E-Infrastructures

H2020-INFRAEDI-2018-1

INFRAEDI-2-2018: Centres of Excellence on HPC

EoCoE-II

Energy oriented Center of Excellence : toward exascale for energy

Grant Agreement Number: INFRAEDI-824158

D4.3

Report on I/O optimization and fault tolerance

Project and Deliverable Information Sheet

EoCoE-II	Project Ref:	INFRAEDI-824158
	Project Title:	Energy oriented Centre of Excellence: towards exascale for energy
	Project Web Site:	http://www.eocoe2.eu
	Deliverable ID:	D4.3
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M24 31/12/2020
	Actual Date of Delivery:	M26 28/02/2021
	EC Project Officer:	Matteo Mascagni

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

Document Control Sheet

Document	Title :	Report on I/O optimization and fault tolerance
	ID :	D4.3
	Available at:	http://www.eocoe2.eu
	Software tool:	LaTeX
Authorship	Written by:	Henrik Büsing (FZJ), Konstantinos Chasapis (DDN), Kai Keller (BSC)
	Contributors:	Jean-Thomas Acquaviva (DDN), Leonardo Bautista (BSC), Yen-Sen Lu (FZJ), Sebastian Lührs (FZJ), Benedikt Steinbusch (FZJ)
	Reviewed by:	PEC, PBS

Document Keywords: I/O, SIONlib, IME, FTI, Data, Fault Tolerance, Flash Storage, Cache

Contents

List of Figures

List of Tables

1 Executive summary

In this deliverable we provide technical details of the activities taking place in the framework of the I/O optimization and fault tolerance tasks in WP 4 of EoCoE-II.

Large simulation can produce a vast amount of data in forms of actual simulation outcome or by capturing the intermediate state of the simulation in so called checkpoint/restart files. The I/O phase of an application can be a significant factor while scaling its performance. In many cases parallel applications perform serial and synchronous I/O where the compute phase has to wait until the I/O phase is finished. Within the I/O optimization and fault tolerance activities of EoCoE-II we try to lower the overall I/O footprint of the scientific challenge applications but also add new optimized options to support fault tolerance capabilities. To achieve this target we leverage capabilities of different I/O libraries, utilize new types of I/O subsystems and implement new libraries to be integrated into the scientific challenge applications in WP1.

Improving the I/O phase performance can include the transformation of serial and synchronous I/O to parallel and asynchronous, which will allow to overlap compute and I/O time of an application. Such effort is being presented in the first part of the deliverable using the ESIAS-WRF application of the scientific challenge Meteo for Energy. We leverage the WRF capabilities underneath of ESIAS to utilize NetCDF4 for serial I/O and PnetCDF for parallel I/O and an so called quilt mode to support asynchronous I/O. We adapted these capabilities into the ESIAS ensemble run framework. In the experiments, we see a total application speedup improvement of 2.8 and 1.5 depending on the parameters. The asynchronous I/O is able to shorten the run-time by a factor of 2.1 to 1.3 depending on the use case. Beside such a pure software based optimization we are also interested to leverage new types of Exascale aware I/O technologies, therefore we illustrate the potential benefits of flash storage integration in the I/O path. We leverage DDN's Infinite Memory Engine (IME) product, an all flash distributed cache that sits in between the compute nodes and the classical parallel file system. To allow an easier adaptation process for interested applications, we implemented the IME native API underneath of the SIONlib I/O library to allow a direct utilization of IME through the API of SIONlib. We measured the performance improvement that IME offers in comparison to the classical parallel file system (GPFS) when using SIONlib. From our evaluation, we clearly see that IME outperforms GPFS in all cases. For a single compute node IME performs up to 1.8 x the maximum performance of GPFS. When using multiple nodes IME can deliver more than four times the throughput of GPFS. Moreover, experimenting with SIONlib parameters we also observed that the performance of IME is more independent of the individual I/O operation layout while GPFS is greatly impacted.

The last part of the deliverable is dedicated to the progress report on fault tolerance. This deliverable covers fault tolerance in ensemble computations, which also introduced a close cooperation with WP5. In the previous deliverable 4.1, we presented a prototype implementation for fault tolerance in Melissa-DA, a distributed I/O-avoiding framework for ensemble data assimilation. In this deliverable we present an advanced version of the implementation which uses threads and dedicated MPI processes to hide the checkpoint costs. We present results, that show, that we can hide the checkpoint cost entirely behind the frameworks normal execution. Moreover, we show that inside a well-defined region of the execution flow, we can recover without any re-computation.

2 Acronyms

Table 1: Acronyms for the partners and institutes therein.

Acronym	Partner and institute
AMU:	Aix-Marseille University
BSC:	Barcelona Supercomputing Center
CEA:	Commissariat à l'énergie atomique et aux énergies alternatives
CERFACS:	Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique
CIEMAT:	Centro De Investigaciones Energeticas, Medioambientales Y Tecnologicas
CoE:	Center of Excellence
DDN:	Data Direct Networks
EDF:	Électricité de France
ENEA:	Agenzia nazionale per le nuove tecnologie, l'energia e lo sviluppo economico sostenibile
FAU:	Friedrich-Alexander University of Erlangen-Nuremberg
FSU:	Friedrich Schiller University
FZJ:	Forschungszentrum Jülich GmbH
IBG-3:	Institute of Bio- and Geosciences Agrosphere
IEK-8:	Institute for Energy and Climate Research 8 (troposhere)
IEE:	Fraunhofer Institute for Energy Economics and Energy System Technology
IFPEN:	IFP Énergies Nouvelles
INAC:	Institut nanosciences et cryogénie
INRIA:	Institut national de recherche en informatique et en automatique
IRFM:	Institute for Magnetic Fusion Research
MdlS:	Maison de la Simulation
MF:	Meteo France
MPG:	Max-Planck-Gesellschaft
RWTH:	Rheinisch-Westfälische Technische Hochschule Aachen, Aachen University
UBAH:	University of Bath
UNITN:	University of Trento

Table 2: Acronyms of software packages

Acronym	Software and codes
ESIAS:	Ensemble for Stochastic Interpolation of Atmospheric Simulations
EURAD-IM:	EURopean Air pollution Dispersion-Inverse Model
FTI:	Fault Tolerance Interface
FUSE:	Filesystem in Userspace
Gysela:	GYrokinetic SEmi-LAgrangian
GPFS:	General Parallel File System, brand name: IBM Spectrum Scale
ICON:	Icosahedral Nonhydrostatic model
IME:	Infinite Memory Engine
I/O:	Input and Output
MELISSA:	Modular External Library for In Situ Statistical Analysis
NetCDF:	Network Common Data Format
ParFlow:	PARallel Flow
PDI:	PDI Data Interface
SIONlib:	Scalable I/O library for parallel access to task-local files
WRF:	Weather Research and Forecast model

Table 3: Acronyms for the Scientific Terms used in the report.

Acronym	Scientific Nomenclature
API:	Application Programming Interface
CPU:	Central Processing Units
DA:	Data Assimilation
EFK:	Extended Kalman Filter
EnFK:	Ensemble Kalman Filter
HPC:	High Performance Computing
HPST:	High Performance Storage Tier
MPI:	Message Passing Interface
NVM:	Non Volatile Memory
NWP:	Numerical Weather Prediction
SSD:	Solid State Drive
PFS:	Parallel Filesystem
WP:	Work Package

3 Introduction

This deliverable covers two main tasks of WP4: Task 4.2 I/O refactoring and optimization and Task 4.3 fault tolerance. Therefore the deliverable is split into two main sections.

Section 4 reports the overall progress and technical details in the I/O refactoring and optimization task in context of asynchronous I/O capabilities and the utilization of intermediate flash storage based burst buffer. In subsection 4.1 we demonstrate the results of using parallel and asynchronous I/O for the ESIAS-WRF application. We utilized NetCDF-4 for serial I/O and PnetCDF for parallel I/O. Asynchronous I/O is implemented using the quilt mode of WRF. These technologies were tested and the quilt mode was adapted to the ESIAS ensemble handling framework. The evaluation of Flash Storage Integration (SIONlib with IME) is presented in subsection 4.2. The experiments were conducted as one of the first users on the High Performance Storage Tier (HPST) of Jülich Supercomputing Centre (JSC) utilizing the JUWELS supercomputer system. In the analysis we compared the performance of the classical parallel file system in contrast with IME and in combination with the new SIONlib integration, which was presented in D4.1. The analysis will provide the basis to judge on the potential of an scientific challange application integration.

The second main section, section 5, covers the details the developments and performance evaluation on the FTI-Melissa integration. The contributions in this deliverable contain a schematic overview of the data assimilation workflow in Section 5.1.1, an introduction to Melissa-DA in Section 5.1.2, an introduction to the utilized checkpoint features in Sections 5.1.3 and 5.1.4 and a detailed description of our fault tolerance implementation into Melissa-DA in Section 5.2.

3.1 Link to other work packages

The work done in the context of the deliverable is directly connected to other work packages in context of EoCoE-II: The main scientific challenge applications, handled by WP1, which benefit from the optimization work are ESIAS for the asynchronous I/O tests and GyselaX, which will test the utilizeation of the SIONlib IME interface. The fault tolerance integration into Melissa are prepared for ParFlow. For this we also worked in close collaboration with WP5 concerning the Melissa framework developed there.

4 I/O refactoring and optimization

The main target of this task is the overall reduction of the I/O footprint to reduce computational costs on larger scales, either by leveraging software technologies and library optimizations or by adapting new types of Exascale aware I/O devices.

4.1 Parallel and asynchronous I/O in ESIAS-WRF

Simulations with multiple ensemble members may produce a significant amount of output data, such as restart or result files. Generally, the simulation is interrupted until reading and writing of files is finished and then continues. This I/O part may consume a crucial part of the total simulation time, in particular if only one process is responsible for reading and writing. Possible solutions include parallel and asynchronous I/O. Parallel I/O uses multiple processes and thus accelerates the I/O intensive part. Asynchronous I/O utilizes dedicated processes for the handling of I/O. Thus, calculations and reading and writing may be carried out at the same time.

This section presents results for the ESIAS-WRF application, which is an ensemble-aware version of the original WRF code, using parallel and asynchronous I/O. We compare serial I/O using NetCDF4 with parallel I/O using PnetCDF. The total speedup for the whole simulation is 2.82 and 1.54 using one and five ensemble members correspondingly. Using asynchronous I/O with so called quilt servers, a WRF builtin option to offload I/O to dedicated I/O handling processes, we can reduce runtime from 687 s to 327 s (speedup of 2.10) for one ensemble member. Providing quilt servers for every ensemble member, we have for the case with five ensemble members a reduction from 1321 s to 1040 s (speed-up of 1.27) using one quilt server per ensemble member. Providing additional cores for the quilt servers can further reduce runtime to 849 s (overall speedup of 1.56).

4.1.1 NetCDF-4 and PnetCDF

All tests were performed in the JUWELS cluster system [6] at the Jülich Supercomputing Centre. For parallel netCDF support on JUWELS we used the following modules and environment variables:

```
module load parallel-netcdf/1.11.0
module load HDF5/1.10.5

export PNETCDF=/gpfs/software/juwels/stages/2019a/software/...
parallel-netcdf/1.11.0-ipsmpi-2019a.1
export PHDF5=/gpfs/software/juwels/stages/2019a/software/...
HDF5/1.10.5-ipsmpi-2019a.1
```

:

Figure 1 compares two examples with one and five ensemble members using 48 and 20 cores. The serial I/O run with one ensemble member takes 687 s with 2.39 s per wrfout write and 231 writes in total yielding 11.80 GB of output. The corresponding parallel I/O run takes 244 s with 0.69 s per wrfout write and 231 writes in total yielding 27.29 GB of output. The write speedup is 3.46 and total speedup 2.82. The increase in size is 2.31.

The serial I/O run for five ensemble members takes 1321 s with 2.16 s per wrfout write with, again, 231 writes in total, yielding 59.79 GB output. The corresponding parallel run takes 856 s with 0.33 s per wrfout write and 231 writes in total yielding 153.04 GB of output. The write speedup is 6.55 and total speedup 1.54. The size increase is 2.56.

4.1.2 Asynchronous I/O via quilt mode

For asynchronous I/O capabilities the builtin quilt feature of WRF was tested. The feature allows to offload the I/O of the computational processes to dedicated I/O handling processes, which

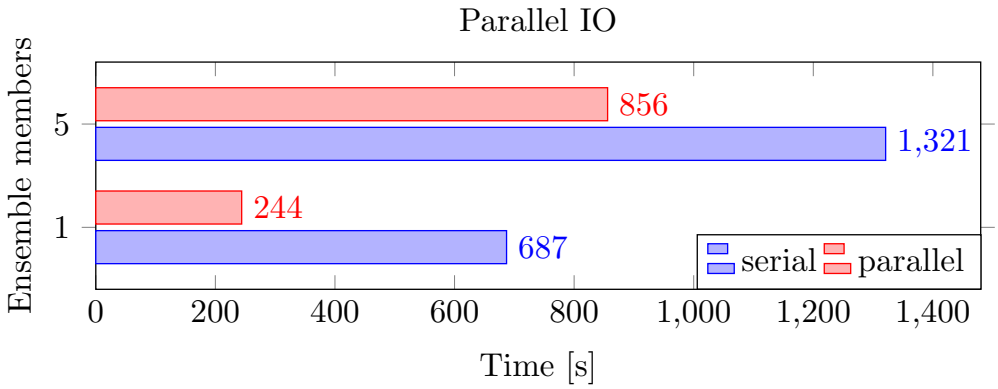


Figure 1: Serial and parallel performance using NetCDF-4 and PnetCDF for one and five ensemble members using 48 and 20 cores in total.

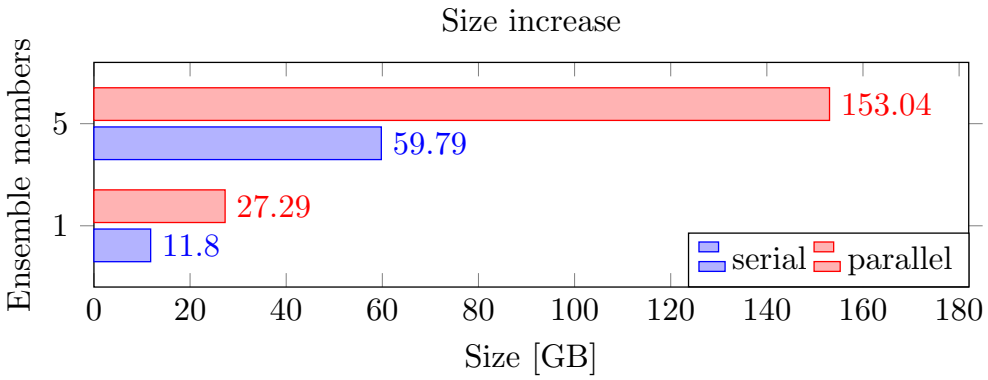


Figure 2: Size increase due to using PnetCDF compared to NetCDF-4 for one and five ensemble members.

allows to overlap I/O and further calculation. In contrast these quilt servers will utilize additional resources. The WRF builtin quilt mode does not work out of the box in the existing setup of ESIAS-WRF due to the ensemble related MPI communicator handling. We introduce a bugfix, which alters the communicator used for the ensemble. Before `MPI_COMM_WORLD` was used by default. This also included the quilt servers being part of the ensemble communicator. This leads to a hangup in an MPI_ALL_REDUCE call. With the bugfix the quilt server processes are removed from the ensemble handling layer and we were able to circumvent the hangup. We compared the run with one ensemble member and 48 cores with NetCDF-4 with runs with one, two, four and eight quilt servers. Figure 3 shows the results. While using one and two quilt servers reduces run-time for this test example using more quilt servers increases run-time when compared to the optimum case of two quilt servers.

Additionally, we examine the case with five ensemble members. Here, we distinguish two options. The first one uses 20 cores in total and the quilt servers use part of the available resources. The second option uses more cores when the number of quilt servers increases, e.g., when using one quilt server per ensemble member, we have five quilt servers in total and use 25 cores instead of 20. Figure 4 shows timing results for five ensemble members and 20 cores with NetCDF-4. Shown are results for zero, one, two, four and eight quilt servers per ensemble member. This yields zero, five, ten, 20 and 40 quilt servers in total. The results show a decrease in runtime from 1 321 s to 1 040 s when using one quilt server per ensemble member, having four and eight mostly for completeness as these setups oversubscribe the available resources. This uses already 25 % of the resources for the quilt servers. Increasing the number of quilt servers does not improve the run-time even compared

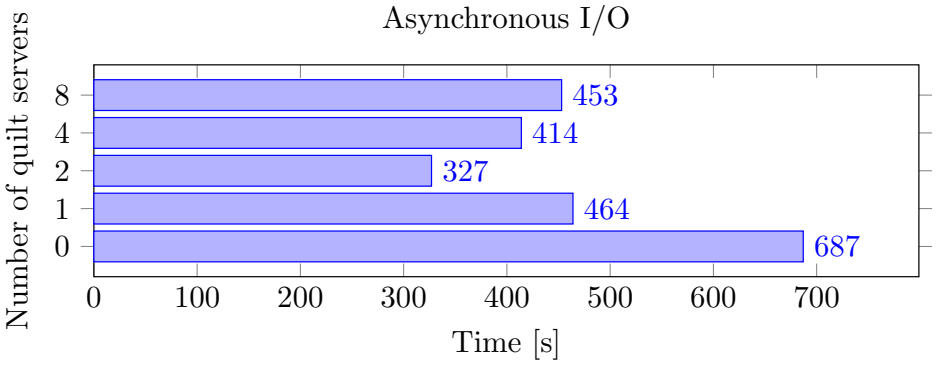


Figure 3: Timing results for one ensemble member and 48 cores with NetCDF-4 and zero, one, two, four and eight quilt servers.

to the run with no quilt servers at all. Thus, we examine the case where we provide additional resources for the quilt servers.

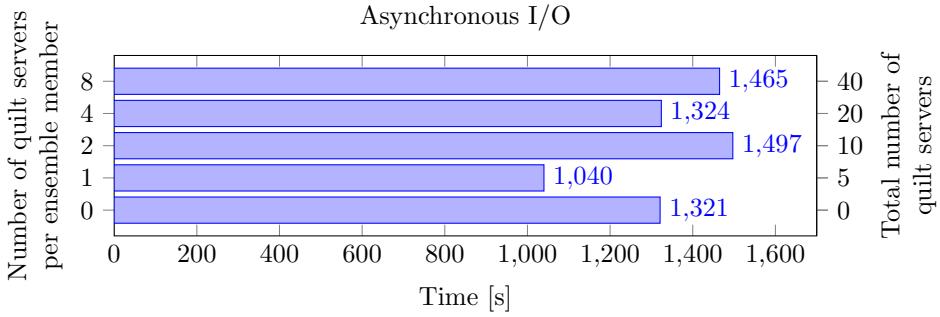


Figure 4: Timing results for five ensemble members and 20 cores with NetCDF-4 and zero, one, two, four and eight quilt servers per ensemble member, yielding zero, five, ten, 20 and 40 quilt servers in total.

Figure 5 shows results for zero, one, two, four and eight quilt servers per ensemble member, but this time the number of cores increases with the number of quilt servers. For zero, five, ten, 20 and 40 quilt servers in total, we use 20, 25, 30, 40 and 60 cores, respectively. The optimum run-time for this case is the one with four quilt servers per ensemble member, i.e., 20 quilt servers in total, which provides a one to one mapping between computation and I/O. Of course for regular production this would be too costly.

Using quilt servers together with PnetCDF does not work for WRF 3.7.1, which is the version of WRF used together with ESIAS so far. This bug is also mentioned in [2]. A general upgrade of WRF within ESIAS is planned which allows to test this option in the newer setup. In addition the approach will be tested next on larger scales.

4.2 Evaluation of Flash Storage Integration (SIONlib with IME)

In this section, we illustrate the benefits of flash storage integration in the I/O path. In detail, we evaluate the performance improvement of the integration of DDN's Infinite Memory Engine (IME) [10] with the SIONlib parallel I/O library in contrast to the utilization of a classical parallel file system. In the last deliverable (D4.1), we explained the technical details of the integration of

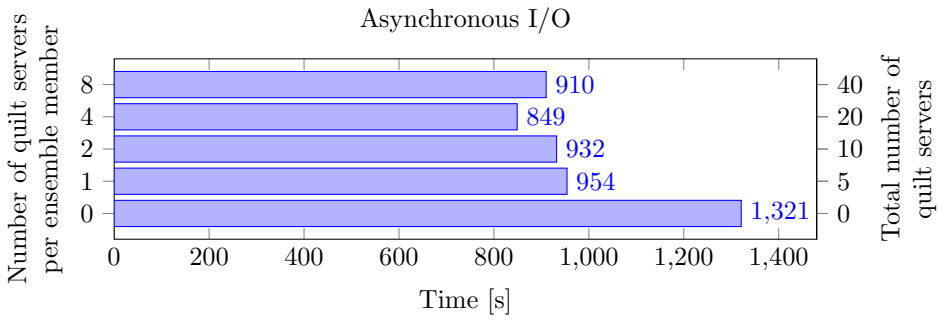


Figure 5: Timing results for five ensemble members with NetCDF-4 and zero, one, two, four and eight quilt servers per ensemble member, yielding zero, five, ten, 20 and 40 quilt servers in total and using 20, 25, 30, 40 and 60 cores, respectively.

the IME native API with SIONlib. For completeness the following subsections provide background information regarding IME and SIONlib. Then, we discuss our experiments methodology. Afterwards, we demonstrate the evaluation and finally we conclude by summarizing the results. The work mainly targets the Fusion for Energy scientific application GyselaX to significantly speed up the checkpoint generation process utilizing SIONlib and IME. This integration will be tested within the final project phase.

4.2.1 IME

As visualized in Figure 6 IME, in essence, is a Scale-Out Flash Cache Layer using NVMe SSDs inserted between the compute cluster and the Parallel File System (PFS). IME is configured as a cluster with multiple NVMe servers. The purpose of IME is to accelerate difficult I/O patterns: small/random/shared file/high concurrency taking advantage of flash device characteristics and also benefits from the thin software I/O management layer. IME handles the I/O data traffic and forwards the metadata requests to the parallel file system underneath.

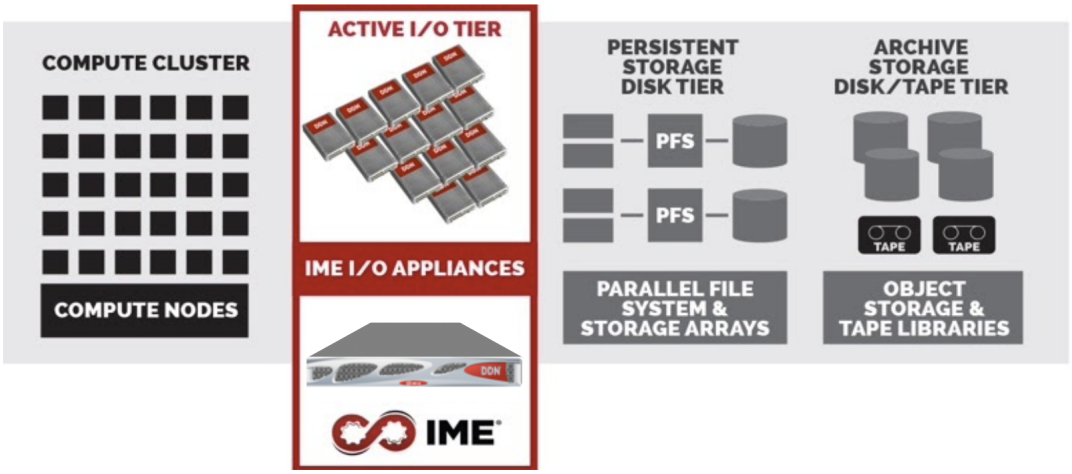


Figure 6: IME as part of the I/O path from compute nodes to parallel file system.

All compute nodes can access data on the IME servers. As described in the previous deliverable, there are two major methods that clients can utilize in order to interact with IME. Without any modification to the applications, they can benefit from IME using the IME fuse client. However,

the IME fuse client adds overhead which involves a system call and also the use of the fuse module. Figure 7 shows a few of the layers within the kernel space that have to be crossed to serve an I/O system call. Apart from the cost of the hardware operation itself, the software overhead is also significant. Fuse adds a copy of the data from user space to kernel space using a 128 KB buffer and also implies further locks.

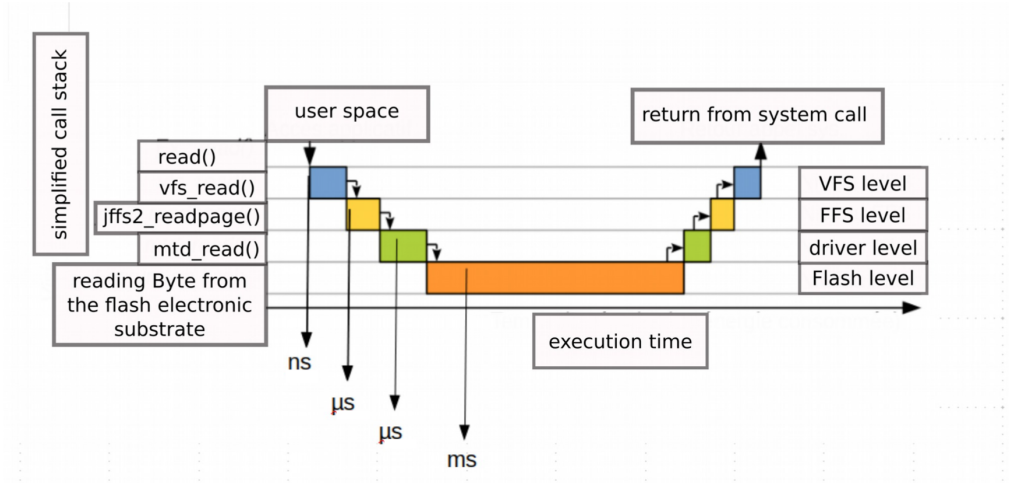


Figure 7: Summary of function calls and layers needed to serve an I/O call from user space up to storage device in respect to the execution time.

In addition to the system call overhead, any fuse file system implementation has to pay an extra penalty for crossing twice from user to kernel space and vice versa. As illustrated in Figure 8, any fuse call from the user application has to cross from the user space to kernel, is forwarded to the fuse kernel module, propagated to the fuse user space library, served from the user level implementation of the file system and finally returned through all the aforementioned layers. IME tackles this overhead by offering applications a custom I/O interface called *IME native*. This interface is very similar to POSIX. It connects the application directly with the IME client eliminating additional layers and minimizing overhead (see Figure 9). In this evaluation we demonstrate the benefits of IME native interface vs. IME fuse and we compare IME with the back-end file system without IME utilization.

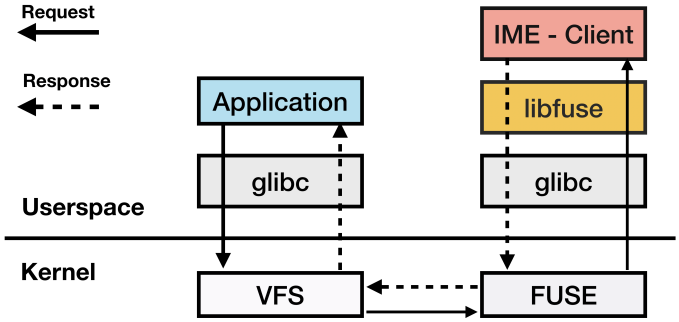


Figure 8: IME FUSE I/O path.

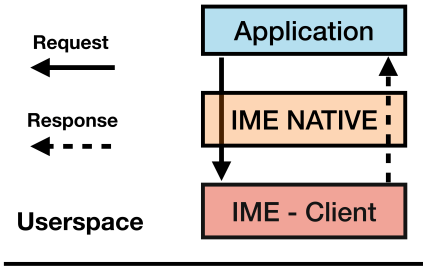


Figure 9: IME Native API I/O path.

IME offers a rich set of features including various fault tolerant capabilities. It supports data device failures with automatic rebuild to maintain parity and also server level fault tolerance. IME calculates parity blocks for bulk data transfers and distributes them across multiple IME servers to

offer data device fault tolerance. Moreover, IME can replicate its internal metadata among several IME servers, offering server level fault tolerance.

4.2.2 SIONlib

SIONlib is an I/O library that aims to provide a task local file view utilizing only a single or a small number of physical files. The main motivation of SIONlib is that metadata operations in parallel file systems are very costly and also that a single shared file access can lead to contention when writing to the same file-system block in parallel.

The file format of SIONlib is depicted in Figure 10. SIONlib metadata are stored at the beginning and at the end of the SIONlib physical file. Chunks are fixed sized file partitions that are assigned per task to provide the task local file view. Each chunk is assigned to a specific task, and each task can use more than one (depending on the amount of data that it would write). Chunks are aligned in file system block boundaries to avoid contention of tasks writing in the same file system block.

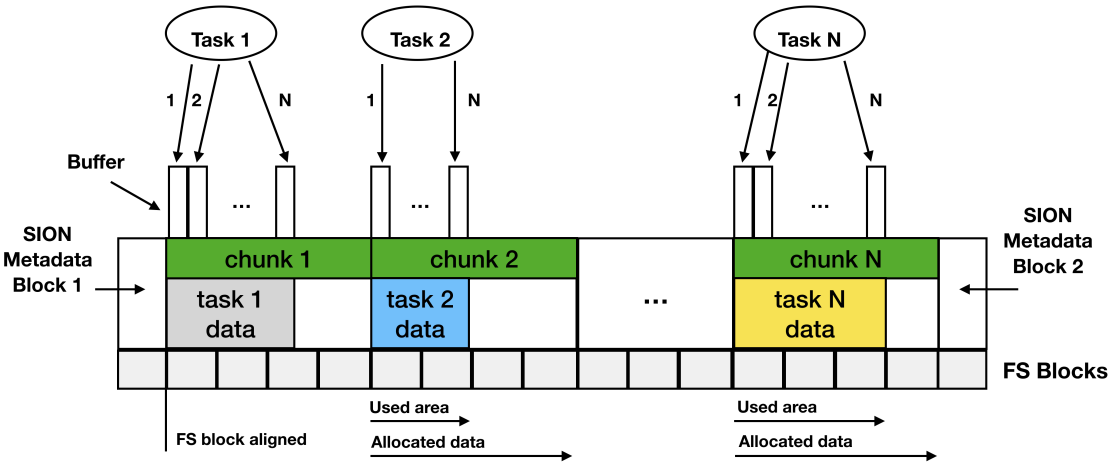


Figure 10: SIONlib physical file format.

4.2.3 Evaluation methodology

To perform our evaluation, we leverage *partest* a benchmark provided as part of SIONlib that can exercise different I/O patterns. *partest* can be parameterized in various options. For our experiments we exploit the following parameters:

- the type of I/O to be performed: single shared file (using the SIONlib format, shown above) or file per process,
- the chunk size: defines the contiguous file area assigned per task,
- the buffer size: specifies the amount of data transferred per I/O call (see Figure 10),
- the interface to be selected for the low level I/O calls: POSIX or IME native.

By experimenting with different chunk/buffer sizes, we are able to identify the sensitivity of them in relation to the low level interface used. We focus our evaluation on the write path, since this is the most demanding from the application perspective during the check-pointing phase, which will be the use case of the scientific challenge application GyselaX.

In our experiments we explore three different use cases:

- using the parallel file system Spectrum Scaler v5.0 (formally known as GPFS [11]) without the use of IME,
- the performance that IME delivers using the fuse interface, and also
- the improvement when using IME with the native interface and the new SIONlib integration.

4.2.4 Experimental Results

All the experiments were conducted on the JUWELS Supercomputer cluster of Jülich Supercomputing Centre [6]. The IME installation, so called High performance STorage Tier (HPST), which is funded partly through the ICEI project, consists of 54 IME servers that are configured with up to two node failures and 9 + 1 parity encoding. Each IME server consists of: i) Intel Xeon Silver 4108, 1.80 GHz ii) 96 GB DDR4 RAM iii) 10 x Intel 2TB NVMe PCIe 3.1 used for data storage iv) one Toshiba XG5 1TB single-ported NVMe SSD PCIe 3.1 used for IME metadata storage. The evaluation were performed as one of the first HPST users.

The compute nodes hardware platform is equipped with: i) dual socket Intel Xeon Platinum 8168 CPU, 2 x 24 cores, 2.7 GHz ii) 96 GB DDR4 RAM and iii) InfiniBand EDR (Connect-X4) allowing up to 12.5 GB/s network transfers.

For all the experiments we wrote 128 GB per node that is sufficient to fill caches and provides stable results across multiple runs. Since the system was tested during regular production both the parallel file system and the IME servers are shared with other jobs running in parallel. However, we do not expect a significant performance interference from that.

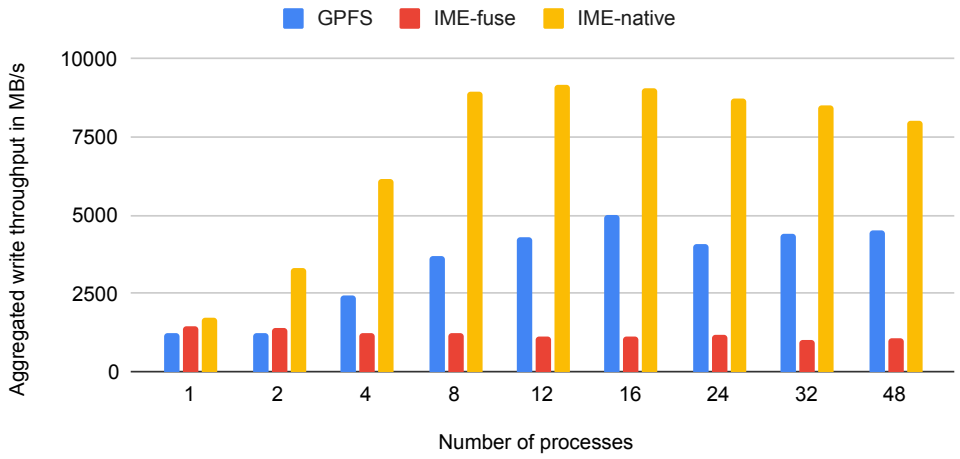


Figure 11: Aggregated write throughput per interface varying the number of processes (strong scaling) used for single node, single shared file with chunk size of 2 GiB and buffer size of 1 MiB.

In Figure 11 we present the aggregated write throughput reported by *partest* varying the number of processes (strong scaling) from 1 to 48 (corresponding with the number of cores in a single node). For this scenario, we utilized a single node and we operated in a single shared file (using the SIONlib format) with chunk size of 2 GiB and buffer size of 1 MiB (both chunk size/buffer size have been selected to deliver the highest throughput). As we can see with a single process, IME native is 18% faster than IME fuse and 40% faster than GPFS. GPFS is able to achieve the maximum performance of 5 GB/s with 16 processes, where IME native reaches 9 GB/s with the same configuration resulting in an improvement factor of 1.8. IME native performs best with 12 processes, giving 9.1 GB/s effective throughput that is only 30% less than the theoretical maximum that the network interface offers. This figure accounts also for the extra data blocks, sent due to

parity encoding and also the overhead of IME server metadata replication. Moreover, the IME system was currently in an initial tuning phase and we expect to reach even higher throughput, once this process has been completed. We also observed that the IME fuse performance does not increase when using more processes. This is due to a *mutex* lock within the fuse module that prevents parallel writes in a shared file. We are in discussions with the fuse module maintainers to provide a solution to this issue, but it has not yet been resolved.

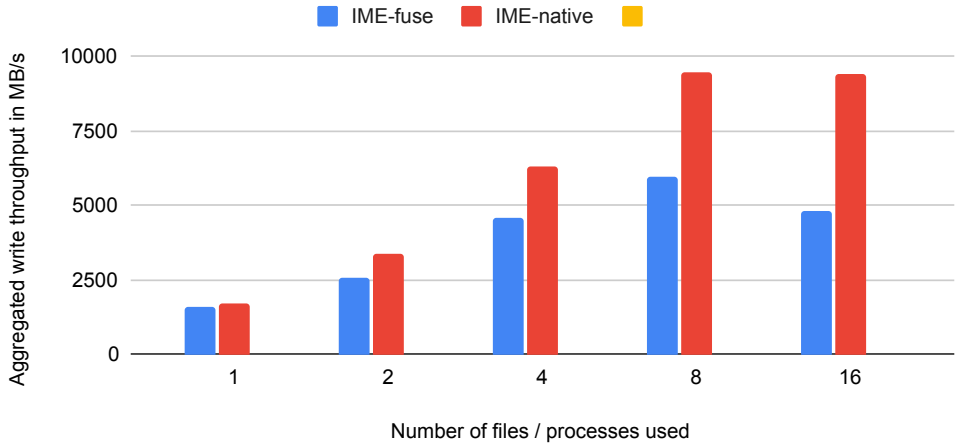


Figure 12: Aggregated write throughput varying the number of processes (strong scaling) used per interface for a single node, file per process with chunk size 2 GiB and buffer size of 1 MiB.

To provide a clear perspective of the IME fuse performance, that will not be impacted by the *mutex* lock of the fuse module that prevents parallel writes in a single shared file, we execute a similar test as above but with a private file per process rather than a single shared file. The chunk size (2 GiB) and buffer size (1 MiB) are exactly the same as before and we compare only IME fuse with IME native. As we can see in Figure 12, IME fuse performance significantly improves with a factor of around 1.7 when we move from 1 to 2 and 4 processes. It reaches the maximum of 5.8 GB/s with 8 processes and then it drops again. IME native outperforms fuse in all cases with a benefit of 10% for a single process up to almost double performance with 16 processes. We expect similar performance behaviour of IME fuse with the single shared file, when the *mutex* lock of fuse can be bypassed.

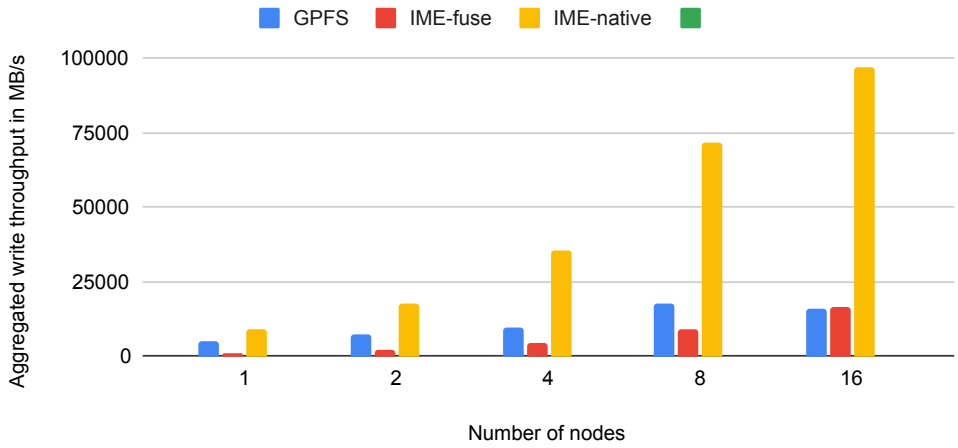


Figure 13: Aggregated write throughput varying the number of nodes used (weak scaling), single shared file with 16 processes per node, chunk size of 2 GiB and buffer size of 1 MiB.

In the next set of tests we explore the scalability varying the number of nodes used (weak scaling). We maintain the same amount of data per node like in the previous experiments (128 GB), the same number of processes per node (16) and the same chunk size (2 GiB) and buffer size (1 MiB). Figure 13 illustrates the aggregated write throughput varying the number of nodes used from 1 to 16 using a single shared file. IME native performance scales linearly with the number of nodes reaching 71 GB/s with 8 nodes, only 30% less than the theoretical maximum similar to a single node. It continues to deliver faster throughput with 16 nodes by a factor of 1.35 in comparison with 8 nodes. GPFS performance also improves when we move from one to two nodes by a factor of 1.48, but the trend does not continue when we move to 4 nodes with only an around 20% improvement. It reaches a maximum performance of 17.4 GB/s with 8 nodes and the performance drops to 16.6 GB/s when we double the number of nodes. IME fuse improves by a factor of 1.7 x ,when using two nodes, in comparison to a single node and doubles the performance as we double the number of nodes up to 16.

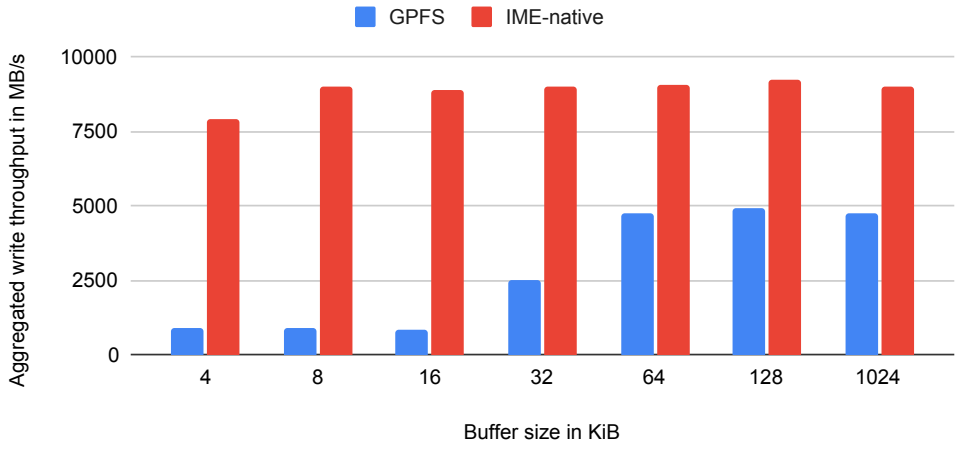


Figure 14: Aggregated write throughput varying the buffer size for single node, single shared file with 16 processes and chunk size of 2 GiB.

To evaluate the significance of buffer/transfer size, we run tests varying its size from 4 KiB to 1 MiB. The tests were executed in one compute node with 16 processes sharing a single file and

a constant chunk size of 2 GiB. The results of the aggregated write throughput are visualized in Figure 14. IME native is not impacted, when using small buffer sizes, and delivers max throughput of around 9 GB/s. On the contrary, GPFS requires at least 64 KiB buffer size to reach maximum throughput of 4.8 GB/s and performs poorly when using buffer size less than 32 KiB delivering at max 1 GB/s.

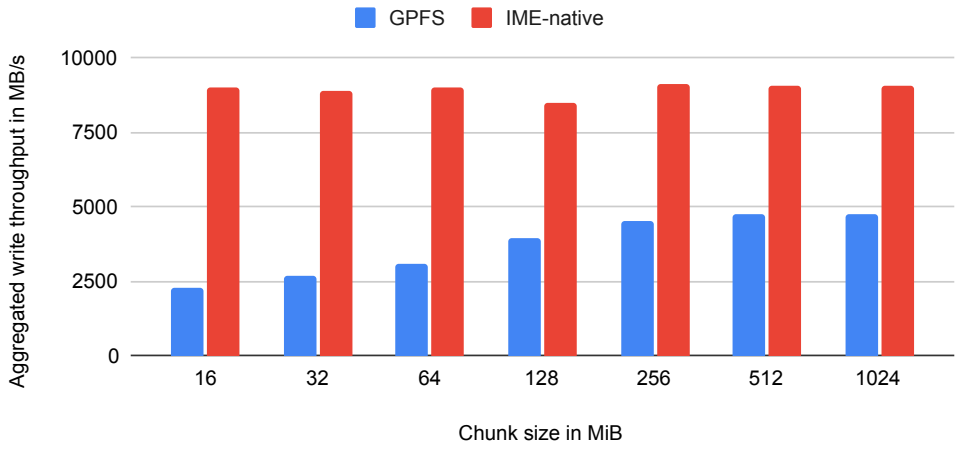


Figure 15: Aggregated write throughput varying the chunk size for single node, single shared file with 16 processes and buffer size of 64 KiB.

The importance of the chunk size is demonstrated in Figure 15. We vary the chunk size from 16 MiB to 1024 MiB while keeping a constant buffer size of 64 KiB. We retain the rest of the parameters to the same values as in the experiment above (single node with 16 processes that share one file in SIONlib format). Similar to the case above, the chunk size does not have any impact on the performance of IME native. However, GPFS doubles the performance when increasing the chunk size from 16 MiB to 256 MiB. With chunk sizes larger than 256 MiB, the performance of GPFS remains the same.

4.2.5 Conclusions

In all the experiments that we performed, IME outperforms GPFS. For the single node tests, IME delivers up to 1.8 x of the maximum performance of GPFS. In the multi nodes tests, IME can deliver more than 4 times the performance of GPFS for 8 nodes. Apart from that, we demonstrate that IME performance is not impacted by the buffer size or the chunk size in contrast to GPFS that smaller values in these parameters resulted in poor performance. Lastly, we see that IME native performs always better than IME fuse resulting in an improvement from 10% up to 90%. IME fuse does not yield scalable performance when using the single shared file (SION format) due to the mutex lock imposed by the fuse module.

The results provide a very good baseline when introducing Exascale aware I/O devices like the IME based cache infrastructure into the scientific challenge applications. Due to the SIONlib integration, a direct integration underneath of PDI is possible as well. The SIONlib integration was publicly released in SIONlib 2.0.0-rc.3[1]. This approach will now be evaluated for the GyselaX application in cooperation with WP1.

[1] `https://apps.fz-juelich.de/jsc/sionlib/docu/2.0.0-rc.3/release_notes_page.html`

5 Fault Tolerance

In the last deliverable (D4.1), we reported on the prototype integration of FTI into Melissa-DA of WP5 and Alya. We also reported on the implementation of a new feature into FTI, the asynchronous creation of shared HDF5 checkpoint files. In this deliverable, we report on our progress on the FTI-Melissa integration and we will present results of measurements we have performed to test its performance. We will begin by introducing the concepts and frameworks we use, to provide a basis for our work. We have worked tightly with the experts from WP5 on the FTI integration and we will continue collaborating by maintaining a weekly video meeting. The work presented in deliverable D4.1 is already integrated into the master branch of Melissa-DA and will be part of the release that will be published within the next weeks.

5.1 Background

In this section we will introduce the concept of data assimilation, the Melissa-DA framework and the concepts of elastic recovery and asynchronous checkpointing.

5.1.1 Data Assimilation and the Ensemble Kalman Filter

An essential part of climate research is to make predictions and reanalyses of environmental systems with numerical methods. The equations of climate models are typically highly non-linear, high dimensional, and very sensitive to initial conditions. To make accurate predictions, initial states with low uncertainties near to the true state are necessary. However, the observational data is sparse and afflicted with uncertainties. Consequently, observational data is not sufficient alone to provide reliable predictions. A mean to reduce uncertainty is data assimilation (DA). DA combines the observational data with results from the numerical model and computes the best estimate for the posterior conditional *probability distribution function* (PDF) $p(x_t|y_{1:t})$. The PDF essentially represents the minimized likelihood of the model state, conditioned on the available observations until time t [1]. Kalman filtering (KF) is among the most common techniques used for DA. The foundation of the formalism is represented by the state space equations:

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

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

Here, x_t represents the model state and y_t the observed state. q_t and r_t are the respective uncertainties and are represented by Gaussian distributions, thus, $\mathcal{M}$ and $\mathcal{H}$ are assumed to be linear operators. In climate models, however, these operators are in general non-linear functions, and hence associated with non-Gaussian distributions. The *Extended Kalman Filter* (EKF) uses the Taylor expansion to linearise the operators. However, the resulting operators are associated with high dimensional covariance matrices of state and observation vectors, and dealing with the full matrices is virtually impossible for very large systems. The *Ensemble Kalman Filter* (EnKF), uses approximates for the covariance matrices, represented by the statistical moments of an ensemble of states. The dimension of the state covariance matrix is significantly reduced and corresponds to the dimension of the *sample covariance matrix* of the ensemble. A detailed technical description of the EnKF formalism can be found in many textbooks and articles (for instance [7, 12, 4, 3]).

5.1.2 Melissa-DA

In numerical data assimilation, the ensemble of states is typically propagated between the two stages through the I/O layer, and forecast and analysis are executed separately (a.k.a. *offline mode*). Also common is executing forecast and analysis within the same binary, while propagating the states through the MPI layer (a.k.a. *online mode*). Both methods have certain advantages and

drawbacks. The first method possesses intrinsic fault tolerance, since each step can be restarted upon the stored ensemble, but, uses I/O for state propagation, which can oppose large overheads. The second method, on the other hand, does provide better performance, but misses the intrinsic fault tolerance and thus needs to be protected in another way towards failures. Melissa-DA (or briefly *Melissa*) takes an intermediate approach. As in the first method, the forecast and analysis steps run in separate binaries, however, are executed at the same time. The ensemble is propagated through the network using ZeroMQ instead of MPI. The framework provides some level of intrinsic fault tolerance without the need to store the states to the file system. Melissa is based on a server-client architecture. In the forecast step, the server acts as a task scheduler, scheduling the input state ensemble to the clients. The clients, called *runners*, are instances of the model simulation that wait for getting assigned a state from the server and thereupon compute the forecast. The number of runners can be chosen and is typically smaller than the ensemble size, thus, each runner will receive several states from the server. Once the forecast state ensemble is complete, the server performs the data assimilation and thereupon propagates the analysis state ensemble back to the runners. Figure 16 visualizes the concept. For a more detailed description of Melissa-da, we refer to Friedmann et al [5], or to the deliverables from WP5.

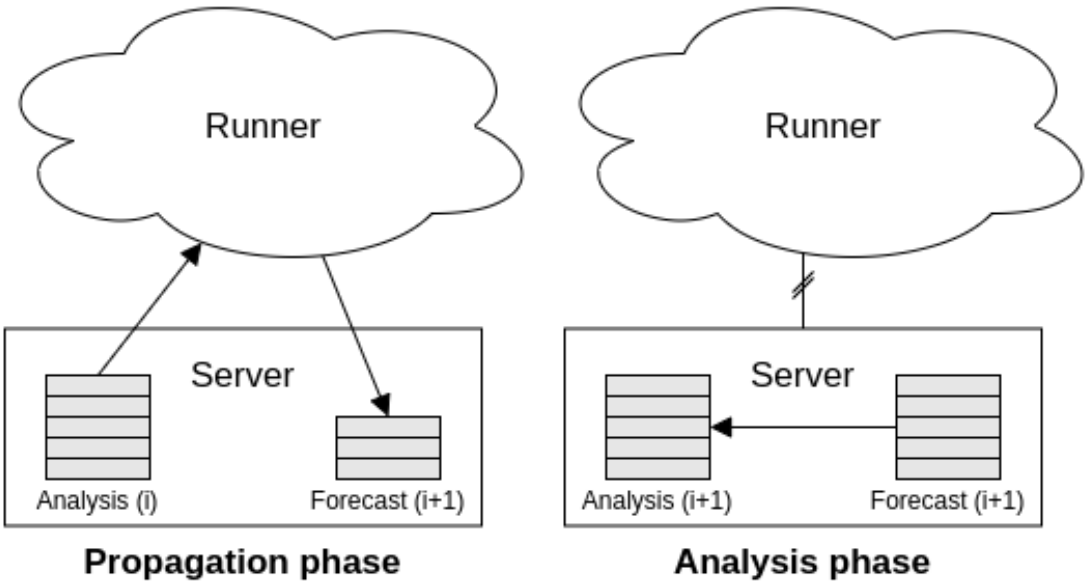


Figure 16: Server-runner concept of Melissa-DA. Each iteration, the server distributes the analysis ensemble to the runners, which in turn compute the forecast as input for the next data assimilation step.

5.1.3 Elastic Recovery

Elastic recovery refers to restarting from a checkpoint with a different number of processes as the checkpoint has been created with. Depending on the data structure inside the checkpoint file, this is not trivial. For instance, considering a multidimensional grid and a decomposition of the grid into hypercubes for each of the participating processes. Multiple datasets further increase the complexity of the problem. In order to enable elastic recovery from a shared file, one has to compute the offsets of the process local share of the datasets inside the checkpoint file in a general form, so that it applies for various domain decompositions. As we presented in D4.1, we have implemented new API functions into FTI that facilitate the application of elastic recovery from shared HDF5 files. We have published an article about our extensions to the FTI API and a study of elastic recovery at HIPC [8]. In Melissa, we leverage this functionality to recover the forecast states on the server. As we will explain later, the forecast states are stored by the runners, thus,

will be created with a different number of processes as will be used for the recovery on the server side.

5.1.4 Asynchronous Checkpointing

Here, asynchronous checkpointing refers to checkpoints which are performed in two stages while the second stage is performed *asynchronous* to the application using dedicated processes or threads. We refer to the first stage as *pre-processing* and to the second stage as *post-processing*. Asynchronous checkpointing can be applied for checkpoints that involve further actions besides storing the data to the file system, for instance in partner-checkpointing, where the checkpoint is stored locally on node storage, and a copy is send to the partner node. In Melissa, we use dedicated processes to perform the HDF5 checkpoint creation asynchronously. During the pre-processing, the checkpoint data is stored locally by the application processes. Afterwards, the FTI head processes consolidate the data to a globally shared HDF5 file on the *parallel file system* (PFS).

5.2 FTI - Melissa Integration

Our implementation involves modifications in all modules of the Melissa framework. Melissa comprises three modules:

- Launcher
- Server
- Runner.

In the next paragraphs we will gradually introduce the three modules along with our modifications.

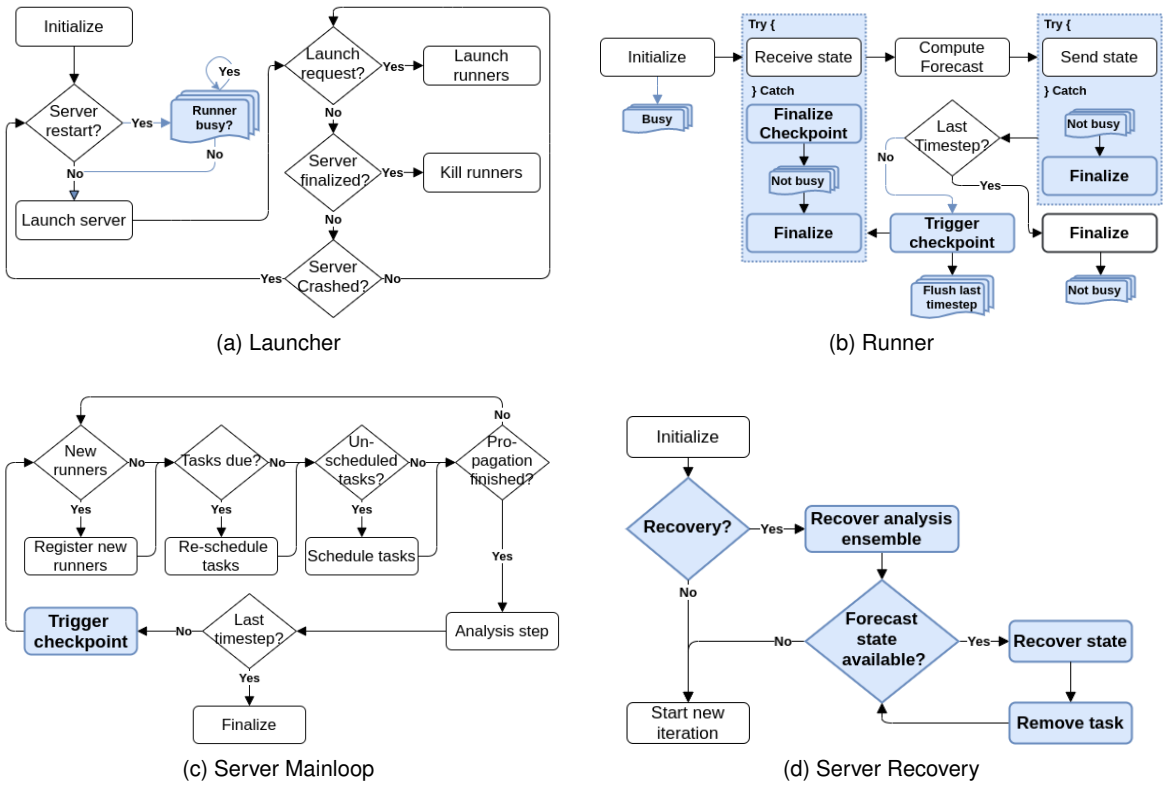


Figure 17: Flow charts of the Melissa-DA modules Launcher, runners and server. The elements in blue indicate our FT extensions to the framework.

5.2.1 Launcher

The launcher is the control unit of the Melissa-DA framework. It is programmed in python and comprises several classes. The main class constructor takes as arguments simulation parameters such as the number of ensemble members, number of server processes, number of runners, etc.. Upon construction, the launcher starts server and runner instances and remains waiting for events that need to be handled. The execution flow of the launcher is visualized in Figure 17a. The launcher takes an important role in the Melissa FT mechanism. The server and the launcher maintain a heartbeat connection, if the heartbeat does not arrive at some point, the launcher assumes a server failure and revokes all running Melissa instances (including all server and runner instances). Afterwards the framework is restarted. Before our additions, the launcher killed all instances, including the runners, as soon as the server failure has been recognized. We have extended this mechanism in order to allow the runners to complete the forecast on the current state. With this, depending on the point of the failure in the execution flow, the server can resume execution where it has been left off. We call this property *zero-waste recovery*, a recovery without re-computation. This works as follows. When the launcher detects a server failure, before triggering the recovery, the launcher checks for every runner id, if the respective runner is busy. The launcher waits until all the runners either set their status to *not busy*, or until a timeout passes. This assures that before the runners end after the server failure, the forecast state is stored and can be recovered by the server upon the restart.

5.2.2 Server

The server workflow is divided into:

- Propagation of states and
- Analysis (data assimilation).

During the state propagation, the analysis state ensemble is *propagated* to the runners to generate the forecast state ensemble, which in turn is *propagated* back to the server. When the propagation has finished, the server generates the next analysis state ensemble by assimilating the observations. Data assimilation in Melissa-DA is implemented using the *Parallel Data Assimilation Framework* (PDAF) [9]. The server implements two distinct FT mechanisms. The first mechanism protects melissa towards runner failures and the second one protects the server itself.

Runner failures The server orchestrates the propagation of the state ensemble using a task scheduler. Each task is assigned the state-id, the runner-id and a timestamp. The tasks have to be completed during a certain amount of time. Each iteration the server checks the due times of the tasks, reschedules tasks that have exceeded the time limit, and notifies the launcher that the runner is not responsive anymore. The launcher sends a kill request to the respective runner and launches a new one. The new runner sends a connection request to the server and the server incorporates the new runner in the team. With this, runner failures are handled during runtime. Figure 17c shows a diagram of the server mainloop and helps to comprehend the mechanism.

Server failures The server checkpoints the analysis state ensemble with FTI into the HDF5 file format. The state ensemble is checkpointed asynchronously, i.e., leveraging dedicated FTI head processes for the checkpoint post-processing (compare Section 5.1.4). The server processes write one file per process inline to the node local storage layer, and the FTI processes consolidate the local files to a shared file on the PFS in the background. We take advantage of the fact, that the analysis states are not changed during the forecast step and delegate the pre-processing to a thread, so that also the pre-processing stage of the checkpoint is performed asynchronously.

In that way we can hide the checkpoint cost completely behind the Melissa workflow. The pre-processing with FTI involves collective MPI calls, hence, care needs to be taken in handling the MPI communicators appropriately. It is crucial to firstly, initialize MPI using MPI_Init_threads, and secondly, duplicate the MPI communicator for the FTI processes. It is important, that the server and the FTI processes use different MPI communicator contexts. Furthermore, we applied the flag MPI_THREAD_MULTIPLE, to provide the highest level of thread support.

5.2.3 Runner

The workflow of the runners is simple. The simulation model implements the Melissa API to expose the model states to the framework and the API consists merely of the two functions:

- melissa_init
- melissa_expose

The call to melissa_expose is used to exchange the states with the server. The function essentially consists of a send and receive operation. On initialization, the runners receive the initial states of the ensemble and after the model simulation, return the forecast state. In the following the runner receive the analysis states and send back the next forecast state and so on and so forth. The runner perform a checkpoint of the currently processed forecast state between the send and the receive operation. In that way we can exploit the idle time when the runners wait for new states. The runner checkpoint only one state of the ensemble at a time, in contrary to the server which stores the full ensemble at once. Thus, the checkpoint on the runner side is significantly faster. The server is memory bound, the runners, on the contrary, are CPU bound and do not generally have spare cores on the nodes. However, we might have one spare core per node for dedicated FTI head processes. Thus, we do not enable checkpoint threads for the runners, however, support FTI head processes. To provide overall fault tolerance for the framework, it is sufficient to checkpoint the analysis state ensemble on the server side. However, to achieve zero-waste checkpointing, and to reduce re-computations for the case we need to roll back to a former application state, we need to store the forecast as well. We will come back to the different failure scenarios in Section 5.2.5. Figure 17b shows a diagram of the runner workflow. We have wrapped the send and receive operations in try and catch blocks. Upon a server failure, an exception is thrown from ZeroMQ. The exception is caught and the runners shutdown gracefully after completing the forecast and the checkpoint of the state.

5.2.4 Recovery

After recognizing the server failure, the launcher initiates the recovery of the framework. As we explained in Section 5.2.1, the launcher waits for the runners to complete the current state's forecast and checkpoint. Afterwards the runner terminate and the launcher starts a new server and several runner instances. The server initially recovers the last available analysis state ensemble. Afterwards it recovers the available forecast states. As the forecast state checkpoints have been created by the runners, the recovery of them on the server side takes place with a different number of processes, hence is elastic. Figure 17d shows the recovery diagram.

5.2.5 Failure Scenarios

There are essentially three different failure scenarios for the server:

A The failure happens during the propagation phase, before the checkpoint has been completed,

B the failure happens during the propagation phase, after the checkpoint completion, and

C the failure happens during the analysis phase.

The zero-waste scenario is B. In this case, we recover to the same application state as we have left off. The worst case scenario is A, in which we need to rollback to the beginning of the analysis from the previous iteration. We do not need to repeat the propagation, since we have the complete forecast ensemble checkpointed by the runners. In scenario C we have to rollback to the beginning of the analysis from the current iteration. Figure 18 summarizes the scenarios graphically.

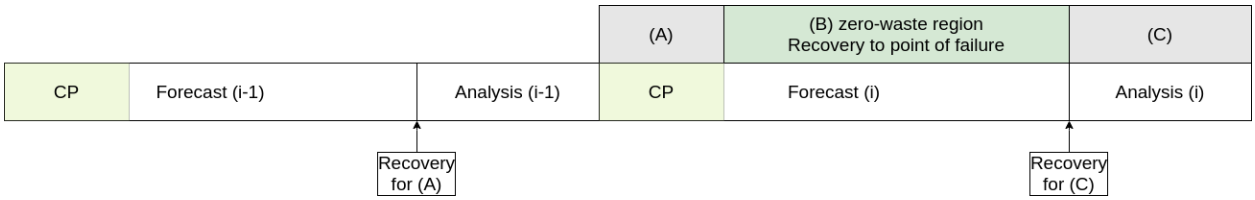


Figure 18: The 3 characteristic regions where the failure can take place. Failures in region A result to a rollback to the end of the propagation of iteration $i-1$. From failures in B can be recovered to the point where the failure has happened (zero-waste recovery), and for failures in C we need to rollback to the end of the propagation of the current iteration.

5.3 Evaluation of the FTI - Melissa-DA integration

5.3.1 Terminology and Data Assimilation Parameters

We measured the checkpoint overhead using the following FT parameters on the server side:

- Server using checkpoint threads **T1**
- Server not using checkpoint threads **T0**

The server has been implemented using FTI heads always. The following parameters were used for the runners:

- Runners using FTI head processes **H1**
- Runners not using FTI head processes **H0**

The experiments without protection are labeled **NOFTI**. All the experiments have been conducted with 64, 128, 256, and 512 ensemble members. The parameters of the experiments are listed in Table 4

Parameter	Value
State dimension	1024*1024*1024
number of observations	21474 (0.002%)
State size	8 Gb
Observation size (Kb)	168 Kb

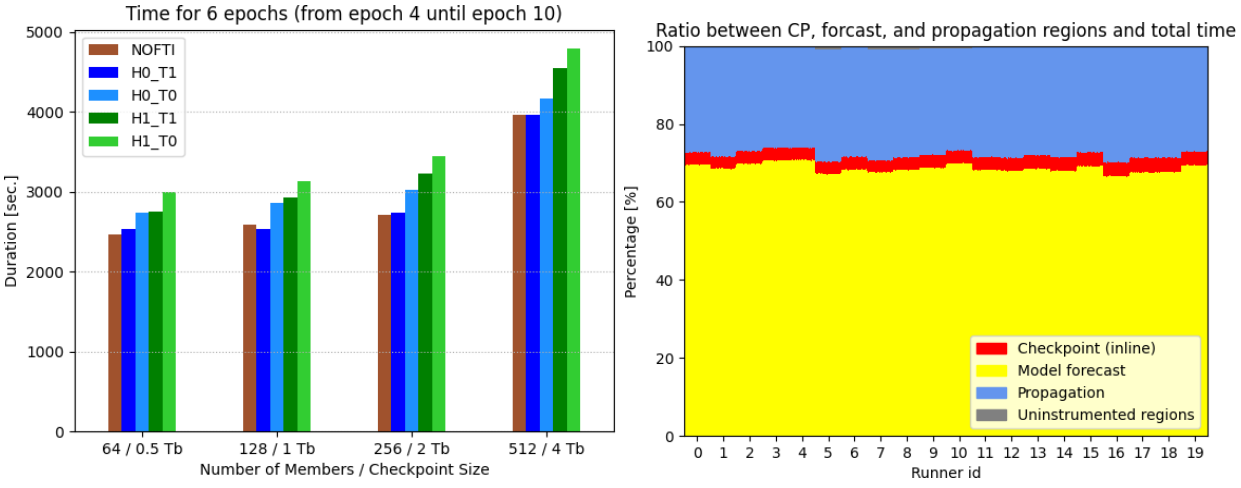
Members	Checkpoint	Nodes
64	0.5 Tb	48
128	1 Tb	96
256	2 Tb	192
512	4 Tb	384

Table 4: Parameters for the experiments (left) and scale of the experiments (right).

5.3.2 Results

Leveraging checkpoint threads on the server side, we can effectively hide the checkpoint cost. This can be seen in Figure 19a, where we listed the runtimes for the different experiments. The times

for the experiments without FTI (NOFTI) and those with server threads (H0_T1) are essentially the same at all scales. In Figure 19b we can see the relative amount of time spent in forecast, checkpoint and propagation during the runner execution. As we can see, the time for checkpointing (in red) is very low, it accounts only for about 3% of the runtime. Moreover, the checkpoint cost is partly hidden when the runners wait to receive a new state from the server. Thus, the checkpoint cost is effectively even smaller than 3%. Leveraging FTI head processes should speed up the checkpointing for the runners even further, however, we can see in Figure 19a, that this is not the case (compare experiments H1_T0 and H1_T1). To investigate this unexpected result, We have conducted experiments that simulate the FTI head processes by splitting the MPI communicator of the runners. Furthermore, we have implemented a dummy model with a sleep statement to exclude any influence of our model on the issue. With this configuration, we arrive to the conclusion, that splitting the communicator is inducing the drop in performance. However, we have not yet been able to find the cause of it. We will continue working on resolving this issue in the next months. We expect to improve the performance even further once we have eliminated the bug. In parallel the approach will be tested within one of the Melissa enabled scientific challenge applications such as ParFlow.



(a) (b) 100% stacked (512 members)

Figure 19: (a) shows the durations from beginning of epoch 4 to the beginning of epoch 10 (6 epochs) for the different experiments and scales. (b) shows a 100% stacked plot for the runner execution, resolving the ratios of total time for model forecast, propagation and checkpoint to total execution time. The plot shows the first 20 runners for the 512 member experiment without FTI head processes for the runners (H0_T1).

References

[1] 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.

[2] Tricia Balle and Pete Johnsen. Improving I/O Performance of the Weather Research and Forecast (WRF) Model. *Cray User Group*, 2016.

[3] 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.

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

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

[6] Jülich Supercomputing Centre. JUWELS: Modular Tier-0/1 Supercomputer at the Jülich Supercomputing Centre. *Journal of large-scale research facilities*, 5(A135), 2019.

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

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

[9] l. nerger, w. hiller, and j. schröter. *pdaf - the parallel data assimilation framework: experiences with kalman filtering*, pages 63–83.

[10] Wolfram Schenck, Salem El Sayed, Maciej Foszczynski, Wilhelm Homberg, and Dirk Pleiter. Early evaluation of the "infinite memory engine" burst buffer solution. In *International Conference on High Performance Computing*, pages 604–615. Springer, 2016.

[11] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[12] 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.