Horizon 2020
European Union funding
for Research & Innovation

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

D4.4
Final report on I/O improvement for EoCoE codes

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.eocoe.eu
	Deliverable ID:	D4.4
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M42 30/06/2022
	Actual Date of Delivery:	M42 30/06/2022
	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 :	Final report on I/O improvement for EoCoE codes
	ID :	D4.4
	Available at:	http://www.eocoe.eu
	Software tool:	LaTeX
Authorship	Written by:	Julien Bigot (CEA), Kai Keller (BSC), Yen-Sen Lu (FZJ), Sebastian Lührs (FZJ), Christian Witzler (FZJ)
	Contributors:	Jean-Thomas Acquaviva (DDN), Massimo Celino (ENEA), Konstantinos Chasapis (DDN), Agostino Funel (ENEA), Leonardo Bautista Gomez (BSC), Francesco Iannone (ENEA), Miroslaw Kupczyk (PSNC), Silvio Migliori (ENEA), Yacine Ould Rouis (CEA), Karol Sierocinski (PSNC)
	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

This deliverable provides details on the overall outcome of the I/O and data-handling-related activities within the EoCoE-II project. Like in former deliverables of this work package, the report focuses on four primary I/O objectives: Improvement of I/O accessibility, improvement of I/O performance, resiliency handling and overall data size reduction. In addition, this report also highlights the links between the technical I/O improvements and the various scientific challenges (SC) of EoCoE-II. On the one side, the deliverable will report all remaining results which were not been reported so far and on the other side will summarize the overall work package outcomes by creating links to the former deliverables.

In the context of I/O accessibility, the PDI Data Interface (PDI) was the main tool which was heavily improved during the runtime of EoCoE-II. Besides SC-driven new features, such as the support of NetCDF files, the library was also significantly improved in the context of its technology-readyness level and the first stable major release was made publicly available. Being a generic approach to treating I/O data handling, the library can also help many other HPC applications outside of EoCoE-II to separate their simulation and data demands. In this report, we mainly show the advantages of using PDI underneath the GyselaX application as part of the Fusion SC, where PDI helped to utilize HDF5 efficiently. Also the latest library improvements, for example, the close work together with work package 5 to support a PDI Melissa backend is presented as well.

For the I/O optimization activity, we worked on software solutions but also leveraged new types of storage hardware solutions. With GyselaX we evaluated and benchmarked the benefits of the latest PDI integration. For ESIAS-met we followed on the work to utilize asynchronous writing capabilities together with having an ensemble approach for the WRF backend underneath. Finally, with Gysela and ParFlow we tested the capabilities of the High Performance Storage Tier (HPST), a global storage cache layer at JSC, and succeeded in having a significant performance impact on such a type of intermediate storage system.

The presented work on fault tolerance and resiliency handling in this deliverable mainly focuses on ensemble-driven applications, such as ESIAS-met. Here a close collaboration with work package 5 allowed us to implement an efficient particle filter solution where WP4 took mainly care of the resiliency aspects.

The final section will sum up the in-transit and in-situ work, where in-transit compression, as already presented in the last deliverable, helped to circumvent I/O bottlenecks in ParFlow and in-situ visualization helps to avoid writing data to disk. The approach of the designed in-situ visualisation pipeline, which also utilizes PDI on the application side, was successfully tested with Alya and ParFlow.

Overall with the different activities we created multiple direct links to the Meteo, Water, Fusion and Wind scientific challenges but also developed transversal solutions which could be utilized outside of the frame of EoCoE-II.

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 (CEA)
INRIA:	Institut national de recherche en informatique et en automatique
IRFM:	Institute for Magnetic Fusion Research (CEA)
MdlS:	Maison de la Simulation (CEA, CNRS)
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
SC:	Scientific Challenge
SSD:	Solid State Drive
PF:	Particle Filter
PFS:	Parallel Filesystem
WP:	Work Package

3 Introduction

This final deliverable of work package 4 (WP 4) the I/O and data handling work package of EoCoE-II provides an overview of the overall outcome of all the I/O related activities and their benefit for the different SC applications. The deliverable is a direct follow up of the former WP4 deliverables [3] and [4], which covers the technical aspects of the different activities in more detail.

Similar to all former deliverables the sections of this deliverable are divided alongside the four major tasks of the work package: First in section 4 all improvements on the data interface PDI and the support work towards the SC applications are shown. Section 5 covers the work related to I/O refactoring and optimization mainly in the context of leveraging I/O library capabilities as well as the utilization of Exascale aware I/O hardware approaches with tests on the intermediate high-performance cache layer. Section 6 focuses on the fault tolerance handling, mainly in the context of ensemble aware applications and the particle filter development work. Finally, section 7 describes the overall outcome of in-situ visualisation activities.

The individual impact on each SC is given in section 3.2.

3.1 Link to other work packages

The work done in the context of the deliverable is directly connected to other work packages of EoCoE-II: The main scientific challenge applications, handled by WP1, which benefit from the optimization work, are GyselaX, ESIAS-met, ParFlow, Alya and SHEMAT. The fault tolerance work as well as the PDI integration work for ensemble-based applications was done in close collaboration with WP 5. In cooperation with the scientific challenges, the integration of PDI was handled as part of WP 2 with the direct support of members of WP 4.

3.2 Impact overview on SC codes

The following table 4 provides an overview of the different activities if these were directly connected to a particular SC. All of these activities were finalised and are reported in the current or former deliverables, given by the individual references to a section within this deliverable or the reference to a former deliverable. In addition to these direct connections, additional transversal I/O related activities were implemented and reported as well in the following sections. The Material SC is not covered by the table as there were no direct interactions planned based on the original proposal.

	Meteo SC	Fusion SC	Water SC	Wind SC
Data Interface		• GyselaX PDI integration user support: 4.1 • GyselaX I/O improvement: 4.3.1	• ParFlow PDI integration user support: 4.1 • SHEMAT PDI integration user support: 4.1 • NetCDF I/O format support for ParFlow: 4.3.2	
I/O refactoring and optimization	• Asynchronous I/O in ESIAS-met: 5.2	• GyselaX I/O benchmarking: 5.1.1 • GyselaX cache storage utilization: 5.3	• ParFlow cache storage utilization: 5.4	
Fault tolerance	• Resilience for ensmble applications: 6			• FTI Integration in Alya: [1]
In-situ & in-transit data manipulation			• In-transit compression ParFlow: 7.1 • In-situ visualization ParFlow: 7.2.3	• In-situ visualization Alya: 7.2.2

Table 4: Overview of WP4 and SC interactions

3.3 Code locations

The download location of the main (non-third-party) tools and libraries, which were extended and used in the frame of WP4, are given in the following table:

Tool/ library	Webpage
PDI	`https://pdi.dev/`
FTI	`https://github.com/leobago/fti`
SIONlib	`https://fz-juelich.de/jsc/sionlib`
Melissa-DA	`https://gitlab.inria.fr/melissa/melissa-da`

Table 5: Overview of WP4 libraries and tools

4 Data Interface

This section focuses on the PDI Data Interface (PDI)[1] primarily developed by CEA and PSNC to ensure separation of concerns in data handling matters. PDI offers a declarative API to a) allow applications to expose memory buffers where they store their data and b) identify when significant steps are reached in the simulation. I/O operations involving these buffers are specified in a dedicated file (specification tree) loaded at runtime instead of being interleaved with the simulation code. A plugin system makes existing libraries available to implement these operations, potentially mixing multiple libraries in a single execution.

This approach supports the modification of I/O strategies without even recompiling the simulation codes. The potential changes can go as far as the replacement of the libraries used for I/O or the replacement of file I/O by in-situ data processing. This completely decouples high-performance simulation codes from I/O concerns. It enables I/O optimization specialists to improve code performance without impacting the original developers and the readability of the code. Overall, this greatly improves code portability, maintainability and composability.

The "Data Interface" task within WP 4 deals with a) the support for integration of PDI in SC applications, b) the development of new features required by SC applications, and c) the overall improvement of PDI core to support this. This is done in close cooperation with WP 2 where the integration of PDI in each SC application is handled. The remaining of this section offers an overview of the work that was implemented in the framework of EoCoE-II, where also more details were already reported in [1] and [3].

4.1 User and application support

WP 4 does not handle the integration of PDI in SC codes per se. This is instead done in dedicated tasks of WP 2 since it requires a deep understanding of the codes. PDI developers do however offer dedicated support to the users tackling this work. One-to-one support is provided by email, through the PDI community slack channel[2] and via in-person or remote visio co-working sessions. As already reported in previous deliverables, direct support has been provided for the integration of PDI in SC codes including GyselaX, ParFlow, and SHEMAT. Overall, this support can be broken down into about 1 person month of dedicated in-person or remote visio co-working sessions, and a difficult to measure time for continuous presence on Slack and email support during opening hours with more than 49k messages exchanged.

Providing this one-to-one support has the side benefit of easing the identification aspects of PDI that are not automatically clear to users. This is then a great asset to provide better documentation and support to the overall PDI community. During EoCoE-II, the content of the PDI documentation website (`https://pdi.dev/master/`) has been much improved, with new sections for installation, usage in user projects and examples. All existing documentation has been updated to match the new developments. Many presentations of PDI have been provided to increase awareness amongst HPC developers and trainings are offered through the PRACE training centre initiative. These aspects are covered in more detail in deliverables D6.3 and D6.4.

In addition, the direct interaction with code developers and PDI integrators offers a unique opportunity for code/library co-design with the ability to identify the aspects and features to add to PDI for maximum application impact. One such aspect identified as a priority by users in the project is related to the technology readiness level. While many dedicated libraries have been developed in collaborations with HPC application developers, these often remain prototypes. As a result, application developers often favour the direct integration of new features in their code instead of relying on libraries whose future is unclear.

[1] `https://gitlab.maisondelasimulation.fr/pdidev/pdi`
[2] `https://join.slack.pdi.dev/`

Since PDI aims to go beyond a research proof-of-concept and to be usable in serious production settings, a huge effort has been deployed to ensure a high library quality in parallel to the development of new features. This is only possible through strict testing and continuous integration policies. As reported in previous deliverables, tests are developed together with each new feature, and about 1000 unit and functional tests are now run for each change to the library in 25 different environments. In addition, micro-benchmarks are now also executed to ensure no performance regression sneaks into the code. This minimizes the risk of bugs occurring and makes a quick reaction possible.

This level of testing is achievable only thanks to the development of an advanced deployment process for PDI that was initially developed in reaction to user demand. An advanced PDI source distribution automatically detects missing dependencies and builds them alongside PDI and its plugins. Many Docker containers[3] are provided to offer different user contexts. Spack recipes[4] are offered to ease deployment on supercomputers. Binary packages[5] are provided for many Linux distributions including Debian, Ubuntu, and Fedora.

4.2 Library core overall improvements

While the PDI plugin-based architecture (Figure 1) means that most new features could be introduced in dedicated plugins, some of them are transverse and were developed directly in the PDI core. This does of course include all bugfixes that were implemented during the EoCoE-II lifetime, but also a lot of code re-factoring and internal improvement made to ensure no technical debt is accumulated. Thanks to this approach, new features can still be implemented and the maintainability of the library itself remains high. This does also include the "support for complex data structure in PDI", and "PDI plugins interface unification" activities, as well as work done all over PDI to improve error messages, logging and user information.

4.2.1 Improved data model

PDI data model is at the core of the library. The dynamic type description system makes it possible to interface user code and plugins relying on distinct languages. At the beginning of EoCoE-II, the PDI data model was somewhat limited. As reported in previous deliverables, it has been greatly improved and gained support for new data types along with the needs of SC codes.

In addition to the native language types, scalar data types matching C and C++ standard library types have been added. This makes the interfacing of codes relying on them much more portable. Pointer, record and tuples datatypes have also been added. These datatypes can be described by the user to match complex data layouts used in their codes including Fortran, C & C++ pointers, Python objects, Fortran derived types, C and C++ structs or C++ classes.

User-defined data types have been added to PDI to let users define a type once and use it multiple times. This is especially useful in combination with the new concept of user attributes. Somewhat similarly to Java annotations or

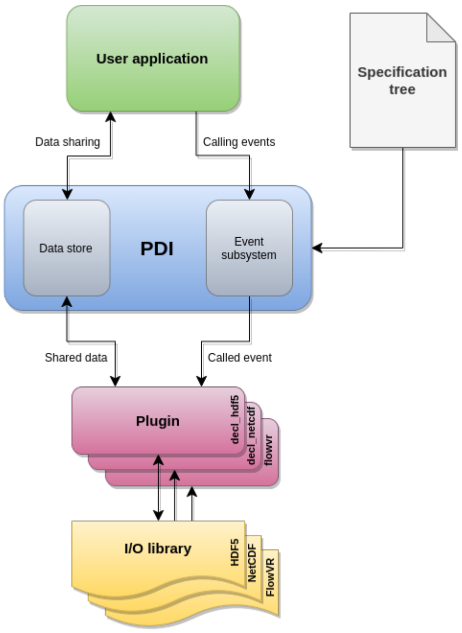


Figure 1: PDI Architecture overview

[3]https://github.com/pdidev/dockerfiles/
[4]https://github.com/pdidev/spack/
[5]https://github.com/pdidev/pkgs/tree/repo/

```
#pragma pdi on

typedef struct Var8
{
#pragma pdi type : int64
    int my_int;
    char char_tab[20];
    char my_char;
} var;

#pragma pdi size:[42]
int **array_of_pointer_of_array[21];

var my_var;

#pragma pdi off
```

Listing 1: Example of PDIC annotations

Python decorators, attributes can be used to extend types and values with user-specified information. This can then be used directly from PDI-script language or the plugins and will be further described later.

PDI `$-expression` system has also received improved support for the new types. Access to record members, array indexing and slicing, as well as pointer indirection, is now possible directly from YAML. Transtyping has been improved with the ability to evaluate string boolean values or to transform MPI Communicators exposed in one language to the type expected in another for better Fortran / C, C++ / Python interoperability.

Support for PGAS-style distributed array views is now also available. This is especially useful for plugins that require a global view of the distributed data such as Deisa discussed later in section 4.3.4. The introduction of this concept at the level of PDI ensures that very little change is required from the user to change the plugin (e.g. from I/O with parallel HDF5 to in-situ analysis with Deisa) once the data is described.

Specific care was given to the Fortran API now that supercomputers offer an environment supporting the Fortran 2008 norm. The user API was simplified with automatic type extraction for arrays thanks to the support of assumed rank data arguments. The library code itself was slightly simplified and redundancies were reduced by the use of assumed types.

Finally, an external contribution by Kevin Barre, made possible thanks to the free/libre and open-source nature of PDI, led to the availability of PDIC. PDIC is a CLang-based compiler that handles PDI-dedicated annotations like those illustrated in Listing 1. These annotations are then used to extract data types without having to specify any redundant information in the YAML file, hence limiting the verbosity of this file by a great amount. For example, the annotations of Listing 1 replace the long YAML presented in Listing 2.

4.2.2 Interface improvement and rationalization

A second aspect handled during EoCoE-II is the effort for the improvement and rationalization of the interface. Indeed, while the plugin-based approach of PDI enables one to implement many features independently, it can lead to incoherent interfaces if no specific care is taken. The experience with PDI integration in SC codes has shown that this can be a limitation to the replacement of an I/O strategy and an important uniformization effort has been undertaken during the project. This aspect has been well covered in previous deliverables already, and the reader is encouraged to refer to these for further detail.

The C API has been streamlined by introducing a single function (`PDI_multi_expose`) whose call replaces a sequences of calls to `PDI_expose` enclosed in the now deprecated `PDI_transaction_begin` and `PDI_transaction_end`. The extension of the PDI type system to handle distributed data has

```
structs:
 Var8:
  type: record
  name: Var8
  alias: [var]
  fieldsize: 3
  buffersize: 28
  packed: false
  members:
   char_tab: { type: array, subtype: char, size: 20 }
   my_char: { offset: 24 , type: char }
   my_int: { type: int64 }
data:
 array_of_pointer_of_array: { type: array, subtype:  { type: pointer, subtype:  {
    type: array, subtype: int, size: 42 } }, size: 21 }
 my_var:
  type: record
  name: Var8
  alias: [var]
  fieldsize: 3
  buffersize: 28
  packed: false
  members:
   char_tab: { type: array, subtype: char, size: 20 }
   my_char: { offset: 24 , type: char }
   my_int: { type: int64 }
```

Listing 2: YAML generated by PDIC for the annotations of Listing 1

moved a lot of plugin-specific interfaces to the shared PDI interface. This is also the case with user attributes. Thank to the combination of all these elements, the Decl'NetCDF and Decl'HDF5 plugins can now offer nearly identical interfaces.

A configuration validation script has also been developed to check the syntactic and semantic validity of the YAML file. A hook system provided to the plugin ensures that the YAML syntax and semantics are valid including in the plugin part before even launching PDI-enabled codes.

4.2.3 Better user information reporting

Another critical point identified by the users is the requirement for better information reporting to better understand exceptional situations, errors and problems. A huge effort has been undertaken during the last part of the project toward this goal.

The logging format and verbosity of PDI can now be tweaked from YAML. It can be specified globally or separately for each plugin thanks to an inheritance system. The set-value plugin that supports PDI-script features has gained the ability to dynamically change these aspects to let the user focus on logging on a specific moment in time during execution.

Many additional error messages have been added or improved. The addition of user-relevant context to messages, that previously included technical information difficult to exploit, required a lot of development but has made the library much more pleasant to use. For example, dereferencing a null pointer now provides the user with the name of the variable whose invalid access lead to the error. Invalid content written in the YAML file by users is now reported with information about the line and column where it was found.

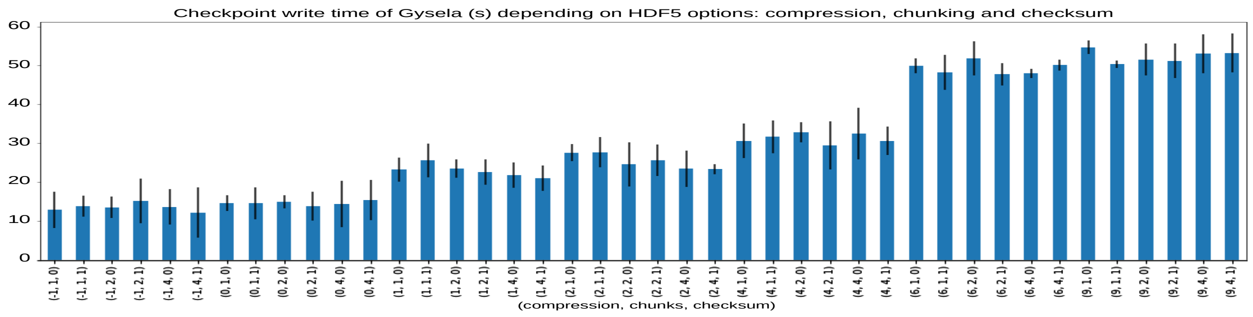


Figure 2: Impact on the write-time of various values for three options exposed by Decl'HDF5: deflate (compression), chunking or checksum. This parameter sweep used in the performance evaluation further discussed in Deliverable D2.3 and Section 5.1.1 of this deliverable has been made really easy to perform thanks to the new options introduced.

4.3 Data handling features in plugins

Finally, the last category of features developed during the EoCoE-II project is directly related to data handling, in plugins. This includes improvement to plugins that pre-existed before EoCoE-II, such as the Decl'HDF5, FTI or MPI plugins as well as the development of new plugins from scratch, such as the Decl'NetCDF, Deisa, FlowVR, Melissa, or Sensei plugins. This work was undertaken in part to implement the"PDI NetCDF capabilities", "PDI In-situ capabilities", and "PDI Melissa capabilities" activities.

4.3.1 Existing I/O solutions improvement

The first category of improvements covers new I/O features that were implemented in existing plugins to offer more extensive support for the interfaced libraries. For example, the FTI plugin keeps evolving to follow the features offered by the FTI library. The latest version is tested against the latest version of FTI (1.6 at the time of this writing).

The Decl'HDF5 plugin has been extensively reworked. It now supports a new, more expressive configuration format in the specification tree with simplified default values for ease of use in common cases. This new format supports memory and dataset selection that can be used to handle parallel access to files for distributed data.

Direct support for record datatypes has been added to the plugin (C structs, Fortran derived datatypes or C++ classes). These can now be stored directly in HDF5 compound data types. HDF5 attributes are now also supported and values can be set for those from the plugin.

Various HDF5 I/O options have also been exposed to YAML through Decl'HDF5, such as deflate (compression), chunking or checksum. This made it possible to easily evaluate the impact of these options on the codes. This is for example illustrated for the GyselaX code in Figure 2 and is also evaluated in section 5 from the benchmarking point of view to judge the overall performance impact.

The Decl'HDF5 plugin also gained support for configurable behaviour when the file or dataset in a file to write already exists. Following HDF5 behaviour, it now also supports writing data from memory to files with different shapes as long as the overall number of elements matches. It supports accessing metadata information about some datasets before accessing their content, for example, to dynamically allocate the memory required for this content.

The MPI plugin was also improved. Support for trans-typing opaque MPI types that have a different representation depending on the language used was added. Support for rank-dependant configuration has been provided.

4.3.2 NetCDF I/O format support

A new Decl'NetCDF plugin has been developed. It offers a feature set very similar to that of Decl'HDF5 but wrapping the NetCDF library instead of HDF5. The organization of the specification tree for both plugins is similar enough that switching from one plugin to the other is easy.

One difference between Decl'HDF5 and Decl'NetCDF plugins is that types and dimensions are implicit in HDF5 but have to be explicitly stored and named in NetCDF. Thanks to the addition of user attributes, this can be handled nearly transparently. A `decl_netcdf.type` attribute is simply added to the type and the plugins handle its creation and storage automatically. This is illustrated by the example in Listing 3.

```yaml
data:
  particle:
    type: struct
    +decl_netcdf.type: \
         particle_xyz
    members:
    - x: double
    - y: double
    - z: double
plugins:
  decl_netcdf:
  - file: "file_name.nc"
    write: [particle]
```

Listing 3: Example of NetCDF type name attribute

4.3.3 Ensemble run support

Ensemble runs simulations offer an interesting approach to leverage Exascale hardware. To efficiently implement such simulations, for each ensemble member, its disk-I/O in a stand-alone simulation must be replaced by network interactions with an integration server. Melissa is such a server developed in WP 5.

A PDI plugin has been developed to offer transparent access to Melissa. Thanks to this plugin, applications designed for stand-alone execution can be integrated into an ensemble without modification or even recompilation required. The plugin comes in two flavours. The Melissa-SA plugin offers access to the sensibility analysis variant of Melissa that requires read-only access to the data. The Melissa-DA plugin offers access to the data-assimilation variant of Melissa that requires read-write access to the data. Both flavours are publicly available in the PDI git repository and are planned for inclusion in the next PDI major release.

4.3.4 In-situ data analysis support

Multiple plugins dedicated to the in-situ analysis of data have been devised. This is especially useful with the increasing performance gap between compute and storage resources in supercomputers. Smart data reduction strategies become a must-have to reach Exascale without storage bandwidth being the bottleneck.

In the simplest case, the Set-value plugin offers access to a minimal built-in scripting language (PDI-script) built around `$-expressions` and directly embedded in YAML. In the lifetime of the EoCoE-II project, the plugin has been enhanced with support with better, Python-style formatting based on C++ `{fmt}`[6] library. It has been extended to be able to execute code at initialization, on an event, data share, release and at finalization. It has also been extended to support a more intuitive execution order of instructions.

The pycall plugin offers access to the Python language and ecosystem from the application process. It instantiates a Python interpreter to execute the code provided by the user in the YAML file. It handles the trans-typing and interface between explicit memory management languages (C, C++, Fortran) and garbage collected Python. It is built on top of the pybind11[7] and numpy libraries to ensure the best possible performance and limit memory overheads to a minimum.

The User-code plugin offers very similar features, but for code written in compiled languages (C, C++, Fortran, ...). The code is compiled separately from the application in a dedicated object

[6] `https://fmt.dev/latest/index.html`
[7] `https://github.com/pybind/pybind11`

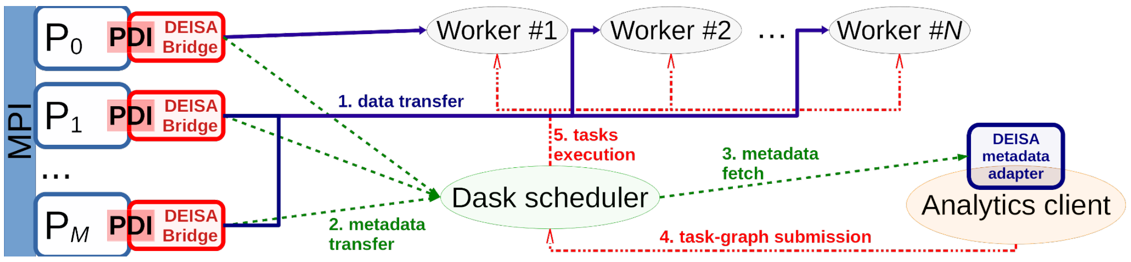


Figure 3: Architecture of Deisa

file and is called directly from the YAML. It can then use the `PDI_access` function of PDI to access data shared by the simulation.

The FlowVR plugin has been developed for data processing in dedicated processes that are either located on the same nodes as the simulation code or even on dedicated nodes through the FlowVR[8][7] workflow library. Data exposed by PDI can be handled by FlowVR modules in a distributed workflow and FlowVR modules can be easily built by wrapping PDI-enabled data processing code from YAML.

A SENSEI[9] plugin[10] has been designed to offer acces to the SENSEI in-situ library. This is covered in more detail in Section7.

Finally, the Deisa[8] (**D**ask-**e**nabled **in** **s**itu **a**nalytics) plugin has been developed to offer access to the Dask infrastructure directly from a PDI-enabled code. The architecture of Deisa is illustrated in Figure 3. When data is provided to the plugin, it is sent directly to Dask workers, bypassing the file system while metadata is sent to the central scheduler. Instead of the usual HDF5 interface, the Analytics client uses a Deisa metadata adapter that offers a similar interface but fetches metadata from the scheduler instead of the disk. The normal Dask behaviour then proceeds with the client building a task graph, sending it to the scheduler that distributes the tasks to the workers for execution. This is described in more detail in [8].

4.4 Conclusion

To summarize, PDI supports interactions between loosely coupled modules sharing the same memory space. Using this approach and thanks to the flexibility offered by the `$-expression` mechanism, plugins can easily implement new behaviour using and modifying the buffers exposed by the simulation code.

In the framework of EoCoE-II, the PDI core has seen a lot of improvement to support the SC codes. Plugins are now available to wrap various general-purpose I/O libraries (HDF5, SIONlib, NetCDF, ...) and specific-purpose I/O libraries (FTI for checkpointing, ...) A specific emphasis has been put on plugins to support in-situ analytics (PDI-Script, User-code, pycall, SENSEI, FlowVR, Deisa, ...) and ensemble runs (Melissa-SA and Melissa-DA). The integration of the library in SC codes was part of WP 2, but this has been made possible and in tight cooperation with this work package. This has led to a lot of feedback, making possible the improvement of the PDI library in a very successful example of code/library co-design.

[8]`http://flowvr.sourceforge.net/FlowVRDoc.html`
[9]`https://sensei-insitu.org`
[10]`https://github.com/pdidev/pdi2sensei`

5 I/O refactoring and optimization

This task mainly focuses on the overall I/O runtime and possible ways to improve or refactor the data handling by leveraging different I/O library capabilities and the introduction of intermediate hardware-based cache solutions.

All planned activities, besides the PDAF work in the context of SHEMAT-Suite, could be targeted and are reported in this deliverable or were already reported in [4]. The PDAF implementation into SHEMAT-Suite together with activity 2.5.2 of WP 2 could not be targeted due to a shift in the related SC work and more involvement in the context of the PDI integration work into SHEMAT-Suite.

5.1 GyselaX I/O performance evaluation and optimization

5.1.1 Benchmarking

Since the redesign of GyselaX also involved a redesign of the overall PDI integration the overall performance optimization could be moved into the PDI plugin work. Therefore the main target of this activity became to allow a continuous evaluation of the GyselaX I/O performance during the implementation and adaptation of the PDI framework. A benchmarking activity was established to measure the I/O impact on different systems.

As an initial baseline, the former version of Gysela (v30.0) and the initial PDI implementation work of EoCoE-I were benchmarked. Since the checkpointing capabilities of Gysela produce the main storage footprint, the time for restarting file writing was taken into account.

The benchmarks sessions were carried out on the ENEA CRESCO6 cluster[11] during the daily regular operation using a different number of nodes: from 2 to 32 nodes, using 2 MPI tasks per node and 24 threads per tasks and two data size use cases producing either up to 66 GB of data or up to 261 GB of checkpointing data.

Release 30.0 Time for Restart file writing		PLAIN_HDF5				PDI/Decl'HDF5			
#nodes	**#cores**	SMALL I/O (511x512x128x127)		LARGE I/O (511x1024x128x127)		SMALL I/O (511x512x128x127)		LARGE I/O (511x1024x128x127)	
		GB	**sec**	**GB**	**sec**	**GB**	**sec**	**GB**	**sec**
2	96	33	31.7	66	67.7	33	116	66	2476
4	192	66	33.7	131	65.6	66	1005	131	2352
8	384	66	26.6	131	58.2	66	382	131	1099
16	768	66	10.8	131	35.7	66	8.4	131	235.3
32	1536	-	-	261	106	-	-	261	196.3

Table 6: Gysela benchmark results for the old PDI integration on CRESCO6.

As shown in Table 6 the utilized I/O runtime of the PDI-based implementation within the former version of Gysela was higher in direct comparison to a direct implementation of HDF5.

In a second benchmarking round the newer GyselaX implementation (in the frame of the redesign Gysela was renamed to GyselaX)[12], developed as part of EoCoE-II, was utilized to rerun a given set of benchmarks. Since the overall GyselaX scheme changed in between, no direct comparison of the former setup was possible and therefore a new benchmark case was selected producing

[11]CRESCO6 consists of 434 nodes for a total of 20832 cores. It is based on the Lenovo ThinkSystem SD530 platform. Each node is equipped with 2 Intel Xeon Platinum 8160 CPUs, each with 24 cores with a clock frequency of 2.1 GHz, 192 GB of RAM and low-latency Intel Omni-Path 100.

[12]Commit `e6bbbb4d` of `https://gitlab.eocoe.eu/gysela-developpers/gysela`

3 TiB of data. Besides CRESCO6, also MARCONI SKL[13] was utilized for execution. On MARCONI the jobs were submitted on 192 compute nodes using 2 MPI tasks per node and 24 threads per task. This version of GyselaX utilized a direct HDF5 implementation for the checkpointing capabilities and did not involve PDI. These runs were meant to create a new baseline of the general GyselaX improvements (without having the PDI-related impact) and to allow also additional performance evaluation using Darshan[14]. Darshan is an I/O profiling tool which allows evaluating the impact of certain read/write and metadata operations, especially it can intercept HDF5 calls to evaluate the performance of these function calls in particular.

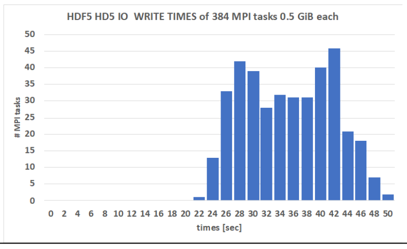
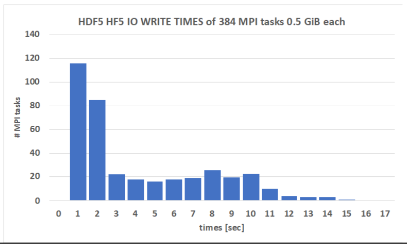
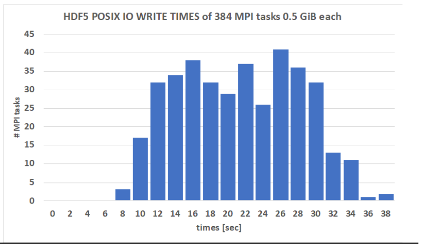
512x1024x128x128x48 output size 3 TiB	**192 nodes - 384 MPI - 9216 cores**		
	total time [s]	**total time (without init & diag) [s]**	**time for restart file writing [s]**
CRESCO6(/gporq2)	2736	1924	265
MARCONI SKL (no Darshan)	2362	1667	53
MARCONI SKL 9441477	2476	1756	51
MARCONI SKL 9436808	2494	1749	51
MARCONI SKL 9431143	2503	1772	52
MARCONI SKL 9443364	2629	1685	51
MARCONI SKL 9443284	2421	1663	114
MARCONI SKL 9443526	2484	1659	183
MARCONI SKL 9430577	2549	1747	123
MARCONI SKL 9433585	2610	1770	156

Table 7: GyselaX benchmark results without PDI integration.

The results of these benchmark runs are given in Table 7. For MARCONI the runs were performed multiple times during regular system production time. Of course in direct comparison to CRESCO6, the I/O subsystem of MARCONI is more performant, nevertheless, the runs highlighted a significant I/O time fluctuation on MARCONI (green runs vs. red runs). This fluctuation is given by the overall system performance e.g. other running applications in parallel. Therefore it can't be predicted if a benchmark run will face this issue, however, the main bottleneck for this issue can be analysed: Since the runs were instrumented by Darshan it was possible to evaluate the reason for this fluctuation by comparing two runs:

[13]https://www.hpc.cineca.it/hardware/marconi
[14]https://www.mcs.anl.gov/research/projects/darshan/

9441477; total time: 2476s; file writing: 51s

H5D	H5F	POSIX
		

9443526; total time: 2484s; file writing: 183s

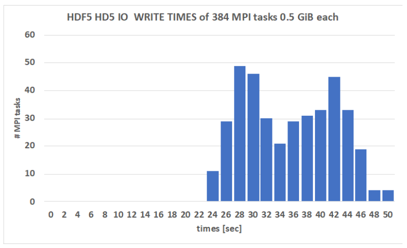
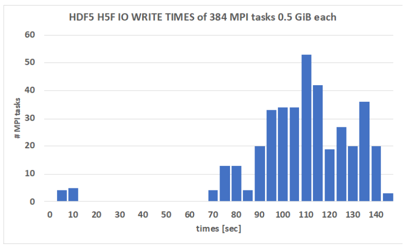
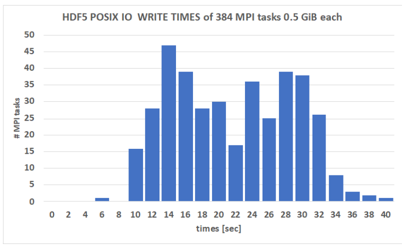
H5D	H5F	POSIX
		

Table 8: Darshan histograms for modules H5F, H5D and POSIX functions underneath of two benchmark runs of GyselaX on MARCONI

The histograms, given in Table 8, highlight that the actual writing time, as contained in the HFD functions of HDF5 (e.g. `hfdwrite`) as well as the POSIX functions underneath perform rather similar between the fast and the slow runs. The major impact is given by the H5F function calls, which take significantly more time in case of the slow runs. These HDF5 function calls are normally related to the actual file creation process which points to a metadata handling problem during the runtime of the slower jobs within the MARCONI filesystem, but no general issue within the I/O performance of GyselaX.

Finally, using the same benchmark setup, the new PDI-enabled version of GyselaX[15] was utilized together with the newly redesigned version of PDI.

[15]Commit `fec86b02` of `https://gitlab.eocoe.eu/gysela-developpers/gysela`

512x1024x128x128x48 output size 3 TiB	192 nodes - 384 MPI - 9216 cores		
	total time [s]	**total time (without init & diag) [s]**	**time for restart file writing [s]**
MARCONI SKL older GyselaX release no PDI integration	2362	1667	53
	2476	1756	51
	2494	1749	51
	2503	1772	52
	2629	1685	51
	2421	1663	114
	2484	1659	183
	2549	1747	123
	2610	1770	156
MARCONI SKL latest GyselaX release, PDI disabled (USE_PDI OFF)	1491	952	60
	1532	975	57
	1667	951	223
	1689	943	230
	1725	944	257
	1766	968	278
	1829	1084	243
	1863	973	228
	1881	1192	170
	2970	1744	62
MARCONI SKL latest GyselaX release, PDI enabled (USE_PDI ON)	1467	970	8
	1484	955	29
	1486	952	10
	1494	972	5
	1505	954	18
	1534	981	5
	1553	978	68
	1587	1011	38
	1609	969	106
	2964	1727	34

Table 9: Comparison of GyselaX benchmark runs without PDI integration (previous version), latest GyselaX version and `USE_PDI OFF`, latest GyselaX version and `USE_PDI ON`.

Table 9 shows that the even newer release of GyselaX, used in the PDI integration benchmarks, was optimized in terms of the computational runtime (*total time (without init & diag)*). The fluctuation between different runs during regular production on MARCONI is still clearly visible. The PDI-enabled runs (`USE_PDI ON`) highlighted a significant performance improvement for the actual I/O time, having the best runs with a runtime of 5 s. This improvement is mainly due to the general I/O restructuring work as part of the PDI integration and the optimization capabilities of the HDF5 plugin in PDI. Similar performance improvements would also be possible within a plain HDF5-based version of GyselaX, but PDI allows to disconnect any optimization work from the actual application and therefore allows faster development work as well as separation between the scientific aspects of the application and the more HPC I/O performance-oriented aspects. More details on the PDI-based improvements are given in section 4.3.1. Overall the PDI implementation and the overall GyselaX I/O bandwidth in direct comparison to its initial EoCoE-I approach could be significantly improved.

5.2 Asynchronous I/O in ESIAS-met

The large ensemble simulation of weather forecasting requires not only large resources for computing but also heavy performance in I/O. A typical weather simulation includes hundreds of physics variables produced per timestep, which can cause the same I/O time as the simulation time. For example, the I/O time of results and restart files requires 3.9 s and 17.8 s, respectively, but the actual run time of the same simulation takes only 6.15 s when the output timestep is per 15 minutes[16].

The flagship code ESIAS-met is developed based on the Weather Research and Forecasting (WRF) model (version 3.7.1). The code is hard-wired and modified with an ensemble module written in Fortran90 and the MPI communicator is modified at the top level over WRF. In this case, the asynchronous I/O options, which are available as part of WRF using the so-called quilt mode, are not able to be activated automatically since the dedicated I/O tasks can not be assigned by the MPI communicator within WRF. A limited workaround to circumvent this issue was introduced in [4], however since it involves changes to the WRF base code, a Lightweight Ensemble Framework (LEF), a python-based objective-oriented module, was created to solve this issue to allow easy utilization of the WRF asynchronous I/O capabilities. This approach allowed us to test the impact of the asynchronous I/O operations already in parallel to the ongoing Melissa enabling work as part of work package 5.

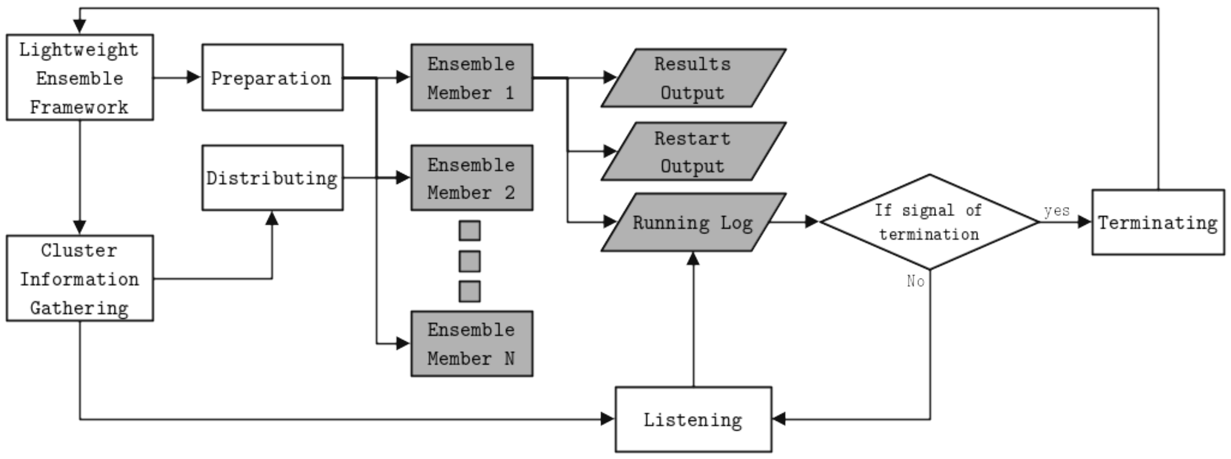


Figure 4: The scheme of Lightweight Ensemble Framework (LEF). The white boxes are the submodules of LEF, and the grey boxes are related to the ensemble members as executors and individual data I/O. The Light Weight Ensemble framework is the root of this framework which controls the simulation workflow. The Preparation submodule can create individual subfolders for each ensemble member; the ”Distributing” submodule assigns the nodes and CPUs for running the simulation after the configuration for using the HPC is acquired by the ”Cluster Information Gathering” function; The Listening submodule is used to monitor the simulation by checking the simulation log files to decide the next steps, e.g. terminating the simulations.

The scheme in the Figure 4 shows how this framework works for the individual executor to simulate. The LEF includes four sub-modules, including the information gathering sub-module,

[16]This experiment is performed using the European domain of 15 km horizontal resolution under 120 s per maximum timestep. The domain size is $220 \times 220 \times 49$ grid cells for this single domain.

preparation sub-module, distributing sub-module, and the listening sub-module. The information-gathering sub-module can analyze the HPC information and gather the information to run the HPC based on the user's demands, e.g. the number of cores or nodes per ensemble member, and create the input files for the modelling codes. The preparation sub-module can copy and prepare ensemble members on demand. The listening sub-module will monitor the simulation based on the logs to keep the LEF alive. When the listening sub-module is informed about the termination of the simulation model, the framework will be terminated. The LEF allows to reduce the duplication of input files amongst ensemble members and adopts an easy test method for upgrading the codes without upgrading the hard wiring inside the simulation codes.

5.2.1 Strong scaling performance

The following scaling performance is employed by performing short-term weather forecasting simulations in the real European Domain (total grid cells are $220 \times 220 \times 49$ and I/O with 15 minutes timestep for a total of 24 hours of simulation. The restart files are generated for every 12 hours of simulation. The first scaling performance investigates the effect of different asynchronous I/O options on the total simulation time of the WRF model. The simulation is fully performed on the JUWELS system using its cluster module (Intel Xeon Platinum 8168 CPU, 2×24 cores, 2.7 GHz, 96 GB RAM).

Figure 5 shows the scaling by performing simulation with and without enabled asynchronous I/O. Q0 indicates the WRF simulation without the function, and Q1, Q2, and Q3 indicate the simulation with the asynchronous I/O handled by 1, 2, and 3 cores per member, respectively. The effect of using multiple nodes per member on the simulation time is also investigated.

The simulation results show a great impact of reducing the median simulation time from 829.7 seconds without the asynchronous I/O to 569.3, 525.3, and 530.5 seconds by 1, 2, and 3 I/O cores per member, respectively. The greatest reduction is 36.67% by 2 cores per member. The increase of the nodes per member does not guarantee the reduction of total simulation time. The maximum simulation time increased from 885.9 seconds to 994.4 seconds by using 2 nodes per member but decrease to 677.8 seconds by using 4 nodes per member. The maximum simulation time from the ensemble members with asynchronous I/O is less than the minimum simulation time for the ensemble members without asynchronous I/O. The increase in nodes per member does not guarantee the reduction of simulation time since the communication between nodes will increase the total simulation time though the simulation during a time steps decreases.

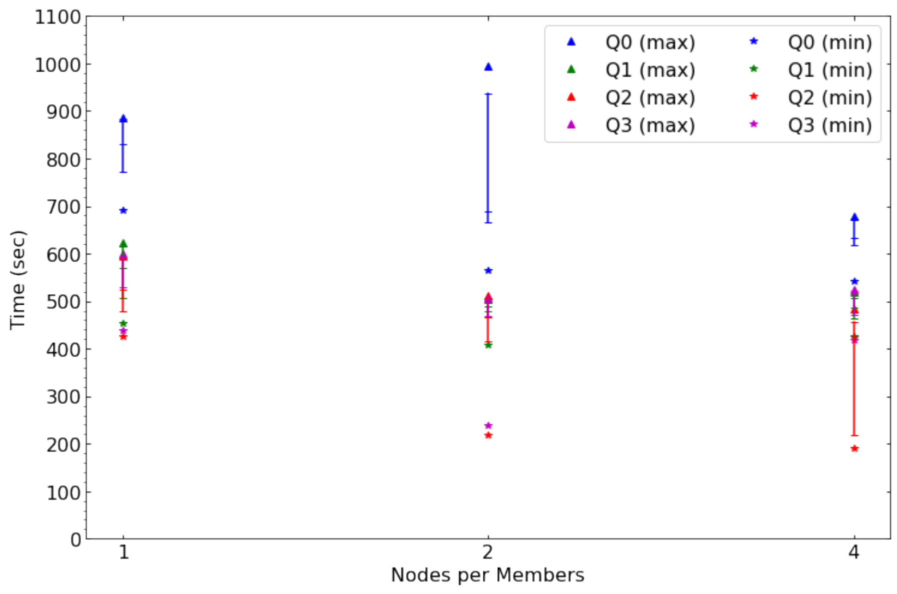


Figure 5: The strong scaling performance of ESIAS-met by LEF. The blue, green, red, and magenta represent the asynchronous I/O option as 0, 1, 2, and 3 dedicated I/O cores per simulation. Triangle is the max time from the ensemble members; the star is the min time from the ensemble members, and the bar indicates the first quantile, median, and third quantile from the bottom to the top bars.

5.2.2 Weak scaling performance

The weak scaling performance is evaluated by increasing the problem size with the increase of the computing resource. The stress on the ensemble simulation size can also be investigated by this benchmark test. The original ESIAS-met is employed with the same input data as a comparison to the LEF (the abbreviation as WRF-LEF). As the problem size increase to 32 members in the ensemble simulation, the performance of WRF-LEF is as good as ESIAS-met except for performing with only 4 members. However, the performance of WRF-LEF using asynchronous I/O is outperforming the ESIAS-met when performing simulations with 8 to 64 ensemble members with a maximum reduction of 38.93%. The simulation with 128 ensemble members increased the simulation time up to 32.90%, however also with having a significant fluctuation as all these tests were performed during regular production times on JUWELS. A technical improvement should be carried out to increase the stability of I/O or to reduce the simulation time while having the advantage of simplified ensemble simulation without difficulty in modification of the code.

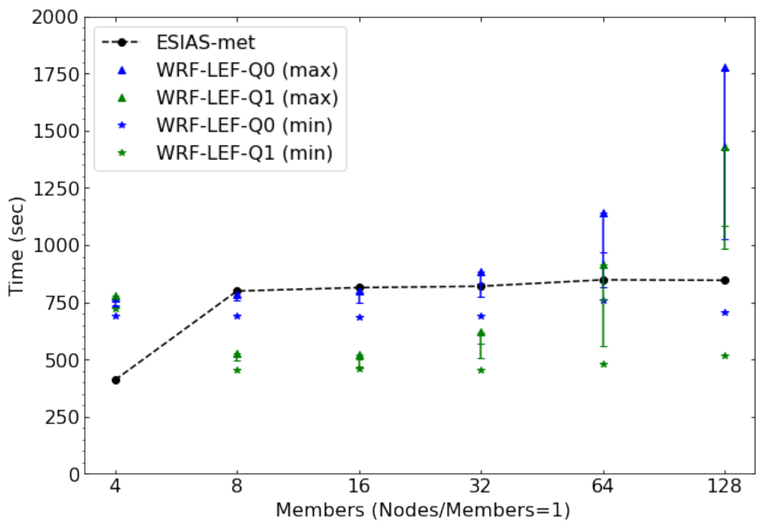


Figure 6: The weak scaling performance of WRF by LEF (blue without the asynchronous I/O and green with the asynchronous I/O) compared to ESIAS-met (black). Triangle is the max simulation time from the ensemble members; the star is the min time from the ensemble members, and the bar indicates the first quantile, median, and third quantile from the bottom to the top bars.

5.2.3 Remarks and conclusions

The exascale computing in climate simulation and weather forecasting requires a good I/O efficiency and storage capacity [14, 13]. The asynchronous I/O is an essential function that can reduce the I/O time to prevent the loss of computing time while performing data output by producing the simulation results and by producing the restart files for fault tolerance. A reduction of 30% of total simulation time can be an advantage in the direction of exascale computing when performing large ensemble simulations. The simulation with a lightweight ensemble framework can reduce the complexity in the software architecture and remain the advantage of the modelling codes.

5.3 Gysela cache storage utilization

Pure software-based I/O optimization isn't the only solution to circumvent I/O bottlenecks. Instead, we also focused on the capabilities of hardware-based solutions mainly intermediate cache layers which provide much higher bandwidth rates due to flash storage technologies as well as fast interconnect solutions. Since DDN is a partner within the EoCoE-II project and we had the opportunity to be one of the first users of the High-Performance Storage Tier (HPST) platform at JSC, which is based on the IME technology by DDN, we focused on this solution for our tests. As written in [4] we evaluated the capabilities of the HPST system and also implemented the IME native API into SIONlib. The results, presented in the former deliverable were also published in the meantime in [6].

The final step of this activity was the evaluation of the given approach, using an SC application. Due to its high storage and I/O bandwidth demands for writing intermediate checkpoints, Gysela was selected. Since this work ran in parallel to the actual GyselaX development work and the PDI implementation work, an older EoCoE-I based build of Gysela was taken into account which was able to leverage either HDF5-based task local files (similar to the newer PDI-based implementation) as well as SIONlib as it was implemented within EoCoE-I.

For the benchmark setup, the former version of Gysela (version 32.0), as well as the latest version of the adapted SIONlib library, were compiled on the JUWELS system [9]. To allow to easily run the benchmark setup in a reproducible way, the setup was ported into the JUBE benchmarking environment [11] to orchestrate the actual benchmark workflow and to extract the checkpoint writing time. For the benchmark itself, a weak scaling approach was selected, using 8 tasks per node and 6 threads per task to utilize all 48 cores per JUWELS node. Two checkpoints were created per run, writing around 512 MiB per task on up to 512 nodes which sum up to around 2.2 TB per checkpoint. For comparison, the HDF5-based task local scheme was evaluated on the general Spectrum Scale/GPFS-based SCRATCH filesystem of JUWELS and the HPST using the available POSIX FUSE mount. The same tests were done using the SIONlib API without any IME native capabilities and finally also by using the IME native API underneath of SIONlib. Since the runs were done during regular production time on JUWELS, all runs were executed multiple times and the best/shortest runtime was selected.

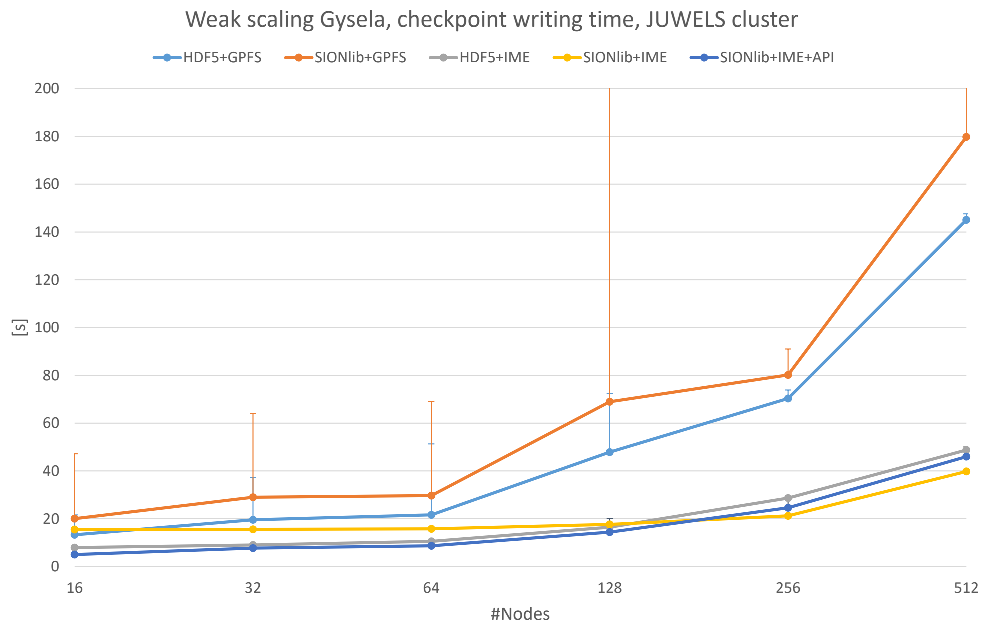


Figure 7: Gysela HPST (IME) vs Spectrum Scale (GPFS) comparison using the checkpoint writing time for a weak scaling run on JUWELS.

Figure 7 shows the overall time spend on writing the two checkpoints using the different filesystems and library solutions. The classical filesystem approaches based on GPFS, clearly become a bottleneck on larger scales. Here SIONlib performed worst due to the used configuration by using only a single output file for the overall checkpoint which creates metadata and filesystem block constraints. A task local setup as given by the HDF5 approach performs slightly better, but also here metadata problems, due to a large number of files, as well as general bandwidth limitations impact the scaling capabilities. Also, the I/O time fluctuation, given by the error bars in the plot, highlights the impact of other applications on a shared resource such as the parallel filesystem. Having Exascale aware runs in mind the given I/O weak scaling behaviour is far from ideal.

In contrast, the utilization of the HPST and IME technology shows a much better scaling, especially on larger scales. For smaller scales, the newly implemented IME native API underneath of SIONlib performed best, which is also expected as the utilization of the native API lowers the client-site specific overhead, while on larger scales the overall bandwidth and the server-site overhead become more relevant. On large scales, SIONlib slightly outperforms a task local approach as the metadata handling for a large number of files still relies on the metadata handling of the classical parallel filesystem behind the intermediate cache layer. On 512 JUWELS nodes, the checkpoint writing time also increases but on a much lower scale in comparison to the general parallel filesystem. In addition to the better performance, all runs could also be performed with nearly no fluctuation in the actual writing time.

The utilization of the tested cache infrastructure for Gysela highlights a great performance boost opportunity for future large-scale production runs of GyselaX.

5.4 ParFlow cache storage utilization

Besides Gysela and direct utilization of the IME API, we also tried to utilize the IME storage solution within the HPST at JSC using other EoCoE-II SC applications. For this, we picked the ParFlow example which we also utilized within the compression work as mentioned in section 7 and compared the overall runtime of a weak scaling test case on top of the HPST and the regular IBM Spectrum Scale-based parallel filesystem. In this case, no changes to the application or libraries were made. The benchmark was performed using the BFP format of ParFlow, which utilizes generic POSIX calls underneath which are handled via a FUSE[17] mount within the HPST. Utilization of NetCDF4 was not possible within this approach as parallel NetCDF4 relies on HDF5, which relies on MPI-IO which utilizes an IME-aware version of MPI-IO on the JSC systems[18]. However, this approach was not fully stable at this point and therefore we stayed with the POSIX based BFP format option instead. The benchmarks were performed on the JUWELS system at JSC (JUWELS cluster) using 48 tasks per node and were again orchestrated by the JUBE benchmarking environment. To utilize the HPST FUSE mount point only the relevant output paths need to be adapted followed by a sync of the actual cache towards the regular file system (to drain the cache and to secure the results in the regular parallel storage solutions).

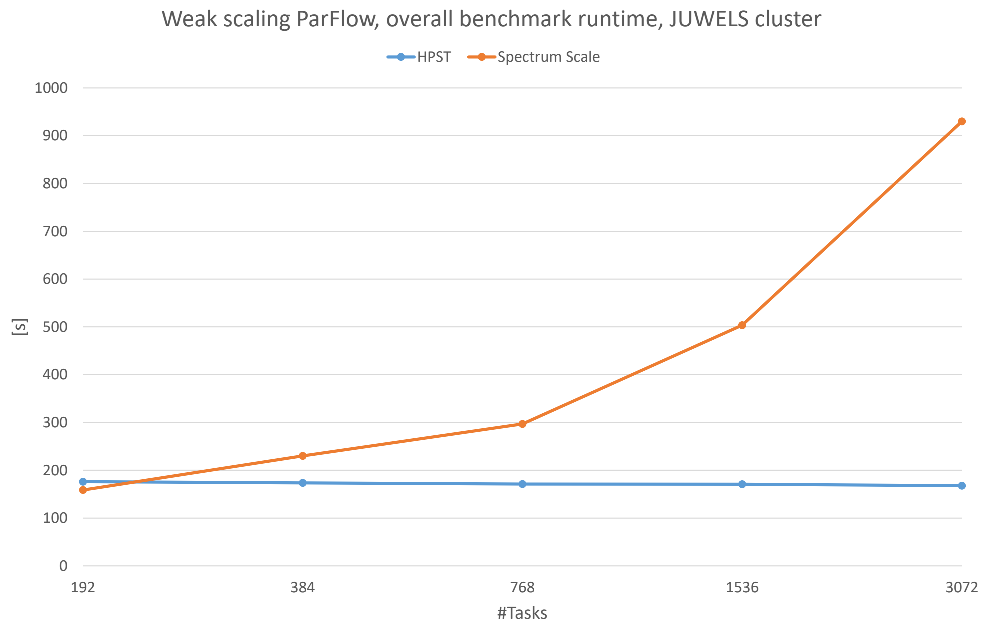


Figure 8: ParFlow HPST (IME) vs Spectrum Scale comparison using the overall benchmark runtime for a weak scaling run on JUWELS.

Figure 8 shows that the HPST performance, acting as a cache layer in between the regular parallel filesystem and the application, clearly outperforms the classical filesystem. For the given data size of the weak scaling test, it scales ideally for the given number of tasks while the classical parallel filesystem soon starts to become the major bottleneck of the application. The bad scaling behaviour of the BFP format was also seen during the compression tests in section 7 where this

[17] `https://www.kernel.org/doc/html/latest/filesystems/fuse.html`

[18] `https://apps.fz-juelich.de/jsc/hps/just/cscratch.html#ime-native-interface`

problem was solved using a different file format and compression on top. Here the cache layer acts as a second, hardware-based, solution to overcome the problem.

Overall a global fast cache infrastructure, if available, as given by the HPST, using the IME technology, helps to circumvent I/O problems quite simple without a major impact on the application workflow.

6 Fault Tolerance

The work presented in this section is a collaboration with the meteo scientific challenge, and work package 5 in the context of ensemble handling. Together we developed a particle filter that operates fault-tolerant and is efficient at a very large scale. Our implementation is based on the Melissa-DA architecture. This work has partially been presented in [2]. However, we repeat basic concepts and motivation to make the report self-contained, while focussing on the resilience aspects, which were the major contribution of work package 4.

6.1 Introduction

Given an output and a transformation function, finding the input states represents a so-called inverse problem. A wide range of approaches to address this central problem exists. Statistical Bayesian methods stand out as they provide uncertainty measures of the proposed input in form of a probability density function (PDF). An important concept from this category is particle filtering (PF). PF is a statistical Bayesian method combining uncertainties of both the dynamical system and observations, providing an estimate of the system's state PDF. Several realizations of the dynamical system, called particles, with differently perturbed internal states, are propagated up to a time where observation data are available. The particles are then weighted corresponding to their distance to the observations. The weights are used to generate a new sample of particles that better match the observations. This process repeats while observations are available.

Particle filters are promising candidates to account for strong non-linearities of climate models when assimilating observations. However, while for ensemble Kalman filters the required ensemble size grows linearly with the problem size, it grows exponentially in particle filters [12]. As a consequence, PF need to handle a very large number of particles efficiently.

We strain our PF implementation with a realistic use-case, interfacing with the Weather Research and Forecasting (WRF, version 3.7.1) model [16], which is also the main simulation kernel used by ESIAS-met. WRF is a widely used weather model for operational forecasting and research. Using our particle filter, we are able to run 2 555 particles on 20 442 compute cores for WRF simulations on a European domain with 87 % efficiency.

6.2 Important Properties of Particle Filters

In this section, we give a brief introduction on the particle filter formalism, focusing on properties that we exploit in our implementation. For a comprehensive introduction, we refer to [17]. Let $\mathcal{M}$ be a numerical model, that propagates a particle p from state $\mathbf{x}_{p,t-1}$ at time $t-1$ to state $\mathbf{x}_{p,t}$ at time t:

$$\mathbf{x}_{p,t} = \mathcal{M}(\mathbf{x}_{p,t-1}) \tag{1}$$

To decide for good particles, we assign a weight $w_{p,t}$ to the particle associated with $\mathbf{x}_{p,t}$. The weight is given by the likelihood $p(\mathbf{y}_t|\mathbf{x}_{p,t})$ of observing $\mathbf{y}_t$ at state $\mathbf{x}_{p,t}$:

$$w_{p,t} = p(\mathbf{y}_t|\mathbf{x}_{p,t}) \tag{2}$$

For the bootstrap particle filter, the proposal PDF $p(\mathbf{x}_t)$ of the particles at time t is given by [17]:

$$p(\mathbf{x}_t) = \frac{1}{P}\sum_{p=0}^{P-1} \delta(\mathbf{x}_t - \mathbf{x}_{p,t}) \tag{3}$$

Where P is the number of particles in the sample and $\delta(\mathbf{x})$ the Dirac delta distribution. With the help of Bayes Theorem:

$$p(\mathbf{x}_t|\mathbf{y}_t) = \frac{p(\mathbf{y}_t|\mathbf{x}_t)p(\mathbf{x}_t)}{p(\mathbf{y}_t)} \tag{4}$$

we are able to construct the posterior PDF $p(\mathbf{x}_t|\mathbf{y}_t)$ using Equations (2) and (3):

$$p(\mathbf{x}_t|\mathbf{y}_t) = \sum_{p=0}^{P-1} \hat{w}_{p,t}\, \delta(\mathbf{x}_t - \mathbf{x}_{p,t}) \tag{5}$$

With $\hat{w}_{p,t}$ being the normalized weight:

$$\begin{aligned}
\hat{w}_{p,t} &= \frac{p(\mathbf{y}_t|\mathbf{x}_{p,t})}{P\, p(\mathbf{y}_t)} &(6)\\
&= \frac{w_{p,t}}{P \int p(\mathbf{y}_t|\mathbf{x}_t)\, p(\mathbf{x}_t)\, d\mathbf{x}_t} &(7)\\
&= \frac{w_{p,t}}{\sum_{q=0}^{P-1} p(\mathbf{y}_t|\mathbf{x}_{q,t})} &(8)\\
&= \frac{w_{p,t}}{\sum_{q=0}^{P-1} w_{q,t}} &(9)
\end{aligned}$$

We want to emphasize, that in contrast to other DA techniques, particles remain unchanged during the update step. Particles that have departed too much from the observations (low weights) are discarded, and the sample set is narrowed around the best particles (high weights). The associated states, however, are not changed. We exploited the following properties of particle filters:

(a) Weights $w_{p,t}$ depend only on the associated particle and observations

(b) The filter update only depends on the weights $w_{p,t}$

(c) The associated states $\mathbf{x}_{p,t}$ remain unchanged by the filter update

Property (a) follows directly from Equation (2). Property (b) results from decoupling the weight calculation from the filter update (decentralization). The update itself only consists of the weight normalization and particle resampling. Finally, property (c) is an intrinsic property of particle filters, since particles are either withdrawn or selected, but not changed. In the following sections, we will show how we can exploit those properties to mitigate resilience and improve the efficiency of particle filters.

6.3 Architecture

The properties expressed before helping us to design the adequate architecture suited for the problem. Recall that we apply the numerical model $\mathcal{M}$ to propagate the particles (Equation (1)). each $\mathcal{M}$ only depends on the particle to propagate, but not on other particles. Therefore, we can execute the model for each particle independently. Property (a) is similar as the weight computation only depends on the propagated weight, but not on other weights or particles. Thus, we can perform both the application of $\mathcal{M}(\mathbf{x}_{p,t-1}) \to \mathbf{x}_{p,t}$ and the computation of the weight $w_{p,t}(\mathbf{x}_{p,t}, \mathbf{y}_t)$, for each particle of the sample isolated from the others. On the other hand, property (b) states that we need to gather the weights of all particles to perform the resampling. However, once the weights are gathered in one location, the remaining steps are computationally cheap. We merely need to compute the normalized weights $\hat{w}_{p,t}$ (Equation (9)) and sample new particles from the generated PDF (Equation (5)).

This structure represents several decentralized and independent operations; particle propagation and weight computation, and one centralized and lightweight operation; weight normalization and resampling. The appropriate architecture to handle this structure is a server/client workflow. The server and clients, hereinafter called runners, are introduced in [2]. We want to focus here on the distributed cache that we implemented on the runner nodes. The cache is used to reduce the idle time of runners due to file system operations and to move the transfer of particles to the parallel filesystem (PFS) into the background (asynchronous IO).

6.4 Distributed Particle Cache

To allow multiple propagations of a particle on different runners, it is necessary to make them globally available. A straightforward approach to accomplishing this is storing them on global storage. However, on large clusters, global storage is subject to large throughput variability due to the high workload and the limited bandwidth. Node local storage, on the other hand, is only used by the processes that run on the nodes, and the bandwidth can be stacked. Therefore, storing the particles locally results in stable IO performance, and the bandwidth scales linearly with the number of nodes. To store the particles locally, and still provide them globally, we implemented an asynchronous particle cache on the runner nodes. For this, we split the runner communicator into one for the model and one for the cache controller. The model processes propagate the particles and store the associated states in local storage. The cache controller then stages the states from local to global storage asynchronously.

Furthermore, we allow keeping multiple particles in the cache. This allows us to exploit property (c). This property states that particles remain unchanged during the resampling. If we keep all particles after their propagation in the cache, we are certain, that all the drawn particles are available locally. They are distributed in the caches of the runners. Hence, we can schedule the propagations to the runners optimized by the availability in the caches. In general, the cache size is limited, and not all the particles can be kept locally. Therefore, we need an appropriate cache eviction strategy. The server possesses knowledge about the particles in the local caches, as all the particles are stored locally in the first step and the runners message weight and particle-id to the server after the propagation. The server has further knowledge about upcoming propagations. Leveraging the server knowledge, we implement the following eviction strategy:

(1) dispensable particles

(2) particles with the lowest weight.

(3) randomly selected particles.

Where dispensable particles are particles that either have been (1) withdrawn during the last resampling, (2) propagated to timesteps $t-1$ or earlier, or (3) have finished all the associated propagations. To ensure that for the latter, all related actions are completed, we implemented a protocol ensuring that by the time the server acknowledges the successful propagation, the associated state has been safely stored in global storage. Dispensable particles will not be associated with future propagations and can therefore be evicted from the cache. If no such particle can be found in the cache, we chose particles for which the propagation is finished and that have the lowest weight. Those particles have a low probability to be drawn during the resampling and will therefore not likely be propagated in the next cycle. Finally, if neither particles from (1) or (2) can be found, we decide to evict a random particle.

From a high-level perspective, the cache can be seen as a distributed level-1 (L1) cache for particle propagations. It keeps recent data; in case of eviction it keeps critical data (particles to propagate soon), and it prefetches critical data. Furthermore, the contents of the cache are always persisted in global storage, providing fault tolerance for cache losses (runner failures). As the runners do not know about the status of the particle filtering (propagations, resampling), they cannot decide about eviction and prefetching. Therefore, the cache controller requests both particles to evict and prefetch from the server.

6.5 Framework Resilience

The resilience of the framework relies on the fast identification of failures from runner and server instances. Runner failures are detected in two different ways. Runner crashes are recognized by the launcher, which is monitoring their execution using the cluster scheduler. Unresponsive runners are identified by the server relying on timeouts for the particle propagations. If propagations exceed the timeout, the server requests the launcher to terminate the respective runner. In both cases, the launcher eventually starts a new runner instance. The new runner connects to the server and is incorporated into the runner pool. As we explained earlier, all particles are stored on the PFS, thus, the new runner can fetch any particle assigned and begin with the propagation.

Server failures are detected similarly, either directly if the server crashes, or if the server exceeds a timeout. The timeout is mediated by a periodical exchange of signals between the launcher and server (heartbeats). If the server fails, the launcher terminates all runner instances and restarts the framework as a whole. The server frequently stores the status of the propagation in checkpoints, and in case of failures, the framework can recover to the point of the last successful propagation.

Finally, a launcher failure is detected by the server monitoring the heartbeat connection between the launcher and server. In case of a missing heartbeat, the server checkpoints the current particle state and shuts down. On the other hand, the runners detect that the server terminated, again leveraging timeouts, and shut down. While the launcher restarts the framework automatically for runner or server failures, If the launcher fails, the framework needs to be restarted manually.

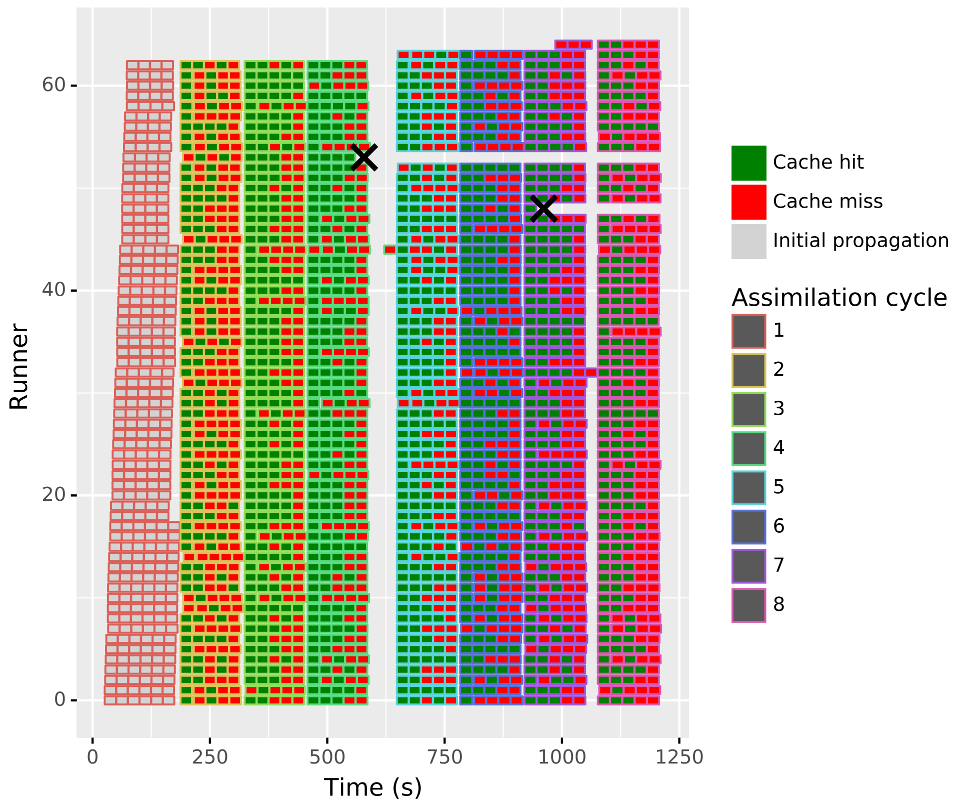


Figure 9: Gantt chart showing the trace where two runners crashed (black cross) and 2 restarted (top 2 runners).

We tested the fault tolerance and elasticity of our PF implementation with 63 runners crashing 2 runners (Figure 9). 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 detecting the runner crash (the 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 an 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 than 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 a rather small variability of about 10%. with other simulation codes, or if executing on heterogeneous resources, propagations might show a much stronger variability.

6.6 Implementation details

The local runner caches rely on the Fault Tolerance Interface (FTI) [5]. FTI is a multilevel checkpoint-restart library supporting asynchronous checkpointing to the global storage. Other caveats are checkpoint creation using HDF5, differential checkpointing, and other reliability levels. We modified FTI to allow its utilization as cache controller backend. For the asynchronous checkpoint creation, application processes store the checkpoint locally, and FTI processes stage the local data to global storage asynchronously (post-processing). The post-processing is triggered automatically as soon the application has stored the data in local storage. However, we need further control, as we need to inform the server of the completed propagation and the weight. The FTI processes run an event loop. Events are triggered in form of MPI communication between the application and FTI processes. The events are identified by certain tags. To intercept this mechanism, we added a feature to FTI, allowing us to register a callback function. This callback function is called inside the event loop and can trigger user-defined events using unique tags. With this, it becomes possible to use application checkpointing in all available levels of reliability that FTI provides, and implement the cache controller on the dedicated FTI processes.

6.7 Conclusion

Together with work package 5, 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 [15]. 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 2 555 particles on 20 442 cores with 87% scaling efficiency.

6.8 Earlier Contributions

Besides this work in collaboration with the Meteo SC, we also focused on additional fault tolerance topics in the frame of EoCoE-II.

Meteorological applications leverage file I/O for both resiliency and scientific output. We presented new FTI API extensions, empowering scientists to expose additional information for the datasets to the checkpoint files (e.g., decomposition, variable names, etc.) using HDF5, and to control the data layout. As the checkpointed data is often also the object of the scientific analysis, we gain two benefits; the application is protected, providing all the checkpoint features of FTI (checkpoint I/O), and we store the data into HDF5 files enabling for post-execution analyses (scientific I/O) [1].

Furthermore, we presented an extension for the FTI API allowing the writing of the protected variables individually to the checkpoint file (incremental checkpointing). This was necessary to integrate fault tolerance with FTI into Alya. Moreover, we leverage asynchronous checkpointing of FTI in Alya, to move the checkpoint creation into the background [1].

We also integrated checkpointing with FTI into Melissa-DA. As in Alya, we apply asynchronous checkpointing. In addition, we move the pre-processing step (local staging of the checkpoint data) in the background as well. With this, we achieve complete hiding of the checkpoint overhead. Our implementation further ensures also a minimum in recomputation. We presented two scenarios that both hide the checkpointing overhead, however, differ in checkpoint size and recovery time [1, 4]. We further present an analysis of the benefits of the scenarios, and a detailed performance evaluation at a large scale in a separate publication [10].

Finally, we added IME support to FTI. The checkpoints can be created using the IME I/O by enabling the corresponding setting inside the FTI configuration file [1].

7 In-situ & in-transit data manipulation

7.1 Overall outcome of in-transit compression

Data compression can be a good and rather easy solution to lower the overall storage footprint of an application while keeping all relevant information in place (in case of a lossless compression). Doing this compression in-transit during the regular runtime of the simulation avoids introducing additional post-processing steps and can even lower the IO demands of the application in addition.

In the context of EoCoE-II, this approach was implemented and tested within the ParFlow application and reported in detail in [3]. Since the approach does not introduce any new dependency as it utilizes the latest features of the NetCDF-4 library, it could be easily integrated into the regular ParFlow repository and is now available for every Parflow production run. Besides the benchmark results, which were already reported in the former deliverable, additional benchmarks on JUWELS (using the same weak scaling ParFlow benchmark as before) also highlighted the benefit of utilising the NetCDF-4 framework in general in contrast to the more basic PFB-based standard Parflow output format as shown in Figure 10. For the benchmark, this runtime improvement was even topped by utilizing the ZLIB-based in-transit compression (in addition to the lower storage footprint).

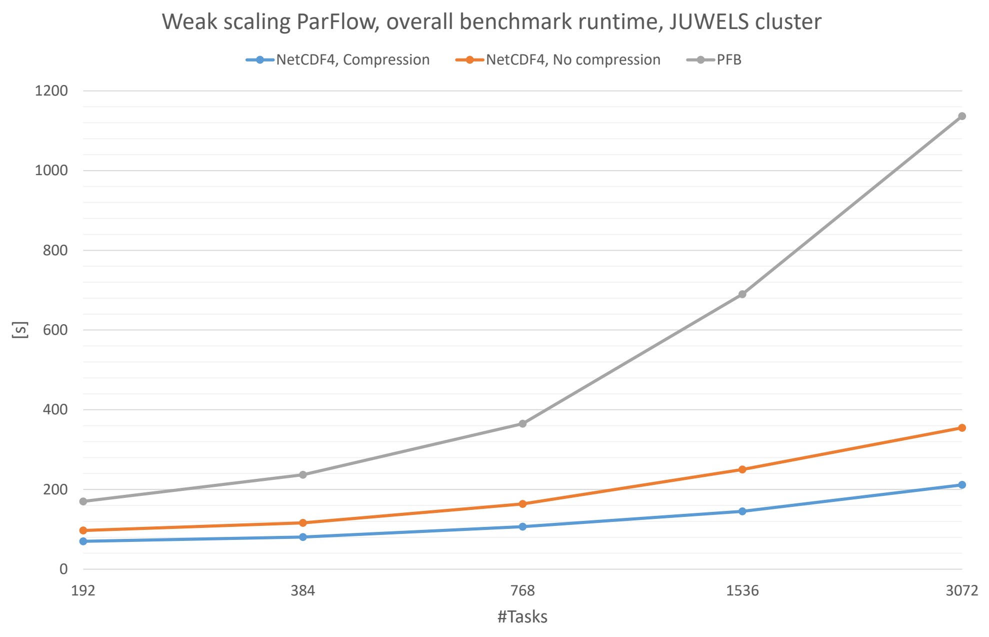


Figure 10: ParFlow compression comparison using the overall benchmark runtime for a weak scaling run on JUWELS.

Overall in-transit compression shows clear benefits for the given use case of ParFlow without introducing too many new site effects or utilization dependencies. A similar approach would be able for other applications, especially for other HDF5 or NetCDF-4 driven applications, as long as they follow certain rules of their I/O pattern as described in [3]. Nevertheless, the compression quality still depends on the available data and can vary significantly between different variables. Lossy compression approaches, such as ZFB, can lower the storage footprint even further.

The parallel in-transit compression approach was only tested for ParFlow. ESIAS-met, which can also use NetCDF-4 underneath as part of the WRF environment, was limited to serial I/O capabilities for the given implementation while parallel file access is performed via the parallel-netcdf library. Newer versions of WRF also allow parallel utilization of NetCDF-4 and can therefore also benefit from parallel in-transit compression.

7.2 Overall outcome of in-situ visualization

Within the framework of EoCoE-II, PDI2SENSEI was developed, which makes it possible to use a flexible in-situ visualization. The necessary changes to the simulation code itself are limited to the integration of PDI because PDI2SENSEI provides the possibility to use in-situ visualization outside of the PDI context. A full descriptions of the setup was described in [3] and an overview is given in the section below and also mentioned in section 4.3.4 alongside the different in-situ approaches supported by PDI.

7.2.1 In-transit capabilities

To keep the influences of the visualization on the running simulation as low as possible and isolated, the preferred option for using PDI2SENSEI is an in-transit visualization. This decouples simulation and visualization by separating them on different nodes, so both have their computing resources, and the coupling is reduced to the data transport. In PDI2SENSEI, the use of in-transit is optional but recommended, but the decision may be different in individual cases.

To retain the flexibility provided by SENSEI in the in-transit case, SENSEI is used twice. The first time as a layer from PDI2SENSEI to the selected data transport solution, in this case, ADIOS2, and the second time on the receiver side to receive the data and pass it on to the desired visualization or processing software. The whole setup is shown in Figure 11.

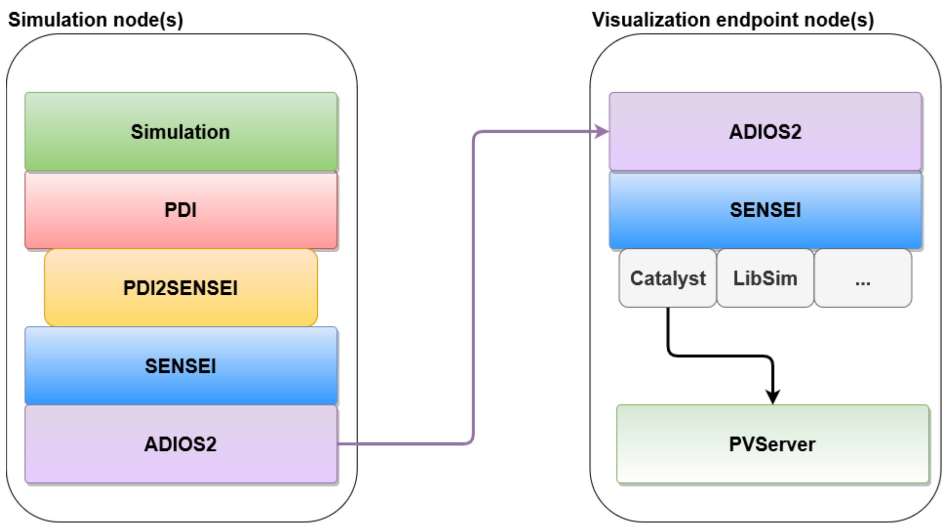


Figure 11: Overview of the used components in PDI2SENSEI. Showing a possible configuration using in-transit and therefore separate nodes for visualization and the simulation. The Simulation uses PDI to pass on the data, which is used by PDI2SENSEI to set up all needed structures to pass the data through SENSEI and ADIOS2 to the visualization endpoint node(s). There the data is received and passed through SENSEI again, to allow a flexible choice for the visualization backend, here using ParaView's Catalyst, running a predefined visualization script and connecting to a PVServer for additional interactive visualization.

7.2.2 Alya

With the help of PDI2SENSEI, Alya has been extended with an in-situ visualization. This allows Alya to use a predefined visualization to directly generate images of the flow. In addition, it is possible to start an interactive visualization and thus look live into the simulation. For this purpose, PDI2SENSEI transforms the Alya internal geometry into a vtkUnstructuredGrid. This is necessary for a non-changing geometry but once. Further details can be found in [18].

Besides the initial conversion, the impact on the running simulation is low, even for very frequent updates every tenth time step of the simulation the impact on the time of the simulation runs was in all tests below five %. The changes visible in the visualization were not significant, which allows to reduce the update interval further and get the performance impact below one % in the case of an update every 50 time steps. This does not take into account, that additional resources are needed for the endpoint nodes, running the actual visualization. Here we used one visualization node for every 24 nodes running Alya itself. To determine the value of these additional nodes, it must be taken into account that even in a conventional workflow, additional resources are required for subsequent visualization and evaluation. With an (interactive) in-situ visualization, however, these resources are used directly during the simulation.

One possible disadvantage of using in-situ visualization, in this case, is, that the conversion and subsequent copy of the Alya geometry into a VTK data structure take part of the main memory of the systems, which could be problematic in case the simulation leaves less memory for use than Alya. In the observed cases, this was not a problem for Alya, at least on the JURECA-DC nodes their memory was not a problem, even with this increased need.

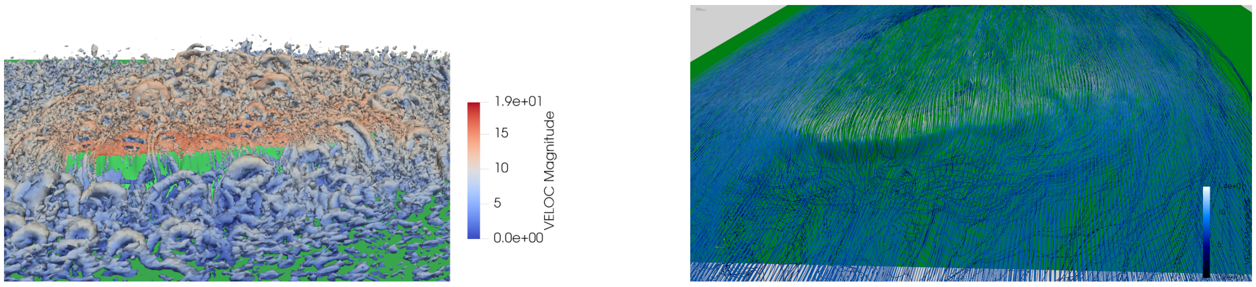


Figure 12: In-situ visualization of Alya running a flow simulation over the Bolund peninsula, once visualized using streamlines, the other time using the Q-criterion to show the velocity magnitude for the turbulence.

7.2.3 ParFlow

A similar approach was also taken for ParFlow. Here an already existing PDI implementation was extended by events for the control of the in-situ routines. The regular geometry makes it much easier to use the data in PDI2SENSEI, as it is no longer necessary to transform the geometry data into a suitable VTK format. Instead, it is sufficient to pass the geometry parameters through PDI to PDI2SENSEI.

Because the data structure was easy to translate into a VTK data structure, the impact on a ParFlow simulation is very small as there is no copy of the data that needs to be generated, instead, all data that is not metadata can be passed through. This helps to keep the memory footprint of PDI2SENSEI low and allows to skip the initial setup cost for the data structure. While this is a big advantage, it bears a small risk. Depending on the actual visualization that is set up, this transformation could happen on the visualization nodes, leading to an unexpected increase in memory need on these nodes. In the worst case, this could stop the visualization, but the in-transit nature and the separation of the simulation and the visualization will keep the simulation running, so that the visualization can be restarted with a different visualization script or more nodes to increase the available memory for the visualization, later on.

References

[1] Intermediate report on I/O improvement for EoCoE codes. `https://www.eocoe.eu/wp-content/uploads/2022/04/D4.1-Intermediate-report-on-IO-improvement-for-EoCoE-codes.pdf`, 2020. D4.1.

[2] Melissa-DA: First stable version. `https://www.eocoe.eu/wp-content/uploads/2022/04/D5.2-M24-Melissa-DA-Stable.pdf`, 2021. D5.2.

[3] Report on data interface and in-situ capabilities. `https://www.eocoe.eu/wp-content/uploads/2022/04/D4.2-Report-on-data-interface-and-in-situ-capabilities.pdf`, 2021. D4.2.

[4] Report on I/O optimization and fault tolerance. `https://www.eocoe.eu/wp-content/uploads/2022/04/D4.3-Report-on-I_O-optimization-and-fault-tolerance.pdf`, 2021. D4.3.

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

[6] Konstantinos Chasapis, Jean-Thomas Acquaviva, and Sebastian Lührs. Integration of parallel i/o library and flash native accelerators: An evaluation of sionlib with ime. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, CHEOPS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Matthieu Dreher and Bruno Raffin. A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations. In *CCGrid 2014 - 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, United States, May 2014. IEEE Computer Science Press.

[8] Amal Gueroudji, Julien Bigot, and Bruno Raffin. DEISA: dask-enabled in situ analytics. In *HiPC 2021 - 28th International Conference on High Performance Computing, Data, and Analytics*, pages 1–10, virtual, India, December 2021. IEEE.

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

[10] Kai Keller, Adrian Cristal Kestelman, and Leonardo Bautista-Gomez. 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)*, pages 131–140, 2021.

[11] Sebastian Lührs, Daniel Rohe, Alexander Schnurpfeil, Kay Thust, and Wolfgang Frings. Flexible and Generic Workflow Management. In *Parallel Computing: On the Road to Exascale*, volume 27 of *Advances in parallel computing*, pages 431 – 438, Amsterdam, Sep 2016. International Conference on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep 2015, IOS Press.

[12] Andrew J Majda and Xin T Tong. Performance of ensemble kalman filters in large dimensions. *Communications on Pure and Applied Mathematics*, 71(5):892–937, 2018.

[13] 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.

[14] T. C. Schulthess, P. Bauer, N. Wedi, O. Fuhrer, T. Hoefler, and C. Schär. 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, 2018.

[15] 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.

[16] 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.

[17] 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.

[18] Christian Witzler, J. Zavala-Aké, Karol Sierociński, and Herbert Owen. Including in situ visualization and analysis in pdi. In *High Performance Computing*, pages 508–512, Cham, 2021. Springer International Publishing.