1.42e-04					
15	7.71e-02	1.34e-02	9.56e-04					
NPME								
Cycle	**FPZIP 32**	**FPZIP 40**	**FPZIP 48**	**ZFP 32**	**ZFP 40**	**ZFP 48**	**HP**	**SP**
1	7.68e-06	2.34e-08	1.33e-10	4.74e-08	3.44e-10	1.05e-12	6.56e-03	5.26e-07
3	1.38e-03	4.38e-06	1.52e-08	3.81e-06	2.66e-08	9.60e-11	4.11e-01	1.02e-04
5	4.86e-02	2.34e-04	1.11e-06	3.01e-04	1.68e-06	8.27e-09	7.35e-01	3.80e-03
7	4.77e-01	6.16e-03	3.15e-05	5.90e-03	3.42e-05	1.10e-07	7.72e-01	1.17e-01
9	6.76e-01	1.04e-01	5.05e-04	1.31e-01	6.60e-04	4.09e-06	8.00e-01	4.97e-01
11	7.47e-01	5.05e-01	1.73e-02	5.19e-01	3.01e-02	2.35e-04	8.24e-01	6.47e-01
13	8.00e-01	6.35e-01	1.27e-01	6.59e-01	1.31e-01	1.04e-03	8.08e-01	7.38e-01
15	7.96e-01	7.24e-01	4.59e-01	7.15e-01	4.57e-01	1.29e-02	8.05e-01	7.79e-01
	ZFP 1e-6	**ZFP 1e-8**	**ZFP 1e-10**					
1	1.54e-06	7.20e-09	7.62e-11					
3	2.52e-04	1.88e-06	8.73e-09					

5	1.06e-02	6.61e-05	5.37e-07					
7	2.25e-01	1.77e-03	1.24e-05					
9	5.64e-01	3.68e-02	3.10e-04					
11	6.70e-01	3.59e-01	7.01e-03					
13	7.53e-01	5.80e-01	9.36e-02					
15	7.80e-01	6.83e-01	3.73e-01					
Pearson Correlation Coefficient								
Cycle	**FPZIP 32**	**FPZIP 40**	**FPZIP 48**	**ZFP 32**	**ZFP 40**	**ZFP 48**	**HP**	**SP**
1	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	0.99986	1
5	1	1	1	1	1	1	0.98953	1
7	0.99992	1	1	1	1	1	0.89889	1
9	0.99750	1	1	1	1	1	0.66111	0.99991
11	0.97405	0.99993	1	0.99991	1	1	0.38879	0.99815
13	0.87646	0.99870	1	0.99839	1	1	0.20570	0.98261
15	0.67014	0.98790	0.99995	0.98595	0.99994	1	0.11110	0.91457
	ZFP 1e-6	**ZFP 1e-8**	**ZFP 1e-10**					
1	1	1	1					
3	1	1	1					
5	1	1	1					
7	0.99999	1	1					
9	0.99974	1	1					
11	0.99550	0.99999	1					
13	0.96534	0.99963	1					
15	0.85887	0.99574	0.99998					

Table A.1: Average values of the statistical qualifiers NRMSE, NPME and ρ_{X_c} for selected compression parameters. The rows show the evolution of the qualifiers by assimilation cycles.

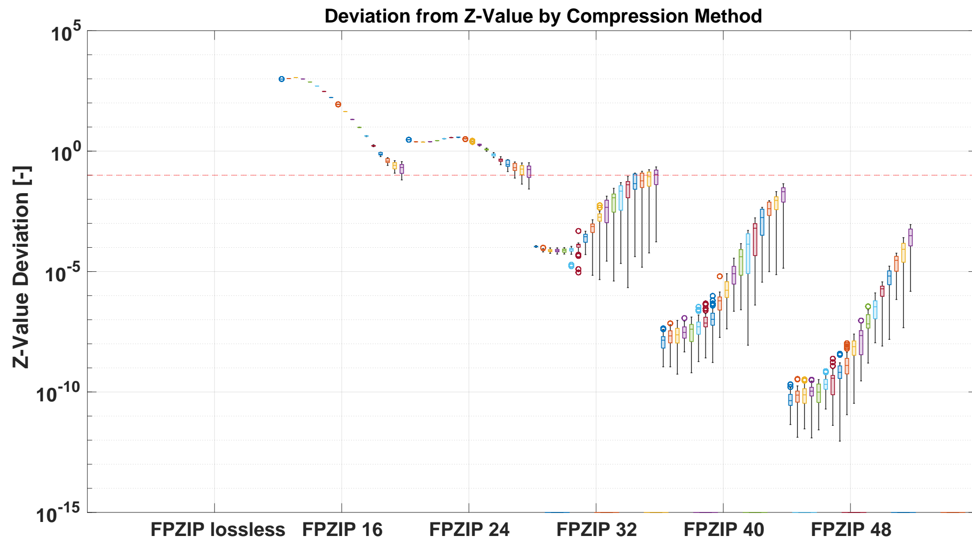


Figure A.1: Z-Value deviation, $\Delta\mathrm{RMSZ}^{p}_{X_c}$ (Equation 9.11) for FPZIP. The colors indicate values at different cycles.

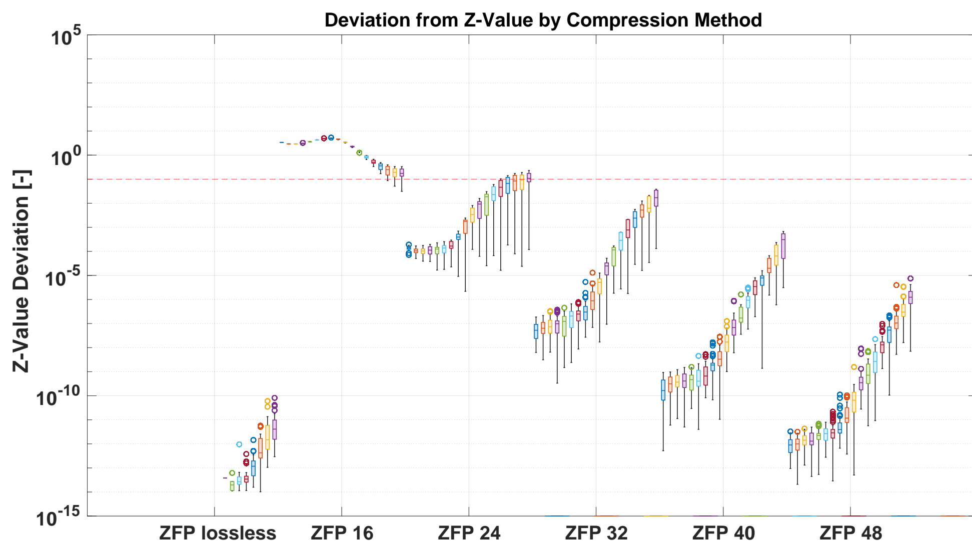


Figure A.2: Z-Value deviation, $\Delta\mathrm{RMSZ}^{p}_{X_c}$ (Equation 9.11) for ZFP in precision mode. The colors indicate values at different cycles.

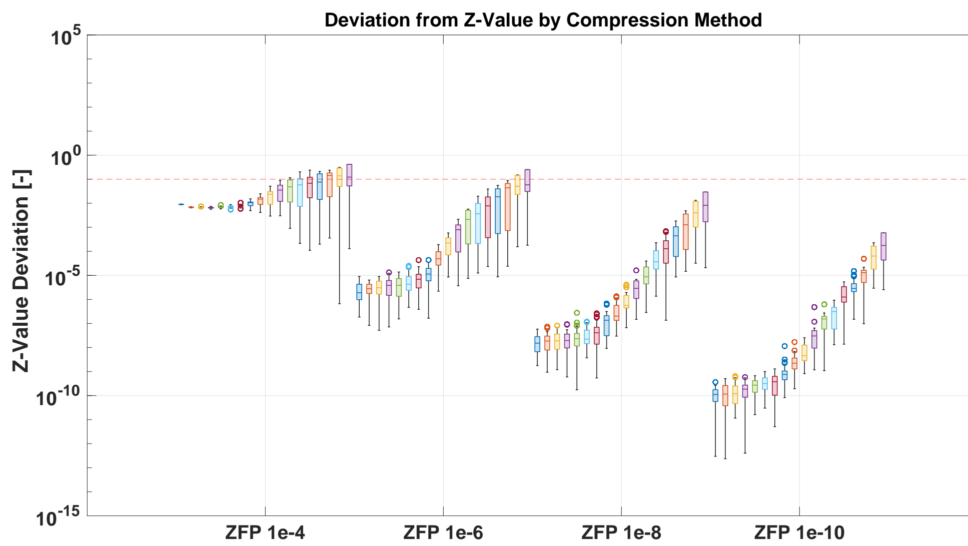


Figure A.3: Z-Value deviation, $\Delta \text{RMSZ}^{p}_{X_c}$ (Equation 9.11) for ZFP in accuracy mode. The colors indicate values at different cycles.

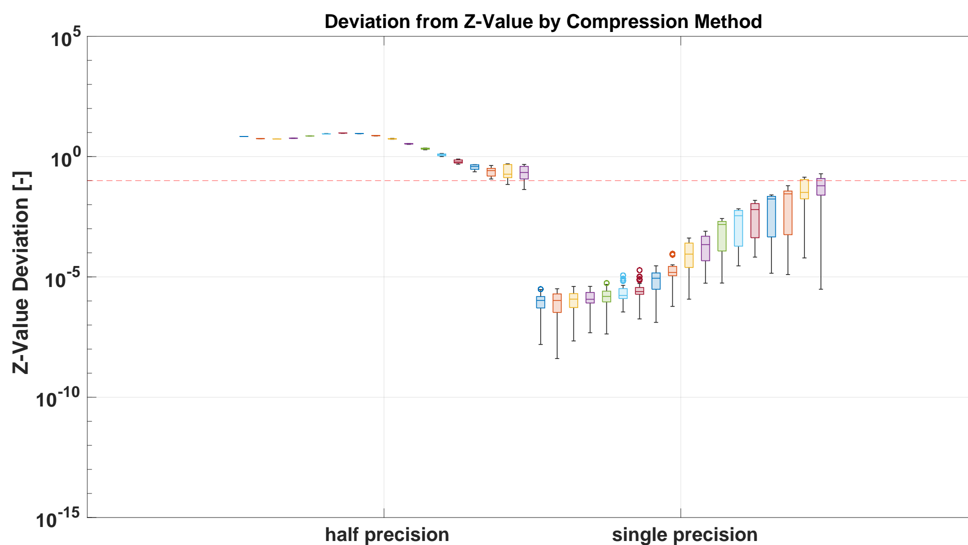


Figure A.4: Z-Value deviation, $\Delta \text{RMSZ}^{p}_{X_c}$ (Equation 9.11) for half and single precision. The colors indicate values at different cycles.

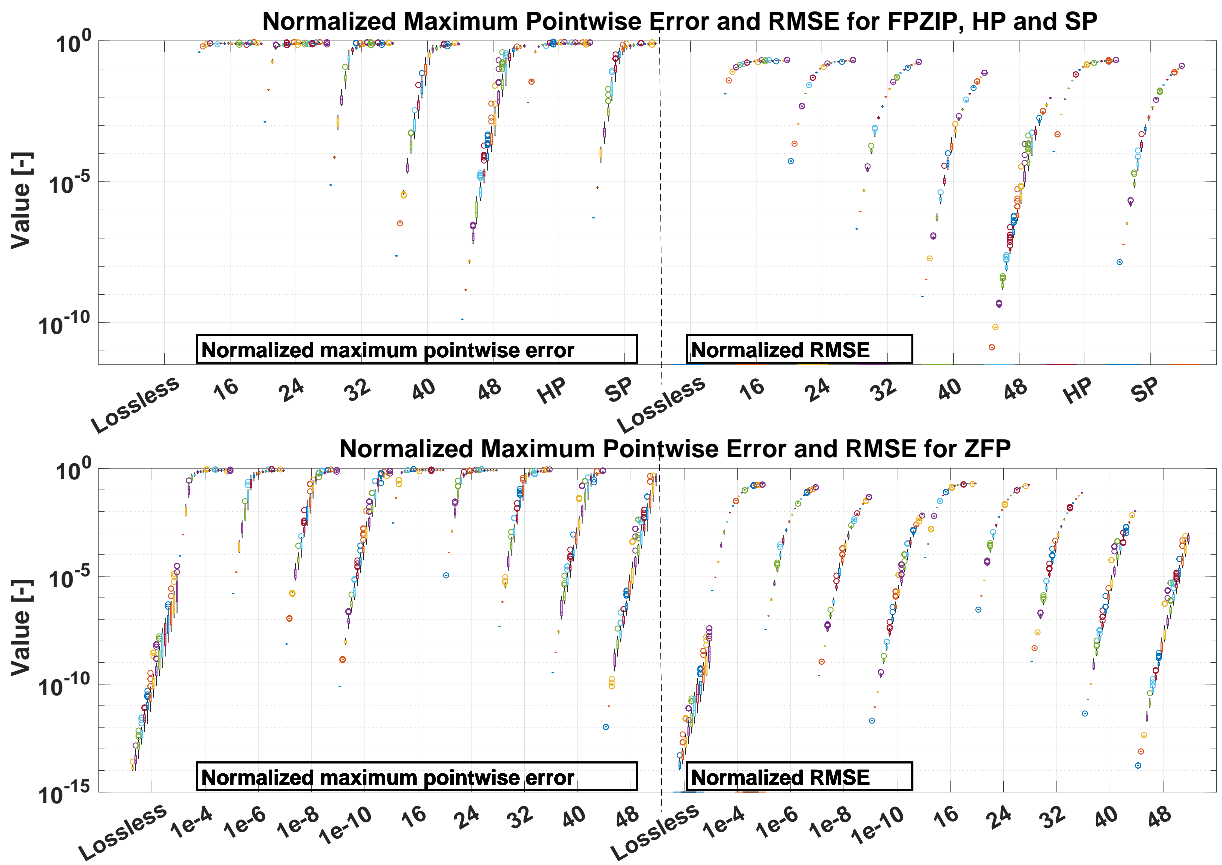


Figure A.5: Normalized maximum pointwise error and normalized root mean square error for (a) FPZIP, half and single precision, and (b) ZFP in accuracy and precision modes. The colors indicate values at different cycles.

Bibliography

[1] Robert Lucas, James Ang, Keren Bergman, Shekhar Borkar, William Carlson, Laura Carrington, George Chiu, Robert Colwell, William Dally, Jack Dongarra, Al Geist, Rud Haring, Jeffrey Hittinger, Adolfy Hoisie, Dean Micron Klein, Peter Kogge, Richard Lethin, Vivek Sarkar, Robert Schreiber, John Shalf, Thomas Sterling, Rick Stevens, Jon Bashor, Ron Brightwell, Paul Coteus, Erik Debenedictus, Jon Hiller, K. H. Kim, Harper Langston, Richard Micron Murphy, Clayton Webster, Stefan Wild, Gary Grider, Rob Ross, Sven Leyffer, and James Laros III. DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) Report: Top Ten Exascale Research Challenges. Technical report, USDOE Office of Science (SC) (United States), February 2014.

[2] Z. Miao, J. Calhoun, and R. Ge. Energy analysis and optimization for resilient scalable linear systems. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 24–34, 2018.

[3] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. Resilience-aware resource management for exascale computing systems. *IEEE Transactions on Sustainable Computing*, 3(4):332–345, 2018.

[4] C. McNairy. Exascale fault tolerance challenge and approaches. In *2018 IEEE International Reliability Physics Symposium (IRPS)*, pages 3C.4–1–3C.4–10, 2018.

[5] Saurabh Hukerikar and Christian Engelmann. Resilience design patterns: A structured approach to resilience at extreme scale. *arXiv preprint arXiv:1708.07422*, 2017.

[6] Lili Ju, Wei Leng, Zhu Wang, and Shuai Yuan. Numerical investigation of ensemble methods with block iterative solvers for evolution problems. *Discrete and Continuous Dynamical Systems - B*, 25(12):4905, 2020. Company: Discrete and Continuous Dynamical Systems - B Distributor: Discrete and Continuous Dynamical Systems - B Institution: Discrete and Continuous Dynamical Systems - B Label: Discrete and Continuous Dynamical Systems - B Publisher: American Institute of Mathematical Sciences.

[7] Brian Ancell and Gregory J. Hakim. Comparing Adjoint- and Ensemble-Sensitivity Analysis with Applications to Observation Targeting. *Monthly Weather Review*, 135(12):4117–4134, December 2007. Publisher: American Meteorological Society Section: Monthly Weather Review.

[8] Ryan D. Torn and Gregory J. Hakim. Ensemble-Based Sensitivity Analysis. *Monthly Weather Review*, 136(2):663–677, February 2008. Publisher: American Meteorological Society Section: Monthly Weather Review.

[9] M. S. Cao, L. X. Pan, Y. F. Gao, D. Novák, Z. C. Ding, D. Lehký, and X. L. Li. Neural network ensemble-based parameter sensitivity analysis in civil engineering systems. *Neural Computing and Applications*, 28(7):1583–1590, July 2017.

[10] Jinjun Ren, Yuping Wang, Mingqian Mao, and Yiu-ming Cheung. Equalization ensemble for large scale highly imbalanced data classification. *Knowledge-Based Systems*, 242:108295, April 2022.

[11] Omer Sagi and Lior Rokach. Ensemble learning: A survey. *WIREs Data Mining and Knowledge Discovery*, 8(4):e1249, 2018. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1249.

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

[13] Peter Jan van Leeuwen. Particle Filtering in Geophysical Systems. *Monthly Weather Review*, 137(12):4089–4114, December 2009. Publisher: American Meteorological Society Section: Monthly Weather Review.

[14] 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.

[15] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Nov 2010.

[16] Franck Cappello, Kathryn Mohror, and Bogdan Nicolae. Overview — veloc documentation, 2019. Available at `https://veloc.readthedocs.io/en/latest/`.

[17] Peter Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, December 2014. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[18] Peter Lindstrom. FPZIP, 2017. Language: en.

[19] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. Compressing the incompressible with ISABELA: in-situ reduction of spatio-temporal data. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 366–379, Berlin, Heidelberg, August 2011. Springer-Verlag.

[20] Sheng Di and Franck Cappello. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 730–739, May 2016. ISSN: 1530-2075.

[21] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Computing and Visualization in Science*, 19(5):65–76, 2018.

[22] Xin Liang, Ben Whitney, Jieyang Chen, Lipeng Wan, Qing Liu, Dingwen Tao, James Kress, Dave Pugmire, Matthew Wolf, Norbert Podhorszki, and Scott Klasky. MGARD+: Optimizing Multilevel Methods for Error-bounded Scientific Data Reduction. *arXiv:2010.05872 [cs]*, November 2020. arXiv: 2010.05872.

[23] Edward N Lorenz. Deterministic nonperiodic flow. *Journal of atmospheric sciences*, 20(2):130–141, 1963.

[24] R. N. Bannister. A review of operational methods of variational and ensemble-variational data assimilation. *Quarterly Journal of the Royal Meteorological Society*, 143(703):607–633, 2017. _eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.2982.

[25] H. Ngodock, I. Souopgui, M. Carrier, S. Smith, J. Osborne, and J. D'Addezio. An ensemble of perturbed analyses to approximate the analysis error covariance in 4dvar. *Tellus A: Dynamic Meteorology and Oceanography*, 72(1):1–12, January 2020. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/16000870.2020.1771069.

[26] Ganesh Gopalakrishnan, Ibrahim Hoteit, Bruce D. Cornuelle, and Daniel L. Rudnick. Comparison of 4DVAR and EnKF state estimates and forecasts in the Gulf of Mexico. *Quarterly Journal of the Royal Meteorological Society*, 145(721):1354–1376, 2019. _eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.3493.

[27] Andrew Lorenc. Relative merits of 4d-var and ensemble kalman filter. Technical report, NWP Internal Report, 2003.

[28] Andrew C. Lorenc. The potential of the ensemble Kalman filter for NWP—a comparison with 4D-Var. *Quarterly Journal of the Royal Meteorological Society*, 129(595):3183–3203, 2003. _eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1256/qj.02.132.

[29] D. M. Barker, W. Huang, Y.-R. Guo, A. J. Bourgeois, and Q. N. Xiao. A Three-Dimensional Variational Data Assimilation System for MM5: Implementation and Initial Results. *Monthly Weather Review*, 132(4):897–914, April 2004. Publisher: American Meteorological Society Section: Monthly Weather Review.

[30] Kayo Ide, Philippe Courtier, Michael Ghil, and Andrew C. Lorenc. Unified Notation for Data Assimilation : Operational, Sequential and Variational (gtSpecial IssueltData Assimilation in Meteology and Oceanography: Theory and Practice). *Journal of the Meteorological Society of Japan. Ser. II*, 75(1B):181–189, 1997.

[31] 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. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/94JC00572.

[32] Geir Evensen. Using the extended Kalman filter with a multilayer quasi-geostrophic ocean model. *Journal of Geophysical Research: Oceans*, 97(C11):17905–17924, 1992. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/92JC01972.

[33] Geir Evensen. Open boundary conditions for the extended kalman filter with a quasi-geostrophic ocean model. *J. Geophys. Res*, 98:16–529, 1993.

[34] Peter Jan van Leeuwen, Hans R. Künsch, Lars Nerger, Roland Potthast, and Sebastian Reich. Particle filters for high-dimensional geoscience applications: A review. *Quarterly Journal of the Royal Meteorological Society*, 145(723):2335–2365, 2019. _eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.3551.

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

[36] Melanie Ades and Peter Jan Van Leeuwen. An exploration of the equivalent weights particle filter. *Quarterly Journal of the Royal Meteorological Society*, 139(672):820–840, 2013.

[37] Peter Jan van Leeuwen. Nonlinear data assimilation in geosciences: an extremely efficient particle filter. *Quarterly Journal of the Royal Meteorological Society*, 136(653):1991–1999, 2010.

[38] DOE/NNSA, Lab announce partnership with Cray to develop NNSA's first exascale supercomputer.

[39] Guillaume Aupy. *Resilient and energy-efficient scheduling algorithms at scale*. PhD thesis, École Normale Supérieure de Lyon, 2014.

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

[41] Andrew J. Majda and Xin T. Tong. Performance of Ensemble Kalman filters in large dimensions. *arXiv:1606.09321 [math, stat]*, May 2017. arXiv: 1606.09321.

[42] P. Sakov, F. Counillon, L. Bertino, K. A. Lisæter, P. R. Oke, and A. Korablev. TOPAZ4: an ocean-sea ice data assimilation system for the North Atlantic and Arctic. *Ocean Science*, 8(4):633–656, August 2012. Publisher: Copernicus GmbH.

[43] Laurent Bertino and Jiping Xie. Operational Forecasting of Sea Ice in the Arctic Using TOPAZ System. In Ola M. Johannessen, Leonid P. Bobylev, Elena V. Shalina, and Stein Sandven, editors, *Sea Ice in the Arctic: Past, Present and Future*, pages 389–397. Springer International Publishing, Cham, 2020.

[44] James W. Hurrell, M. M. Holland, P. R. Gent, S. Ghan, Jennifer E. Kay, P. J. Kushner, J.-F. Lamarque, W. G. Large, D. Lawrence, K. Lindsay, W. H. Lipscomb, M. C. Long, N. Mahowald, D. R. Marsh, R. B. Neale, P. Rasch, S. Vavrus, M. Vertenstein, D. Bader, W. D. Collins, J. J. Hack, J. Kiehl, and S. Marshall. The Community Earth System Model: A Framework for Collaborative Research. *Bulletin of the American Meteorological Society*, 94(9):1339–1360, 09 2013.

[45] G. Danabasoglu, J.-F. Lamarque, J. Bacmeister, D. A. Bailey, A. K. DuVivier, J. Edwards, L. K. Emmons, J. Fasullo, R. Garcia, A. Gettelman, C. Hannay, M. M. Holland, W. G. Large, P. H. Lauritzen, D. M. Lawrence, J. T. M. Lenaerts, K. Lindsay, W. H. Lipscomb, M. J. Mills,

R. Neale, K. W. Oleson, B. Otto-Bliesner, A. S. Phillips, W. Sacks, S. Tilmes, L. van Kampenhout, M. Vertenstein, A. Bertini, J. Dennis, C. Deser, C. Fischer, B. Fox-Kemper, J. E. Kay, D. Kinnison, P. J. Kushner, V. E. Larson, M. C. Long, S. Mickelson, J. K. Moore, E. Nienhouse, L. Polvani, P. J. Rasch, and W. G. Strand. The Community Earth System Model Version 2 (CESM2). *Journal of Advances in Modeling Earth Systems*, 12(2):e2019MS001916, 2020. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2019MS001916.

[46] S. G. Yeager, G. Danabasoglu, N. A. Rosenbloom, W. Strand, S. C. Bates, G. A. Meehl, A. R. Karspeck, K. Lindsay, M. C. Long, H. Teng, and N. S. Lovenduski. Predicting Near-Term Changes in the Earth System: A Large Ensemble of Initialized Decadal Prediction Simulations Using the Community Earth System Model. *Bulletin of the American Meteorological Society*, 99(9):1867–1886, September 2018. Publisher: American Meteorological Society Section: Bulletin of the American Meteorological Society.

[47] 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.

[48] Dan Fu, Justin Small, Jaison Kurian, Yun Liu, Brian Kauffman, Abishek Gopal, Sanjiv Ramachandran, Zhi Shang, Ping Chang, Gokhan Danabasoglu, Katherine Thayer-Calder, Mariana Vertenstein, Xiaohui Ma, Hengkai Yao, Mingkui Li, Zhao Xu, Xiaopei Lin, Shaoqing Zhang, and Lixin Wu. Introducing the New Regional Community Earth System Model, R-CESM. *Bulletin of the American Meteorological Society*, 102(9):E1821–E1843, September 2021. Publisher: American Meteorological Society Section: Bulletin of the American Meteorological Society.

[49] 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.

[50] Christopher D. Roberts, Retish Senan, Franco Molteni, Souhail Boussetta, Michael Mayer, and Sarah P. E. Keeley. Climate model configurations of the ECMWF Integrated Forecasting System (ECMWF-IFS cycle 43r1) for HighResMIP. *Geoscientific Model Development*, 11(9):3681–3712, September 2018. Publisher: Copernicus GmbH.

[51] Masaki Satoh, Hirofumi Tomita, Hisashi Yashiro, Hiroaki Miura, Chihiro Kodama, Tatsuya Seiki, Akira T. Noda, Yohei Yamada, Daisuke Goto, Masahiro Sawada, Takemasa Miyoshi, Yosuke Niwa, Masayuki Hara, Tomoki Ohno, Shin-ichi Iga, Takashi Arakawa, Takahiro Inoue, and Hiroyasu Kubokawa. The Non-hydrostatic Icosahedral Atmospheric Model: description and development. *Progress in Earth and Planetary Science*, 1(1):18, October 2014.

[52] 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.

[53] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. C. Bates, G. Danabasoglu, J. Edwards, M. Holland, P. Kushner, J.-F. Lamarque, D. Lawrence, K. Lindsay, A. Middleton, E. Munoz, R. Neale, K. Oleson, L. Polvani, and M. Vertenstein. The Community Earth System Model (CESM) Large Ensemble Project: A Community Resource for Studying Climate Change in the Presence of Internal Climate Variability. *Bulletin of the American Meteorological Society*, 96(8):1333–1349, August 2015. Publisher: American Meteorological Society Section: Bulletin of the American Meteorological Society.

[54] Martin Leutbecher. Ensemble size: How suboptimal is less than infinity? *Quarterly Journal of the Royal Meteorological Society*, 145(S1):107–128, 2019. _eprint: https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.3387.

[55] Takemasa Miyoshi, Guo-Yuan Lien, Shinsuke Satoh, Tomoo Ushio, Kotaro Bessho, Hirofumi Tomita, Seiya Nishizawa, Ryuji Yoshida, Sachiho A. Adachi, Jianwei Liao, Balazs Gerofi, Yutaka Ishikawa, Masaru Kunii, Juan Ruiz, Yasumitsu Maejima, Shigenori Otsuka, Michiko Otsuka, Kozo Okamoto, and Hiromu Seko. “Big Data Assimilation” Toward Post-Petascale Severe Weather Prediction: An Overview and Progress. *Proceedings of the IEEE*, 104(11):2155–2179, November 2016. Conference Name: Proceedings of the IEEE.

[56] 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.

[57] 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.

[58] Philipp Neumann, Peter Düben, Panagiotis Adamidis, Peter Bauer, Matthias Brück, 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.

[59] Yongjun Zheng, Clément Albergel, Simon Munier, Bertrand Bonan, and Jean-Christophe Calvet. An offline framework for high-dimensional ensemble Kalman filters to reduce the time to solution. *Geoscientific Model Development*, 13(8):3607–3625, August 2020. Publisher: Copernicus GmbH.

[60] 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.

[61] Habib Toye. Efficient Ensemble Data Assimilation and Forecasting of the Red Sea Circulation. November 2020. Accepted: 2020-11-23T06:03:14Z.

[62] decimate/what_is_decimate.rst at dist · samkos/decimate.

[63] Peter D. Düben and Andrew Dawson. An approach to secure weather and climate models against hardware faults. *Journal of Advances in Modeling Earth Systems*, 9(1):501–513, 2017. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/2016MS000816.

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

[65] Théophile Terraz, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin. Melissa: large scale in transit sensitivity analysis avoiding intermediate files. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 1–14, New York, NY, USA, November 2017. Association for Computing Machinery.

[66] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[67] James S. Plank, Jian Xu, and Robert H.B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing, 1995.

[68] Kurt B. Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. libhashckpt: Hash-based incremental checkpointing using gpu's. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 272–281, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[69] Fti file format documentation. `http://leobago.github.io/fti/ftiff.html`.

[70] Mark Adler and Jean-loup Gailly. Zlib data compression library. `https://github.com/madler/zlib`, 1995-2017.

[71] Anastase Nakassis. Fletcher's error detection algorithm: How to implement it efficiently and how toavoid the most common pitfalls. *SIGCOMM Comput. Commun. Rev.*, 18(5):63–88, October 1988.

[72] OpenSSL 1.0.2 manpages - md5. `https://www.openssl.org/docs/man1.0.2/crypto/md5.html`. Accessed: 2018-04-17.

[73] Sriram Ramanujam and Marimuthu Karuppiah. Designing an algorithm with high avalanche effect. *IJCSNS International Journal of Computer Science and Network Security*, 11(1):106–111, 2011.

[74] Shay Gueron. Speeding up crc32c computations with intel crc32 instruction. *Information Processing Letters*, 112(5):179 – 185, 2012.

[75] zlib home page. `https://zlib.net`. Accessed: 2018-05-3.

[76] bsc.es, marenostrum4 user's guide. `https://www.bsc.es/user-support/mn4.php#systemoverview`. Accessed: 2018-04-27.

[77] bsc.es, marenostrum iv (2017) system architecture. `https://www.bsc.es/marenostrum/marenostrum/technical-information`. Accessed: 2018-04-27.

[78] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[79] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[80] Stefano Markidis, Giovanni Lapenta, and Rizwan-uddin. Multi-scale simulations of plasma with ipic3d. *Mathematics and Computers in Simulation*, 80(7):1509 – 1519, 2010. Multiscale modeling of moving interfaces in materials.

[81] Deep projects. `http://www.deep-projects.eu/`.

[82] Boost - serialization. `https://www.boost.org/doc/libs/1_67_0/libs/serialization/doc/index.html`. Accessed: 2018-05-17.

[83] Weikuan Yu, J. S. Vetter, and H. Sarp Oral. Performance characterization and optimization of parallel i/o on the cray xt. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, April 2008.

[84] Hongzhang Shan and John Shalf. Using ior to analyze the i / o performance for hpc platforms. 2007.

[85] L. Bautista Gomez, K. Keller, and O. Unsal. Performance study of non-volatile memories on a high-end supercomputer. In *Workshop on Performance and Scalability of Storage Systems 2018 (WOPSSS'18), Frankfurt, Germany*, June 2018.

[86] Kurt B Ferreira. *Keeping Checkpointing Viable for Exascale Systems*. PhD thesis, The University of New Mexico, 2011.

[87] Bogdan Nicolae, Adam Moody, Gregory Kosinovsky, Kathryn Mohror, and Franck Cappello. Veloc: Very low overhead checkpointing in the age of exascale. *arXiv preprint arXiv:2103.02131*, 2021.

[88] Joshua Hursey, Richard L. Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G. Solt. Run-through stabilization: An mpi proposal for process fault tolerance. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 329–332, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[89] William Gropp and Ewing Lusk. Fault tolerance in message passing interface programs. *The International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[90] A. D. Selvakumar, P. M. Sobha, G. C. Ravindra, and R. Pitchiah. Design, implementation and performance of fault-tolerant message passing interface (mpi). In *Proceedings. Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region, 2004.*, pages 120–129, 2004.

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

[92] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in mpi applications. *International Journal of High Performance Computing Applications*, 30(3), 1 2016.

[93] Sourav Chakraborty, Ignacio Laguna, Murali Emani, Kathryn Mohror, Dhabaleswar K. Panda, Martin Schulz, and Hari Subramoni. Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous mpi applications. *Concurrency and Computation: Practice and Experience*, 32(3):e4863, 2020. e4863 cpe.4863.

[94] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, August 2013. Publisher: SAGE Publications Ltd STM.

[95] The HDF Group. Hierarchical data format version 5, 2000-2010.

[96] Christoph Ertl, Jérôme Frisch, and Ralf-Peter Mundani. Design and optimisation of an efficient hdf5 i/o kernel for massive parallel fluid flow simulations. *Concurrency and Computation: Practice and Experience*, 29(24):e4165, 2017.

[97] Mark Howison. Tuning hdf5 for lustre file systems. 2010.

[98] Kshitij Mehta, John Bent, Aaron Torres, Gary Grider, and Edgar Gabriel. A plugin for hdf5 using plfs for improved i/o performance and semantic analysis. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 746–752. IEEE, 2012.

[99] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[100] Oak Ridge National Laboratory. Adios 2: The adaptable input/output system version 2 — adios2 2.5.0 documentation, 2018.

[101] Unidata | NetCDF.

[102] Jianwei Li, Wei-keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 39–39, 2003.

[103] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.

[104] BSC Operations. Technical information marenostrum 4, 2019. Available at `https://www.bsc.es/marenostrum/marenostrum/technical-information`.

[105] Elvis Rojas, Esteban Meneses, Terry Jones, and Don Maxwell. Analyzing a five-year failure record of a leadership-class supercomputer. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 196–203. IEEE, 2019.

[106] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[107] Camille Coti. chapter Fault Tolerance Techniques for Distributed, Parallel Applications, pages 221–252. IGI Global, Hershey, PA, USA, 2016.

[108] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626. IEEE, 2012.

[109] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kalé. Using migratable objects to enhance fault tolerance schemes in supercomputers. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2061–2074, 2015.

[110] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Proactive process-level live migration in hpc environments. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[111] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kalé. Using migratable objects to enhance fault tolerance schemes in supercomputers. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):2061–2074, July 2015.

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

[113] 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.

[114] Dieter Fox, Sebastian Thrun, Wolfram Burgard, and Frank Dellaert. Particle filters for mobile robot localization. In *Sequential Monte Carlo methods in practice*, pages 401–428. Springer, 2001.

[115] Adil Rasheed, Omer San, and Trond Kvamsdal. Digital twin: Values, challenges and enablers from a modeling perspective. *Ieee Access*, 8:21980–22012, 2020.

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

[117] ZeroMQ - `https://zeromq.org/`.

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

[119] 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.

[120] 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.

[121] 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.

[122] 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.

[123] 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.

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

[125] Ibrahim Hoteit, Tim Hoar, Ganesh Gopalakrishnan, Nancy Collins, Jeffrey Anderson, Bruce Cornuelle, Armin Köhl, 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.

[126] John Marshall, Alistair Adcroft, Chris Hill, Lev Perelman, and Curt Heisey. A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research: Oceans*, 102(C3):5753–5766, 1997. _eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/96JC02775.

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

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

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

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

[131] 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.

[132] 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.

[133] 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.

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

[135] Adil Rasheed, Omer San, and Trond Kvamsdal. Digital Twin: Values, Challenges and Enablers From a Modeling Perspective. *IEEE Access*, 8:21980–22012, 2020. Conference Name: IEEE Access.

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

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

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

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

[140] Valentijn R. N. Pauwels, Rudi Hoeben, Niko E. C. Verhoest, and François P. De Troch. The importance of the spatial patterns of remotely sensed soil moisture in the improvement of discharge predictions for small-scale basins through data assimilation. *Journal of Hydrology*, 251(1):88–102, September 2001.

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

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

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

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

[145] Mark Asch, Marc Bocquet, and Maëlle Nodet. *Data assimilation: methods, algorithms, and applications*, volume 11. SIAM, 2016.

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

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

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

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

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

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

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

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

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

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

[156] Milan Klöwer, Miha Razinger, Juan J. Dominguez, Peter D. Düben, and Tim N. Palmer. Compressing atmospheric data into its real information content. *Nature Computational Science*, 1(11):713–724, November 2021. Number: 11 Publisher: Nature Publishing Group.

[157] John L. Schnase, Tsengdar J. Lee, Chris A. Mattmann, Christopher S. Lynnes, Luca Cinquini, Paul M. Ramirez, Andre F. Hart, Dean N. Williams, Duane Waliser, Pamela Rinsland, W. Philip Webster, Daniel Q. Duffy, Mark A. McInerney, Glenn S. Tamkin, Gerald L. Potter, and Laura Carrier. Big Data Challenges in Climate Science. *IEEE geoscience and remote sensing magazine*, Volume 4(Iss 3):10–22, September 2016.

[158] Francesca Eggleton and Kate Winfield. Open Data Challenges in Climate Science. *Data Science Journal*, 19(1):52, December 2020. Number: 1 Publisher: Ubiquity Press.

[159] Ryohei Okazaki, Takekazu Tabata, Sota Sakashita, Kenichi Kitamura, Noriko Takagi, Hideki Sakata, Takeshi Ishibashi, Takeo Nakamura, and Yuichiro Ajima. Supercomputer Fugaku CPU A64FX Realizing High Performance, High-Density Packaging, and Low Power Consumption. Fujitsu Technical Review, Fujitsu Limited, March 2020.

[160] Sam Hatfield, Aneesh Subramanian, Tim Palmer, and Peter Düben. Improving Weather Forecast Skill through Reduced-Precision Data Assimilation. *Monthly Weather Review*, 146(1):49–62, January 2018. Publisher: American Meteorological Society Section: Monthly Weather Review.

[161] Sam Hatfield, Peter Düben, Matthew Chantry, Keiichi Kondo, Takemasa Miyoshi, and Tim Palmer. Choosing the Optimal Numerical Precision for Data Assimilation in the Presence of Model Error. *Journal of Advances in Modeling Earth Systems*, 10(9):2177–2191, 2018. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2018MS001341.

[162] Masuo Nakano, Hisashi Yashiro, Chihiro Kodama, and Hirofumi Tomita. Single Precision in the Dynamical Core of a Nonhydrostatic Global Atmospheric Model: Evaluation Using a Baroclinic Wave Test Case. *Monthly Weather Review*, 146(2):409–416, February 2018. Publisher: American Meteorological Society Section: Monthly Weather Review.

[163] Sameh Abdulah, Qinglei Cao, Yu Pei, George Bosilca, Jack Dongarra, Marc G. Genton, David E. Keyes, Hatem Ltaief, and Ying Sun. Accelerating Geostatistical Modeling and Prediction With Mixed-Precision Computations: A High-Productivity Approach With PaRSEC. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):964–976, April 2022. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[164] Allison H. Baker, Haiying Xu, John M. Dennis, Michael N. Levy, Doug Nychka, Sheri A. Mickelson, Jim Edwards, Mariana Vertenstein, and Al Wegener. A methodology for evaluating the impact of data compression on climate simulation data. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, HPDC ’14, pages 203–214, New York, NY, USA, June 2014. Association for Computing Machinery.

[165] Allison H. Baker, Dorit M. Hammerling, Sheri A. Mickelson, Haiying Xu, Martin B. Stolpe, Phillipe Naveau, Ben Sanderson, Imme Ebert-Uphoff, Savini Samarasinghe, Francesco De Simone, Francesco Carbone, Christian N. Gencarelli, John M. Dennis, Jennifer E. Kay, and Peter Lindstrom. Evaluating lossy data compression on climate simulation data within a large ensemble. *Geoscientific Model Development*, 9(12):4381–4403, December 2016. Publisher: Copernicus GmbH.

[166] Andrew Poppick, Joseph Nardi, Noah Feldman, Allison H. Baker, Alexander Pinard, and Dorit M. Hammerling. A statistical analysis of lossily compressed climate model data. *Computers & Geosciences*, 145:104599, December 2020.

[167] Christopher K. Wikle and L. Mark Berliner. A Bayesian tutorial for data assimilation. *Physica D: Nonlinear Phenomena*, 230(1):1–16, June 2007.

[168] Brian R. Hunt, Eric J. Kostelich, and Istvan Szunyogh. Efficient data assimilation for spatiotemporal chaos: A local ensemble transform Kalman filter. *Physica D: Nonlinear Phenomena*, 230(1):112–126, 2007.

[169] Roland Potthast, Anne Walter, and Andreas Rhodin. A Localized Adaptive Particle Filter within an Operational NWP Framework. *Monthly Weather Review*, 147(1):345–362, January 2019. Publisher: American Meteorological Society Section: Monthly Weather Review.

[170] Russ Rew, Glenn Davis, Steve Emmerson, Cathy Cormack, John Caron, Robert Pincus, Ed Hartnett, Dennis Heimbigner, Lynton Appel, and Ward Fisher. Unidata NetCDF, 1989. Language: en Medium: application/java-archive,application/gzip,application/tar.

[171] ADIOS: The Adaptable I/O System | Computer Science and Mathematics.

[172] The HDF5® Library & File Format.

[173] RIKEN Center for Computational Science. Fugaku Supercomputer. `https://www.r-ccs.riken.jp/en/fugaku/`, 2021. [1 April 2021].

[174] Edward N. Lorenz. Designing Chaotic Models. *Journal of the Atmospheric Sciences*, 62(5):1574–1587, May 2005. Publisher: American Meteorological Society Section: Journal of the Atmospheric Sciences.

[175] Jinyang Liu, Sihuan Li, Sheng Di, Xin Liang, Kai Zhao, Dingwen Tao, Zizhong Chen, and Franck Cappello. Improving lossy compression for sz by exploring the best-fit lossless compression techniques. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 2986–2991, 2021.

[176] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX*, 12:100561, July 2020.

[177] Xavier Delaunay, Aurélie Courtois, and Flavien Gouillon. Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netCDF-4 or HDF5 files. *Geoscientific Model Development*, 12(9):4099–4113, September 2019. Publisher: Copernicus GmbH.

[178] Oriol Tintó Prims, Mario C. Acosta, Miguel Castrillo, Stella Valentina Paronuzzi Ticco, Kim Serradell, Ana Cortés, and Francisco J. Doblas-Reyes. Discriminating accurate results in nonlinear models. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 1028–1031, July 2019.

[179] James W. Hurrell, M. M. Holland, P. R. Gent, S. Ghan, Jennifer E. Kay, P. J. Kushner, J.-F. Lamarque, W. G. Large, D. Lawrence, K. Lindsay, W. H. Lipscomb, M. C. Long, N. Mahowald, D. R. Marsh, R. B. Neale, P. Rasch, S. Vavrus, M. Vertenstein, D. Bader, W. D. Collins, J. J. Hack,

J. Kiehl, and S. Marshall. The Community Earth System Model: A Framework for Collaborative Research. *Bulletin of the American Meteorological Society*, 94(9):1339–1360, September 2013. Publisher: American Meteorological Society Section: Bulletin of the American Meteorological Society.

[180] Dingwen Tao, Sheng Di, Hanqi Guo, Zizhong Chen, and Franck Cappello. Z-checker: A framework for assessing lossy compression of scientific data. *The International Journal of High Performance Computing Applications*, 33(2):285–303, March 2019. Publisher: SAGE Publications Ltd STM.

[181] Jie Dou, Ali P. Yunus, Dieu Tien Bui, Abdelaziz Merghadi, Mehebub Sahana, Zhongfan Zhu, Chi-Wen Chen, Zheng Han, and Binh Thai Pham. Improved landslide assessment using support vector machine with bagging, boosting, and stacking ensemble machine learning framework in a mountainous watershed, Japan. *Landslides*, 17(3):641–658, March 2020.

[182] Jinsong Yu, Yue Song, Diyin Tang, and Jing Dai. A Digital Twin approach based on nonparametric Bayesian network for complex system health monitoring. *Journal of Manufacturing Systems*, 58:293–304, January 2021.