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

D4.1

Intermediate 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.eocoe2.eu
	Deliverable ID:	D4.1
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M18 30/06/2020
	Actual Date of Delivery:	M18 30/06/2020
	EC Project Officer:	Evangelia Markidou

* - 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 :	Intermediate report on I/O improvement for EoCoE codes
	ID :	D4.1
	Available at:	http://www.eocoe2.eu
	Software tool:	LaTeX
Authorship	Written by:	Julien Bigot (CEA), Konstantinos Chasapis (DDN), Agostino Funel (ENEA), Francesco Iannone (ENEA), Kai Keller (BSC), Sebastian Lührs (FZJ), Karol Sierocinski (PSNC), Benedikt Steinbusch (FZJ), Bérénice Vallier (RWTH), Christian Witzler (FZJ)
	Contributors:	Jean-Thomas Acquaviva (DDN), Pierre-Aimé Agnel (DDN), Fiorenzo Ambrosino (ENEA), Leonardo Bautista Gomez (BSC), Johanna Bruckmann (RWTH), Jens Henrik Göbbert (FZJ), Tomasz Paluszkiewicz (PSNC)
	Reviewed by:	PEC, PBS

Document Keywords: I/O, PDI, FTI, Data, Fault Tolerance, Insitu

Contents

List of Figures

List of Tables

1 Executive summary

This intermediate mid-term report provides an overview of the I/O and dataflow work package of EoCoE-II. The work presented within this report focuses on the four major I/O objectives, which are targeted by the individual work package activities: Improvement of I/O accessibility, improvement of I/O performance, resiliency handling, overall data size reduction. The individual improvements are directly linked to specific scientific challenges of EoCoE-II or serve as a generic improvement, useable by multiple applications.

In context of I/O accessibility many new requested and generic features were added to the portable data interface PDI, which were then integrated and usable by different scientific challenge applications, mainly for requests in context of the Gysela and the ParFlow application. The features either allow to perform new types of data handling operations (such as specific data formats) or helps to simplify the library utilization and integration in general (by allowing advanced logging and dependency handling). The overall interface work allows to move the data handling aspect into a central middleware, while further improvements could be handled in detached plugins, not necessarily in the individual applications. Beside the new features of the library also support for the PDI application integration was handled as part of this activity.

The improvement of I/O performance started with two activities: Due to the relevance of PDI within the overall EoCoE-II software stack, a benchmarking activity for PDI within the IOR benchmark was established to better judge on the overhead of the library. This integration can also serve as an additional testbed for further PDI improvements. Another large activity was the integration of the IME API into SIONlib. IME is a special hard- and software environment dedicated to filesystem based caching, developed and distributed by DDN. As this type of technique is quite relevant for large-scale applications (to be able to read or write chunks of data as quickly as possible), we integrated the IME specific API into the SIONlib I/O library to allow us to utilize the API within an existing library approach. This integration will be tested next for Gysela on a large-scale IME system.

For the resiliency and fault tolerance work the main tool used and extended in the frame of the project is the fault tolerance interface FTI. In this first half of the project, FTI was integrated into Alya for checkpointing capabilities. In addition, the main development work focused on the combination of FTI and the Melissa framework, which is used for the ensemble handling. This integration will be relevant for example for Parflow were Melissa will be utilized. Beside different other improvements for the library also a new IME backend was added as well, to make FTI aware of this type of hardware infrastructure.

To reduce the overall datasize an in-transit compression activity started for the ParFlow application. By leveraging the capabilities of the latest NetCDF library compression could be integrated into ParFlow, which allows creating compressed output files in parallel. This approach will be benchmarked next.

Overall, many different data related aspects were already achieved during the first half of the project. Many new features within the different libraries were added and now have to be integrated and benchmarked within the scientific challenge applications.

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
EFCOSS:	Environment For Combining Optimization and Simulation Software
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
HYPERstreamHS:	Dual-layer MPI large scale hydrological model including Human Systems
ICON:	Icosahedral Nonhydrostatic model
IME:	Infinite Memory Engine
I/O:	Input and Output
JupyterLab:	web-based user interface for Project Jupyter
MDFT:	Molecular Density Functional Theory
MELISSA:	Modular External Library for In Situ Statistical Analysis
Nemo5:	NanoElectronics MOdeling Tools 5
neXGf:	non-equilibrium eXascale Green's functions

OpenFOAM:	Open Source Field Operation and Manipulation
ParFlow:	PARallel Flow
ParaView:	Multi-platform data analysis and visualization application
PDAF:	Parallel Data Assimilation Framework
PDI:	PDI Data Interface
PPMD:	Performance Portable Molecular Dynamics
ReaxFF:	Reactive Force Field
SENSEI:	Provides simulations with a generic data interface for in-situ visualization
SHEMAT:	Simulator of HEat and MAss Transport
SIONlib:	Scalable I/O library for parallel access to task-local files
SOWFA:	Simulator fOr Wind Farm Application
SPS:	Solar Prediction System
TELEMAC:	TELEMAC-MASCARET system
TerrSysMP:	Terrestrial Systems Modeling Platform
WaLBerla:	A Widely Applicable Lattice-Boltzmann Solver
WanT:	Wannier Transport
WPMS:	Wind Power Management System
WRF:	Weather Research and Forecast model

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

Acronym	Scientific Nomenclature
ABL:	Atmospheric Boundary Layer
AD:	Automatic Diffentiation
AOT:	Aerosol Optical Thickness
PBE:	Perdew-Burke-Ernzerhof functional
BLYP:	Becke-Lee-Yang-Parr functional
COT:	Cloud Optical Thickness
CLM3.5:	Community Land Model version 3.5
CPU:	Central Processing Units
CSP:	Concentrated Solar Power
DA:	Data Assimilation
DFT:	Density Functional Theory
DMC:	Dynamic Monte Carlo
FSI:	Fluid-Structure Interaction
GPU:	Graphical Processing Unit
HLST:	High Level Support Team
HPC:	High Performance Computing
ITER:	International Thermonuclear Experimental Reactor
KMC:	Kinetic Monte Carlo
LES:	Large Eddy Simulations
MD:	Molecular Dynamics
MPI:	Message Passing Interface
NEGF:	Non-Equilibrium Greens functions
NREL:	National Renewable Energy Laboratory
NWP:	Numerical Weather Prediction
OED:	Optimal Experimental Design
PBC:	Periodic Boundary Conditions
PDAF:	Parallel Data Assimilation Framework
pdf:	probability density functions

PF-CLM:	Parflow-Community Land Model
QMC:	Quantum Monte Carlo
QM:	Quantum Mechanics
SHJ:	Silicon HeteroJunction
SOL:	Scrape-Off Layer
WP:	Work Package

3 Introduction

Datahandling in context of I/O operations or complex workflow approaches is a difficult task when applications run on large system scales. For many scientific applications the time, which is necessary to perform I/O operations can even outperform the calculation time itself. Therefore in the context of Exascale preparation a dedicated view on I/O and data challenges is necessary. For this work-package four (WP4) of the EoCoE-II project tries to cover certain major objectives in context of the overall datahandling of the scientific challenge (SC) applications: Improvement of the I/O accessibility, I/O performance, resiliency, data size reduction.

This first WP4 deliverable should provide a status overview concerning the different tasks and activities and their impact for the different SCs. For this the sections of this deliverable are divided alongside the four major tasks of the deliverable: First in section 4 all improvements on the data interface PDI and the support work towards the SC applications is shown. Section 5 covers the work related to I/O refactoring and optimization mainly in the context of leveraging new I/O libraries but also by allowing to utilize new types of Exascale aware I/O Hardware approaches such as the IME-System from DDN. Section 6 focuses on the resiliency aspect to show how applications can be secured against system faults, which can have a significant impact when running on larger scales. Finally section 7 describes the state of the compression and insitu visualisation activities to lower the overall datasize where possible.

Each section is introduced by a Gantt chart, which reports the current state of the activity timelines. The color used within the charts represents the individual connection towards the different SCs: Blue for Water, Grey for Wind, Red for Fusion, Green for Meteo, Yellow for Material, White for generic activities or activities which serve multiple SCs.

4 Data Interface

This task focuses on the Portable Data Interface (PDI)[1] developed by CEA and PSNC. 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 architecture is illustrated in Figure 2.

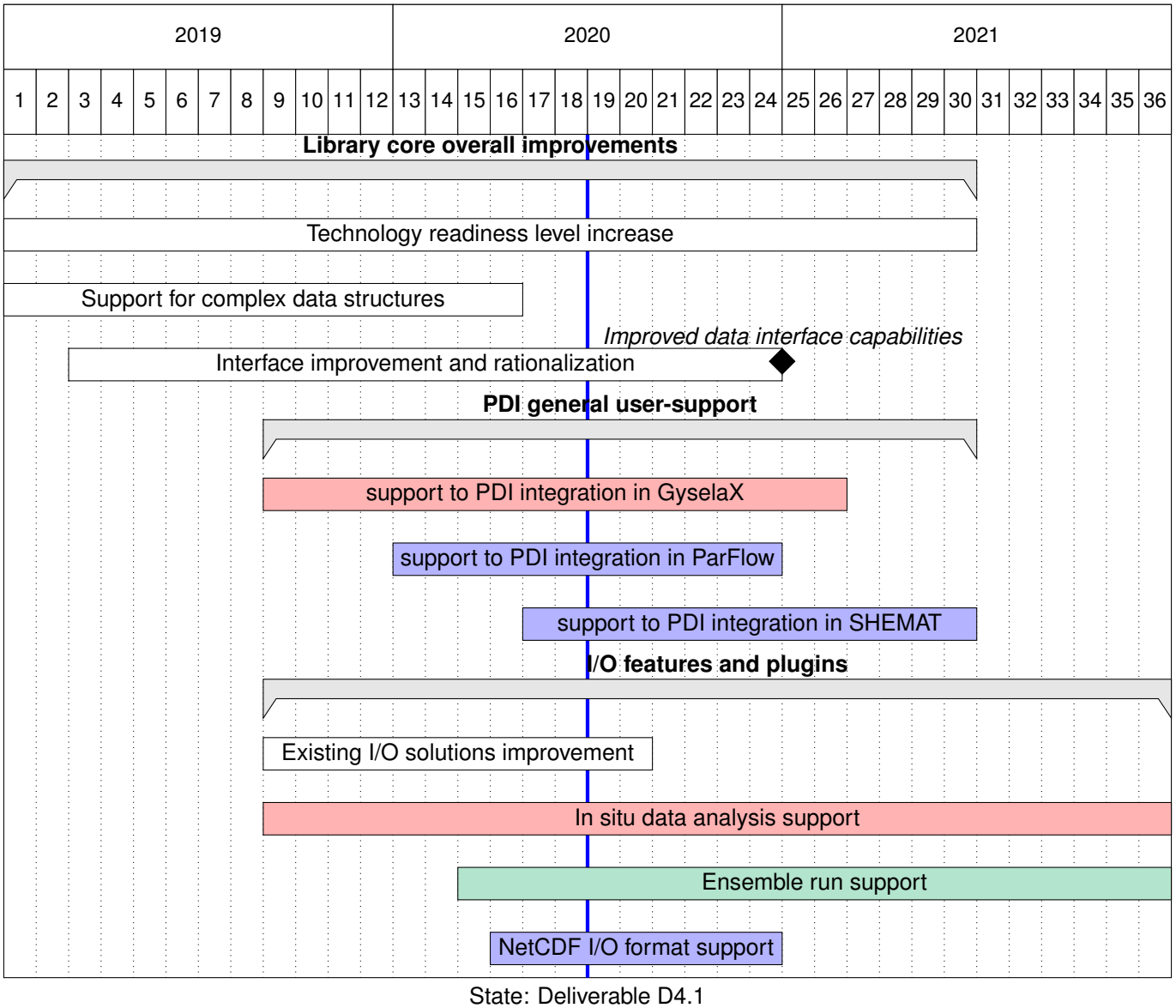

Figure 1: Gantt chart: Task 1 - Data interface

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

[1] `https://gitlab.maisondelasimulation.fr/pdidev/pdi`

performance without impacting their readability by their original developers. Overall, this greatly improves code portability, maintainability and composability.

The "Data Interface" task within WP 4 deals with **a)** the overall improvement of the PDI library core in reaction to requirement identified by SC applications, **b)** support to the integration of PDI in SC applications, and **c)** the development of new features for SC applications. This is done in close cooperation with WP 2 where the integration of PDI in each individual SC application is handled.

As illustrated in Figure 1, the task started with a focus on general improvements to the library to adapt it to the requirements of the project SCs. In a second step, as WP2 started to integrate PDI in the SC codes, user-support became another important aspect of the task. The repository now contains more than 48k lines of code and received more than 194k line modifications over 464 git commits in this framework. While user-support remains an important aspect, focus is now shifting toward the development of new I/O features and plugins for SC codes.

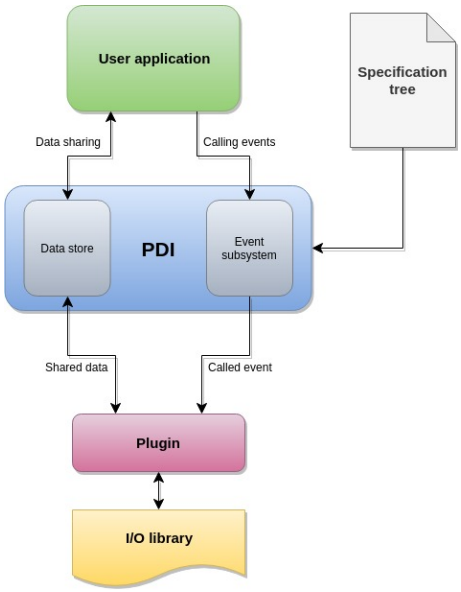

Figure 2: PDI Architecture from the user point of view

4.1 Library core overall improvements

This first aspect of the work tackled in EoCoE-II deals with improvements to the PDI library that do not directly introduce new I/O-related features. Each of these improvements instead improve the accessibility of plugin-provided features to the codes. These developments where guided by requirements identified during the preliminary integration of PDI in Gysela in EoCoE-I and the new integration work in EoCoE-II.

Technology readiness level increase

The PDI library aims to go beyond a research proof-of-concept and to be usable in serious production settings. A huge effort has thus been developed to increase the library technology readiness level. This is especially challenging as PDI is under heavy development; new features are added and other aspects of the software are modified every week. Maintaining development at such a heavy pace is only possible through a strict testing and continuous integration policy. This ensures that the library keeps working as expected. In the PDI development process, each new feature is implemented together with a set of tests to ensure that the new code works correctly. This minimizes the risk for bugs to occur and makes a quick reaction possible if any is found. Documentation[2] is also written alongside each change and feature.

Over 570 unit- and functional-tests have been developed for PDI along the project. All tests have to pass in height different contexts for new code to be integrated into the library master branch. This is accomplished by using a combination of GitLab CI, Jenkins and Docker containers. The Docker containers[3] were specifically tailored to provide context representative of the users and to support older and newer systems.

The PDI approach to improve code modularity eases the integration of third party libraries into simulation codes. This lifts a reason not to use libraries. Another reason often invoked in the HPC community does however remain. Libraries can make the installation of a simulation code with its dependencies difficult; this was specifically mentioned by the GyselaX and SHEMAT use-cases. For this reason a strong effort has been put on packaging PDI, its plugins and the libraries it interfaces.

[2] `https://pdi.julien-bigot.fr/master/`
[3] `https://github.com/pdidev/dockerfiles/`

The PDI source distribution automatically detects missing dependencies and builds them alongside PDI and its plugins. Spack recipes[4] have been developed to ease deployment on supercomputers. Binary packages[5] are provided for many linux distributions including Debian, Ubuntu, Fedora and CentOS.

Interface improvement and rationalization

The plugin-based approach adopted in PDI to expose library features ensures each library can be handled completely independently. This does however also mean that if no specific care is taken, the plugins might end-up offering completely incoherent interfaces. The experience with PDI integration in SC codes has shown that this can be a limitation to the replacement of an I/O strategy by one based on a different library such as HDF5 vs. SIONlib in Gysela or HDF5 vs. NetCDF in ParFlow. To prevent that, the interfaces of all plugin have been reviewed and common features have been identified to be provided to the user with similar interfaces.

PDI integration in SC codes has also shown that without a deep knowledge of the library internals, it can sometimes be difficult to understand some corner-case behaviour. The library has seen some additions to make its behaviour more clear. Logging has been updated and unified in the library and its plugin. The spdlog library is now used to support the same formatting and configuration for every messages whether they come from the PDI library core or a plugin. Logging verbosity is now configurable at the MPI rank and plugin granularity to ease debugging. Each plugin can use its own format to always link a specific message to a specific plugin. A trace plugin is also provided to follow exactly what happens inside the library during an execution.

Another issue that Gysela users faced due to the run-time loading of the specification tree was a potential late discovery of errors. A Python script has been developed to check the syntactic and semantic validity of the file. Each plugin can provide its own validation script to check the plugin-specific part in the file. Thanks to this script, users can ensure before launching their code that the specification tree is valid and will not lead to an early abort of the simulation run and batch scheduler job.

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`. This prevents potential misuses where the user could erroneously call disallowed functions in the sequence.

During the integration of PDI in GyselaX and ParFlow, we identified situations where metadata values would have to be tested to choose whether to enable some I/O operations before the metadata value was first exposed. To solve this, codes had to carefully expose data and metadata in an order compatible with the tests performed for I/O operations in the specification tree. This went against the separation of code and I/O concerns advocated by PDI. A new solution has therefore been devised where metadata values can be directly initialized from the specification tree to solve this chicken-and-egg problem.

Support for complex data types

Unlike GyselaX where all data is stored in contiguous multi-dimensional arrays, the ParFlow use-case enabled us to identify a requirement for handling more complex data structures such as C "structs" with indirections and variable length vectors. Two new features were added to simplify usage and enable users to expose only the root field of such a structure instead of having to iterate over it and expose the buffers at its leaves: data records and indirections support.

Support for data record was introduced in PDI type system and specification tree. A record is constituted of multiple members, each identified by a unique name, that each have a type and

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

whose memory address is computed relatively to the base address of the record. This description makes it possible to support C structs but also Fortran derived types or C++ classes.

Data indirection has also been introduced in the type system. Data indirection is handled as a specific case of scalar data type that is not valued directly but instead references values at another location in memory. The indirection is handled in an opaque way in PDI so that plain C/C++ or Fortran pointers can be handled by this type but also more complex types such as C++ smart pointers for example.

The handling of indirections as a specific case of scalar data types was made possible by the introduction of support for "complex" data types. A complex data type is one that need special care before deallocating its memory or in order to copy it, similar to a C++ non-trivial type. Support has been added in PDI to specify functions handling data copy and destruction instead of the default plain memory copy and no-op destruction. This is for example used by the MPI plugin to provide support for the various opaque data types of the MPI specification.

The support of data types in the specification tree have also been improved to support the use-cases of SC applications. Character arrays exposed by the application can now be used as strings in the specification tree for example to set the name of an output file like in the case of ParFlow. Floating point arithmetic is also supported in the specification tree for example to use them as conditions for I/O as in the case of GyselaX.

4.2 Support work in context of SC applications requests

The integration of PDI in EoCoE-II SC application codes is handled in WP2 since this requires a deep understanding of the codes. The PDI developers do however offer dedicated support to the users tackling this work. Support is provided via email, through the growing PDI community slack channel[6] with over 25 users and more than 7k exchanged messages and via dedicated remote visio support and training.

IOR benchmarks: A performance benchmarking activity for PDI through the well-known IOR benchmark suite has been initiated. IOR allows to generate certain I/O patterns while measuring the performance of different I/O libraries underneath. Dedicated support was provided to co-design benchmarks for PDI which is not an I/O library like other libraries supported by IOR, but an interface to actual I/O libraries. Support for writing a dedicated PDI plugin was initially expected, a dedicated developer documentation for plugin writers[7] was written for the occasion. After further discussion, it was however identified that the existing "User-code" plugin would offer all features required. Support was offered to make the best use of the "User-code" plugin in this context. The specifics of the PDI benchmarking activity are described in section 5.3 of this deliverable.

GyselaX: GyselaX being based on Gysela, an initial integration of the PDI library had already been done in EoCoE-I where support for checkpoint through PDI was implemented. The goal with GyselaX is however to go much further. PDI will be used to simplify the code by removing other I/O implementations from the code-base and by relying on PDI plugin mechanism instead. PDI will be used not only for I/O, but also to modularize the code at many levels including the separation of aspects currently strongly-coupled in the code-base such as "diagnostics". This endeavour is the most advanced use of the library in a production code to date. After re-organization of the project due to one of the main developers of GyselaX leaving, this sub-task in WP2 was however moved directly to the PDI development team at CEA/MdlS. The notion of user-support does thus not really make sense anymore in this context and one should instead refer to the deliverable of WP2.

[6] `https://pdidev.slack.com/`

[7] `https://pdi.julien-bigot.fr/master/how_to_create_plugin.html`

ParFlow: Dedicated support was offered to the integration of PDI in the ParFlow code-base. This support took the form of online coding sessions coupling a ParFlow developer from WP2 and a PDI developer from WP4. The chosen approach was to integrate the library in the code and testing it with the "Decl'HDF5" plugin that was the most advanced at the time. Most of the work focused on the support of the advanced data types used by ParFlow, both regarding their description in the specification tree and the addition of missing features to PDI. A dedicated branch of the PDI library with the required additions was created for this use-case before the features could be progressively integrated in the main branch of the library. The additional developments now required for NetCDF and dedicated ParFlow file-format (PFD) support can henceforth be directly provided on WP4 side as this involved modifications of the specification tree only with no changes on the ParFlow code side.

SHEMAT: Dedicated online training sessions were provided to support the inclusion of PDI in SHEMAT. The sessions specifically focused on:

- the installation process for PDI and its plugins,
- providing a better understanding of PDI philosophy, capabilities and supported features,
- the understanding and completion of PDI tutorial.

Now that PDI expertise has been transferred in SHEMAT development team, the next step will focus on the integration in the actual application. These dedicated session provided useful feedback to improve PDI tutorial[8] and documentation.

4.3 I/O features and plugins

In addition to the work previously mentioned that focuses on the improvement of the core PDI library to support more requirement from application codes and offer support to their developers, another aspect of the work done focused on new development to actually handle the data exposed by codes. In this category, two sub-tasks focus on the support of existing data workflows: the improvement of existing I/O solutions and the support for NetCDF, while two other sub-tasks focus on innovative solutions targeted at pre-Exascale and Exascale machines: in situ data analysis and ensemble runs support.

Existing I/O solutions improvement

A first category of improvements are new I/O features that were implemented in existing plugins to offer a more extensive support of the interfaced libraries. For example, the FTI plugin that was designed in 2015 and whose feature-set covered the features available then has been reworked from scratch to cover the complete feature-set of FTI-1.3. This work was made possible thanks to the synergies due to the EoCoE project where an engineer came to work on both PDI and FTI; thus acquiring the deep knowledge that made this possible. The new features of FTI are now available in codes whose checkpoint process is PDI-enabled like Gysela.

The "Decl'HDF5" plugin was also revamped with:

- support for a new more expressive configuration format in the specification tree,
- improved support for both sequential and parallel HDF5,
- support for memory and dataset selections,

[8] `https://github.com/pdidev/tutorial`

- support for storage of records (C structs, Fortran derived datatypes or C++ classes) in HDF5 compound data types.

These new feature constitute the core of the HDF5 implementation in the transition from Gysela to GyselaX. They have also been used to provide the first PDI output format for ParFlow and will likely be used in SHEMAT also.

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.

In situ data analysis support

While supporting various general purpose I/O libraries (HDF5, SIONlib, NetCDF, ...) and domain-specific I/O libraries (FTI, ...) is important, PDI truly shows its potential in terms of modularity increase with in situ data analysis. With the increasing performance gap between compute and storage in supercomputers, the application of smart data reduction strategies is becoming a must have to reach Exascale performance without being limited by I/O. In situ data analysis and transformation through PDI enable users to apply data transformations including data reductions on the buffers generated by simulation codes before writing them to disk. This approach is at the core of the modularization of the GyselaX code but will also used in ParFlow for the dedicated file format support at least and is also used for the IOR benchmarks. This approach is specifically supported by three plugins that have been developed for PDI in the framework of the project.

The “pycall” plugin enable users to apply process-local transformations on the data exposed by the simulation code using the Python language and ecosystem. The plugin takes care of instantiating a Python interpreter in each process of the simulation application if required. It uses this interpreter to execute the data handling procedures directly described in Python in the specification tree. It handles the memory allocation interface between languages with explicit memory management like C, C++ and Fortan and the automatic garbage collected management of Python. The plugin supports Python numpy arrays to ensure the best possible performance and takes care of the required trans-typing to expose objects from C/C++/Fortran to Python and inversely.

The “User-code” plugin enable users to apply process-local transformation on the data exposed by the simulation code using compiled languages that support the C calling interface like C, C++ or Fortan. This is very similar to the “pycall” plugin, but can be used either to access features from libraries with no Python API or when the performance offered by Python are not sufficient. The plugin works by executing functions provided in a separate object file and called by their name from the specification tree. The functions can use the `PDI_access` function of PDI to access data shared by the simulation.

These two plugins offer very similar features and APIs and can be used in a chain of data manipulations where the data exposed by the code has one treatment applied to it before being exposed to PDI again to be transformed again, and again and again until the last treatment in the chain writes the data to permanent storage. This approach makes it easy not only to insert new treatments in situ but also to start developing these treatments offline before integrating them in situ. It is indeed very simple when using PDI to go from a workflow where data is written to disk by the simulation code, then read transformed and written again by the post-processing tool to another one where the data does not have to go through disk just by changing a few lines in the specification tree.

In addition to these two plugins for in-process data processing, “FlowVR” plugin has been developed for in transit data processing on dedicated processes that are either located on the same nodes as the simulation code or even on dedicated nodes. This plugin interfaces with the FlowVR 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 the specification tree.

The approach supported by the combination of these three plugins enable application developers to focus on the core application development while supporting efficient in situ data processing without having to embed the complexity in the simulation code. If the technical foundations are now in place, an important aspect of the second half of the project will consist in the use of this approach in production settings. Needs for the GyselaX code have already been clearly identified, and discussions are planned with the other SC codes to determine whether this would be pertinent in their case.

Ensemble run support

Similarly to in situ processing, ensemble-run simulations as advocated in WP5 offer an interesting approach to leverage Exascale hardware. This can however only be achieved if communications between the ensemble member simulations and the aggregation process do not go through disk. This is the service offered by the Melissa software developed in WP5, but adding Melissa support to an existing simulation code in order to use it as an ensemble member requires dedicated modification to its code-base.

A PDI "Melissa" plugin is therefore developed in parallel to Melissa-DA itself. Once again the synergies offered by EoCoE enable us to share manpower and make the implementation of such software at the interface between two work-packages much easier. The plugin has not been integrated in the main PDI branch since the Melissa-DA API and feature-set is not frozen yet, but its development alongside the main Melissa-DA code should ensure that as soon as this is done, the plugin will be ready.

The next step will consist in using this plugin from the production ParFlow code when using it as part of an ensemble run. Since the data is already exposed from ParFlow side and when the plugin is ready, one can expect this work to go relatively smoothly and should provide a good evaluation of the advantages of the PDI approach.

NetCDF I/O format support

The "Decl'NetCDF" plugin is currently under development. It wraps the NetCDF library in a declarative way similarly to what the "Decl'HDF5" plugin does for HDF5. The organization of the specification tree for both plugins is similar enough that switching from one plugin to the other should be easy. This plugin is tested along its development with the ParFlow use-case. The goal is to generate through this plugin the exact same NetCDF files as those currently generated by the in-code ParFlow NetCDF support so as to seamlessly transition from the current situation to the PDI enabled version.

4.4 Future roadmap

The second half of the project regarding the data interface will follow the plan outlined in Figure 1. Increase of the technology readiness level will keep being an important aspect to ensure all other developments undertaken can be used in a production environment. The work on the improvement and rationalization of the interface will continue to converge for a feature and ABI freeze in time for the "improved data interface capabilities" milestone of the project that should be concretized by the release of version 1.0 of the library.

The integration of the library in SC codes will continue with a shift from the basic integration in the codes toward the development of features dedicated to the improvement of I/O in these codes. Support resources will always be made available to new SC codes that would like to integrate PDI in their code base even if this requirement has not been identified yet.

Interfacing the NetCDF library will be finished and new libraries might be interfaced if requirements emerge from the SC codes. In the current situation however, the plan is to put a strong effort on in situ data analytics and ensemble run support. For in situ data analytics, we will evaluate the possibility to interface PDI with dedicated distributed data analytics libraries like Dask[9]. This approach could bring great rewards as this would open a new world of possible by making one of the major parallel data analytics library with support for distributed data and machine learning available to codes. It is however a challenge as the philosophy and thus the concepts implemented by these libraries considerably differ from those used in HPC.

[9] https://dask.org/

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. Beside of leveraging different I/O library capabilities and testing different configuration options the tasks also focuses on the introduction of new types of I/O storage hardware such as intermediate cache devices like IME from DDN, which will become very relevant in context of Exascale ready approaches.

Leveraging this new type of storage infrastructure by introducing its API into the SIONlib I/O library was also the main activitiy of this task, which was performed within the first half of this project. Beside this activity the task mostly started to evaluate the overall PDI middleware performance by introducing PDI into IOR. The SC related activties had to be shifted to the second half of the project as the hiring process in context of the different SCs and the general I/O integration work took longer then expected.

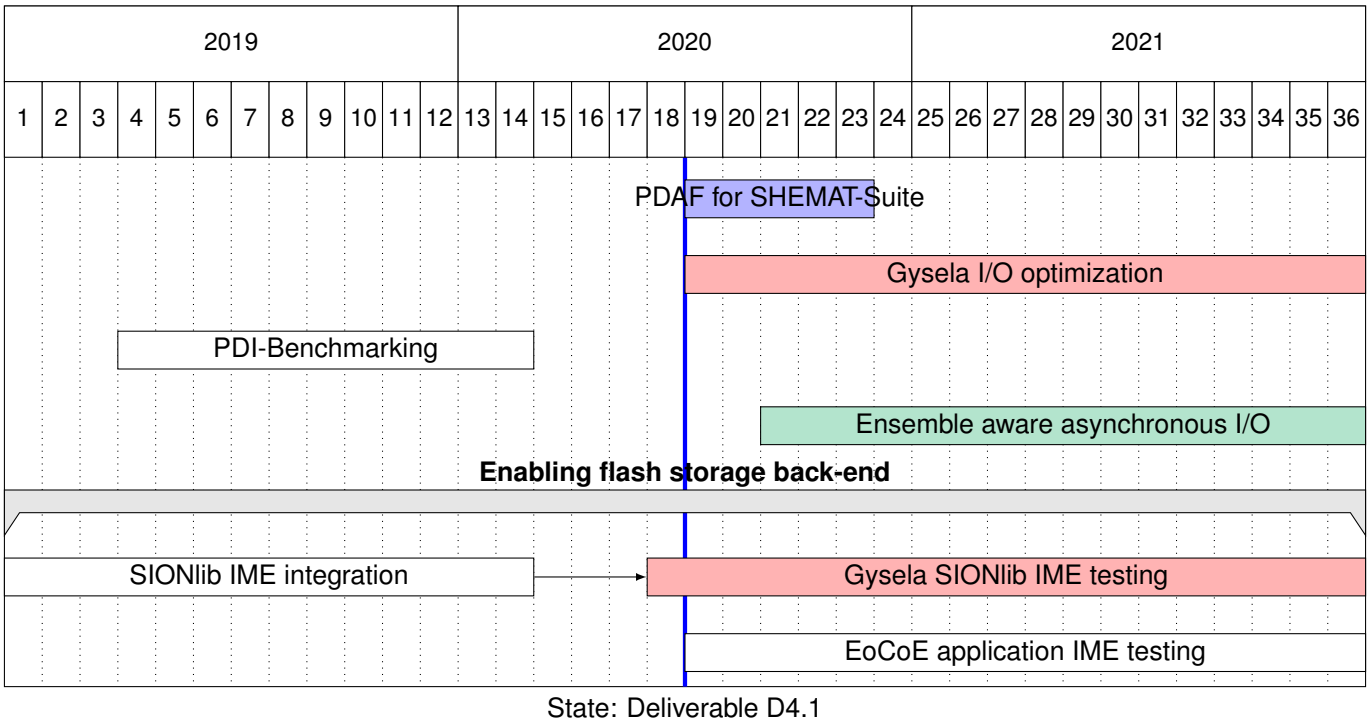

Figure 3: Gantt chart: Task 2 - I/O refactoring and optimization

5.1 PDAF Status for SHEMAT

A post-doc has been hired in the end of 2019 to fulfill this task of integrating the Parallel Data Assimilation Framework (PDAF) into SHEMAT-Suite. This activity is closely connected to activity 2.5.2 in WP2. The integration will be explained in deliverable D2.2 of WP2. In the following section, we aim to highlight the principle of PDAF and its benefits for the data assimilation with SHEMAT-Suite in context of geothermal reservoir modeling. We will present:

- Principle of the PDAF software ;
- PDAF assimilation data structure ;
- Parallel data assimilation system and integration into SHEMAT-Suite ;
- Benefits of the PDAF optimization process on SHEMAT-Suite.

The deliverable presents a preliminary work as the implementation is still on going in WP2. The main task will begin from mid-May to November 2020. The status of the work is as follows: the integration of PDAF into SHEMAT-Suite is currently on going. The integration process itself is handled within WP2. After successful integration, we aim to test the performance by using two- and three-dimensional transient test models.

To be noted, the efficiency and productivity decreased due to the Covid-19 pandemic and subsequent lockdown during the past months. The delay of progress in task 2.5.2 including the PDAF implementation will have a delay which will result in a lack of time for finalizing this current task.

Principle of the PDAF software

The Parallel Data Assimilation Framework (PDAF) is a software environment for ensemble data assimilation. More details on PDAF are available on the official website: `http://pdaf.awi.de/trac/wiki` and the reference work of [6]. It simplifies the implementation of the data assimilation with existing numerical models which can run on big parallel computers such as SHEMAT-Suite. With this, users can obtain a data assimilation system with less work and can focus on applying data assimilation. To simplify the implementation of parallel data assimilation, PDAF contains fully implemented and optimized ESKF algorithms in particular the Ensemble Kalman Filter (EnKF) and nonlinear filters. This task focuses on integrating the EnKF via PDAF; other filters available in PDAF might be added to SHEMAT-Suite in the future. The main point is that PDAF provides a parallel ensemble data assimilation framework. The current version of EnKF in SHEMAT-Suite is not itself parallel but serial ([4]). Thus, the current EnKF version does not allow us to perform the data assimilation with large ensembles of reservoir-scale models. This is why we aim to integrate the PDAF providing parallel data assimilation for SHEMAT.

PDAF assimilation data structure

Fig. 4 corresponds to the logical structure of the assimilation system of PDAF. The latter is based on a consistent structure of the three components of the data assimilation system: (i) Model: the numerical model includes the initialization and integration of all model fields. The model corresponds to the dynamics of the simulated system ; (ii) Filter algorithm: the observations from the system contains additional information ; (iii) Observations: the filter algorithms combine both the model and observational information. The filter algorithms are part of PDAF, while the model routines and the ones handling observations are provided by the user. A standardized interface for all filter algorithms connects the three components.

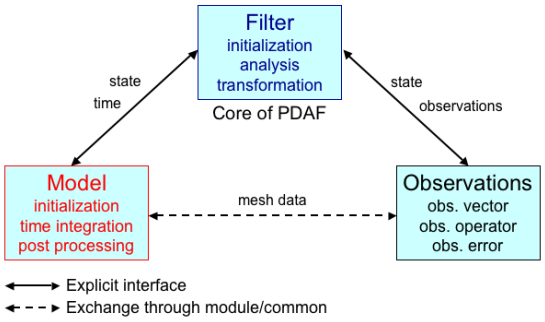

Figure 4: Logical separation of the assimilation system. From the official website: `http://pdaf.awi.de/trac/wiki`.

Only minimal changes to the model source code are required when combining a model with

PDAF in its online mode. In the online mode, the model code is extended by calls to PDAF core routines. A single executable is compiled. While running this single executable the necessary ensemble integrations and the actual assimilation are performed. An offline mode is possible separating programs for model and filtering. The offline mode avoids changes to the model code but leads to a smaller computing performance. We decided to follow the online coupling approach (data exchange via main memory and not running the model as a single executable) in order to avoid frequent re-initializations of the model and a significant overhead in I/O operations which both degrade the performance of the constructed data assimilation system for SHEMAT-Suite.

Parallel data assimilation system and integration into SHEMAT-Suite

Fig. 5 shows the parallel assimilation system of PDAF. PDAF not only provides fully implemented and parallelized ensemble filter algorithms, but also provides support for a 2-level parallelization for the assimilation system: (i) Each model task can be parallelized; (ii) Several model tasks are executed concurrently. Thus, ensemble integrations can be done fully parallel. In addition, the filter analysis step uses parallelization. In the online mode of PDAF, all components are included in a single program.

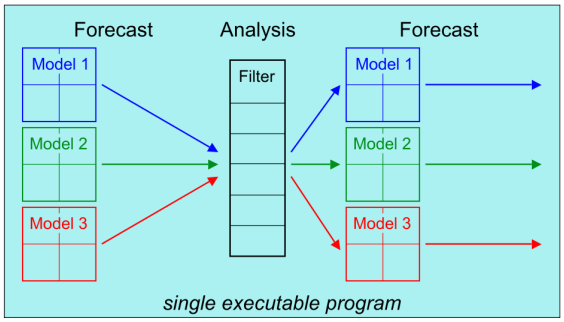

Figure 5: Parallel data assimilation system of PDAF from [7].

Here is explained in more details how PDAF will be integrated into SHEMAT-Suite:

- At the very beginning of the initialization phase of the model, a PDAF routine needs to be called that establishes the parallel communication within the model and the data assimilation algorithms. In this phase, PDAF creates three parallel communicators: The model communicator, the coupling communicator and the filter communicator. The general layout of these communicators is depicted in Fig. 6 for a model setup with three ensemble members and four processors by model realization. The coupling communicator is the communicator for exchanging data between the processors in the filter communicator and the remaining ensemble members before and after the assimilation step. Then all processors first read a common input file which holds information about specific settings for the data assimilation run. Afterwards, the data structures for the data assimilation in PDAF are created.

- After this initialisation phase of PDAF implemented to SHEMAT-Suite, the time loop over the assimilation cycles takes place. For each assimilation cycle, first SHEMAT-Suite is advanced to the next observation time. Then the data assimilation algorithm in PDAF is called. In this step the model state vectors are collected on the filter communicator with the help of the coupling communicator (see Fig. 6). The EnKF filter is performed. After the filtering step, the updated state vector is transferred back to the corresponding model variables. SHEMAT-Suite associated to PDAF then proceeds to the next assimilation cycle. When all assimilation cycles are finished, the data structures of the individual components of SHEMAT-Suite associated to PDAF are deallocated and the process is turned off.

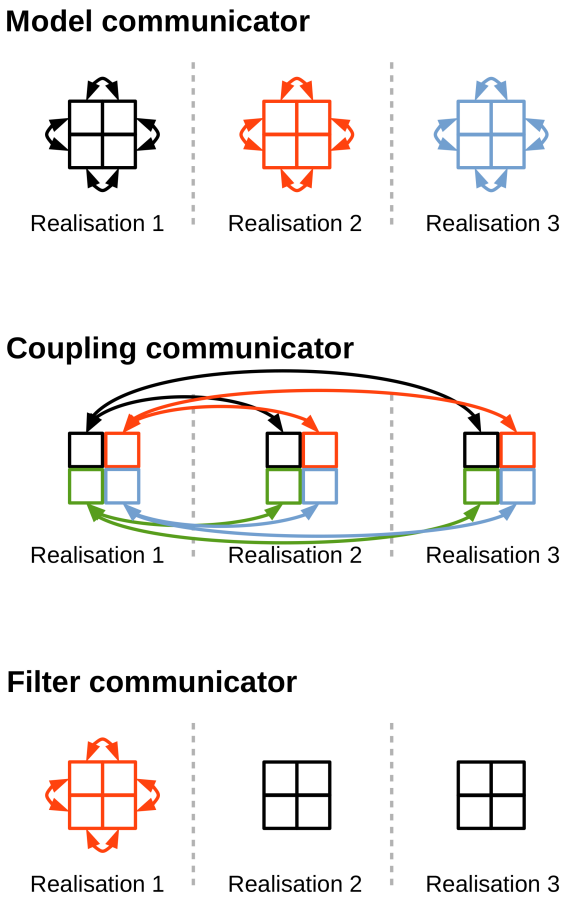

Figure 6: Communicators in PDAF for a parallel setup with 3 ensemble members and 4 processors per ensemble member from [5].

Benefits of the PDAF optimization process on SHEMAT-Suite

SHEMAT-Suite (Simulator for HEat and MAss Transport) is a transport simulation code for a wide variety of thermal and hydrogeological problems in two and three dimensions ([3]; [8]). Specifically, SHEMAT-Suite can simulate flow, heat and species transport through saturated porous media. It uses Fortran, and OpenMP. MPI is used in certain branches but not in the master. SHEMAT-Suite includes parameter estimation and data assimilation approaches, both stochastic (Monte Carlo, ensemble Kalman filter) and deterministic (Bayesian inversion using automatic differentiation for calculating derivatives).

We aim to integrate PDAF instead of the ensemble Kalman filter (EnKF) already installed in SHEMAT-Suite for the stochastic inversion. Then, PDAF will allow to optimize the process of stochastic inversion and data assimilation. Even if it does not have so much impact on the entire Input/Output (I/O) process, the optimization with PDAF is related to the data assimilation. Moreover, the parallelism of the data assimilation system by PDAF allows the improvement of the I/O processes of the ensemble runs. Thus, the PDAF optimization can be comprehend as I/O and DataFlow related.

The main benefit of PDAF is that a very good scalability is provided through the complete parallelism of all parts of the assimilation system such as the ensemble integration and filter algorithms. It has been developed to simplify the implementation of data assimilation systems. It can be used to test assimilation methods but is also applicable for realistic data assimilation applications. The online mode of PDAF allows us to provide only minimal changes to the SHEMAT-Suite

source code.

A similar work of implementation has been made on the Terrestrial System Modelling Platform (TerrSysMP) published in the work of [5]. They highlight many benefits to the implementation of PDAF. First, the infrastructure for data assimilation has only to be initialized once and the data assimilation system is continuously integrated forward in time without the need of system calls to the model or re-initialization of any of the system components. PDAF is fully parallelized which significantly reduces the memory requirements of the system. [5] provides also a scaling study on the massively parallel environment. They show also that the scientific simulations at large-scale benefits from an efficient implementation of the filtering step provided by PDAF.

5.2 Ensemble aware asynchronous I/O

Writing large portions of data can utilize a significant amount of time of the overall application runtime. Typically, the general calculations are stopped until the reading or writing part is finalized. In contrast to this, overlapping I/O operations with further calculations can help to better utilize the available ressources. In this activity it will be tested to handle at least the main output generation of the ESIAS application in an asynchronous manner, either by offload the relevant I/O routines to dedicated processes or by utilize remaining idle time of the same processes if available. The activity was shifted to the second half of the project due to some delay in the hiring process for the ESIAS application.

5.3 PDI-Benchmarking

The I/O runtime is being deployed by using the Parallel Data Interface (PDI) as a middleware layer able to decouple specific I/O functions from the various EoCoE applications. PDI supports a plugin system to integrate existing I/O libraries, such as: POSIX or HDF5. Several new plugins and improvements for PDI are under development as part of WP4. To better judge on the overhead of the PDI middleware, beside including it into the EoCoE applications, a benchmarking setup was developed using the standard I/O benchmarking tool IOR[10].

IOR is typically used for testing the performance of file systems using various interfaces (MPIIO, HDF5, PnetCDF, POSIX) and different access patterns. This allows to utilize IOR to run different I/O access pattern in a reproducible manner. This setup allows to easily test different access patterns without involving the full overhead of a real application.

Setup and development

The integration of PDI in IOR has been developed by using POSIX as backend interface. The PDI `usercode` plugin was utilized to introduce the relevant POSIX calls. This is the most easiest way to introduce user specifc routines underneath of the PDI middleware layer without the need to develop a full new PDI plugin. Listing 1 contains the full plugin code.

```
1  void xfer_pdi() {
2      // arguments of the function
3      int* access;  PDI_access("access", (void**)&access, PDI_IN);
4      long* length; PDI_access("ts", (void**)&length, PDI_IN);
5      long* offset; PDI_access("offset", (void**)&offset, PDI_IN);
6      int* fd;      PDI_access("file", (void**)&fd, PDI_IN);
7      long l=*length;
8      char* pptr;
9      if (*access == READ) { //read
10         PDI_access("value", (void**)&pptr, PDI_OUT);
11     } else { // write
12         PDI_access("value", (void**)&pptr, PDI_IN);
13     }
14     long long rc;
```

[10] IOR benchmark: `https://github.com/hpc/ior`

```
15      long long remaining = (long long)*length;
16      // function body
17      if (*access == READ) {
18          if (verbose >= VERBOSE_4) {
19              fprintf(stdout,"task %d reading from offset %lld\n",rank,*offset + length -
    remaining);
20          }
21          rc = read(*fd, pptr, *length);
22      } else {
23          if (verbose >= VERBOSE_4) {
24              fprintf(stdout, "task %d writing to offset %lld\n", rank, *offset + length -
    remaining);
25          }
26          rc = write(*fd, pptr, *length);
27      }
28      // release all metadata
29      PDI_release("access");
30      PDI_release("file");
31      PDI_release("ts");
32      PDI_release("offset");
33      PDI_release("value");
34      // update return value
35      PDI_expose("return", &rc, PDI_OUT);
36  }
```

Listing 1: PDI usercode plugin for Posix interface

In addition to the plugin, the relevant PDI calls had to be introduced into IOR in form of an additional IOR backend (`aiori-PDI.c`). For this the IOR prototype functions

- `void *PDI_Create (char *testFileName, IOR_param_t * param);`
- `int PDI_Mknod (char *testFileName);`
- `void *PDI_Open (char *testFileName, IOR_param_t * param);`
- `static void PDI_Fsync (void *fd, IOR_param_t * param);`
- `static IOR_offset_t PDI_Xfer (int, void *, IOR_size_t *, IOR_offset_t, IOR_param_t *);`
- `IOR_offset_t PDI_GetFileSize (IOR_param_t * test, MPI_Comm testComm, char *testFileName);`
- `PDI_Delete (char *testFileName, IOR_param_t * param);`
- `void PDI_Close (void *fd, IOR_param_t * param);`
- `option_help * PDI_options (void ** init_backend_options, void * init_values)`

were implemented. Finally to utilze the new usercode based plugin underneath of the implemented PDI calls a YAML baser PDI-configuration file was added.

Execution

The benchmark runs were perfomed on HPC CRESCO6 system by ENEA. CRESCO6 consists of 434 nodes for a total of 20832 cores. It is based on 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. The high performance file system is based on GPFS Spectrum Scale with 6 NSD nodes able to provide parallel I/O.

In order to run IOR benchmarks on the CRESCO6 system, a JUBE[11] configuration of the IOR benchmarks has been deployed. JUBE has been configured to run IOR on distributed mode,

[11] JUBE Benchmarking environment: `http://www.fz-juelich.de/jsc/jube`

with sequential jobs scheduled within the LSF platform resource manager. The results of the IOR benchmark integrated with the PDI library for the POSIX interface have been compared with the default IOR POSIX case without involving the PDI layer to measure not only performances but also the overhead introduced by PDI. The experiments have been conducted for the following configurations:

- number of nodes (N) = 1, 2, 4, 6, 8
- task per node (TS) = 1, 2, 4, 8, 16, 32

The total number of tasks (TT) for each combination is thus (TT) = (N)·(TS). For each combination (TT) the block size has been set to the four distinct values: 128 kiB, 1 MiB, 4MiB, 256 MiB to explore the effect of increasing I/O buffers. One file per task was created. Four benchmark sessions were carried out. Three of them during the stop of production of ENEA HPC CRESCO6 cluster, when the IO bandwidth of the infrastructure is noiseless caused by users jobs. The overall datasize for each task was limited to 512 MiB.

Results

The average value of the four benchmark sessions are plotted in figure 7, whilst the error bars are the variance of the four benchmark results. The runs show only a minimal bandwidth difference between the default POSIX and the PDI based approach. For most runs this difference is even accumulated by the variance of the individual runs. The overall impact of the PDI middleware is very low, while the most time consuming aspects depends most on the individual plugin implementation.

Now having this IOR based PDI benchmark approach also allows to verify other PDI plugins in the future, by adapting the used PDI configuration file and will help as an additional example for the application implementations.

Figure 7: Average IOR write/read bandwidth benchmark results for different blocksizes utilizing POSIX and PDI based backends.

5.4 Gysela I/O optimization

This activity will focus on the Gysela application. The I/O approach of Gysela (mainly used for checkpointing) will be benchmarked and different library specifc configuration options will be tested.

5.5 Flash Storage Integration (SIONlib with IME)

In order to benefit from the performance that flash storage devices provide, we leverage DDN's Infinite Memory Engine (IME) product. We place IME as a distributed cache at the lower part of the I/O path between the client application and the storage servers. IME can be utilized directly without any modification to any POSIX-compliant application code by using IME-FUSE implementation on the client side. However, all FUSE implementations require crossing the user space to the kernel space as illustrated in Figure 8. This incurs a limitation to the performance gain IME can achieve with the IME-FUSE implementation.

To overcome this issue, IME exports also a so called *IME Native API I/O* that provides the best possible performance of IME. This non-POSIX compliant, low-level I/O protocol bypasses the kernel, communicating directly with IME Client as depicted at Figure 9. IME Native API directly implements the functional chain from the client applications to IME eliminating any performance loss due to extra layering. To use the IME Native API I/O interface, the applications require modifications in the I/O calls and a link with the DDN-supplied IME client libraries.

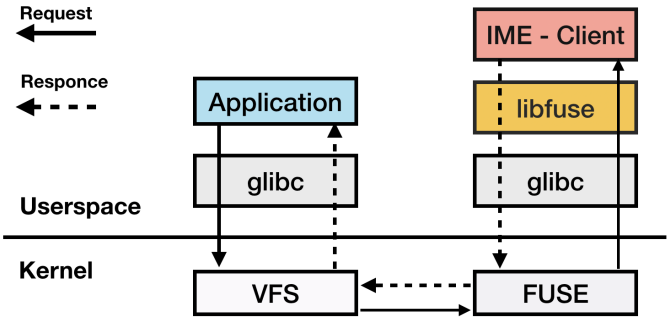

Figure 8: IME FUSE I/O path.

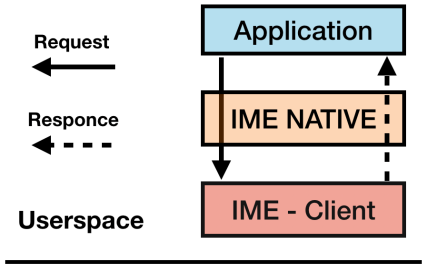

Figure 9: IME Native API I/O path.

For our case the *application/user* of IME is SIONlib (Scalable I/O library for parallel access to task-local files). SIONlib has an abstraction layer on top of the POSIX and Standard C APIs to perform I/O to the storage system. We have modified SIONlib to include also IME Native API as an I/O interface as shown in Figure 10. To use the IME Native API interface, the user has to include the string `"ime"` among the `file_mode` argument of any of SIONlib's `open` functions. Table 4 lists the functions from the SIONlib abstraction layer that had to be modified, the functions that are called when the POSIX interface is being used and the equivalent functions in the IME Native API.

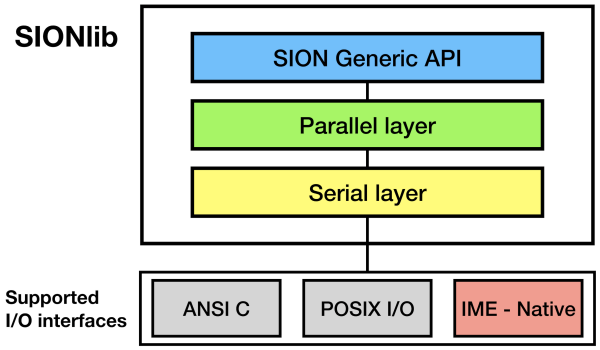

Figure 10: SIONlib I/O architecture with IME Native.

Table 4: SIONlib abstraction layer functions, POSIX, and IME Native API I/O.

SIONlib functions	POSIX functions	IME - Native API functions
_sion_file_open()	open()	ime_client_native_open()
_sion_file_close()	close()	ime_client_native_close()
_sion_file_read()	read()	ime_client_native_read()
_sion_file_write()	write()	ime_client_native_write()
_sion_file_stat_file()	stat()	ime_client_native_stat()
_sion_file_flush()	fsync()	ime_client_native_fsync()
_sion_file_set_position() _sion_file_get_position()	lseek()	ime_client_native_lseek()

In the next months we will evaluate the performance benefit of IME. For this it was necessary to receive access to a large IME installation, which now becomes available at the Jülich Supercomputing Centre in form of the so called "High performance storage tier" (HPST), which will be made available to the PRACE resources utilized by EoCoE-II. First, we will use a micro-benchmark already included in SIONlib called *partest*. Moreover, we are extending the IOR benchmark so that it will support also SIONlib as an I/O interface. With the modified version of IOR we will be able to run synthetic benchmarks and test the integration of IME and SIONlib more rigorously. As a last step, we will leverage the SIONlib support in PDI and test the Gysela application.

For the same reasons as for SIONlib, IME-Native API has also been added as an I/O interface for *FTI*. For more details see section 6.

Beside the SIONlib based approach, the FUSE based access will be benchmarked as well by running different SC applications on top of the HPST system.

6 Fault Tolerance

Data resiliency and fault tolerance are meant to avoid significant data loses after a hardware or software related problem. Typically, this is achieved by writing intermediate calculation data to storage elements to keep those data available in the case of a failure.

This task focuses on enabling performant fault tolerance capabilities for different SC use cases. For this the Fault Tolerance Interface (FTI) is utilized, adpated and improved.

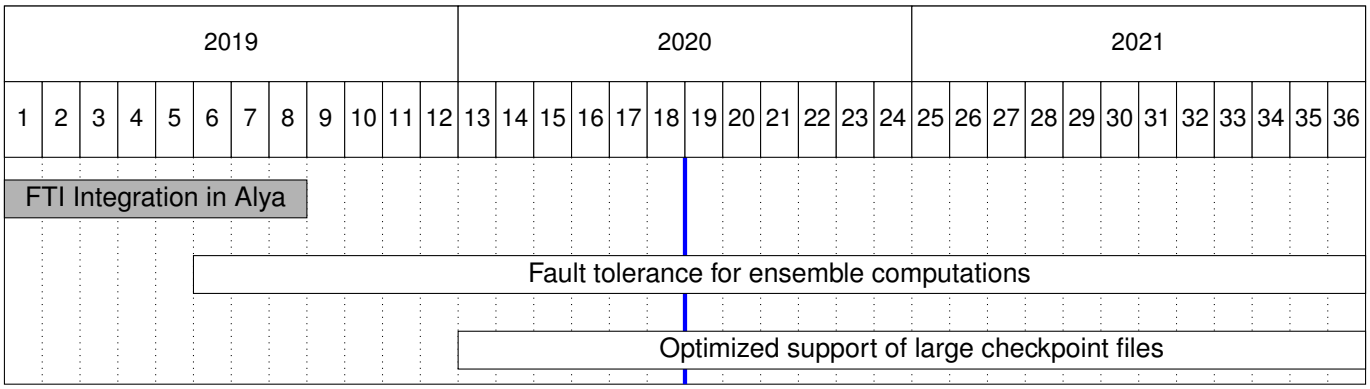

Figure 11: Gantt chart: Task 3 - Fault Tolerance

6.1 Fault Tolerance Interface (FTI)

FTI contains interfaces to several I/O libraries. Most importantly for this work, the HDF5 interface operating in N-1 mode (N processes write into one file), and the *incremental checkpointing* (ICP) mechanism. The HDF5 N-1 mode allows performing elastic restarts and to structure the data inside a global portable checkpoint file. ICP allows adding protected variables independently to the checkpoint file, and overlapping computation and checkpoint IO in some cases.

HDF5 Shared Checkpoint-File

The motivation behind the integration of HDF5 into FTI was to provide access to the checkpoint files to perform preliminary data analysis or visualization. However, it can also be used to perform elastic restarts, i.e., the restart with a different number of processes. FTI provides a set of functions that allow to define a global dataset and add subsets to it. A simple 1-D example of how to realize this is given in listing 2. Assuming that for this simple example the number of total elements is a multiple of the number of processes, the subset-size and offset is automatically aligned to the number of processes of the current run. With the information that is exposed by `FTI_DefineGlobalDataset()` and `FTI_AddSubset()`, the library will load the correct subset from the checkpoint upon recovery determined by the current decomposition.

```
1  /* assuming Ng is multiple of number of processes */
2  unsigned long Ng;                  // global number of elements
3  unsigned long Nl = Ng/num_procs;   // local number of elements
4  unsigned long offset = rank * Nl;
5
6  int* subset = (int*) malloc( Nl * sizeof(int) );
7
8  FTI_DefineGlobalDataset(0, 1, &Ng, "SHARED DATASET", NULL, FTI_INTG);
9
10 FTI_Protect(0, subset, Nl, FTI_INTG);
11 FTI_AddSubset(0, 1, offset, &Nl, 0);
```

Listing 2: 1-D example of adding a subset to a shared dataset.

FTI allows leveraging one process per node to create the single file on the parallel file system (PFS) in the background, i.e., asynchronous to the application. In that case, the application processes write the owning checkpoint data to node storage (N-N) and the dedicated FTI processes consolidate the local checkpoint data to one shared HDF5 file on the PFS. This accelerates the checkpoint creation significantly.

Incremental Checkpointing (ICP)

Sometimes, the buffers that will be incorporated into the checkpoint are private or transient, for instance when serializing classes in C++. In that case, it is necessary to add the buffers incrementally to the checkpoint. However, the ICP mechanism can also be exploited to overlap checkpoint IO and computation. For example when part of the checkpoint data is involved in GPU computations, independent data can be added to the checkpoint in parallel to the GPU computation (for instance, force and velocity in n-body simulations). Listing 3 shows a simple example of ICP with FTI.

```
int i[3];
FTI_Protect(0, &i[0], 1, FTI_INTG);
FTI_Protect(1, &i[1], 1, FTI_INTG);
FTI_Protect(2, &i[2], 1, FTI_INTG);
FTI_InitICP(level, is_time_for_checkpoint());
FTI_AddVarICP(0);
FTI_AddVarICP(1);
FTI_AddVarICP(2);
FTI_FinalizeICP();
```

Listing 3: Example for incremental checkpointing with FTI.

6.2 FTI Integration in Melissa-DA

Melissa-DA is a derivate of Melissa-SA [9], a large scale sensitivity analysis framework that operates upon a server-runner structure. Melissa-DA adapts the concept of Melissa-SA as it implements a similar server-runner structure, however, to perform data assimilation for ensemble computations. For details concerning the Melissa-DA architecture we refer to section 2 of deliverable D5.1 of WP5. The characteristic of fault-tolerance in melissa-DA is two-fold. Firstly, the server-runner interoperability provides intrinsic fault-tolerance on the runner side. Secondly, the server is protected through checkpoint-recovery (CR). The latter is implemented within the EoCoE II project using the *Fault Tolerance Interface* [1, 2] (FTI)

Implementation

As explained before, Melissa-DA comprises a server instance and an API that operates as interface to the server instance. The runners, that advance the ensemble states, implement the API to communicate with the server. As the runners per se are fault tolerant, we implement FTI on the server side for checkpointing. Since Melissa-DA is programmed in C++, we have designed the implementations in form of classes with an abstracted interface to FTI. The interface is shown in listing 4.

```
// FTI class
class FTmodule;
// public functions of the class
void init( MpiManager & mpi, int & epoch_counter );
void protect_background( MpiManager & mpi, std::unique_ptr<Field> & field );
void store_subset( std::unique_ptr<Field> & field, int dataset_id, int runner_rank );
void initCP( int epoch );
void flushCP( void );
void finalizeCP( void );
void recover( void );
```

```
11 void finalize( void );
```

Listing 4: Interface to FTI in Melissa-DA.

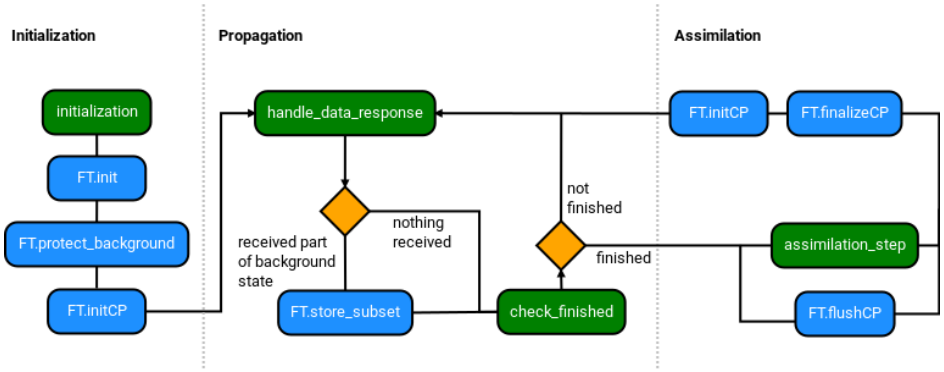

Figure 12: Flow of the fault tolerance module, embedded in the server instance.

Asynchronous Checkpointing with Threads

Figure 12 shows the UML diagram for the threaded checkpointing mechanism embedded into melissa. To enable threads, Melissa-DA needs to be configured with the flag `WITH_FTI_THREADS=1`. We implemented a simple task scheduler, that allows to submit functions to a task queue. The tasks are executed first-in-first-out by a team of threads. As we can see in the figure, FTI is getting initialized before the server loop. The loop is essentially divided into two segments. The reception of the background states, and the data assimilation step after all the background states have been received. As soon the server receives a part of a background state, the function `store_subset` submits a task to the scheduler. The task comprises the storage of the subset to the checkpoint. Hence, the subsets are written by threads, while the server continues the reception of the missing parts for the background states. When the background states have been completed, a task is submitted by `flushCP` and the server starts the data assimilation process. The task, submitted by `flushCP`, waits until all the subsets have been written to the checkpoint and afterward closes the checkpoint file. This, again, is happening asynchronously to the server. When the assimilation step is completed, the execution meets at a synchronization point, established in function `finalizeCP`. Afterward, a new epoch starts.

Mapping of the Data to the Checkpoint File

The communication between the server and the runners is $n - m$. That is to say, each runner comprises n ranks and the server comprises m ranks. The left graphic of Figure 13 shows an example for runners with $n = 3$ and a server with $m = 2$. The mapping of the background state on runner and server side, is a linear projection. Consequently, some ranks of the runners send to different server ranks. Since we write each time the server receives a part of a background state from a runner, we need to maintain the runner-server mapping when writing a part to the checkpoint file. That means, we create subsets with offset and count, corresponding to the parts that are received by the server. Thus, we create a global dataset for each ensemble state inside the checkpoint and add the subsets, corresponding to the parts of the background state. An example for a possible send and receive is shown in the right graphic of Figure 13.

Restart with Any Number of Processes

We load the data from the checkpoint files to the ensemble states in the same way as we have stored them. The information we need, offset and count from the subsets (i.e., parts of the

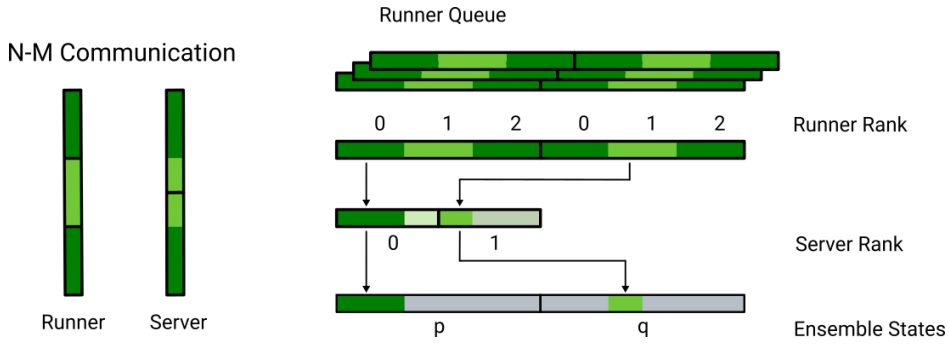

Figure 13: *left* Communication pattern between runner and server. *right* send and receive flow for background state.

background states), is already computed in malissa-DA at initialization. Thus, we merely need to expose the subsets to FTI passing this information and perform the recovery for each ensemble state. The data of each ensemble member is stored contiguously in the checkpoint, and upon recovery, we can load the subsets according to the current decomposition. This is independent of the number of ranks the server has been running at creation time of the checkpoint. Thus, the restart can be realized with a different number of processes.

6.3 FTI Integration in Alya

To implement Checkpoint Restart (CR) in Alya, we extended FTI to associate a string with the ID of a protected variable, and we integrated a checkpointing workflow into Alya. To enable CR support, we need to configure Alya with FTI support before compilation. We provided a configuration script that automatizes this process. Besides compiling Alya with FTI, the user needs to provide some extra information on Alya's application configuration to enable the CR runtime. The example from Listing 5 shows the additions to the configuration file for instructing Alya to use FTI (FTICR) as CR library, and perform a checkpoint every 2 iterations.

```
RUN_DATA
ALYA:        sphere
RUN_TYPE:    FTICR, FREQUENCY=2
END_RUN_DATA
```

Listing 5: Enable FTI in Alya's configuration file.

Implementation

Again we employ FTI's ICP mechanism. Apart from this, the CR implementation is straight forward. Upon invocation, Alya initializes FTI by calling `FTI_Init`. Inside the mainloop, Alya is divided into a time step and a post-action step. The post-action step comprises processing from modules that need to perform additional action at the end of each time step. The modules used, can be different depending on the problem at hand, thus, every module needs to implement calls to `FTI_Protect` and `FTI_AddVarICP`, in order to include data inside the checkpoint. Each time the time step has finished, Alya calls `FTI_InitICP` to open the incremental checkpointing phase (i.e., only if the iteration matches the checkpoint condition). Now every module will perform any post-action necessary, and add its protected data to the checkpoint file. At the end of this step, the checkpointing phase is closed by calling `FTI_FinalizeICP`. With this, it is easy to extend Alya with modules and include them into the CR mechanism. However, all currently available Alya modules have been equipped with CR measures and can be used. The recovery is similar to the checkpoint procedure. All modules are invoked at the restart, before entering the mainloop, to retrieve data

from the checkpoint file. When all modules finished recovering the variables, Alya starts execution from the last checkpointed time step.

Asynchronous Checkpointing

Additionally, Alya can be used in asynchronous CR mode. For this to work, Alya's global communicator has been replaced by the FTI communicator, which is returned by `FTI_Init`. FTI can be operated in this mode by enabling a setting in the configuration file. Enabled, the checkpoints are created in two steps. The first step comprises the storage of the Checkpoint files on the nodes by the application processes (i.e., local SSD, NVMe, RAMdisk, etc.). After this, The application continues execution, and in parallel, FTI dedicated processes convert the local files to the requested checkpoint level. Thus, levels of high reliability (partner, encoded, or PFS checkpoints) can be created in the background and will take, from the application point of view, essentially the same amount of time. Alya can be operated in synchronous and asynchronous mode without recompilation, just by enabling or disabling the respective setting inside the FTI configuration file.

6.4 IME Plugin for FTI

FTI has been extended by an interface to IME. The interface, directly calls functions from the IME native API. Hence, if FTI operates in IME IO mode, the checkpoint files are staged to the PFS through the IME file server. The implementation into FTI followed the implementation of the existing FTI POSIX interface. FTI assigns function pointers to the generic write, initialize and finalize functions. The implementation of new IO interfaces becomes realatively simple with this. We have simply provided the necessary IME functions that correspond to the respective functions from the POSIX interface, and added a new setting inside the configuration file. At FTI initialization, the IME functions get assigned to the generic checkpoint functions and the checkpoint is performed through the IME interface. For the time being, the recovery leverages the POSIX interface when operating in IME IO mode.

7 In-situ & in-transit data manipulation

Beside techniques to increase the I/O performance of an application, in-situ & in-transit data processing allows to couple post-processing or analyse elements as soon as data are available in the memory of the running simulation. This enables to reduce the data to be written to disk, to compute on-line high-level descriptors needed for the post-mortem analysis, saving scientist time, as well as enable live monitoring of the simulation run through scientific visualization tools.

The tasks focuses on two different aspects: In-situ visualization to monitor a running application and in-transit data compression to reduce the overall data size on-the-fly.

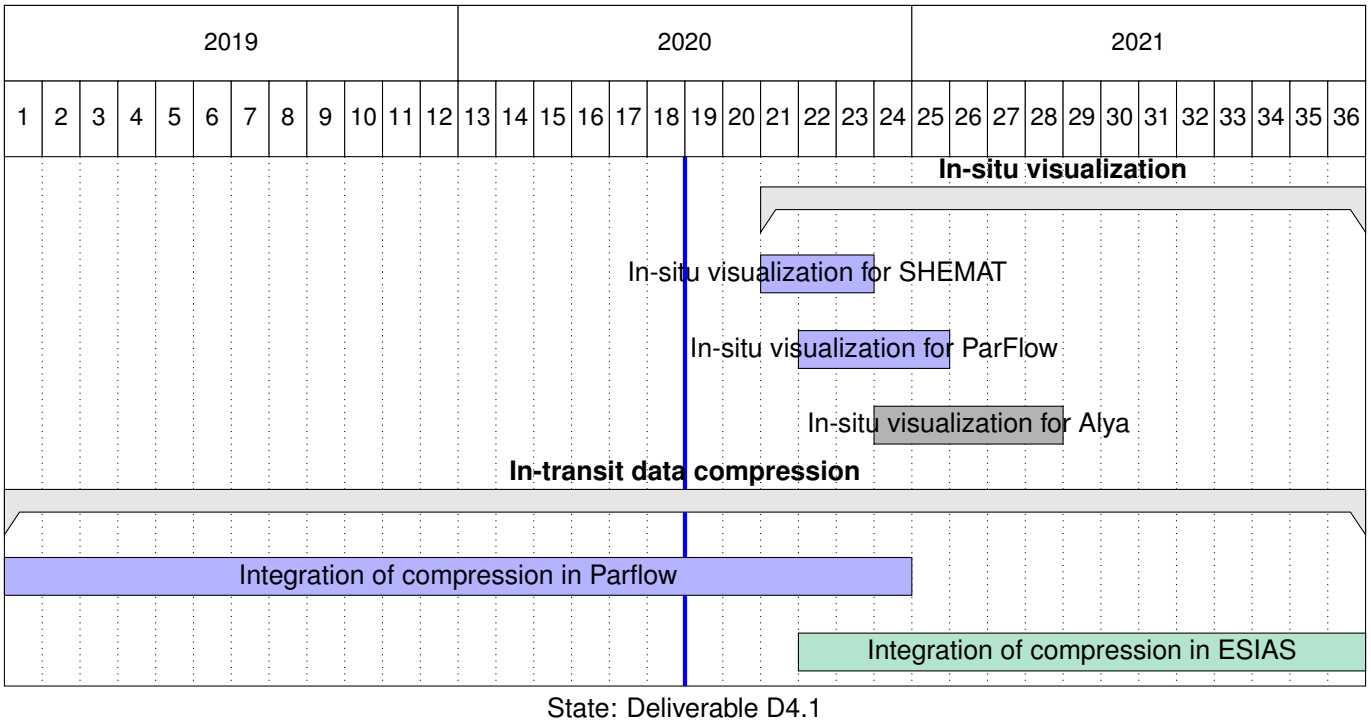

Figure 14: Gantt chart: Task 4 - In-situ & in-transit data manipulation

7.1 In-situ visualization

In-situ visualization is about reducing storage by running the visualization simultaneously with the simulation and watching the results live. This makes it possible to see the data live without the reductions that would be necessary to make due to the storage.

Normally several steps are necessary to use this visualization. For example, you would have to reserve a computing node and then start a vncserver. This is accessed via a vncviewer, which has to be installed on your local computer. Since this access happens over the internet, you also need a ssh-tunnel for the encryption. After that the visualization software has to be started on the computer node before you can start with the actual visualization.

To simplify this access to the computing resources at FZJ, there is an installation of JupyterLab[12]. This allows registered users easy access to the HPC systems. Only a web browser is required. In JupyterLab you have the possibility to use an interactive visualization. This uses pvlink[13] and ParaView for server-side rendering and streams this visualization encrypted to the user. Server-side rendering allows a more complex visualization to be calculated on the HPC systems without changing the required computing power of the end device. It is also possible to use the ParaView

[12] https://jupyter-jsc.fz-juelich.de/

[13] https://gitlab.version.fz-juelich.de/jupyter4jsc/j4j_extras/pvlink

in-situ interface Catalyst in JupyterLab. So it is possible to visualize a running simulation with Catalyst or SENSEI directly in the web browser[14]. In addition, a dashboard can be created, with which it is then possible to create and use a web GUI, which is then an avenue for clearly arranged live monitoring.

7.2 In-transit data compression

The idea of in-transit data compression is the reduction of the total I/O data size, which is moved towards the filesystem, during the runtime of the parallel application.

Typically, compression algorithms are used to lower the overall filesystem footprint of the scientific data due to data size restrictions and data manageability within a computing time project. However this type of data postprocessing does not lower the initial amount of data which is read or written to/from disk. In-transit compression compresses the data on-the-fly, which avoids having the total data footprint on the filesystem at any time and can also reduce the involved necessary network transfers towards/from the filesystem. On the other side the compression will increase the necessary computing time, so the approach has to be validated and configured to achieve a significant lower data size while keeping the necessary computing time limited.

For the initial in-transit compression work as part of the work within WP 4, we focus on the ParFlow application as part of the Water4Energy Scientific Challenge, as this application produces large datasets and support multiple output formats. For writing the data to disk ParFlow uses a distributed parallel writing approach, where each process writes a portion of data into a shared file using a specific file format. This type of parallel file access is quite difficult to handle for a regular file compression. Each process only sees its individual data, so the data must be already compressed within each process, without destroying the file format. Due to this reason, the in-transit compression approach of the HDF5 library was selected, which allows to use different compression algorithms while creating a parallel file by multiple processes. So far ParFlow does not directly support the HDF5 format, but it allows to write data by using the NetCDF4 format, which is also the preferred format by the scientific challenge community for their future ParFlow work. NetCDF4 is directly build on top of HDF5, which allows to use all HDF5 capabilities within NetCDF4 as well.

Implementation

During the first tests, NetCDF4 did not directly supported the parallel HDF5 data compression and filter techniques. This support was added by slightly patching the relevant NetCDF4-library files to circumvent the existing blocking behavior. In the meantime the support for parallel data compression was officially added in NetCDF4 release 4.7.4 on March 27, 2020[15]. This avoids the need for a patched version of NetCDF4.

HDF5 supports multiple compression algorithms by using a so called filter[16], which can be applied to the individual data sets. For the first approach, the basic GZIP compression (deflate) is used, as this compression is directly included as part of the basic HDF5 (and NetCDF4) installation, which avoids the need of having additional library dependencies installed. Because the relevant compression library must also be in place to open the created files later on, it was decided to keep the necessary dependencies to a minimum to avoid usability problems. The capabilities of other filter algorithms, such as ZFP, will be tested in a second update.

To support parallel compression the datasets must be chunked and collective calls must be used. Both were already in place within the existing NetCDF implementation in parflow.

[14] https://gitlab.version.fz-juelich.de/jupyter4jsc/j4j_notebooks/-/tree/master/002-Methods/003-Visualization/001-InSitu/001-Catalyst2JupyterLab-Tutorial

[15] https://github.com/Unidata/netcdf-c/blob/master/RELEASE_NOTES.md#474---march-27-2020

[16] https://support.hdfgroup.org/services/filters.html

Changes were only necessary on the NetCDF output routines within `write_parflow_netcdf.c` by enabling the build-in GZIP compression per variable by using `nc_def_var_deflate(ncid,varid,0,1,1);`.

Additional changes were added to make this option configurable.

Results and next steps

The compression was tested first on a scalable small Parflow benchmark test case, which allows to validate the setup and allows first scaling runs. Because this benchmark only handles a very simple geometry the result datasets were easily reducible by a factor of 1000.

To better judge on the performance impact and the compression quality a larger, real world example test case will be evaluated next.

In parallel the new compression options will be integrated back into the main repository of Parflow.

Finally in a last step, other compression algorithms beside GZIP will be tested as well.

References

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

[2] Leonardo Bautista-Gomez. Fti - fault tolerance interface. `https://github.com/leobago/fti`, 2018.

[3] C. Clauser. *Numerical simulation of reactive flow in hot aquifers: SHEMAT and processing SHEMAT*. Springer Science & Business Media, 2003.

[4] J. Keller, H.-J. Hendricks Franssen, and G. Marquart. Comparing seven variants of the ensemble kalman filter: How many synthetic experiments are needed? *Water Resources Research*, 54(1):1–20, 2018.

[5] W. Kurtz, G.-W. He, S.-J Kollet, R.-M. Maxwell, H. Vereecken, and H.-J.-H. Franssen. TerrSysMP-PDAF version 1.0: A modular high-performance data assimilation framework for an integrated land surface-subsurface model. *Water Resources Research*, 9(4):1341–1360, 2016.

[6] L. Nerger and W. Hiller. Software for ensemble-based data assimilation systems-implementation strategies and scalability. *Comput. Geosci.*, 55(1):110–118, 2013.

[7] Lars Nerger, Wolfgang Hiller, and Jens Schröter. The parallel data assimilation framework pdaf - a flexible software framework for ensemble data assimilation. pages 1885–, 04 2012.

[8] V. Rath, A. Wolf, and H. Bücker. Joint three-dimensional inversion of coupled groundwater flow and heat transfer based on automatic differentiation: Sensitivity calculation, verification, and synthetic examples. *Geophysical Journal International*, 167(1):453–466, 2006.

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