Centre of Excellence in Simulation of Weather and Climate in Europe

Phase 2

Architecture

Milestone MS4.1

This project has received funding from the European Union's

Horizon 2020 Research and Innovation Programme

under Grant Agreement No 823988

Lead Beneficiary: *UREAD, Julian Kunkel, Luciana Pedro, Bryan Lawrence*

Other contributing authors:

Deutsches Klimarechenzentrum GmbH (DKRZ), *Nathanael Hübbe*

Met Office (METO), *Glenn Greed, David Matthews*

Centro Euro-Mediterraneo sui Cambiamenti Climatici (CMCC), *Donatello Elia, Sandro Fiore*
United Kingdom Research and Innovation (UKRI), *Neil Massey*

Seagate Systems UK Limited (SEAGATE), *Hua Huang*

DataDirect Networks France (DDN), *Jean-Thomas Acquaviva*

Type: *see the original description in the DoA*

Dissemination level: *Public*

Delivery date Annex 1 (DoA) *Month 15*

Means of verifications: *Report*

Achieved: Yes

If not achieved, indicate forecast achievement date: -

Comments*:* -

Table of Contents

1. Introduction

The main target of this document is the ESiWACE2 consortium[1] team from the different Work Packages (WP), in particular, WP1 (Production runs at unprecedented resolution on pre-exascale supercomputers), for the end-to-end simulation workflows, and WP4 (Data systems for scale) and WP5 (Data post-processing, analytics, and visualization), for the data management systems. The document is mainly intended for internal use, although it is publicly released.

This milestone has the form of a report that provides design considerations and present design choices. It includes an architectural view of the data management components in ESiWACE2 WP4 (Task 1 to Task 5) and expands upon some designs in WP5. The purpose for the documentation of the design and interaction of the components is to provide a common ground in the ESiWACE consortium and to support the development of components that are utilized in different use cases.

Figure 1 provides the Pert chart related to all the data systems activities provided in the project. In particular, it shows the task dependencies among WP1, WP4, and WP5, which respectively relate to (i) climate and weather numerical high-resolution simulations on pre-exascale supercomputers, (ii) data systems at scale with respect to ensemble tools and storage middleware and, finally, (iii) data post-processing, analytics and visualization support for a selected set of applications.

Some of the developments made in the project are considered experimental as they aim at prototyping and demonstrating the benefit of such integrated approaches. In particular, the project plans preliminary versions (across WP4 and WP5) of the ESDM architectural document at M12 (WP5) and at M15 (WP4). It is worth noting that the design will also take into account data requirements coming from the end-to-end numerical simulations defined in WP1.

The rest of the document is structured as follows. In Section 2, an introduction to relevant software components is provided to support the context for further reading. In Section 3, an overview of the relation between the developed software and ESiWACE tasks is given. The design and modifications to software components relevant for WP4 are sketched in Section 4. In Section 5, we continue to describe the design and interactions with WP5 which is a continuation of the Milestone M5.1. The milestone is concluded in Section 6.

1

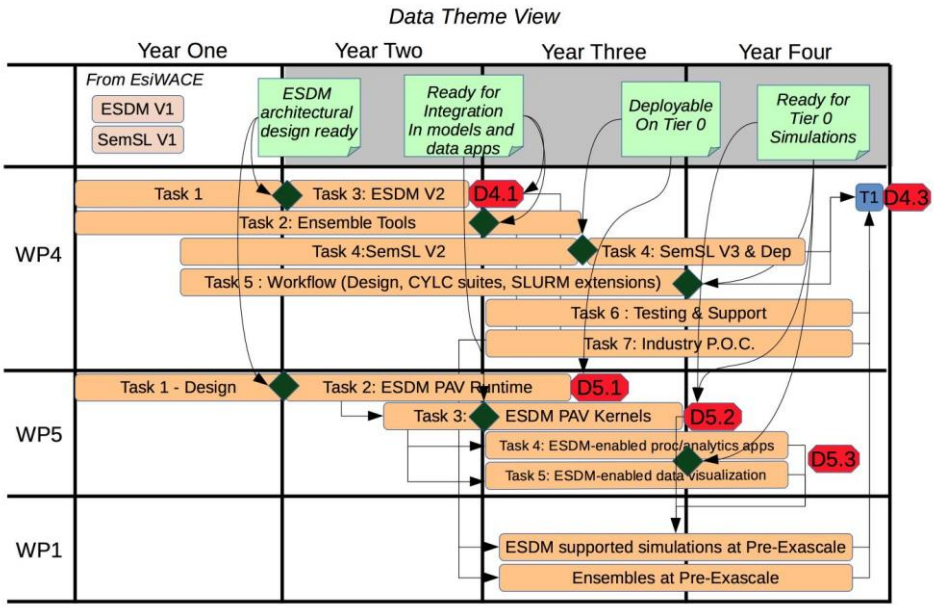

Figure 1. Data Theme View: the task dependencies among WP1, WP4, and WP5

2. Background

In this section, we briefly describe the software and hardware environment relevant for ESiWACE.

2.1 Workflows and Cylc

Cylc [Oliver et al, 2019] is in charge of executing and monitoring cyclic workflows in which each step is submitted to the batch scheduler of a data center. With Cylc, tasks from multiple cycles may be able to run concurrently without violating dependencies, preventing the issue of delays that cause one cycle to run into another. Cylc was written in Python and built around a new scheduling algorithm that can manage infinite workflows of cycling tasks without a sequential cycle loop. At any point during the workflow execution, only the dependence between the individual tasks matters, regardless of their particular cycle points. The information Cylc uses to control a given workflow is the task dependency. In a script file, the developers define, for each task, the parallelism settings and where the data is to be stored.

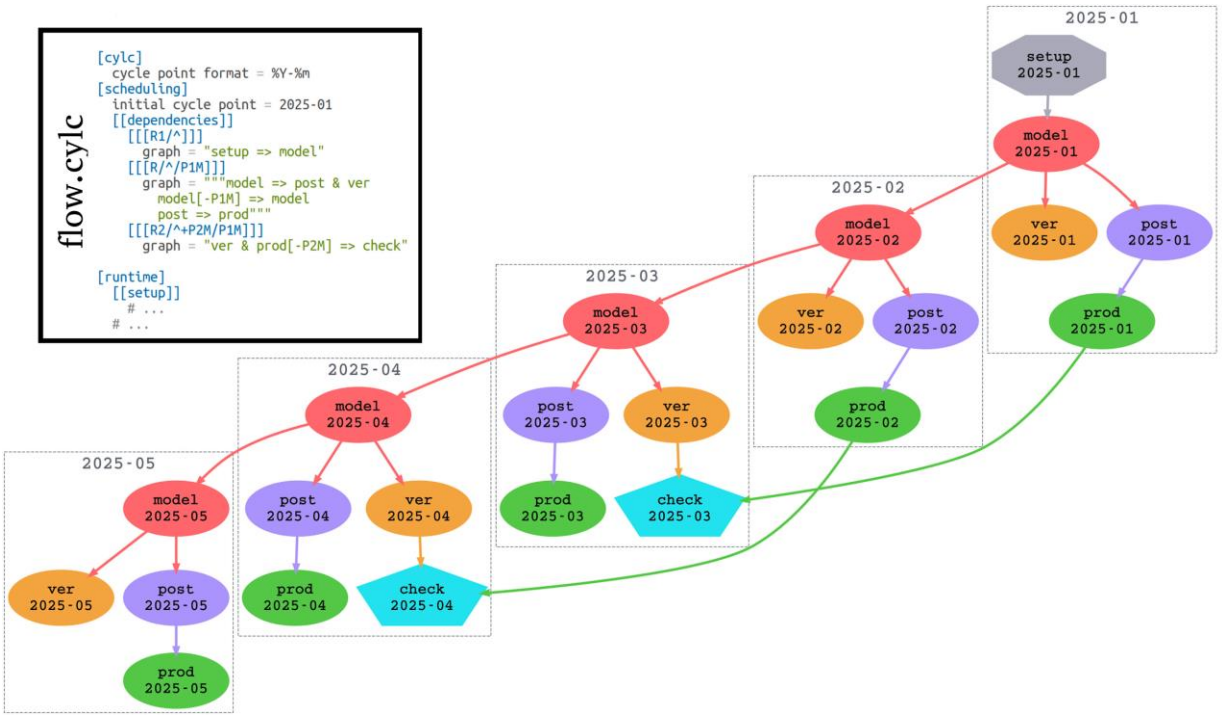

Figure 2: Example of a Cylc workflow with its configuration file

Consider the Cylc workflow for a toy monthly cycling workflow in Figure 2. In this workflow, an atmospheric model (labelled as **model** in the figure) simulates the physics from a current state to predict the future, for example, a month later. In climate research, this process is repeated in the model to simulate years into the future. Once the simulation of any month is computed, the data for this month becomes available and can now be analysed. In this workflow, the task **model** is followed by tasks **post** (post processing), **ver** (forecast verification), and **prod** (product generation), all specified as a workflow in a Cylc configuration file *flow.cylc*.

2.2 Workflow Execution

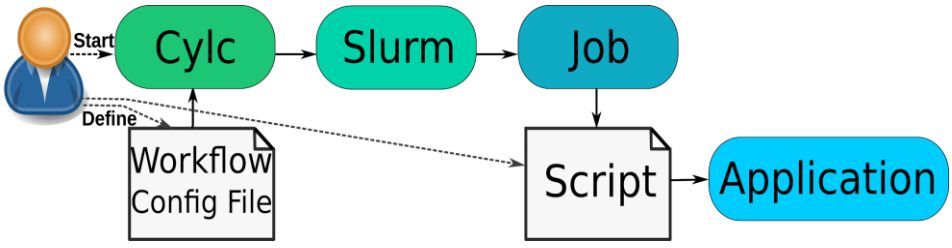

Figure 3: Software stack and stages of execution

While Cylc is directing the execution of workflows, several components are presented in the implementation. The software stack involved in a general workflow is depicted in Figure 3. In the following, each stage of the execution is further described.

- **Scientist** specifies the workflow and provides a command or a script for each task. As part of the Cylc configuration we have the command(s) to be run, any environment variables used by these application(s), and any workload manager directives. After that, a user enacts Cylc to start the workflow.

- **Cylc** parses the workflow configuration file, generates tasks dependencies, defines a schedule for the execution, and monitors the progress of the workflow. Once a task can be executed (dependencies are fulfilled), the workflow engine submits a *job script* for the workload manager with the required metadata that will run the Cylc task script.

- **Workload Manager** such as Slurm [Jette et al, 2002] is responsible for allocating compute resources to a batch job and performing the job scheduling. The selected tool queues the job that represents the Cylc task and plans its execution, considering the scheduling policy of the data center. Once the job is scheduled to be dispatched, i.e., resources are available and the job priority is the highest, it is started on the supercomputer.

- **Job** provides the environment with the resources and it runs the user-provided program or script on one of the nodes allocated for it. Local variables containing information about the environment of the batch job, e.g., the compute nodes allocated, enacts the Cylc provided program/script on one node.

- **Script** executes the commands with one or multiple (potentially parallel) applications to run sequentially. During the creation of the script, Cylc has included variables that describe the task in the workflow. The information is typically fed into the application(s) representing the task, and so defines the storage location. The user script may use commands to generate filenames considering the cycle and may store data in a workflow-specific shared directory.

- **Application** is executed taking the filenames set by the script.

2.3 I/O Stack of a Parallel Application

Climate applications may have complex I/O stacks, as can be seen in Figure 4. In this case, we assume the application uses XIOS [Meurdesoif et al, 2016], which provides domain-specific semantics to climate and weather. It may gather data from individual fields distributed across the machine (exploiting MPI for parallelism) and then uses NetCDF[2] to store data as a file. Under the hood, NetCDF4 uses the HDF5 API with its file format. Internally, HDF5 uses MPI and its data types to specify the nature of the data stored. Finally, data is stored on a parallel file system like Lustre which, on the server-side, stores data in a local file system on block devices such as SSDs and HDDs.

2

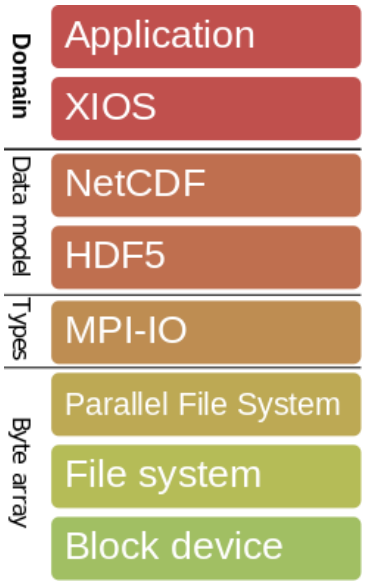

Figure 4: I/O path for an MPI-parallel application

Different applications involved in a workflow may use different I/O stacks to store their outputs. Naturally, the application that uses previously generated data as its inputs must use a compatible API to read the specific data format. In Figure 4, for example, XIOS may perform parallel I/O via the NetCDF4 API, allowing subsequent processes to read data directly using NetCDF4. Within the ESiWACE project, we are developing the Earth System Data Middleware (ESDM)[3] to allow applications with this kind of software stack to exploit heterogeneous storage resources in a data center. The goal of ESDM is to provide parallel I/O for parallel applications, with advanced features to optimize subsequent read accesses. Implemented with a standalone API, it also provides NetCDF integration allowing usage in existing applications. Hence, in Figure 4, the HDF5 layer can be replaced with ESDM.

2.4 Data Center Infrastructure

Data centers provide an infrastructure consisting of computing and storage devices with different characteristics, making them more efficient for specific tasks and satisfying the needs of different workflows. Take, for example, the supercomputer Mistral[4] at DKRZ, that consists of 3,321 nodes and offers two types of compute nodes equipped with different CPUs and GPU nodes. Each node has an SSD for local storage, and DKRZ has additionally two shared Lustre file systems with different performance characteristics. Individual users and projects are mapped to one file system explicitly, and users can access it with **work** or **scratch** semantics. While data is kept on the work file system indefinitely, available space is limited by a quota. The scratch file system allows storing more data, but data is automatically purged after some time. Future data centers are expected to have even more heterogeneity. A variety of

3

4

accelerators (GPU, TPU, FPGAs), active storage, in-memory, and in-network computing technologies will enhance storage and processing capabilities.

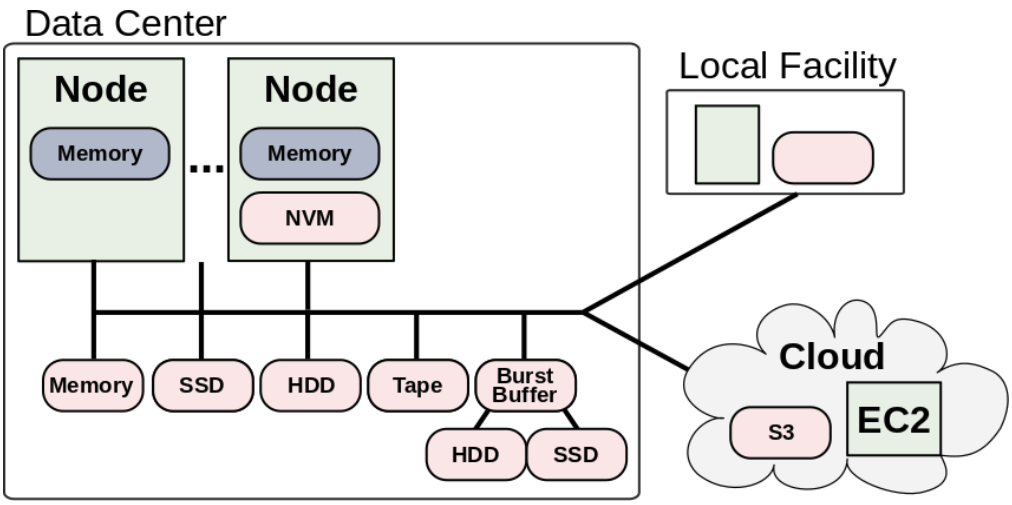

Figure 5: Example of an heterogeneous HPC landscape

Figure 5 shows such a system with a focus on computation and storage. Some of these technologies might be local to specific compute nodes or globally available. Depending on the need, the storage characteristics range from predictable low-latency (in-memory storage, NVMe) to online storage (SSD, HDD), and cheap storage for long-term archival (tape). The tasks within any given workflow could benefit from utilizing different combinations of storage and computing infrastructure.

2.5 Earth System Data Middleware (ESDM)

The Earth System Data Middleware (ESDM) aims to exploit heterogeneous storage landscapes by concurrently utilizing the available storage and allowing meaningful subsets of data to be placed on different storage systems as required. The ESDM targets concurrency from a single parallel application providing dynamic data granularity. The main ideas behind this software-centric co-design effort to address the I/O challenges are:

1. High-level semantics, i.e., awareness of application data structures and scientific metadata.
2. Flexible mapping of data to multiple storage backends.

The ESDM architecture is sketched in Figure 6. While ESDM provides its own interface, it deeply integrates into NetCDF and the Python module for NetCDF, allowing seamless integration into existing workflows. Internally, a NetCDF file is mapped to a container consisting of datasets and metadata. ESDM uses the site configuration to make layout decisions by splitting a dataset into fragments. A fragment is stored on exactly one storage backend, but as a dataset typically consists of many fragments, data of a dataset is

scattered among several backends locations. Further information about ESDM architecture and benefits are provided in Deliverable 4.3[5] of the first phase of ESiWACE.

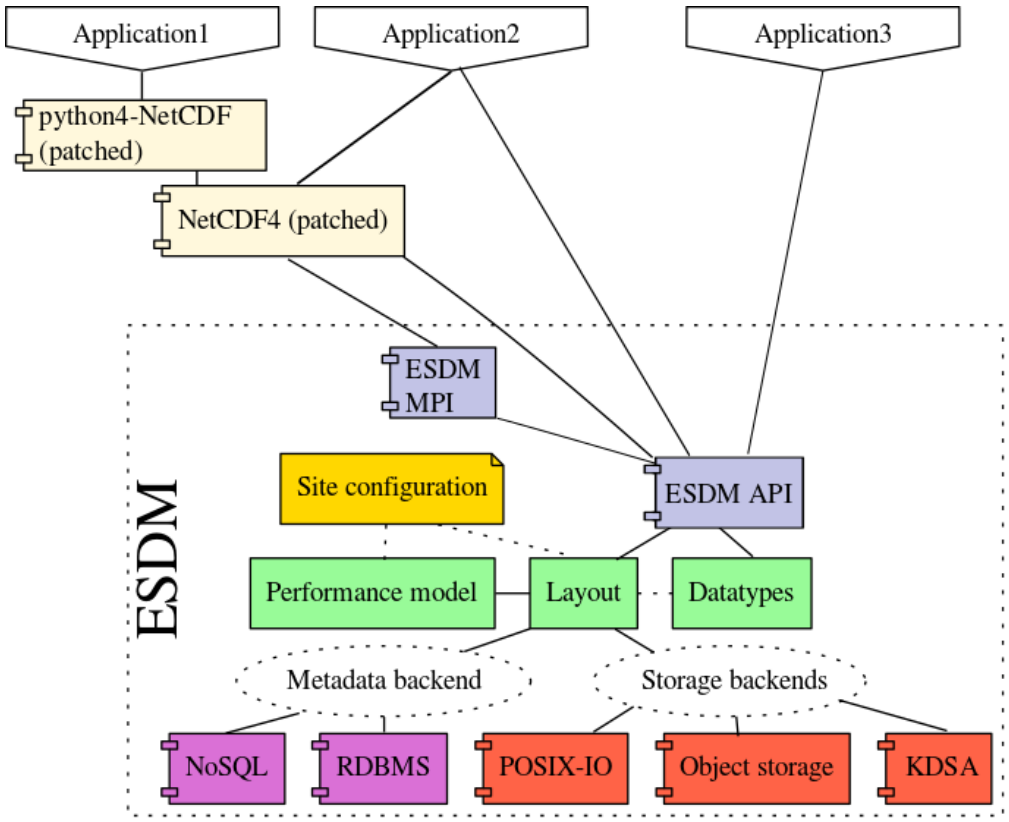

Figure 6: ESDM software architecture

2.6 SemSL

The Semantic Storage Layer (SemSL) is a further development of S3-NetCDF-python. S3-NetCDF-python is a Python library that can be installed directly into user space, via a Python virtual environment. It allows for the writing, reading and discovery of very large NetCDF datasets where the dataset has been divided into smaller fragments and distributed across different storage systems, such as S3 compatible object store or cloud storage, and POSIX disk. The design goals of both S3-NetCDF-python were:

- Be installable in user space, without having to install anything that is only available outside a Python virtual environment. This means that all package dependencies have to be installable via "pip".
- Be suitable for data analysis tasks. This means that it has to fit into existing user workflows, which is achieved by reimplementing the interfaces for file format libraries, such as NetCDF4-python.
- Be suitable for long-term archive of data. This is achieved by each subarray fragment of the dataset being a self describing object.

5

SemSL extends S3-NetCDF-python by having the architecture to support different file formats (e.g. HDF5 as well as NetCDF3 and NetCDF4) and to support different storage systems (e.g. FTP and tape, as well as S3 and POSIX).

2.7 JDMA

The JASMIN data migration application (JDMA) is a multi-tiered storage library which provides a single API to users to move data to a number of different storage systems, query the data that they have stored on those systems, and retrieve the data. These interactions are carried out using a common user interface, which is a command line tool to be used interactively, and a HTTP API to be used programmatically. The command line tool essentially provides a wrapper for calls to the HTTP API.

JDMA was designed with the following criteria in mind:

- The user experience for moving data, regardless of the underlying storage systems, should be identical.
- The user should not be responsible for maintaining the connection to the storage system in the case of asynchronous transfers.
- The user should receive notifications when the transfers are complete.
- Users should be able to transfer data from one storage system to another
- The JDMA system should be distributable, both across computing nodes and across cores on each node.
- Extra storage systems should be able to be added to the JDMA system without extensive re-engineering.

JDMA has been implemented and deployed on the JASMIN super data analysis platform at UKRI-STFC. Two storage backends have been implemented - a tape storage backend, and a backend for S3 compatible object storage. It is actively in use by a number of JASMIN user groups.

2.8 Scientific Compression Interface Library

The main purpose of compression methods is to shrink data size and to save storage space, but they also possess a huge potential to reduce the gap between computational power and I/O performance, because often after compression less data has to be moved. For these reasons, many modern file formats, in particular HDF5 and NetCDF4, provide native support for compression. This is especially beneficial in climate science, where data amounts are huge and are growing constantly. Unfortunately, the compression algorithms used in HDF5 and NetCDF4 are lossless and do not meet the requirements of climate science to full extent.

Within the Advanced Computation and I/O Methods for Earth-System Simulations (AIMES) Project, the Scientific Compression Interface Library (SCIL) was designed and implemented. The main purpose of SCIL is compression/decompression of scientific data, especially, of climate modeling data. It uses different third party compression libraries as well as specifically developed lossy and lossless compression methods. The advantage of the library is that it decouples the specification of required precision and the utilized algorithm; hence, users do not need to be familiar with all the compression algorithms and its

characteristics. Based on user specified quantities, the library selects automatically the best method. For further information, check the AIMES D2.2 Report[6].

In this work, the primary target was NetCDF4 C-API and the intention is to integrate the clever compression functionality of SCIL into NetCDF4 (Task 4.3). This can be done directly in NetCDF4, but due to the dependency, the functionality can also be added indirectly into the HDF5 library. Even more, the architecture of HDF5 allows a loose integration of SCIL, i.e., it can be developed as an independent library. The advantages of this approach are good testability without HDF5 and reuse of the library in other projects. Later, the interface can also be ported to any programming language with C-bindings support, or used by other libraries.

In order to make the compression path work, NetCDF and HDF5 require additional functionality. Firstly, a quantity passing mechanism is required, to pass quantities from application to the SCIL library. Secondly, an HDF5 filter is required for communication with SCIL. The prototypes of the quantity passing mechanism and the HDF5 filter are already implemented (Figure 7). Using them, the application can send the data together with quantities to NetCDF4, NetCDF4 can pass them to HDF5, and HDF5 in turn can pass them to SCIL, yielding compressed data as result. After that, the data can be saved in a file. The decompression step works similar, but in reverse direction.

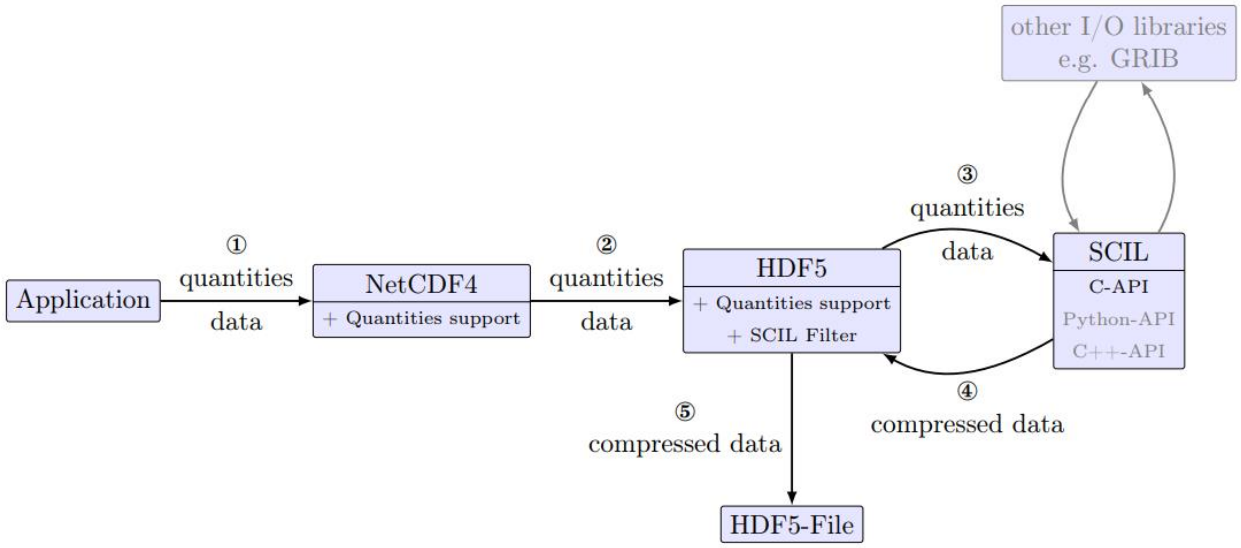

Figure 7: Compression path: gray-out elements are out of scope of the project

In climate science, there is often nothing against reducing the precision of data by lossy compression algorithms, when the impact on the simulation results is negligible. After that, the data can still be compressed with a lossless compression algorithm. Application of both kinds of compression methods can result in a higher compression ratio, compared to compression ratio of only one algorithm. SCIL facilitates this task by supporting a number of lossless and lossy compression algorithms and choosing automatically the best one based on predefined user hints.

The user hints are a set of metrics, that describe required precision, maximum error, consider noise and take some other metrics into account. We already have a functional prototype of the SCIL library. Its compression chain allows the construction of different compression processes for different compression

6

scenarios on the fly. It works well with user hints, which can be set at runtime. The extensible architecture of HDF5 easily allows the insertion of third party filters and plugins. For that purpose, it provides a number of well documented interfaces.

3. Architecture Overview

In this section, we describe a big picture of the proposed architecture and how the tasks and software stack developed in WP4 contribute to ESiWACE use cases of enabling weather and climate simulations on the upcoming (pre-)Exascale supercomputers. It is important to understand how the developed software stack can be used in climate and weather use cases. We consider the following scenarios relevant to ESiWACE:

- **Large-scale simulation**: Running a single large-scale application at the highest resolution, e.g., with an 1-km resolution for demonstration. This requires optimal efficiency and minimal overhead due to I/O. For demonstrators, it is expected that initially only a few well-defined data products will be generated for diagnostics and presentation purposes. Any intermediate data and other data products will remain in the specific center where the large scale run is executed.

- **Local production workflow**: This scenario covers production runs conducted at one location where a group of scientists run a simulation and well-known workflows that generate data products of interest. Some analysis could be done in an ad-hoc fashion, where scientists test a new hypothesis, but most of the scientific workflow is known a priori.

- **Downstream processing**: This scenario covers the long-term processing of data products generated by (large-scale) simulations. Such products may be archived in one site and replicated at others, but the analysis is fully decoupled from the originating simulation, e.g., scientists explore and analyze the data anytime later than the experiment. Typically, scientists perform ad hoc specific analyses to test a hypothesis or to understand the simulation behaviour. Many analyses are done only once (a very typical pattern in climate research). Historically, the performance of such workflows has not been an issue, and analysis is mostly made by tools that support node-local parallelism. Future workflows at scale are likely to have performance issues, but bespoke optimization will still be unlikely for one-off analysis jobs.

Generally, WP4 fits into the data management aspect of these scenarios, allowing scientists to localize and access data in storage hierarchies. WP5 blends into these scenarios by providing means to optimize steps involving data processing and analysis. Specifically, the technology developed in WP4 and WP5 embraces these scenarios as follows:

- **Large-scale simulation**: The simulation parallel codes use ESDM (Task 4.2) allowing to create outputs as efficiently as possible. They may use XIOS which, in turn, uses the ESDM NetCDF library to distribute the data across available storage systems. The limited post-processing may be performed by a supported tool, e.g., CDO [Schulzweida, 2019] or Ophidia [Fiore et al, 2014]. The advances of the PAV API (WP5) can be used to offload a set of computational tasks to storage or to perform an in-situ data analysis via ParaView [Ayachit, 2015].

- **Local production**: Here, the weather or climate workflow may use Cylc to orchestrate the production (Task 4.5). A single task could involve the execution of an ensemble (Task 4.3) that is then processed by XIOS to store only the interesting data subset. In the diverse set of workflows, CDO, Ophidia or Python may be used to analyze the data (WP5) and SemSL (Task 4.4) may be used to access datasets managed by ESDM. Hence, in these cases, the workflows will benefit partially from the developed solutions.

- **Downstream processing**: Semantic analysis tools are used to manipulate the location of data in (potentially complex) storage hierarchy via JASMIN Data Migration Application (JDMA) and analysis may take advantage of SemSL (Task 4.4) to provide an aggregated view of data and hide the location of file granules across the storage systems.

The big picture of these interactions between these developed components is seen in Figure 8. Note that some of the developments made in the project are considered experimental. They aim at prototyping and demonstrating the benefit of such integrated approaches and will not yet be considered ready for general production.

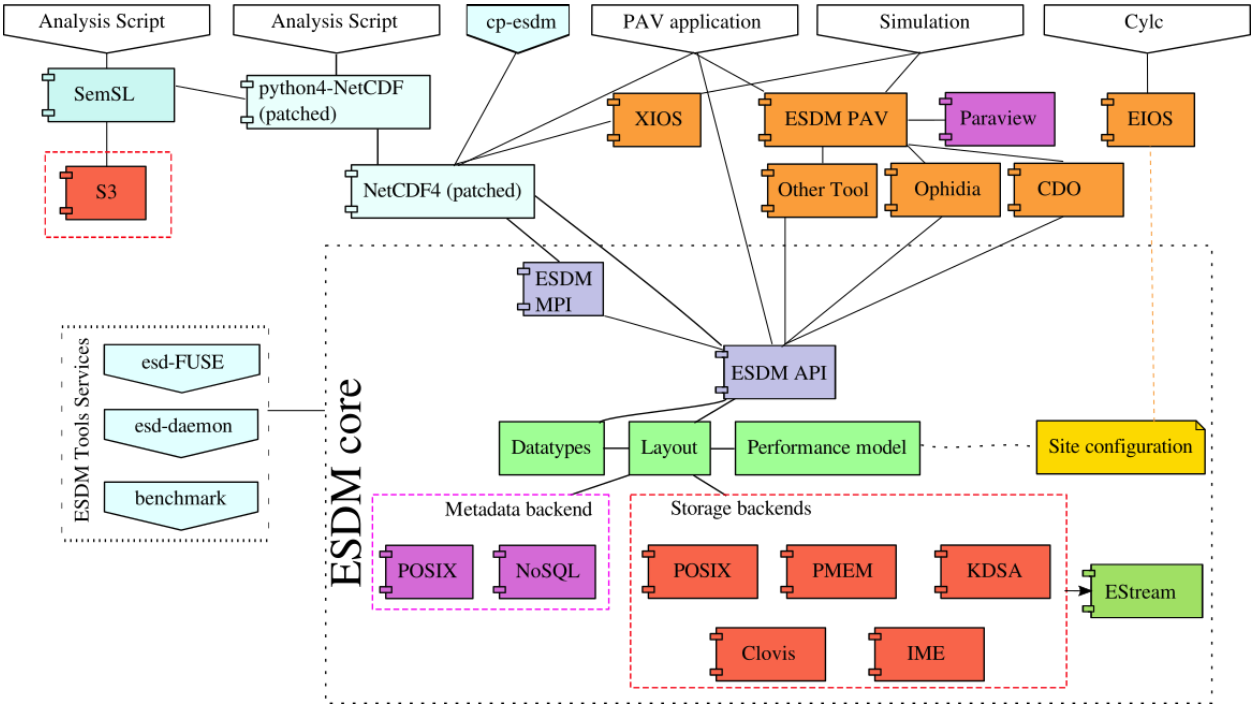

Figure 8: Interaction of the ESiWACE software components related to WP4

To understand the interaction from the perspective of an individual application as shown in Figure 8, additional potential paths and new components are listed in the following:

- An **Analysis Script** written in Python could use SemSL to create an aggregated view on data stored in a set of files or to partition data and storing it on S3.

- Another **Analysis Script** may also use the patched NetCDF4 Python package to access data that is 7stored via ESDM.

- The **cp-esdm** program is a slightly modified **nccopy** tool that supports data export by partitioning the data into multiple NetCDF files allowing to utilize SemSL. Generally, the nccopy tool already allows to import/export data between the ESDM data format and any other NetCDF format. This enables scientists to convert between the data-center specific representation of ESDM and a portable format which is important for Internet exchange or archival.

- A post-processing tool such as **CDO** or **Ophidia** may use the **ESDM PAV API** to reuse some of the functionalities of ESDM PAV which includes the usage of an existing command line tools to realize additional functionality. ESDM PAV will support the native addressing of ESDM allowing to invoke the PAV functionality on top of ESDM objects.

- There are several ESDM Tools Services depicted on the left:

    - The **esd-fuse** tool allows mounting the ESDM namespace into a regular directory using FUSE. It enables semantic-based access to the data.

    - The **esd-daemon** is in charge for checking the consistency of containers and datasets, to perform the garbage collection, and to remove files if their prescribed end of life is reached (according to the specified life cycle information).

    - The **benchmark tool** allows to quantify the performance of individual storage components and the overall ESDM configuration. This supports a data center to check and determine suitable configurations.

- **EIOS** is a new high-level workflow scheduler that interacts with Cylc and enable I/O decisions based on a Cylc workflow. It will reuse the configuration file of ESDM enabling it to understand the characteristics of available storage. It also provides command line tools.

- The **EStream** API provides routines to the ESDM backends to perform various data processing tasks. These will include checksumming, compression (Task 4.3), and user-defined processing. The backends may or may not use EStream. However, they are responsible to perform the tasks that would otherwise be done by EStream, possibly by offloading them to storage servers.

A goal of ESiWACE is to exploit data locality in workflows, including smarter data placement, offloading of analysis to the storage systems, and APIs that generate abstractions from use cases. The interaction of workflows (Task 4.5) and the ESiWACE suggested software stack is shown in Figure 9 (Compare it with the current software stack introduced in Figure 3.)

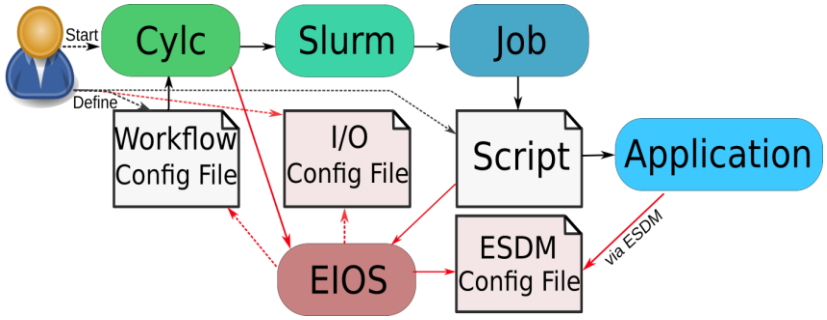

Figure 9: Software stack and stages of execution with the I/O-aware scheduler (EIOS)

Here, EIOS is the ESDM I/O scheduler that interacts with ESDM, Cylc, and the user scripts to communicate scheduling decisions. A task in such a workflow could be a job that performs post-processing via one of the ESiWACE tools and, hence, take advantage of the advances made in WP5. A task could also be the execution of an ensemble of a parallel simulation and the analysis then could cover the statistical differences of the ensemble members as output by XIOS (Task 4.3) or ESDM (maybe via NetCDF) (Task 4.2). The parallel application may also use the PAV API to register post-processing jobs and perform in-situ visualization directly (WP5).

Details about the individual components and changes are discussed in later sections of this milestone.

4. Design

In the following sections, we discuss the architectural changes that are performed as part of ESiWACE2.

4.1 Ensemble Services

A simple template for delivery of ensemble services assumes that a model will use XIOS as its I/O system. It is convenient if the model retains the ability to write data through an alternative I/O system to manage checkpoint files for example, or other highly specialized output.

The basic ensemble workflow comprises several phases:

1. **Diagnostic specification** Decide output fields, spatial and temporal domains, and ensemble processing;
2. **XML file generation** Convert model-specific output generated in (i) to XML for use by XIOS;
3. **XIOS configuration** For performance and consideration of downstream data processing requirements;
4. **Ensemble set-up** This is model specific, depending on details of how physical schemes are initialized and parameterized, but typically through specification of input data files (for an initial-conditions ensemble) or by setting namelist values (for a perturbed parameter ensemble);
5. **HPC** Construct the HPC environment and working spaces;
6. **Ensemble execution** Interact with the HPC job submission system to launch the ensemble and monitor its progress.

The Ros/Cylc workflow configuration and management system is the basis upon which the ensemble scheme is being developed but since the degree of interaction with the workflow engine is not significant, we envisage little difficulty in using alternative workflow tools.

A UM-XIOS ensemble based on the workflow above is running in test mode on the UK National HPC (ARCHER). Based on our experience to date, it will be desirable to develop tools to determine both XIOS client and XIOS server memory requirements, which can be substantial, as a task to aid in the XIOS configuration step.

4.2 ESDM

In this section, we describe the overarching design aspects of ESDM which is a central component of WP4. Since ESDM is a work in progress, some parts of the design described here are supposed to change in the near future. This section highlights these changes by the use of different text colors to avoid confusion about which parts describe the current status and which pertain to the future. Additionally, the coloring also documents in which direction we are planning to update the design of ESDM. Statements about the current state of ESDM which are supposed to change in the future are marked in red, while statements that are not true yet, but should be implemented in the future are marked in green.

ESDM is comprised of several distinct pieces:

- A library that interfaces with user code, allowing it to create/manage/delete fragments, containers, datasets, and dataspaces.
- Metadata backends that are used to store information about containers and datasets.
- Data backends that are used to store the actual user data in the form of fixed size fragments.

Both kinds of backends are provided as libraries which are currently linked in by the build system, and which should eventually be dynamically loaded as instructed by the ESDM configuration file (*esdm.conf*).

The interface to the two kinds of backends is quite similar. When the library is initialized by ESDM, its *init* function returns an esdm_backend_t* which contains a pointer to a *vtable* listing the various backend's entry points (functions) for the respective behavior.

All backends, both data and metadata, provide functions to

- create and check consistency of an ESDM file system
- provide a performance estimate
- finalize the backend

In addition to this, the metadata backends provide functions to create, commit, retrieve, update, destroy, and remove containers and datasets. Likewise, the data backends provide functions to

- create, retrieve, update, and delete fragments

- create, load, and free fragment metadata

ESDM is designed to execute the code of several different backends concurrently as threads as appropriate, so backend code must be written to be multithreading safe. The dispatch of actions to the different backends happens within the ESDM scheduler. The scheduler is an internal logical component and it is not visible to either user code nor backend code.

4.2.1 Configuration

ESDM provides a mechanism for configuration in JSON format that is designed to be easily expandable to new/improved backends. Usually, it is read from the file "_esdm.conf" when esdm_init() is called. This is supposed to be changed to a sensible sequence of search paths. Another option is to provide the JSON text as a string to esdm_load_config_str() before calling esdm_init(). ESDM can also be reset with a different configuration by first finalizing the library with esdm_finalize(), possibly setting the configuration with esdm_load_config_str(), and finally calling esdm_init() again.

Internally, the configuration always involves a call to esdm_config_init_from_str(). If esdm_load_config_str() is used, that will immediately call through to esdm_config_init_from_str() and save the result in the global esdm_instance_t object. esdm_init() then sees that the configuration has already been set, and will proceed immediately. In case that no call to esdm_load_config_str() happened before esdm_init() is invoked, esdm_init() will see that no configuration has been set, load the contents of "_esdm.conf" into memory, and kick off configuration with a call to esdm_config_init_from_str().

The function esdm_config_init_from_str() does not really interpret the configuration string. It only uses the jansson library to parse the configuration string and sets config->json to the resulting json handle. It is then the job of the different *_init() functions of modules to interpret the contents of the config->json container.

The function esdm_modules_init() calls through to esdm_config_get_backends() to get the list of configured backends, and to esdm_config_get_metadata_coordinator() to read further options.

esdm_layout_init(), esdm_performance_init(), and esdm_scheduler_init() do not currently interpret any configuration data themselves.

The process outlined above has several issues:

- The configuration is effectively stored in a global variable, making it intransparent the moments in which specific parts of the configuration are interpreted by distinct parts of the code.
- The sequence of functions that handle configuration data is confusing and has surprising names. For example, esdm_load_config_str() does not "load a configuration string" in any way, it only calls the JSON parser to setup the parsing.
- There is not one single function call that fully parses and interprets the given configuration and returns an object with all that configuration data in usable form.

- The strings that are fetched from the configuration are not stored in the memory managed by ESDM, but rather remain backed by the jansson handle. As such, the jansson handle needs to be kept around until esdm_finalize() is called.

Consequently, the configuration should be redesigned to the following process. There should be exactly one high level function esdm_init_from_str() that takes a configuration string, parses the JSON code, and calls through to esdm_init_from_json(). This later function should then proceed to initialize ESDM, including all configured modules and calling their respective init functions with the appropriate jansson handle. Once esdm_init_from_json() returns, the jansson handle is disposed of. The function esdm_init() should be implemented to search for the configuration file, read its contents, and then call through to esdm_init_from_str().

During these operations, the configuration string and the jansson handle should be passed around explicitly to have a clear flow of information.

4.2.2 Ownership of Objects

Most objects in ESDM are owned by either a single object or exactly one function that creates them and disposes them of again. If any other part of the code needs to get hold of such an object, it is obliged to create a copy for itself.

Nevertheless, there are also a few objects for which such simple mechanics are insufficient. These are containers and dataspaces. The details are provided in the following.

4.2.2.1 Datasets

The reference count of a dataset does not control whether the memory object itself is alive, but rather whether the datasets' data is loaded into memory. The initialization sets the reference count to zero and a dataset can only exist with its enclosing container. The reference count is incremented by the container when the dataset is created and registered by a call to esdm_dataset_create()[7]. This increment should happen via esdm_dataset_ref().

Function esdm_dataset_ref()[8] checks whether the dataset is loaded into memory. If that is not the case, the dataset's metadata is loaded into memory before incrementing the reference count. These functions are called when a dataset is opened from a container.

Surprisingly, esdm_dataset_ref() is also called from esdm_dataset_delete() to load its metadata into memory. This is necessary to subsequently walk the fragment list and delete all the fragments referenced by the dataset.

7

8

The reference count is decremented in esdm_dataset_close(), triggering the unloading of the dataset's metadata.

Containers check that the reference counts of all their datasets have reached zero before they commit suicide. If one of these reference counts is found to be non-zero, the closing is aborted with an error code.

Containers also check their datasets reference counts when they are destroyed, destroying the datasets only if the container is the sole owner of the respective dataset.

The reference count should be renamed to dataRefCount to signal that it does not manage the object lifetime. esdm_dataset_open() and esdm_dataset_close() should increment/decrement the strongRefCount, (un)loading the dataset's metadata as appropriate.

Datasets should also reference their enclosing container while their dataRefCount is nonzero. This will ensure that ESDM does not try to destroy a container unless all of the contained datasets have been closed. This removes the requirement to first close all contained datasets before closing the container from the ESDM user. An open dataset is an indirect reference to the enclosing container.

4.2.2.2 Containers

The reference count is never incremented. It starts out at 1 and the first close/delete destructs the container. The reference count of the container is decremented when the container is closed (esdm_container_close()). This also happens when the container is deleted (esdm_container_delete()). The function esdm_container_delete() has the additional requirement that none of the contained datasets may be referenced by other code when esdm_container_delete() is called. The function esdm_container_close() is to be removed, users are expected to use esdm_container_ref() and esdm_container_unref() appropriately and the cleanup is performed when the last reference is cleared with esdm_container_unref(). The currently existing function esdmI_container_destroy() should be removed, forcing all lifetime management to go through esdm_container_[un]ref(). The reference count is incremented whenever a strong reference is created to a contained dataset, and decremented whenever a contained dataset's strong reference count drops to zero.

4.2.2.3 Dataspaces

Dataspaces are not reference counted, even though their pointers are stored within ESDM currently. This information ensures that dataspace objects are always copied when they are stored within a dataset and/or fragment. The dataset/fragment is then responsible to dispose of its copy of the dataspace. Likewise, esdm_dataset_get_dataspace() should return a copy of the internal dataspace object, ensuring strict value semantics for them.

4.2.3 Data Layout

ESDM is designed to be able to handle any possible regular layout of the multidimensional data it is provided with. This is done by the means of the esdm_dataspace_t objects. These contain four important fields:

- The count of dimensions
- The logical, multidimensional coordinate of the first data point in the fragment/dataset
- The logical, multidimensional size of the fragment/dataset
- The stride that is used to convert logical coordinates into memory offsets

The later field may be NULL, indicating that standard C array index order is to be used. When a stride is provided, it gives a single multiplicator for each coordinate. I.e. to calculate the offset of a 3D coordinate coord, the memory offset would be calculated as coord[0]*stride[0] + coord[1]*stride[1] + coord[2]*stride[2].

The stride field allows for great flexibility: It allows for handling of other indexing orders (like buffers passed in from FORTRAN code), for manipulation of non-consecutive data, for replication of data (simply by specifying a stride of zero for one or more dimensions), or even for logically reversing an array in any amount of dimensions. While ESDM currently has no use for reversing an array, it does make significant use of the flexibilities concerning indexing order, non-consecutive data and data replication for its internal operations.

On the API level, every dataspace is created with the default data layout, i.e. no stride set, contiguous, C order indexing is assumed. A user may then choose to either set a non-default stride explicitly using esdm_dataspace_set_stride(), or to copy stride information from another existing dataset with esdm_dataspace_copyDatalayout(). The later function basically assumes that the destination dataspace is supposed to be used to index into the same memory buffer as the source dataspace, which is also the main use of this function.

Additionally, ESDM provides a simple to use, efficient function for moving data from one dataspace to another, possibly transposing/reversing/replicating the data in the process. This function is esdm_dataspace_copy_data(), which takes two dataspaces and associated memory buffers, and copies all common logical data points from the source buffer to the destination buffer. While this function is exported to users of ESDM, it is also used internally to stitch together data from several fragments, or to split data in memory into several fragments.

4.2.4 Write Path

From a user perspective, data is written by first creating/opening a container, then creating/opening a dataset in/from the container, and finally calling esdm_write() on that dataset.

Internally, the bulk of the organization of the writing is performed within the ESDM scheduler. Currently, this is only implemented in esdm_scheduler_write_blocking(). This function uses esdm_scheduler_enqueue_write() to create the appropriate background tasks, and then waits for their completion. To get information about errors within the background tasks, an io_request_status_t object is created and passed along to the background processes. A nonblocking variant of the function is to be added to the scheduler.

Function esdm_scheduler_enqueue_write() does not just create the background tasks, it may also split the work across different backends and into several fragments. This is done via functions splitToBackends() and esdm_scheduler_makeSplitRecommendation() that add fragments to the dataset, accordingly. Finally, an io_work_t object is created for each task and backend_thread() is either called directly or its call is dispatched to a backend thread. The backend_thread() function calls the respective backend's fragment_commit() to persist the data.

Frequently, the backend will observe that the data that is requested to be written is not stored in memory consecutively. In this case, the backend may decide to first linearize the data before writing it to storage. How this is implemented is strictly down to the backend code, the scheduler does not know or care.

However, ESDM should provide the backends with a routine in the EStream interface to serialize the data for storage. This routine should take a dataspace that describes the source memory layout and return an array of bytes for the backend to store. Depending on configuration and user requests, this routine should also

- perform coroutine callbacks
- perform compression (see our section below for details about compression)
- add a checksum to the data for integrity checking

The backend is free to split the data of the fragment further before passing it to the EStream routine, as long as it will use the same chunking when invoking the reading counterpart of the EStream routine.

Both, the coroutine callbacks and the compression functions should accept an arbitrary dataspace, allowing them to work on non-contiguously stored input data. In this way, we can avoid an additional data copy step to sequentialize the input data when compression is used. If the backend wishes to control the data layout itself, it is still obliged to call an ESDM routine to perform the coroutine callbacks.

4.2.5 Read Path

From a user perspective, reading is done by opening a container, then opening a contained dataset, and finally calling esdm_read() on that dataset.

Internally, the proceedings are similar to the write path. The onus of the organization is on the ESDM scheduler, and the high level call is esdm_scheduler_read_blocking(), because nonblocking reading is to be implemented.

However, in the case of reading, much more work is performed within esdm_scheduler_read_blocking():

- First of all, a set of fragments that contain requested data is computed in esdmI_fragment_makeSetCoveringRegion().

- Then, it is determined whether these fragments actually cover the entire region and, if not, whether a fill value is set for the dataset. If that is the case, the uncovered region is initialized with the fill value; if not, an error is returned.

- Function esdm_scheduler_read_blocking() then proceeds to creating background tasks to read all the fragments in the fragment list. The background tasks unconditionally read the data into contiguous storage, before copying the useful part over to possibly noncontiguous, user supplied memory. With partial reading implemented in the backends, this memory copy will be avoidable.

  ESDM is built on the assumption that overlapping fragments do not contain contradicting data. As such, no effort is made to guarantee reading a datapoint from a specific fragment, nor to avoid reading the same datapoint from several fragments at once. This decision does not have any impact on performance as ESDM favors selecting non-overlapping fragments anyways (see 4.2.6 Fragment Selection below).

- Once the reading is done, esdm_scheduler_read_blocking() may decide to create another fragment containing the data it just read from disk. This happens whenever much more data needs to be read to satisfy the read request than data is delivered back to the user. This serves the use case of a reading program using a different domain decomposition than the writing program did. By performing the write back, a second execution of the reading program, or another program with a similar domain decomposition, will execute much faster than the first read.

Again, the backend_thread() call relays the request to the respective backend. However, in the reading case, the backend_thread() will also execute a callback function that may transpose the data layout to the in-memory layout requested by the user. The backends' interface should be changed to allow for partial reading, controlled by an arbitrary dataspace that is passed along to the backend. This operation should be supported by ESDM with a routine in the EStream interface that

- checks a checksum if present
- performs (partial) decompression if data was compressed
- performs any requested coroutine callbacks

If the backend has decided on the write side to pass the data through the EStream interface as chunks, it must pass the same chunks into the read side of EStream.

Even if the backend decided to take full control of the on-disk data layout, it is still obliged to call an ESDM routine to perform the coroutine callbacks.

4.2.5.1 Compression

We intend to use the Scientific Compression Library (SCIL) library developed in the AIMES project. SCIL provides an interface that allows users to set the expected data properties in terms of data quality and performance. SCIL is a meta compressor that decouples the algorithmic selection from the compression algorithms. In ESiWACE, we 1) integrate SCIL into ESDM to allow scientists to specify the data properties

on datasets, and 2) extend the description of a Cylc workflow to allow scientists to specify the required properties for a dataset in the I/O workflow specification. This information is then communicated to ESDM.

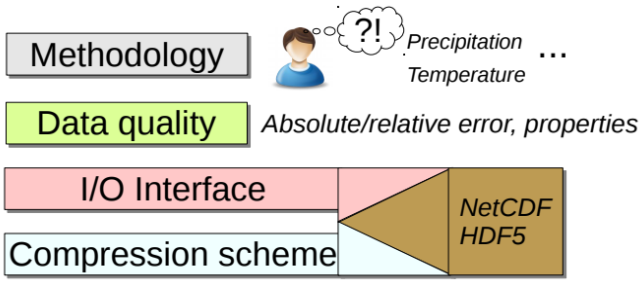

Figure 10: Integrating ESDM (I/O Interface), NetCDF, and SCIL

The following quantities determine compression of each individual point and the example in Figure 11. illustrates how data can be changed, when quantities are applied. For further information about the definition of these quantities, check AIMES D2.2 Report[9].

- Absolute error tolerance
- Relative error tolerance
- Relative error finest absolute tolerance
- Precision bits and precision digits
- Preserved values

1.0000	2.0000	3.0000
5.0000	4.0000	1.0000
7.0000	3.0000	5.0000

(a) Array before compression.

1.0018	2.0036	3.0054
5.0071	4.0065	1.0018
7.0092	3.0054	5.0071

(b) Decompressed array.

$$\eta \leq 0.002$$
$$\varepsilon \leq 0.01$$
$$MSE = 0.00003388$$
$$RMSE = 0.005820739$$
$$\sigma = 1.667988889$$
$$PSNR \approx 60$$

(c) Quantities.

Figure 11: Lossy compression of a 3x3 array and resulting error quantities

The compression option extends the ESDM API when creating a dataset. The user can set the required precision using SCIL quantities. This information will be stored as metadata in the dataset and allows querying the (intended) precision. This will be implemented in the EStream component to actually invoke the SCIL API if compression is needed for a dataset (and, hence, the fragments).

9

A benefit of integrating SCIL into EStream is that backends can utilize a shared facility and that compression of smaller blocks reuses data in the CPU caches increasing the performance for the compression.

4.2.5.2 Stream Processing

The read path with stream processing is very similar to the normal read path. The deviations are as follows:

- The backend is invoked via a separate API which passes the map() and reduce() callbacks, and which returns only a single, yet arbitrarily sized object.
- The backend may decide to perform the map() and reduce() itself, possibly offloading them to a server, or it may simply call an ESDM API that takes a fragment and the callbacks, and performs the processing for the backend.

This way, storage backends can easily be implemented to support stream processing, but can also take advantage of not having to move the full data across the network.

4.2.6 Fragment Selection

ESDM stores data in the form of fragments, each of which fully covers a multidimensional region (a hypercube shape). When reading, a suitable set of these fragments has to be selected. Finding the set of fragments that requires the least amount of data to be read from disk is a weighted set cover optimization problem. As such, finding this optimal solution is NP hard, so ESDM has to use heuristic approaches to find good solutions to this problem.

Solutions that have been explored:

- Finding a small number of minimal solutions (= solutions where each fragment is the sole provider of at least one data point), and then selecting the best of these solutions. Such minimal solutions can quickly be generated by repeatedly dropping a random fragment from the set and checking whether the resulting set still covers the requested region. This algorithm yields good results, but it has a complexity of O(N^2), making it inappropriate for datasets with many fragments.
- Typical use cases tend to create fragments that are right next to each other, but do not overlap. For instance, a 3D dataset might be fragmented into complete 2D slices. In these cases, it is easy to find sets of non-overlapping fragments fully covering the requested region by first selecting one intersecting fragment, and then recursively walking the list of neighbouring fragments. Of course, this requires neighbourhood information to be built and stored along with the fragments. Building this neighbourhood information is an O(N log N) operation that can be split across the N write operations that create the fragments, so a O(log N) neighbourhood management operation is added to each write operation.
- A third variant is to use an internal regular domain decomposition that does not need to coincide with the domain decomposition of the user. This approach resembles what traditional file format libraries do under the hood. The advantage of using a defined regular domain decomposition is

that it makes selection of fragments for reading trivial. However, it adds complexity to the write path. Writes frequently turn into read-modify-write cycles with such a scheme. Also, accesses may read/write much more data than necessary if the internal domain decomposition does not match well with the user's domain decomposition. It may also yield suboptimal performance in the read path if the read request matches the original user's write requests. In this case, deviating from the user's domain decomposition adds unnecessary costs to both writing and reading.

Since with ESDM we strive to avoid many of the fundamental problems that are inherent to the last method, we are going to keep its implementation out of the master branch of the ESDM code repository. We are implementing it for research reasons only, to perform benchmarks which will help us to assess the performance benefits of ESDM. It is not expected that the regular domain decomposition will yield enough benefits to justify its inclusion in a release version.

4.2.7 Optimizations of Backends

Different backends may have different storage characteristics, e.g., bandwidth for large sequential I/O, IOPS for small random I/O, data availability and integrity. ESDM should direct fragments better to different backends according to the storage requirements and the backend storage characteristics to achieve better overall performance.

Let's take the Clovis backend as an example to explain what optimizations we will do.

- Store fragments of the same dataset in the same Clovis object. This will avoid the overhead of creation of a new object for every fragment. Creation of a new object is a medata operation and it requires the Clovis to talk to several Clovis servers. With this optimization, the corresponding Clovis object is created when the dataset is first created, and is used for data I/O afterwards.

- Cache of metadata. Just like inode/dir entry is cached in DRAM in kernel VFS implementation, Clovis object handle is opened and its metadata is cached when it is accessed at the first time. Later access can use this cache directly. Dirty cache will be flushed later, or at the finalization phase.

- Asynchronous operations. Clovis has asynchronously APIs. ESDM scheduler will use the asynchronously APIs to queue multiple operations to Clovis backend. The optimal queue depth will be evaluated.

4.3 Semantic Storage Layer

The Semantic Storage Layer (SemSL) is a further development of S3-NetCDF-python. SemSL is a Python 3 library that can be installed directly into user space, via a Python 3 virtual environment, that allows for the writing, reading and discovery of very large datasets where the dataset has been divided into smaller fragments and distributed across different storage systems, such as S3 compatible object store or cloud storage, and POSIX disk. The design goals of both S3-NetCDF-python and SemSL are:

- Be installable in user space, without having to install anything that is only available outside a Python virtual environment. This means that all package dependencies have to be installable via "pip".

- Be suitable for data analysis tasks. This means that it has to fit into existing user workflows, which is achieved by reimplementing the interfaces for file format libraries, such as NetCDF4-python.

- Be suitable for long-term archive of data. This is achieved by each subarray fragment of the dataset being a self describing object. For NetCDF this means that each subarray file is itself a NetCDF file containing the same metadata (NetCDF attributes) as the master array file and containing the coordinate information for the subdomain that the subarray file represents. The master array file contains all the subarray information, including the location. However, if this master array file is lost, it can be recreated from the subarray files, as long as their locations are known.

S3-NetCDF-python provided a drop-in interface replacement for the NetCDF4-python library supplied by Unidata, to allow reading and writing NetCDF data from and to storage with an S3 API with minimal code changes for the user. SemSL extends this by having a pluggable architecture consisting of the main components:

1. A data model for storing the relationship between the master array file and the subarray fragments.

2. A (number of) storage "backends" that have a common set of operations which can read, write and discover data on storage systems in a unified way.

3. A (number of) file format "frontends" that are drop in replacements for file format libraries, such as NetCDF4-python.

4.3.1 SemSL Data Model

The SemSL data model is based upon the Climate and Forecast Aggregation conventions outlined by [Hassel, 2014]. Information about each subarray (fragment) object is contained within a master array object. The UML for this data model, as implemented in SemSL, is shown in Figure 12.

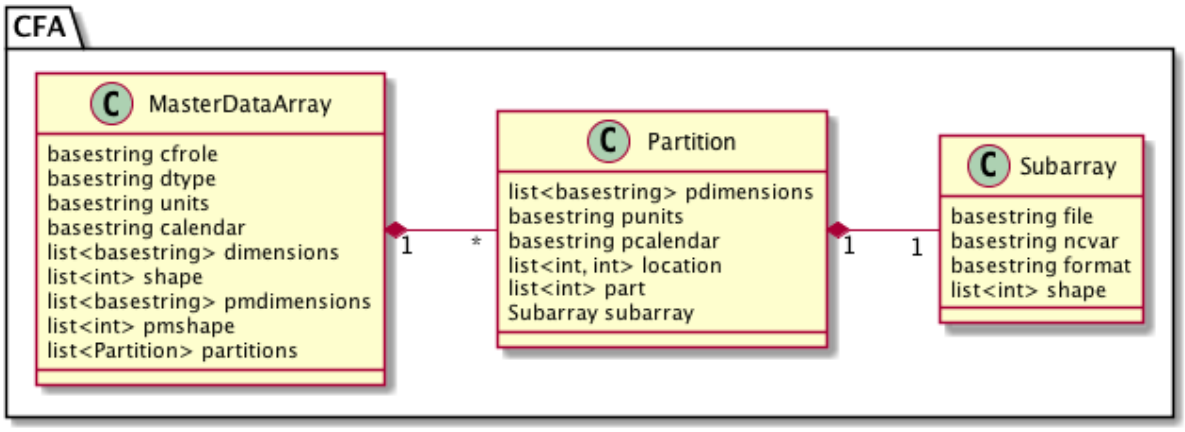

Figure 12 - Data model for the CFA specification of master arrays and subarrays in SemSL

S3-NetCDF-python used version 0.4 of the CFA conventions. This convention encodes the data model in a JSON string that was added to the attributes of a NetCDF variable. However, in some tests of S3-NetCDF

it became apparent that this method could not be used for very large arrays with many subarray files, due to the length of the JSON string causing a memory error in the underlying NetCDF library.

SemSL overcomes this problem by flattening the Subarray and Partition classes (as they have a 1 to 1 relationship) and storing the information in a number of variables that are contained within a group in the Master Array NetCDF file. This group is associated with its variable by a NetCDF attribute of the variable. In addition to overcoming the memory error this has a number of advantages:

1. The partition information (cfa_tmp group) is stored internally (in memory) as a NetCDF dataset, with a group, variables and dimensions.
2. There is no interpretation of the subarray metadata necessary on load – the internal dataset is opened from the dataset on disk or S3.
3. The required metadata is only read when a slice of the master array is taken.
4. The metadata is only written when its slice in the master array is written.
5. Trying to read a slice that has not been written will result in missing values being returned.

4.3.2 SemSL Storage Backends

The "storage backends" in SemSL are implemented as Python file objects, with methods such as read, write, seek and tell. This maintains compatibility with POSIX file systems and allows other storage backend interfaces to be implemented in a manner that is separate from SemSL and therefore reusable. There are currently three backends implemented:

1. POSIX file system (actually the Python io.BufferedIOBase class)
2. Serial / synchronous S3 using Amazon botocore
3. Asyncio version of S3 using aiobotocore

4.3.3 SemSL File Format Frontends

The file format frontends mirror the file format libraries for data formats, such as NetCDF, HDF 5, etc. This allows users to maintain their workflow with none to minimum changes to their code. There is currently just the NetCDF3 and NetCDF4 frontend implemented. In future more file formats could be added, such as GRIB or PP, which are structurally similar to NetCDF.

4.3.4 SemSL Other New Developments

SemSL is approaching a release candidate, and other new features have been added in the past year:

1. Pluggable backends
    a. written as python file objects with seek, tell, read, write, etc.
    b. controlled on a per server basis, via an user config file
2. Pluggable parsers for CFA format

a. Currently just the NetCDF3 / NetCDF4 parser

b. Could allow other file formats (GRIB, ESA SAFE, PP, etc.)

3. Completely cache-less

a. Read/write to disk performed on disk

b. Read/write to S3 performed in memory (memory swapping is currently being worked on)

4.4 Workflow Enhancements

This section describes our first approach to incrementally extend workflows for climate and weather that realizes parts of the project. While individual components such as ESDM and Cylc exist, we have not implemented the described scheduler, yet. To automatically make scheduling decisions, the software stack needs to:

1. Deliver information about dataset life cycle together with the workflow, and

2. Adapt the resulting workflow, individual scripts and application executions to consider the potential for data placement.

4.4.1 System Information

ESDM is used as I/O middleware in the parallel application (with NetCDF or directly) and orchestrates the I/O according to a simplified ESDM configuration file (*esdm.conf*). This file contains information about the available technology in the data center, its I/O characteristics, and will be used to make decisions about how to prioritize I/O targets. In the example presented in Figure 13, we have three storage targets: two global accessible file systems (**lustre01** and **lustre02**), and one local file system (**/tmp**) that can be accessed via the POSIX backend. Each of them comes with a lightweight performance model and the maximum size of data fragments. The metadata section (Line 24) utilizes here a POSIX interface to store the information about the ESDM objects. Internally, ESDM creates so-called containers and dataset objects to manage data fragments.

```
"backends": [
    {"type": "POSIX", "id": "work1", "target": "/work/lustre01/projectX/",
        "performance-model" : {"latency" : 0.00001, "throughput" : 500000.0},
        "max-threads-per-node" : 8,
        "max-fragment-size" : 104857600,
        "max-global-threads" : 200,
        "accessibility" : "global"
    },
    {"type": "POSIX", "id": "work2", "target": "/work/lustre02/projectX/",
        "performance-model" : {"latency" : 0.00001, "throughput" : 200000.0},
        "max-threads-per-node" : 8,
        "max-fragment-size" : 104857600,
        "max-global-threads" : 200,
        "accessibility" : "global"
    },
    {"type": "POSIX", "id": "tmp", "target": "/tmp/esdm/",
        "performance-model" : {"latency" : 0.00001, "throughput" : 200.0},
        "max-threads-per-node" : 0,
        "max-fragment-size" : 10485760,
        "max-global-threads" : 0,
        "accessibility" : "local"
    }
],
"metadata": {"type": "POSIX",
        "id": "md",
        "target": "./metadata",
        "accessibility" : "global"
    }
```

Figure 13: Example of an ESDM configuration file (*esdm.conf*)

ESDM manages a pool of threads that should be created per compute node to achieve good performance and delegates the assignment of optimal block sizes to the storage backend. The number of threads is defined in the configuration file. As an example, based on the number of Object Storage Targets (OSTs) provided by both Lustre systems at DKRZ, performance tests already developed in Deliverable 4.3 using ESDM show that no more than 200 threads in total should be used to perform I/O to extract the best performance. To clarify this behaviour, ESDM distributes a single dataset across multiple storage devices depending on their characteristics. Since ESDM also supports several non-POSIX storage backends, an application can utilize all available storage systems without any modifications to the code.

The configuration file is inquired by an application that utilizes ESDM and steers the distribution of data during I/O. While the current system information and performance model are based on latency and throughput only, it shows that automatic decision making can be made on behalf of the user.

4.4.2 Extended Workflow Description

The user now has to provide information about the datasets required for input and the generated output for each workflow task in a separate file similarly to Cylc's workflow configuration file. An example of an I/O-workflow configuration file (*io.cylc*) is shown in Figure 14. In this file, information about Task 1 is given as an example, and we expect the extra information about all tasks in the same file. Ultimately, this could be integrated into the file *flow.cylc*.

```
1  [Task 1]
2    [[inputs]]
3      topography = "/pool/input/app/config/topography.dat"
4      checkpoint = "[Task 1].checkpoint$(CYCLE - 1)"
5      init = "/pool/input/app/config/init.dat"
6
7    [[outputs]]
8      [[[varA]]] # This is the name of the variable
9        pattern = 1 day
10       lifetime = 5 years
11       type = product
12       datatype = float
13       size = 100 GB
14       precision.absolute_tolerance = 0.1
15
16     [[[checkpoint]]]
17       pattern = $(CYCLE)
18       lifetime = 7 days
19       type = checkpoint
20       datatype = float
21       dimension = (100,100,100,50)
22
23     [[[log]]]
24       type = logfile
25       datatype = text
26       size = small
```

Figure 14: External Cylc I/O-workflow configuration file (*io.cylc*)

In this example, the *io.cylc* file could define a cycle flexibly to be a month or a year according to the *flow.cylc* file. The notation is similar to the specification of Cylc workflows which uses a nested INI format. For each task, inputs and outputs are defined. In the input section, each entry specifies the virtual name that is used by ESDM as a filename inside NetCDF. Line 3, for example, defines that the filename **topography** is mapped to a specific input file and this dataset does not depend on any previous step of the workflow. The next line specifies that the input filename **checkpoint** should be mapped to the checkpoint dataset from the previous cycle (e.g., the checkpoint generated after completed the last year's output). For the initial cycle, the checkpoint file will be empty, and the application will load the **init** data. In the output section, the datasets are annotated with their characteristics more precisely. For each variable, a pattern defining how frequently the data is output according to the workflow must be

provided. Most data is input and output in the periodicity of the cycle, except for **varA**, which is output per day regardless of the cycle. Next, we formally define the expected annotations in all the fields expected in the I/O-workflow configuration file:

- **Name** A basic name for the field/data generated. It is extended by a pattern defined in a variable (Lines: 8, 16, 23).
- **Pattern** The frequency the data is output (Lines: 9, 17).
- **Lifetime** How long the data must be retained on storage (if at all) (Lines: 10, 18).
- **Type** The class type of data, i.e., checkpoint, diagnostics, temporary (Lines: 11, 19, 24).
- **Datatype** The data type of the data (Lines: 12, 20, 25).
- **Size** An estimate of the data size (Lines: 13, 26). This field can be inferred if dimension and datatype are provided.
- **Dimension** The data dimension (Line: 21).
- **Accuracy** Characteristics quantifying the required level of data precision (Line: 14).

Note that the user may not be able to provide all required information. This can be handled by assuming a default safe behaviour. For instance, in the case of missing data precision, data should be retained in its original form. Knowing the dimension or size a priori might be difficult for scientists, e.g., the log file size is unclear. In this case, the user may insert relevant information like small or big, reflecting that any information is better than no information at all. In future, we will explore how to automatically infer the output volume from the input or by using monitoring. By allowing to run using an empty I/O workflow specification and monitoring I/O accesses for one cycle, we can propose an I/O description to the user to simplify the specification.

4.4.3 Smarter I/O Scheduling

As data movement involves a large overhead, both in terms of latency and energy-efficient computing, it requires to optimize data locality, where locality is twofold, spatial and temporal. The spatial locality has to be understood not only between the tiers of hierarchical storage architecture but between data silos such as cloud and the on-premise data lake. In the specific case of ESiWACE, temporal locality depends on the workflow where a component generates a dataset to be consumed later in the workflow. Therefore, data should be pinned to the faster tier to optimize future access. The second locality exploitation is related to the read-many profile where a dataset brought to the fast tier by a component of the workflow should remain local if the same dataset will be read downstream.

To exploit temporal locality, the capacity of a fast tier must be sufficient to handle the data of the related workflow tasks. A space reservation mechanism has to arbitrate the need for immediate usage of the running component and for future usage. Private storage (node-local) implies that the consumer component is scheduled on the same node that the produced component.

The second aspect of data locality is related to spatial placement. A trade-off exists between the cost of bringing the data in and the amount of computation to be applied to the data. Depending on the arithmetic intensity (ratio between the Byte/Flops), offloading the computation to the data hosts makes sense, even if the data hosts have limited computing resources.

General considerations regarding the temporal access pattern observed in the workflow and the potential for optimization is shown in the table:

Access Pattern	Note	Scheduler Implication
Write, read	Typical communication pattern between producer and (multiple) consumers.	Local storage is possible, could communicate data directly between producer and consumer
Read many	Input data	Data should be cached/migrated to a fast storage tier
Read once	Input data	May be cached depending on spatial access pattern
Write through	Data will not be re-read during the workflow but is typically analyzed later (manually)	Data should be drained quickly from the compute nodes and write-through a burst buffer
Write once, read optional	Data will be overwritten, e.g., a checkpoint/restart use case where an application crashes	Data should be drained quickly from the compute nodes, write-through a burst buffer, and keep cached if capacity is sufficient

We aim to realize data placement and migration decisions in a heterogeneous (multi-storage) environment. These goals will be achieved via the proposed I/O-aware scheduler, called here EIOS (ESDM I/O Scheduler). EIOS will make the schedule considering the Cylc workflow and ESDM provided system characteristics. Components of it are involved in different steps of the workflow and the I/O path.

While Cylc schedules the workflow, EIOS can provide hints about prioritizing and co-locating tasks which provide the opportunity for keeping data in local storage. Decisions about data locality will not be made for a whole (and potentially big) workflow. Instead, the system will make decisions by looking ahead to several steps of the workflow, allowing reacting to the observed dynamics of the execution. We utilize

DDN's IME API[10] [Betke et al, 2019] to pin data in Infinite Memory Engine (IME) and to trigger migrations between the IME and a storage backend explicitly.

Ultimately, when a user-script runs, the information about the intended I/O schedule is communicated from EIOS through a modified filename, which is then used by the ESDM-aware application to determine the data placement.

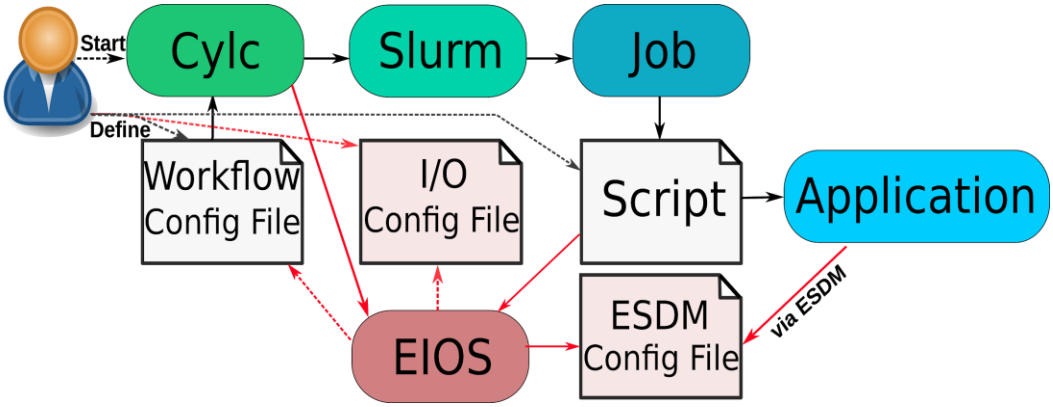

Figure 15: Software stack and stages of execution with the I/O-aware scheduler (EIOS)

4.4.4 Modified Workflow Execution

The steps to execute a workflow enriched with I/O information and perform smarter scheduling is depicted in Figure 9 and repeated here in Figure 15. The suggested alterations can be seen in boxes pointed by red arrows, and the remaining components are the current state-of-the-art for workflows in climate and weather from Figure 3. In the following, we describe the modifications we propose in this project for each component involved in the software stack.

- **Scientist** The user now has to provide an additional file that covers the I/O information for each task and slight changes have to be made to the current scripts.
- **Cylc** EIOS is invoked by Cylc to identify potential optimizations in the schedule before generating the Slurm script.
- **EIOS** The ESDM I/O Scheduler reads the information about the workflow *flow.cylc* and *io.cylc* configuration files) and acts depending on the stage of the execution. EIOS consists of several subcomponents:
    - The high-level scheduler that interfaces with Cylc.
    - A tool to generate pseudo filenames used by the ESDM-aware applications.
    - A data management service (not shown in the figure) that migrate and purge data at the end of the life cycle.

10

EIOS components use knowledge about the system by parsing the *esdm.conf* file. EIOS may decide that subsequent jobs shall be placed on the same node, reorder the execution of some jobs, and provide information for conducting data migration.

- **Slurm** Cylc may now have added an optimization identified by EIOS which is now handled by a modified Slurm. Also, if migrations have to be performed, Slurm will administer them according to the job script specification.
- **Job** A job might run on the same node as a previous job to utilize local storage.
- **Script** Filenames are now generated by a replacement command that calls EIOS to create a pseudo filename. This filename will encode additional information for ESDM about how to prioritize data placement according to data access.
- **Application** The application may either use XIOS, NetCDF with ESDM support or ESDM directly to access datasets. Hence, in Figure 4, the HDF5 layer is replaced with ESDM. ESDM loads the *esdm.conf* file that contains the information about the available storage backends and their characteristics. ESDM extracts the long-term schedule information from the generated pseudo filenames and employs it during the I/O scheduling to optimize the storage considering data locality between tasks. Basically, ESDM can now change the priorities for data placement on different storage locations that would normally be encoded in its configuration file.

4.5 NetCDF

4.5.1 Introduction

As presented in Figure 6, ESDM deeply integrates into NetCDF and the Python module for NetCDF. Internally, a NetCDF file is mapped to a container consisting of datasets and metadata. This section covers aspects of the compatibility between ESDM and NetCDF. For detailed information, please check the final Compatibility Report[11] which compares the NetCDF functionalities with the current version of ESDM and presents details about the tests performed in C, Python and some benchmarks.

Data in NetCDF format is:

- **Self-Describing** A NetCDF file includes information about the data it contains.
- **Portable** A NetCDF file can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
- **Scalable** Small subsets of large datasets in various formats may be accessed efficiently through NetCDF interfaces, even from remote servers.

11

- **Appendable** Data may be appended to a properly structured NetCDF file without copying the dataset or redefining its structure.
- **Sharable** One writer and multiple readers may simultaneously access the same NetCDF file.
- **Archivable** Access to all earlier forms of NetCDF data will be supported by current and future versions of the software.

ESDM breaks with the concepts of Portability, Archivability, and shareability (to some extent) as the aim is to provide a site-specific optimal mapping.

4.5.2 Data Model

This chapter describes how ESDM objects are related to NetCDF objects. For each NetCDF file, ESDM creates a container, and for each NetCDF variable, ESDM creates a dataset. To be able to interact with the NetCDF C code, we introduce the file `esdm_dispatch.c`. This file contains the functions that will be used by ESDM that will be called by the dispatch table when NetCDF runs. For each nc function, we have an ESDM function in the backend of NetCDF with similar parameters, providing the same outcome, but using this new middleware. Some of the ESDM functions use the flag ESDM PARALLEL to check if the code is being parallelized and call the respective ESDM parallel functions if that is the case.

4.5.2.1. ESDM Open

ESDM is first called when nc_open() is called, which triggers the respective function ESDM open and loads the data from the NetCDF file to memory. Then, the following functions are called to create a representation of the NetCDF file according to the ESDM entities.

- **esdm_container_open** Open the container that contains the NetCDF file.
- **esdm_container_dataset_count** Check the number of datasets in the container.
- **esdm_container_dataset_from_array** For each dataset, load it into memory.
- **esdm_dataset_ref** For each dataset, check whether the dataset was already loaded into memory. If that is not the case, the dataset's metadata is loaded into memory before incrementing the reference count.
- **esdm_dataset_get_dataspace** For each dataset, create a copy of the internal dataspace object.
- **esdm_dataspace_get_dims** For each dataset, read the number of dimensions from the dataspace.
- **esdm_dataset_get_name_dims** For each dimension, read its name.
- **esdm_dataset_get_size** For each dimension, read its size.

- **add_to_dims_tbl** For each dataset and for each dimension, insert the dimension into the dimension table, if not already there. The dimension table is constructed in memory to allocate the number of dimensions, sizes, and names.
- **insert_md** For each dataset, insert the information about its metadata.
- **esdm_container_get_attributes** Open the global attributes of the NetCDF file.
- **add_to_dims_tbl** If there exists a dimension that was not previously used in any dataset, insert this dimension into the dimension table.

4.5.2.2. ESDM Close

When nc close is called, the respective function ESDM close is triggered to close the NetCDF file. The following functions are called to close the NetCDF file according to the ESDM entities.

- **ESDM_nc_get_esdm_struct** Loads the data from the NetCDF file to memory.
- **ncesdm_container_commit** Insert the information from the dimension table as a global attribute and make the information inside the container persistent.
- **esdm_container_dataset_count** Check the number of datasets in the container.
- **esdm_container_dataset_from_array** For each dataset, load it into memory.
- **esdm_dataset_close** For each dataset, remove its information from storage.
- **esdm_container_close** Remove the container from storage.

4.5.3 Tests in C

The directory **nc_test4** has originally 104 tests in C language. The tests were first classified according to their output using NetCDF, because some tests turned out to be non-functional.

Therefore, from the 104 tests, 78 tests were used to demonstrate ESDM functionalities. Each test was classified according to their status according to the following four categories:

- **Success** It means the original test is successful when using ESDM (17 tests).
- **Partial Success** It means the original test is successful when using ESDM, but some parts of the test code had to be removed because ESDM does not support that specific feature (25 tests).
- **Inconclusive** It means the original test is not successful when using ESDM, and some features still need to be implemented or revised to provide the expected result. The missing features should be available in the short term (4 tests).
- **Failure** It means the original test is not successful when using ESDM. Some of the missing features should be available in a medium-term and others are not expected to work with ESDM at all (29 tests). Specific information about the tests can also be found in the final compatibility report.

4.5.4 Python

4.5.4.1 Support

There is a Python module for accessing NetCDF files. We provide a patch to the module to support ESDM. The patch and instructions to run the tests are available in the directory esdm-netcdf/dev. The changings in the original patch to include ESDM can be seen in Figure 16. Basically, we add the flags HAS_ESDM_SUPPORT, NC_FORMATX_ESDM, NC_ESDM and fmt == 'ESDM' (ESDM format).

```
+        NC_ESDM # ESDM support for Python
+        NC_FORMATX_ESDM
+IF HAS_PARALLEL4_SUPPORT or HAS_PNETCDF_SUPPORT or HAS_ESDM_SUPPORT:
+IF HAS_PARALLEL4_SUPPORT or HAS_PNETCDF_SUPPORT or HAS_ESDM_SUPPORT:
+                 'NETCDF4'          : NC_FORMAT_NETCDF4,
+                 'ESDM'             : NC_FORMATX_ESDM}
+               'NETCDF4'          : NC_NETCDF4,
+               'ESDM'             : NC_ESDM}
+        elif formatp == NC_FORMATX_ESDM:
+            return 'ESDM'
+            if is_netcdf3 and N > 1 and fmt != 'ESDM':
+    is_netcdf3 = fmt.startswith('NETCDF3') or fmt == 'NETCDF4_CLASSIC' or
    ↪ fmt == 'ESDM'
+    `NETCDF4_CLASSIC`, `NETCDF3_64BIT_OFFSET` or `NETCDF3_64BIT_DATA` or
    ↪ ESDM."""
+        ``NETCDF3_64BIT_DATA`` or ESDM.
+        IF HAS_PARALLEL4_SUPPORT or HAS_PNETCDF_SUPPORT or
    ↪ HAS_ESDM_SUPPORT:
+               parallel_formats = ['ESDM']
+                  IF HAS_PARALLEL4_SUPPORT or HAS_PNETCDF_SUPPORT or
    ↪  HAS_ESDM_SUPPORT:
+               IF HAS_PARALLEL4_SUPPORT or HAS_PNETCDF_SUPPORT or
    ↪ HAS_ESDM_SUPPORT:
+        if self.data_model != 'NETCDF4' and self.data_model != 'ESDM':
+        if self._data_model == 'NETCDF4' or self._data_model == "ESDM":
+        if self._grp.data_model != 'NETCDF4' and self._grp.data_model != '
    ↪ ESDM':
+    has_esdm_support = False
+    f.write('DEF HAS_ESDM_SUPPORT = 1\n')
```

Figure 16: Changes in the patch to include ESDM

4.5.4.2 Procedures to Setup NetCDF Python

We provide a build script:

$ ./build.sh

Python tests are created now inside the directory dev/netcdf4-python/test. Go to that directory.

$ cd netcdf4-python/test

Now, copy or link the esdm.conf file to this directory. A possible example is:

$ cp ../../../build/libsrcesdm test/ esdm.conf .

All Python tests start with the prefix tst . This string has to change to enable us to run pytest. To do that, one option is use the mmv utility. Note that the tests names need to be changed, but the tests files (.nc files) have to remain unchanged. The tool can be installed and used in Debian-based distributions as follows:

$ sudo apt-get install mmv

Now, rename the existing tests using the mmv tool.

$ mmv tst*.py test#1.py

Run the mkfs.esdm utility with the following parameters to generate the directories to store ESDM:

$ mkfs.esdm --create --remove --ignore-errors -g -c esdm.conf

Finally, install the pytest utility:

$ sudo apt-get install python3-pytest

and run it

$ pytest --junitxml results.xml

The file results.xml can now be uploaded by Jenkins which run the results on a daily basis to keep testing the compatibility during ESDM further developments.

5. PAV Interface

This is a continuation of M5.1 sketching the evolving design since the delivery of M5.1.

5.1 Level of Abstraction

Figure 17 shows the different levels of abstractions for the computation of postprocessing, analysis and visualization in regards to the ESiWACE components. The connection between the IME client and server is just an example.

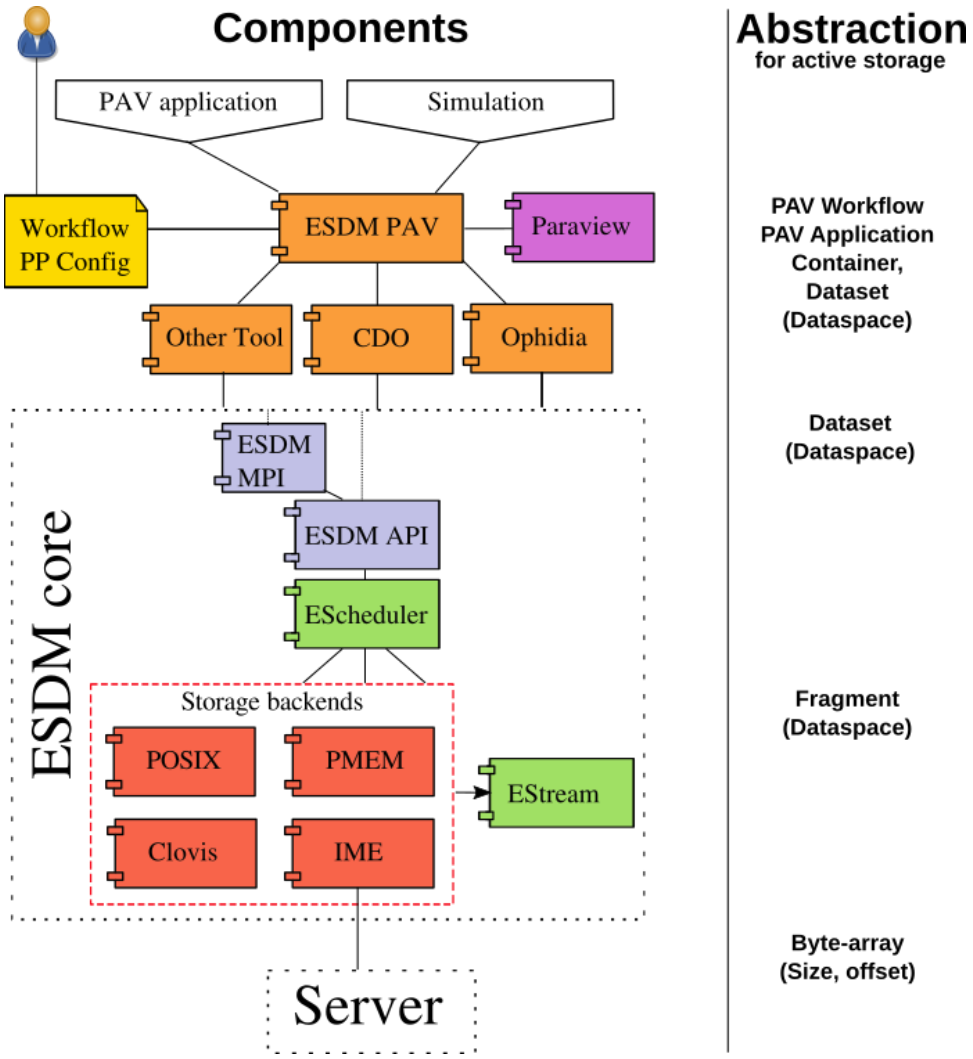

Figure 17: Abstraction for active storage considering ESiWACE components

The following high-level description covers the responsibilities from top down for a workflow.

- A **user** has created a **Workflow PAV Experiment Configuration** describing the operations to be performed on the workflow. In the final version, this could be a DAG of operations and may include data sinks for in-situ visualization using **Paraview** [Ayachit, 2015]. A workflow here may involve multiple datasets each with a dataspace that constraints the region of operation.

- The **user** may provide the configuration to either

    - kick off this workflow for post processing of existing data by invoking a command line tool (pav-run-pp-workflow)

    - register the workflow directly in the simulation by setting an environment variable or calling the PAV API. In this case, once the required datasets are available, the post-processing workflow shall be executed. Note that this requires that the application is PAV-enabled, i.e., uses the API calls to setup post processing.

- An **application** such as a **simulation** utilizes the PAV API to register metadata about individual datasets that provides the necessary information such as topology of the data, allowing the PAV API to understand the datasets submitted via NetCDF/ESDM or ESDM.

- The **PAV API** parses the workflow configuration and invokes the specified kernel (or command line tools) utilizing generated file names representing the ESDM containers and datasets required. Internally, PAV must orchestrate workflows that span multiple datasets by utilizing existing tools or kernels.

- A **PAV application** such as **CDO** or **Ophidia** performs the instructed operation on a subset or the whole workflow. Ultimately, producing a data product that is stored using ESDM again. A kernel might be implemented in CPU or using specialized hardware or GPUs. A tool may also use the ESDM API to offload some computation on individual datasets on ESDM and servers by using ESDM **streaming** or by deploying **coroutines that will execute whenever an IO is performed**. Given that GPU acceleration benefits mostly from data-intensive operations, more complex and compound operations are required. As storage servers do not bring GPUs normally, ESDM streaming or coroutines are limited to CPUs for now.

5.1.1 Streaming Scenario to Accelerate Reads

A client may intend to read data from ESDM and offload operations that partially reduce data while it is fetched. Typically, such operations may involve a map-alike operation that is executed on each element and/or a reduce operation that aggregates all processed elements into a subset of data.

- The **ESDM API** is invoked specifying the dataset and dataspace and a specific function shall be performed on. This includes a map-alike operation that shall be executed on every data object and a reduce operation that aggregates the outcomes of the map operation. Only one process of a parallel application is expected to set up the operation.

  As any reduce operation that is to be used within a map-reduce parallelization, it must be associative to ensure deterministic results in the presence of non-deterministic scheduling. Additionally, at least for the first prototype of ESDM, the reduce operation must also be idempotent to ensure deterministic results in the presence of partially overlapping fragments.

- The **scheduler** identifies the relevant fragments that overlap with the dataspace and makes a selection of these fragments. It enqueues the respective operations in backend-specific threads that call the backend-specific function to perform the operation. It also configures the callbacks for the EStream library such that potential reduction operations will be performed.

- The **storage backend** for a specific backend receives the call to process a fragment's data on a given dataspace. It may decide to invoke a storage-specific API to execute the kernel on the server(s) that host the data. Alternatively, it may use the EStream functionality to perform the operations on the client while data is fetched. In either case, only the relevant data must be processed. Information regarding the required region(s) of a partially overlapping fragment needs to be encoded in the payload to the server. Given the appropriate API for the storage, the client may request to read only the subset of data.

- The **EStream** library provides routines that implement the mandatory processing of the data as it is read or written. What processing is performed is determined by the user on a per-dataset basis, and it may include things like compression, checksumming, and execution of other user-defined

coroutines. There are different routines available to support normal reading/writing and map-reduce operations. In either case, the backend may use a single call to perform all the processing it is obliged to do, or it may handle parts of the processing itself, possibly offloading it to its storage servers.

- A **storage-server** reads objects (a byte array representing ESDM fragment data) and executes the specified kernel on the data. It is expected that such a kernel can easily apply the kernel functionality on the data and returns the resulting product to the client.

5.1.2 Coroutines Scenario to Offload Data-Production to Servers

Coroutines register operations that are invoked on server when data passes through the server realizing an active-storage concept. It can flexibly perform any operation instructed by the client. Basically, when the server receives a data buffer such as using a Remote Direct Memory Access (RDMA) or Remote Procedure Call (RPC), the coroutine must be executed. It could, for example, create an additional file on the server to store additional data products or data indexes. The call to set a coroutine is a collective operation in a parallel application.

- The **ESDM API** is invoked specifying the dataset and dataspace that a specific coroutine shall be performed on. The information about the coroutine is stored inside the dataset. A read coroutine is registered on all existing fragments. A write coroutine is registered for any subsequently created fragment before it is written by the **scheduler**. Therefore, the coroutine is registered on the backend. If a backend does not support coroutines, the scheduler must register the operation on the client(s) (which is the reason why it is a collective operation). Any subsequent read or write operation will then trigger the coroutine.

- The **storage backend** for a specific backend receives the information to register a coroutine (for writing during creation of a fragment or for reading by calling a function). It forwards this call to the backend-specific server by utilizing a vendor-specific implementation.

- The **storage server** must execute the specified coroutine on the data whenever the data is accessed by the specific client. This may be implemented by the storage system by resending information about the coroutine for any read or write operation.

5.2 Refined ESDM PAV Architecture

This section provides a description of an updated ESDM PAV architecture with respect to that presented in the milestone MS5.1 from WP5. The architecture has evolved in order to take into account some new aspects and requirements. In particular, we include the concept of a PAV *experiment* which defines a workflow of post-processing, analytics and visualization applications.

In order to support heterogeneous PAV use cases, the workflow should not only consist of a simple sequence of operations, but also of trees of tasks in which multiple inputs or outputs can be used by each task and even of more complex Directed Acyclic Graph (DAG) of tasks. These concepts represent new functional requirements with respect to those identified in MS5.1.

The *experiment* definition language will be modelled after the requirements from the initial set of PAV tools and applications envisioned in the context of the project. However, it will be defined from the beginning in order to be as general as possible to support also additional PAV tools. To this end, a plugin-

based approach is exploited to address the flexibility required and allow a simple extension of the set of tools handled by the ESDM PAV meaning runtime plugins are going to be implemented for each PAV tool. The initial set of PAV tools supported by ESDM PAV runtime could then be extended easily with new plugins.

Besides the support for PAV workflows, the architecture has evolved to strengthen the integration with the ESDM I/O core functionalities. The lower level kernel API will hence represent an extension to the ESDM library developed in WP4. To this end, the analytical kernels developed in the context of WP5 to support PAV applications will directly exploit the kernel interface jointly with active-storage kernels and I/O functionalities. This will allow supporting in-flight analytics while providing a minimal and integrated API, addressing the efficiency and usability requirements defined in MS5.1.

Figure 18 shows a detailed view of the refined architecture. The workflow is defined through a configuration file called ESDM PAV Experiment Document. The next section provides a draft of the supported abstractions and some examples of how it should look like.

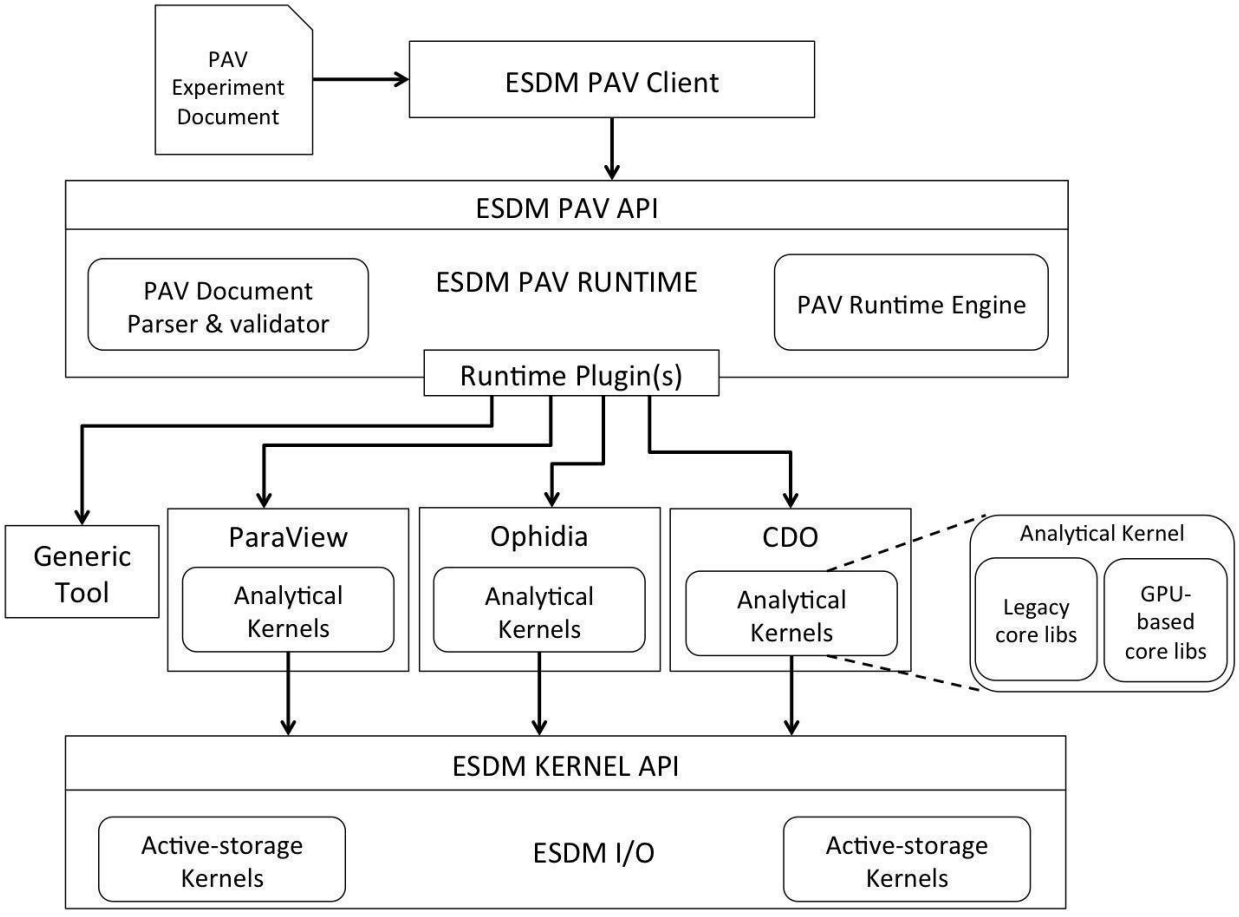

Figure 18: Refined PAV architecture

As shown by the diagram, the ESDM PAV runtime environment will handle the parsing and validation of the PAV document before its execution to check if the document file follows the syntax and respects the

workflow definition rules. For example, if dependencies are properly described and if there are no cyclic dependencies. The single tasks defined in the validated document are then mapped to actual commands which are submitted and managed, according to the dependency definition, by the in-memory runtime engine. Tasks execution will be automatically handled by the runtime environment.

Besides orchestrating the execution of PAV tasks, the ESDM PAV runtime engine will also provide the features to interrupt the workflow and check its progress (e.g. number of tasks completed and number of total tasks in the experiment). All these functionalities will be made available to higher-level applications through the C-based ESDM PAV API.

A subset of the PAV API provides the means to annotate I/O to enable any kind of postprocessing or in-situ analysis. A PAV-enabled application that uses this API is ready to run arbitrary PAV workflows. From the scientists perspective that want to setup a PAV workflow, all that is necessary is to provide an appropriate PAV experiment configuration.

Furthermore, a command line client, the ESDM PAV Client, will be provided to allow the execution of PAV workflows on existing data without the need to write specific C code. In this case, the same experiment configuration file can be applied to operate. Therefore, a user can choose to run a workflow at runtime of the PAV-enabled application or afterwards.

The set of PAV tools supported by the runtime environment can be extended through the definition of new runtime plugins, as described before.

Specific Analytical Kernels for in-flight analytics, both based on GPU core libraries as well as legacy libraries, will be developed during the project to support PAV use cases. These kernels will exploit the ESDM library for I/O operation and the ESDM API Kernel extensions for the implementation of the core functionalities. Analytical kernels can also take advantage of active-storage kernels, if available, for pre-processing and reducing the amount of data transferred from the storage backends to the compute nodes and GPUs.

5.3 ESDM PAV Experiment Definition

The ESDM PAV Experiment document is the document describing the PAV experiment workflow in terms of tasks and dependencies. This configuration file, defined by the user, is provided to the ESDM PAV runtime for the execution of PAV applications.

The format initially chosen for the definition of the document is JavaScript Object Notation (JSON). Other more compact formats, such as YAML, might be taken into consideration later during the project. JSON represents a widely used language-independent interchange format, which provides a flexible way for specifying data in a human-readable way. JSON syntax allows, in fact, to express various data types including: strings, numbers, ordered lists of values (arrays) and structured objects consisting of a collection of key-value pairs. These data types can be also nested to define more complex data structures allowing for great flexibility in the definition of the set of data structures that can be handled.

Concerning the PAV experiment document, the JSON format is used to define the supported workflows abstractions, such as tasks and dependencies. Keywords are defined for each abstraction to represent the information required to manage and run properly the workflow. To this end a preliminary schema for the workflow has been defined. An example of the workflow definition is provided in Figure 19.

```
"tasks":
[
...
    {
        "name": "Time reduction",
          "type": "ophidia",
          "operator": "oph_reduce",
          "arguments":
        [
              "operation=avg",
             ...
        ],
        "inputs":
          [
            "CUBEPID"
        ]
    },
    {
        "name": "Export data",
        "type": "ophidia",
        "operator": "oph_exportnc",
        "dependencies":
        [
            {
                "task": "Time reduction",
                "argument": "cube"
            }
          ],
        "outputs":
          [
            "FILENAME"
        ]
    },
...
]
```

Figure 19: Workflow definition in JSON format

In essence, a workflow is defined as a list of JSON objects called *tasks.* Our example shows a snippet of an Ophidia operators-based workflow consisting of only two tasks. The first one consists of a parallel data reduction operator executed over the datacube identified by *CUBEPID*, while the second one exports a datacube into a NetCDF file called *FILENAME*. Moreover, the second task has to start only after the output of the first task is made available.

The operation to be performed by each task is coded within a proper object structure consisting of the following keys:

- **Name** Represents the name given to that specific task. It is a mandatory key and its values must be unique within the whole PAV document file.
- **Type** Type of PAV tool to use for the execution. This is also a mandatory key. In the example above, Ophidia is the only tool used for the execution. Additional tools, such as CDO, can be used by specifying the proper value.

- **Operator** The (Ophidia) operator to be executed. This is a mandatory field.
- **Arguments** An optional key consisting of an array of one or more values. Each value describes an input argument for the specified operator.
- **Dependencies** optional list of the tasks (parents) that have to be completed before executing the reference task (child). It is possible to set input arguments of the child task to the output value of a parent task by specifying the argument name (“cube” in the proposed example). If not set, the task is considered independent and so it will be executed immediately.
- **Inputs** An array consisting of PID of input cubes or name of input files. In case dependencies are specified this field can be optional; otherwise inputs should be provided.
- **Outputs** An array consisting of the names to be assigned to output files. This field is not mandatory because by default a random name is assigned.

The ESDM PAV runtime exploits the value of the type field to implement the aforementioned plugin-based approach. In fact, the runtime will select the tool to be executed and the command format to be adopted based on the value provided. Of course, the possible values that can be used are restricted to the set of plugins actually available.

5.4 ESDM Extensions for Streaming

As illustrated in MS5.1, ESDM provides an API to perform an extended class of operations that support map-reduce on a dataset in the read-path:

```
esdm_status esdm_read_stream(esdm_dataset_t *dataset, esdm_dataspace_t *space, void *user_ptr,
esdm_stream_func_t *stream_func, esdm_combine_func_t *combine, esdm_reduce_func_t *reduce_func);
```

The signatures of **stream** and **reduce** functions will enable the execution of the stream function on each data element (if not NULL) and the reduce function across all elements (if none is NULL). The user may provide additional information via **user pointer**. Data reduced by **map** is expected to be smaller or equal in size, but not necessarily.

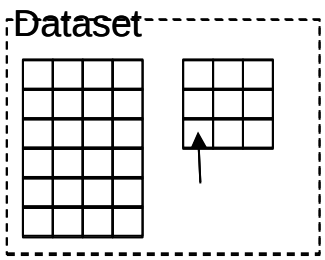

Figure 20: Example of an ESDM dataset and its fragments

ESDM typically partitions the dataset into multiple fragments that can be placed on multiple storage systems. As shown in Figure 20, the ESDM client process needs to perform the final reduction function. For storage systems without computing capabilities, the ESDM library will read the data from the storage system regularly and perform the map operations returning the proper data. The backend may utilize the EStream facility to pipeline the I/O and the analysis more efficiently.

If the storage system provides active-storage capabilities such as Clovis and IME, i.e., they support server-side computation, then the processing operations are forwarded to the server. Since the storage servers operate on byte arrays, ESDM will translate the high-level access to a dataset into low-level byte accesses.

Regardless of the server-sided low-level API provided, ESDM will provide a lightweight wrapper library that provides a single API. The general signature for this low-level interface is sketched here:

```
int server_map_reduce(void*objID, size_t off, size_t size, char const * funcname, char const * libname, size_t max_out_data_size, void * out_buff, size_t * out_size, int8_t * payload, size_t payload_size);
```

An ESDM backend may implement the function allowing the ESDM client to offload the computation to the server.

A call to this function will trigger on the server work on the file/object with the system-specific identifier, read at the specified offset a given size of the data and pass it to **funcname** inside the specified shared library; it will pass on the payload to the function on the server to provide arguments. The function invoked on the server may return a maximum amount of data as specified. Once the function returns with a return code of 0, the result is found in **out_buf** with the size in the variable **out_size**.

A potential invoked function on the server has the signature:

```
size_t function(int8_t * data, size_t off, size_t size, int8_t * out_buff, size_t max_out_data_size, void * payload, size_t payload_size);
```

It is called with the data buffer as input, the offset and size it was read from, and a pointer to the output buffer which shall have size **max_out_data_size**. Also, the payload submitted on the client-side is transferred to the function. The return code is the number of bytes actually written to the output buffer.

The payload must be a single contiguous blob on the heap that can be freed using free().

For the development version of this approach, we will encode arguments to the server function such as a fill value using the native machine format and not transform it into a machine-independent representation.

5.5 Extension for Active Storage

Active storage components allow to offload general data processing in the read or write path flexibly. We make an early draft here focusing on the stream-processing in the read path during the ESiWACE project, as this promises to be more beneficial for now. The core concept is that, upon a server-side I/O, the operations pass through a coroutine that can flexibly perform any operation instructed by the client. We call the concept coroutine as it resembles the general idea of a coroutine: it allows execution of a larger computational task to be suspended after the I/O completed and being resumed for the next I/O call.

Basically, when the server receives a data buffer such as using an RPC, the coroutine should be executed. It could, for example, create an additional file on the server to store additional data products. Generally, it would be advantageous if vendors provide a vendor-agnostic API to perform potentially expected manipulations such as

- Accessing of data objects and appending arbitrary data
- Maintaining a persistent state

On the semantic level of ESDM, we envision that coroutines are registered on either read or write path as follows:

```
esdm_status esdm_co_routine_register_read(esdm_dataset_t *dataset, esdm_co_routine * func, void *
payload, size_t payload_size);

esdm_status esdm_co_routine_register_write(esdm_dataset_t *dataset, esdm_co_routine * func, void *
payload, size_t payload_size);
```

At the moment, we plan to attach only a single coroutine to either read or write path. As a function could invoke arbitrary code, theoretically such a coroutine could orchestrate the invocation of a sequence of coroutines -- we may explore this in the future.

The esdm_co_routine structure contains information about the library and function name to use and the payload is additional data provided to these functions. The payload must be a single contiguous blob on the heap that can be freed using free().

Backends that support the execution of coroutines set the feature flags, respectively: ESDM_BACKEND_SUPPORT_COROUTINE_READ or ESDM_BACKEND_SUPPORT_COROUTINE_WRITE.

If the backend does not support the call, the coroutine needs to be executed on the client side.

On all fragments of the dataset, the coroutine information will be set internally as part of the fragment structure.

The low-level execution by backends that relate to byte-streams works as follows:

1. The client opens the data object (e.g., using POSIX open() call).
2. The client registers a coroutine using the function:

   ```
   int register_co_routine(void*objID, char const * funcname, char const * libname, int8_t *
   payload, size_t payload_size);
   ```

   This function returns 0 if the registration was successful.
3. Any subsequent I/O on the server will then call the coroutine located in the library. The signature of the coroutine is:

   ```
   int coroutine(int8_t * data, size_t off, size_t size, void ** inout_internal_ptr, void *
   payload, size_t payload_size);
   ```

   The function must be called on the server when the client calls register_co_routine using an inout_internal_ptr that contains NULL and setting a data pointer of NULL. This allows the coroutine to load configuration information from the payload and supplementary resources. It may then modify the internal_ptr to store arbitrary data in memory. The internal_ptr pointer is

then provided for subsequent calls to the coroutine which will be invoked using a payload of NULL. When the client closes the data object, the coroutine is called using a NULL pointer in the data field and NULL in the payload. This is the indicator for the coroutine to free all resources. This model generally also supports stateless servers, but requires the server to perform multiple calls to the coroutine to setup and free the required resources. The payload is a machine-native structure.

4. Theclose() call on the client will remove the coroutine.

There are several considerations that must be made regarding the concurrent execution of coroutine and stream processing. For example: would a coroutine see the already preprocessed data or the original data? Can the user configure the order of the execution? Can a coroutine modify the source of data allowing to inject precomputed/cached data?

6. Conclusions

In this milestone, we described the overall interaction between work package tasks and WP4-related software components in the ESiWACE-2 project. Several developed components such as ESDM play an important role in multiple WP tasks and may be useful in different user scenarios. The scientific workflow can start at a Cylc workflow specification that may use several of the developed software components. We also illustrated the architectural and design changes that we have or will perform as part of ESiWACE to these software packages.

References

[Ayachit, 2015] Ayachit, Utkarsh, "The ParaView Guide: A Parallel Visualization Application", Kitware, 2015, ISBN 978-1930934306.

[Betke et al, 2019] Betke, E., Kunkel, J.: Benefit of DDN's IME-Fuse and IME-Lustre file systems for I/O intensive HPC applications. In: Yokota, R., Weiland, M., Shalf, J., Alam, S. (eds.) High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers. pp. 131–144. No. 11203 in Lecture Notes in Computer Science, ISC Team, Springer (01 2019), DOI: https://doi.org/10.1007/978-3-030-02465-9_9.

[Fiore et al, 2014] S. Fiore et al., "Ophidia: A full software stack for scientific data analytics," 2014 International Conference on High Performance Computing & Simulation (HPCS), Bologna, 2014, pp. 343-350.

[Hassel, 2014] D. Hassel., "The CFA conventions for the storage of aggregated fields (v0.4)" http://www.met.reading.ac.uk/~david/cfa/0.4/, 2014.

[Jette et al, 2002] Jette, M.A., Yoo, A.B., Grondona, M.: "SLURM: Simple Linux Utility for Resource Management". In: In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003. pp. 44–60. Springer-Verlag (2002).

[Meurdesoif et al, 2016] Meurdesoif, Y., Caubel, A., Lacroix, R., D'erouillat, J., Nguyen, M.H.: "XIOS tutorial", 2016, http://forge.ipsl.jussieu.fr/ioserver/raw-attachment/wiki/WikiStart/XIOS-tutorial.pdf.

[Oliver et al, 2019] H. Oliver et al., "Workflow Automation for Cycling Systems", in Computing in Science & Engineering, vol. 21, no. 4, pp. 7-21, 1 July-Aug. 2019.

[Schulzweida, 2019] Schulzweida, Uwe. (2019, October 31). "CDO User Guide (Version 1.9.8)". http://doi.org/10.5281/zenodo.3539275.