European Commission

Horizon 2020
European Union funding
for Research & Innovation

E-Infrastructures

H2020-INFRAEDI-2018-1

INFRAEDI-2-2018: Centres of Excellence on HPC

EoCoE-II

Energy oriented Center of Excellence : toward exascale for energy

Grant Agreement Number: INFRAEDI-824158

D4.2

Report on data interface and in-situ capabilities

Project and Deliverable Information Sheet

EoCoE-II	Project Ref:	INFRAEDI-824158
	Project Title:	Energy oriented Centre of Excellence: towards exascale for energy
	Project Web Site:	http://www.eocoe2.eu
	Deliverable ID:	D4.2
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	M24 31/12/2020
	Actual Date of Delivery:	M25 31/01/2021
	EC Project Officer:	Matteo Mascagni

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

Document Control Sheet

Document	Title :	Report on data interface and in-situ capabilities
	ID :	D4.2
	Available at:	http://www.eocoe2.eu
	Software tool:	LaTeX
Authorship	Written by:	Julien Bigot (CEA/MdlS), Sebastian Lührs (FZJ), Karol Sierociński (PSNC), Christian Witzler (FZJ),
	Contributors:	Kacper Sinkiewicz (PSNC)
	Reviewed by:	PEC, PBS

Document Keywords: I/O, Data, In-situ, PDI, FlowVR, Compression, NetCDF, HDF5, Visualization, SENSEI

Contents

List of Figures

List of Tables

1 Executive summary

This Deliverable provides technical details of the activities taking place in the framework of the I/O and dataflow work package of EoCoE-II towards two goals: Offering in-situ processing capabilities, and making these capabilities available to codes of the scientific challenges of the project. The work presented within this report focuses on five main points to reach these objectives.

A key element is the design process of the PDI data interface to support loose coupling of simulation codes with HPC-oriented libraries. This interface provides a single annotation API to easily allow integration of all the in-situ features described in this deliverable for the applications used by the scientific challenges and for other data handling related features developed in EoCoE-II such as the work regarding fault tolerance or the ensemble run support developed in WP5. The report will give a technical overview concerning the developed PDI functionalities such as in the context of process-local and in-situ data processing, two PDI plugins will be covered in more detail: the pycall-plugin and the user-code plugin. These plugins make it possible to execute in-situ data processing on the same processes in parallel to the simulation code in a loosely coupled manner, either in python or in compiled languages (C, C++, Fortran, etc.), as a specific requirement expressed by the Fusion scientific challenge flagship code GyselaX. A even more flexible but also more complex approach is provided by the FlowVR in-situ and in-transit framework which also received support through another PDI plugin. FlowVR allows to write complex distributed data processing workflow for parallel applications. The plugin developed makes it possible to use this framework without having to handle the low-level issues typically associated with its use. Benchmarks demonstrate the negligible overheads of the approach and its potential applicability in the future for codes currently relying on process-local solutions such as GyselaX.

Beside the PDI related work we present the work done in EoCoE-II regarding in-situ data compression in the HDF5 and NetCDF libraries, which allows to lower the overall storage footprint of an application already during the runtime of the simulation in contrast to a post-processing based setup. Multiple compression approaches were evaluated to determine their applicability to answer the needs of the Water for Energy scientific challenge flagship code ParFlow. Based on the results of these benchmarks, compression using ZLIB has been integrated in ParFlow master branch and can now be used in production.

Finally we present the initial work to support in-situ visualization through the SENSEI library. The work done in EoCoE-II combines SENSEI for the in-situ visualization proper and ADIOS2 for data transfer between nodes. It is made available to the scientific challenges through a PDI adapter for the pycall plugin.

Overall, support for many different variations of in-situ data processing were developed in the project according to the requirements of the scientific challenges. Thanks to the PDI data interface, these developments do however not remain isolated as each development benefits the EoCoE-II community as a whole as soon as it is made available through PDI.

2 Acronyms

Table 1: Acronyms of partners and institutes

Acronym	Partner and institute
CEA:	Commissariat à l'énergie atomique et aux énergies alternatives
FZJ:	Forschungszentrum Jülich GmbH
MdlS:	Maison de la Simulation
PSNC:	Poznań Supercomputing and Networking Center

Table 2: Acronyms of software packages

Acronym	Software and codes
ADIOS2:	Adaptable Input Output System, version 2
Catalyst:	ParaView in situ library
FTI:	Fault Tolerance Interface
FlowVR:	a middleware for high performance interactive applications
Gysela:	GYrokinetic SEmi-LAgrangian
HDF5:	the Hierarchical Data Format, version 5
MELISSA:	Modular External Library for In Situ Statistical Analysis
MPI:	Message Passing Interface
NetCDF:	the Network Common Data Form
ParFlow:	PARallel Flow
ParaView:	Multi-platform data analysis and visualization application
PDI:	PDI Data Interface
SENSEI:	Provides simulations with a generic data interface for in-situ visualization
SIONlib:	Scalable I/O library for parallel access to task-local files

Table 3: Acronyms of scientific and technical terms

Acronym	Scientific Nomenclature
AOP:	Aspect-Oriented Programming
CPU:	Central Processing Units
JSON:	JavaScript Object Notation
HPC:	High Performance Computing
I/O:	Input and Output
RDMA:	Remote Direct Memory Access
SC:	Scientific Challenge
TRL:	Technology-Readiness Level
WP:	Work Package
YAML:	YAML Ain't Markup Language

3 Introduction

Data flow and data handling gets more and more complex in modern Exascale-aware HPC applications. Beside the classical basic input/output scheme, data also needs to be transferred between different parts of a complex workflow or needs to be analysed during the execution of the simulation (also named in-situ or in-transit data handling). This leads to a growing number of data handling related APIs and configuration options. The PDI data interface, as designed and implemented within context of the EoCoE project, helps to provide a single middleware API towards the simulation codes which can be utilized for any kind of data handling approach.

This deliverable provides an overview concerning the key elements of the PDI API and their utilization with the major focus on in-situ and in-transit data handling approaches and will also provide details and results on related in-situ and in-transit development work. All developments described in the frame of this report are directly linked to task 1 and task 4 in work package 4 of the EoCoE-II project and focuses mainly on technical details of the newly developed features to provide more details with respect to the previous Deliverable D4.1. The utilization results based in the individual integration of the different scientific challenges will be presented in D4.4: The final report on I/O improvement for EoCoE codes.

Section 4 provides an overview of the usage of the PDI interface and the architecture underneath. This is followed by section 5 and section 6 which provide details on three different PDI plugins in the context of in-situ data processing. These sections, mostly related to PDI, are followed by sections 7 and 8 which provide more details on two in-situ approaches: In-situ data compression within the NetCDF and in-situ visualization with the help of SENSEI, which both allows to lower the overall data footprint of an HPC application.

3.1 Link to other work packages

The work done in the context of the deliverable is directly connected to other work packages in context of EoCoE-II: The main scientific challenge applications, handled by WP1, which benefit from the interface and in-situ related work are GyselaX, ParFlow and SHEMAT. In cooperation with the scientific challanges the integration of PDI was handled as part of WP2 with direct support of members of WP4 to facilitate the transfer of the work tackled in this work package to production settings. This support work also provided a feedback channel for the PDI development work. Finally, as PDI is designed to act as a central data interface, we also worked in close collaboration with WP5 and an interface towards the Melissa framework developed there.

4 The PDI data interface and its use for in situ data manipulation

This section introduces the PDI Data Interface[1] (PDI) developed primarily by CEA and PSNC in the framework of EoCoE-II. It recalls a few aspects regarding the library development in the framework of the EoCoE-II project already reported in Deliverable D4.1 and it specifically focuses on the elements of the architecture of PDI enabling codes to leverage in-situ processing transparently.

PDI plugins and other solutions developed in the framework of EoCoE-II to support various approaches for in-situ processing are specifically discussed in the following sections. One must note that all these solutions –whether specifically developed for PDI or developed in libraries lower in the stack– are potentially accessible through the PDI interface. This approach enables all scientific challenges (SC) of the EoCoE-II project to leverage in situ processing as soon as they integrate the PDI interface in their code as part of WP2 tasks supported by task T4.5.1 "PDI general support" of WP4 (Cf. Deliverable D4.1 for more details on application support). In additions, some SC applications directly access the described improvements in the underlying libraries without going through PDI abstraction.

4.1 PDI interface and usage overview

PDI is designed to support loose coupling of simulation codes with HPC-oriented libraries. To this end, it adopts an approach inspired by aspect-oriented programming (AOP). Aspects that would usually be tangled with the code (*cross-cutting concerns* in AOP terminology) can instead be handled separately and independently. This include concerns such as for example logging, fault tolerance, result storage to disk, result post-processing, interaction with other codes in a code coupling or in an ensemble run setting, etc.

The pieces of code that implement each cross-cutting concern (*advice* in AOP terminology) are provided in PDI plugins. Each plugin implements a specific concern and can rely on dedicated libraries without inducing any dependency beyond this specific plugin. Neither the PDI library, nor the user-code has to be recompiled in order to leverage the plugin underlying library capabilities.

Locations where an advice can be applied (*joint-points* in AOP terminology) are identified using the PDI annotation API. This API makes it possible to **a)** expose the memory buffers where the code stores data and **b)** identify when significant steps are reached in the simulation. This is discussed in further details in Section 4.1.1

The selection of which advice to insert at each joint-point (*pointcuts* in AOP terminology) is specified in PDI *specification tree*. The specification tree is passed to PDI at initialization and hence can be replaced with no need for recompilation. Its content is further discussed in Section 4.1.3.

Overall, using PDI, it becomes possible to:

1. annotate code with PDI annotations independently of any third-party library,
2. compile the code once,
3. choose how to implement cross-cutting concerns, potentially using specialised or even architecture specific libraries in a file that can be easily chosen and changed at run-time.

This approach makes it possible to separate concerns between the domain scientists writing the simulation code and optimization specialists taking care of implementation choices for logging, fault tolerance, result storage, post-processing, code coupling, ensemble runs, etc. It greatly improves code portability, maintainability and composability.

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

4.1.1 Code annotations

PDI code annotations offer a framework to annotate simulation codes and specify when a given computation is finished and its result stored in a given buffer. The API is available for C/C++ (Listing 1), Fortran (Listing 2) and Python (Listing 3). The remaining of the document primarily discusses the C API for brevity, however one must keep in mind that the three languages are supported using very similar APIs.

```c
// Identify a buffer whose content is valid and ready to be used
PDI_status_t PDI_share(const char *buffer_name, void *buffer_address,
                       PDI_inout_t access)
// Identifies a buffer that will be invalidated (reused)
PDI_status_t PDI_reclaim(const char *buffer_name)
```

Listing 1: PDI data annotation functions for C

```fortran
! Identify a buffer whose content is valid and ready to be used
SUBROUTINE PDI_share( buffer_name, buffer, access, err )
    CHARACTER(LEN=*), INTENT(IN)    :: buffer_name
    TYPE(*), ASYNCHRONOUS           :: buffer
    INTEGER(pdi_inout), INTENT(IN) :: access
    INTEGER, INTENT(OUT), OPTIONAL :: err
ENDSUBROUTINE PDI_share

! Identifies a buffer that will be invalidated (reused)
SUBROUTINE PDI_reclaim(buffer_name, err)
    CHARACTER(LEN=*), INTENT(IN)    :: buffer_name
    INTEGER, INTENT(OUT), OPTIONAL :: err
ENDSUBROUTINE PDI_reclaim
```

Listing 2: PDI data annotation functions for Fortran

```python
class pdi:
    # Identify a buffer whose content is valid and ready to be used
    def share(buffer_name: str, buffer, access: pdi.Inout)
    # Identifies a buffer that will be invalidated (reused)
    def reclaim(buffer_name: str)
```

Listing 3: PDI data annotation functions for python

In this API, the `PDI_share` and `PDI_reclaim` functions are used to identify sections in the code where a buffer contains data whose computation is finished and that is ready to be used. Such a *shared section* starts with a call to the `PDI_share` function and ends with a call to the `PDI_reclaim` function with a matching buffer name as illustrated in Listing 6. Inside this section, the content of the buffer whose pointer has been provided to `PDI_share` is shared with PDI and it can be used for I/O or any other purpose. For those familiar with the MPI API, this is very similar to the `MPI_Isend` ... `MPI_wait` logic.

Another option is to share a buffer, allocated in the simulation, for which the content can be provided to the simulation from the outside. The code annotated using PDI specifies the allowed direction of information flow using the `access` parameter of the `PDI_share` call.

- With `PDI_OUT`, the information flows *out* from the simulation; the data generated by the simulation in the buffer can for example be written to disk, or another application coupled with the simulation can read the content of the buffer.

```c
// Briefly exposes a buffer to PDI, this is equivalent to a combination of
// PDI_share immediatly followed by PDI_reclaim
PDI_status_t PDI_expose(const char *buffer_name, void *buffer_address,
                        PDI_inout_t access)
```

Listing 4: PDI expose annotation function for C

```c
// Identify an interesting point in code execution
PDI_status_t PDI_event(const char *event_name)
```

Listing 5: PDI event annotation function for C

- With `PDI_IN`, the information flows *into* the simulation; the buffer can for example be filled by reading data from disk, or in a code coupling another application can write into the buffer.
- The `PDI_INOUT` parameter is also available to allow a combined information flow in both directions in a single shared section.

Inside a shared section, the simulation code must make sure that it does not use the shared buffer in an invalid way. When specifying `PDI_OUT`, the content of the buffer can be used from the outside and the code should not modify it inside the shared section. With `PDI_IN` or `PDI_INOUT`, the content can be modified from the outside, so the code must also refrain itself from reading the content of the buffer as it might be in an inconsistent state. Because it is common for applications to want to limit the size of this section to a minimum, PDI offers the `PDI_expose` function that combines the share and reclaim and thus introduces no shared section.

In addition to these data-oriented annotations, PDI offers the `PDI_event` function whose C interface is illustrated in Listing 5. This function supports pure-event (data-less) annotations. Unlike the data annotation functions, this one does not share any buffer with PDI. It can however be used to mark transitions in the code, locations of interest such as for example the start of the main simulation loop or a point where all processes of a parallel simulation are in a known synchronized state.

Overall, these annotations provide information on the simulation execution. They are however only annotations, a purely declarative API. The code does not impose any choice nor does it require to be restructured to make use of these. This makes it possible to easily insert such annotations in any code whatever the approach taken to initially write it. In contrast to the internal structure of PDI the end user oriented API was not modified during the EoCoE-II time-frame and PDI guarantees complete API stability starting with the release of version 1.0.0 synchronized with Milestone MS5 of the project. This capability to write and annotate code using a table API while adding new behaviour through PDI plugin system characterise the loose coupling support offered by PDI.

4.1.2 Type system

In order to manipulate the buffers it receives, PDI needs to know the memory layout and semantics associated to each of them. For dynamic languages such as Python where this information is available at run-time, PDI directly extracts the information with no additional user action required. For statically compiled languages with no or limited introspection support like C/C++ or Fortran, the information has to be provided in the specification tree as will be presented in Sections 4.1.3.

The type system supported by PDI is very similar to the MPI or HDF5 type systems. It is a structural type-system, equality and other operations depend on the structure and semantic associated to data only, not to potential names given to the types. It offers four main categories of data types: *arrays*, *records*, *scalars* and *references*.

```c
double *data_buffer;
initialize( data_buffer );
while ( !computation_finished )
{
  compute_the_value_of( data_buffer, /*...*/ );
  PDI_share("main_buffer", data_buffer, PDI_OUT); // -- START of shared section
  do_something_without_data_buffer();
  do_something_reading( data_buffer, /*...*/ );
  PDI_reclaim("main_buffer"); // ----------------------  END  of shared section
  update_the_value_of( data_buffer, /*...*/ );
}
```

Listing 6: Definition of a PDI shared section in C. Between the call to `PDI_share` and `PDI_reclaim`, the buffer pointed to by `data_buffer` is shared with PDI.

- An array contains a given number of sub-elements, all of the same type and equally spaced in memory with potential padding at the beginning or end of the array. Each sub-element in the array can be identified by a numerical index.
- A record contains a number of sub-elements of potentially distinct types whose location in memory is specified as a displacement relative to the start of the record. Each sub-element in the record can be identified by a name.
- A scalar is identified by a size and an interpretation that should be given to the memory: an unsigned integer, a 2-complement signed integer, a IEEE-754 floating-point number, a pointer, etc. Unless the semantic of a scalar is opaque, the memory content can be directly interpreted.
- A reference (or pointer) is handled by PDI as a specific kind of scalar. Its content can be interpreted as a reference to a specific location in memory. The type of the referenced content is stored as part of the type. Hence a reference can be interpreted as a memory address or de-referenced to interpret the referenced data.

Each of these types also provide minimum-alignment information and a memory footprint (independent of the content size). This make it possible to describe "sparse types". This can for example be used to describe sub-arrays in a multi-dimensional array. This can also be used to describe a PDI record type `A` that represents only some of the members of a C `struct` while another PDI record type `B` could represent others members of the same C `struct`. The PDI type system is flexible enough to handle instances of these two types interleaved in memory. This for example makes it possible to represent data using an array of structures semantic (AoS) or a structure of arrays (SoA) semantic without any change to the actual memory layout.

PDI data types also support specific copy and destruction functions. They make it possible to handle data whose memory representation can not simply be moved at a different location but requires transformations. This is for example useful to deal with data containing references to their own content that must be changed when the content is moved. This is common in languages like C++ where one manipulates opaque datatypes that provide copy constructor/assignment operators and destructors. This is even more important for languages like python where the memory layout of most types is an implementation detail of the interpreter.

Overall, PDI type-system is full featured and can describe data types from languages as diverse as C, Fortran, C++ or python; including "sparse data types". In addition to the description of the memory layout of data, it associates a semantic interpretation to it and supports copy and deallocation operations. This type-system was greatly expanded in the framework of the EoCoE-II project. While the basic building blocks where in place at the start of the project, support for most

features beyond simple C/Fortran style scalar and arrays was added due to requirements in the SC applications as reported in Deliverable D4.1.

4.1.3 Specification tree

```c
// PDI initialization function, takes the specification tree as sole parameter
PDI_status_t PDI_init(PC_tree_t spec_tree);
```

Listing 7: PDI initialization function

The specification tree is passed to PDI at initialization as illustrated in Listing 7 and can therefore be changed with no need for recompilation. It is structured as a tree (hence its name) where each node can be either an ordered list, an un-ordered mapping or a scalar (the leafs of the tree). The concrete syntax can be anything supported by the *paraconf* library[2] developed in the framework of EoCoE-I. Typically, this is YAML[3] data read from a file on disk.

The YAML syntax can be considered as a superset of the JSON[4] syntax and supports the same three basic kinds of elements: lists, mapping and scalars. Lists and mappings can be either written in-line as in JSON or in an indentation-delimited multi-line block (similar to python) as illustrated in Listing 8. Scalars can be written as-is, as single-, double-quote enclosed or even multi-line strings. Single-line comments are introduced by the hash sign (`#`).

```yaml
# this is an in-line list of 3 elements
list1: [ elem1, 'elem2', "elem3" ]
# this is a similar list using the block syntax
list2:
  - elem1
  - 'elem2'
  - "elem3"
# this is an in-line mapping with 2 elements
map1: { key1: 'val1', key2: val2 }
# and this is a similar mapping using the block sytax
map2:
  key1: 'val1'
  key2: "val2"
```

Listing 8: Example of the YAML syntax

The specification tree can be used to configure behaviour of the library such as logging or error handling. While these aspects of the library related to usability and technology-readiness level (TRL) have been much improved in the course of the EoCoE-II project, they are not covered here. One should instead refer to Deliverable D4.1 regarding these aspects.

As evoked in Section 4.1.2, the configuration tree is also used to specify the type associated to buffers passed to PDI. This is done in the `data` section of the tree as illustrated in Listing 9. In this example, the types of three buffers are specified: a C `int` called `an_integer`, a C `struct` called `a_structure` and a 2D C array of `double` called `a_2D_array`.

While supporting the specification of all types supported by PDI, this approach suffers from usability issues. Any change to the size of the array would for example require to re-write and adapt the file. In order to improve this situation, PDI supports `$-expression` in the specification-tree. A `$-expression` is an expression introduced by the `$` sign that can include simple mathematical

[2] `https://github.com/pdidev/paraconf`
[3] YAML Ain't Markup Language, for more information, see `https://yaml.org/`
[4] the JavaScript Object Notation, Cf. `https://www.json.org`

```yaml
data:
  an_integer: int
  a_structure:
    type: struct
    members:
      - first_member: int
      - second_member: double
  a_2D_array: { type: array, size: [ 4, 8 ], subtype: double }
```

Listing 9: Description of buffer types in PDI specification tree

expressions or references to other buffers exposed by the code as variables. To be referenceable, a buffer should be declared in the `metadata` section instead of the `data` section as illustrated in Listing 10.

```yaml
metadata:
  array_size: { type: struct, members: { x: int, y: int } }
data:
  arr: { type : array, size: [ '$( array_size.x * 4 )', '$(array_size.y)' ] }
```

Listing 10: Usage of $-expressions in the specification tree

Whenever a metadata buffer is shared with PDI, its content is copied so as to be usable even after the end of the shared section where it was made available. This approach makes it possible to specify the types of the buffers in the tree once and for all without having to change it every time a run-time parameter is modified. Instead, one can expose the run-time parameters from the code and use them in `$-expression` in the specification of the type of other data. Since the `$-expression` in the types are re-evaluated every time a buffer is shared, this also makes it possible to describe buffers whose type evolve along time such as a dynamic vector of particles whose size evolve along the simulation.

The `$-expression` system has been much improved in the framework of the EoCoE-II project. In addition to plain variable and array elements access, support has been added for access to members of records or indirection through pointers. A PDI plugin has also been developed to provide initial values for metadata and hence break potential circular dependency cycles.

```yaml
data:
  main_data: { type: array, subtype: double, size: 1000000 }
  secondary_data: int
plugins:
  decl_hdf5:
    file: "my_file.h5"
    write: [ 'main_data', 'secondary_data' ]
  fti:
    config_file: fti_config.ini
    dataset: [ 'main_data', 'secondary_data' ]
    snapshot_on: new_iteration_start
```

Listing 11: Example of the plugin related part of the PDI specification tree

The second role of the specification tree is to express pointcuts that associate behaviour from plugins (advice) to the location in code containing annotations (joint-points). Unlike typical AOP languages, PDI does not define a common pointcut language. The list of plugins to load is determined by the set of keys under the `plugins` section of the specification tree and the actual

behaviour of the plugins is passed as a sub-tree associated to this key. This is illustrated in Listing 11 where two plugins are loaded: `decl_hdf5` and `fti` with a specific sub-tree passed to each plugin. The structure and semantic of the sub-tree each plugin accepts is plugin-specific.

The plugins distributed with PDI offer a very thin pure-interface layer on top of the libraries they wrap as the goal of PDI is not to implement new features only available through it, but instead to make the features of underlying libraries easily accessible.

The following sections will discuss plugins and features specifically suited to in-situ processing. This include the `user-code` and `pycall` plugins enabling users to execute arbitrary code in a compiled language or in python respectively that will be further discussed in Section 5 and 8. This also includes the `FlowVR` plugin that gives access to the dedicated in-situ and in-transit processing library of the same name as further discussed in Section 6. In addition, the `Decl'HDF5` and `Decl'NetCDF` plugins gives access to the in-situ compression features described in Section 7.

The `Decl'HDF5`, `Decl'NetCDF` and `Decl'SION` plugins support I/O to disk using the HDF5, NetCDF and SIONlib libraries respectively. The `FTI` plugin supports fault-tolerance by giving access to the features of the FTI library developed in the framework of EoCoE-II. The `Melissa` plugin is a work-in-progress to provide access to the Melissa library developed in WP5 of EoCoE-II. The `trace` plugin enables logging to follow the execution of a PDI-enabled code. The `MPI`, `serialize` and `set-value` plugins are support plugins useful in combination with other plugins. The `MPI` plugin defines the types required to share MPI objects such as communicators; this can be used in support of the Decl'HDF5, Decl'NetCDF, Decl'SION and FTI plugins. The `serialize` plugin supports serialization that can for example be used to reorganize data prepare data for libraries that expect contiguous blocks of memory such as FTI or SIONlib. The `set-value` plugin makes it possible to define the value of metadata from PDI itself and hence implement minimal logic on PDI side when required.

While all these plugins offer completely different features and therefore different interface, an effort has been made to provide similar interfaces when possible. For example, the `Decl'HDF5` and `Decl'NetCDF` plugins that offer access to two libraries with similar features have been designed to accept close sub-trees that make switching from one library to the other easy. This convergence of specification tree interface is made possible by the huge amount of work handled by the `$-expression` mechanism, leaving only the more simple configuration handling to the plugins themselves.

Overall, the plugins available through PDI have been deeply improved in the framework of the EoCoE-II project. The `Decl'NetCDF` plugin has been written from scratch during the project, specifically to fit the needs of the Parflow SC application as reported in Deliverable D4.1. The `serialize` and `set-value` plugins have also been developed from scratch to fit the needs of EoCoE-II SC applications (Parflow and GyselaX). The `Decl'HDF5` and `Decl'NetCDF` plugins sub-trees have been re-organized to unify their interface and ease replacement of one plugin by the other. The `Melissa` plugin is developed together with the main Melissa library. In addition, an `IOR` plugin has been developed to evaluate the performance and overheads induced by the library as reported in Deliverable D4.1.

4.2 PDI Library architecture

While the previous section focused on the presentation of PDI from a user point of view, this one offers a closer looks at the library internal. It describes the approach taken in PDI core library to support its plugins.

The overall architecture of PDI is illustrated in Figure 1. As one can see on this figure, the PDI library mediates the two aspects of the interaction between the simulation code and the plugins. Data transfer is handled by the PDI "*data store*" while control transfer is handled by PDI "*event subsystem*". The orchestration of these exchanges is driven by the specification tree described in

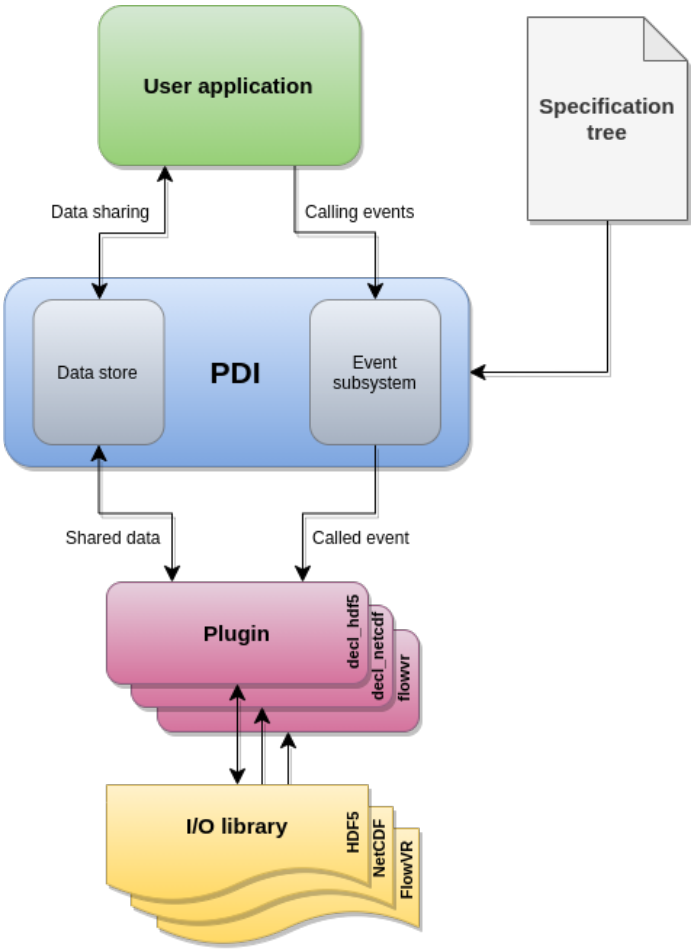

Figure 1: PDI Architecture overview

Section 4.1.3 that makes an important use of the `$-expression` mechanism. The remaining of this section describes these three structuring building blocks of PDI: the data store, the event subsystem and the expression mechanism.

4.2.1 Data store

The data store is the mechanism offered by PDI to handle data transfers between the application code and external data handlers in plugins. It behaves similarly to an in-memory process-local file-system, document store or a python dictionary of references.

The introduction of an intermediary name-space between the application and the plugins ensures loose coupling between these two elements. Instead of directly receiving parameters that the application code would have to specifically pass, the plugins can get the information they require from the data store similarly as two file-coupled applications would interact. In contrast a hard-wired implementation, by directly including the specific data handler into the simulation code, the application developer has to ensure to link the application to all necessary dependencies and to take care of all individual APIs. This makes it even more difficult to exchange or add an individual data handler. While offering the same simplicity as file coupling, PDI data store is designed to limit performance overheads to a strict minimum. No access to the disk or even memory copy is induced by the use of the store.

In fact, PDI data store contains a set of references to buffers shared by the application. Each reference is identified by a unique name and:

- has dynamic type information associated to make it possible to manipulate the memory buffer or interpret its content;
- supports a read-write mutual exclusion mechanism to ensure that a single writer or readers only (including the simulation code) access the buffer at any time;
- implements reference-counting to automatically free the memory of buffers not referenced anymore; this reference counting system supports a detach operation that duplicates the allocated memory as required to reclaim a non-unique reference.

In addition to storing references, the data store can attach to each identifier –whether associated to a reference or not– a type template and a metadata marker. The metadata marker simply defines whether the reference should be removed from the store when the application code reclaims it (data) or whether its memory should be duplicated (metadata). The type template is similar to a dynamic type descriptor as introduced in Section 4.1.2, except that all the values that define it (number of elements in an array, displacement in memory, etc.) are specified as expressions instead of plain numbers. When inserting an untyped reference into the store, the type template is evaluated, all expression values are resolved and the resulting type is used.

Storing an object in PDI store is cheap as it does not triggers any copy, instead the store holds a reference to the exact same object in memory as that manipulated by the code. Each process contains its own instance of the store, inter-process communications might be offered by plugins, but not by PDI itself. This results in extremely low execution-time and memory overheads, even non-measurable in realistic conditions as demonstrated by the results of the performance evaluation reported in Deliverable D4.1.

The annotation API described in Section 4.1.1 can be understood as simple accessors to this data store. The `PDI_share` function is used to insert a copy of a user reference into the store, while `PDI_reclaim` is used to remove and detach an existing reference. In practice, the API offered by PDI to manipulate the data store goes slightly beyond these two functions/ A function to duplicate a reference from the store to the user code (`PDI_access`) is available as well as one to remove the user reference (`PDI_release`).

Overall, this approach makes it possible to exchange data between very loosely coupled modules. Each module can add or access objects in the store and does not need to know which other module created it or how. It offers an interface for data transfer similar to file-coupling with performance similar to those of passing a function parameter by reference. Similar to file-coupling however, it only handles data transfer and does not offer any solution for control transfer; that is the responsibility of PDI event subsystem.

4.2.2 Event subsystem

While the data store handles data transfer between the application code and plugins, the event subsystem handles control transfer. In PDI, the event subsystem is mostly based on the observer design pattern[5]. Plugins or any other piece of code can register themselves to be notified of many types of change associated to a notification. Notifications are emitted when:

1. the library has been initialized or is about to be finalized,
2. a reference becomes available in the store,
3. a reference is requested from the store for an identifier not currently associated to one,
4. a reference is about to be reclaimed and all other associated references will be nullified,
5. a named event is emitted (for example using `PDI_event`).

For each of these notifications, one can register to be notified unconditionally or one can filter the notifications based on the identifier for which the event occurs.

[5] `https://en.wikipedia.org/wiki/Observer_pattern`

To ensure minimal overhead, PDI event subsystem is synchronous (like a function call). Plugins can implement asynchronous behaviour by relying on dedicated threads or processes for example, but PDI does not implement it itself. This choice has been made because the philosophy of PDI is not to provide any feature on its own, but instead to make the features available in underlying libraries accessible.

While very simple, this event sub-system makes it possible for plugins to implement their behaviour. At initialization, plugins interpret their sub-tree and can register to be notified of any event they are interested in. Typically, this can be the availability of a new buffer (2) or a named event (5) in order to read or write in an application-provided buffer. If the plugin implements any sort of asynchronous behaviour it will typically register on the nullification event (4) for the references it keeps in order to ensure they are not nullified before the asynchronous processing ends. Additionally, plugins can allocate and transfer buffers to the application by registering on the missing reference event (3) and by sharing a buffer at this time.

4.2.3 Expression mechanism

```
/* parsing as a REFERENCE is preferred over OPERATION
   parsing as an OPERATION is preferred over STRING_LITERAL
*/
<EXPRESSION>      ::= <REFERENCE> | <OPERATION> | <STRING_LITERAL>

<STRING_LITERAL> ::= ( <CHAR> | '\' '\' | '\' '$'
                       | <REFERENCE>
                       | '$' '(' <OPERATION> ')'
                     )*

/* The operator descending precedence order is:
   1. *, /, %: multiplication, division and modulo,
   2. +, -: addition and subtraction,
   3. <, >: less than and greater than,
   4. =: equality,
   5. &: logical AND,
   6. |: logical OR.
*/
<OPERATION>       ::= <TERM> ( <OPERATOR> <TERM> )*

<TERM>            ::= ( <INT_LITERAL> | <REFERENCE> | '(' <OPERATION> ')' )

<REFERENCE>       ::= '$' ( <IREFERENCE> | '{' <IREFERENCE> '}' )

<IREFERENCE>      ::= <ID> ( '[' <OPERATION> ']' | '.' <ID> )*

# The terminals
<INT_LITERAL> ~= "(0x)? [0-9]+ ( \.  )"
<ID>          ~= "[a-zA-Z0-9_]*"
<CHAR>        ~= "[^$\\]"
<OPERATOR>    ~= "[|&=<>+\-\*/%]"
```

Listing 12: Specification of the PDI expression grammar

The combination of data transfer offered by the data store and control transfer offered by the event subsystem offers the basis for loose coupling of multiple independent modules. The specification tree builds on this basis and orchestrates the interaction between the multiple modules used in an execution. Each plugin is responsible of interpreting its sub-tree, but this is made much simpler thanks to the expression mechanism of PDI.

An expression is usually constructed by parsing a string according to the grammar presented in Listing 12. An abstract syntax tree for the expression is kept in memory. Whenever the expression is evaluated, all references are resolved in the data store, the operations are executed and the value of the expression is generated. This value can then itself be made available through a PDI reference.

Using this approach, the implementation of data type templates is especially easy. This also makes it possible for plugins to make their behaviour depend on the context in a very simple way. By passing any configuration option they receive to the expression parser and delaying the evaluation of the expression until the last time, plugins can let the user build complex logic with only minimal implementation burden in the plugin itself.

4.3 Conclusion

To summarize, the PDI core library described in this section supports interactions between loosely coupled modules sharing the same memory space. This is mediated by the data store that acts somewhat like a file-system for data transfer and the event subsystem based on the observer pattern for transfer of control. Using this approach and thanks to the flexibility offered by the `$-expression` mechanism, plugins can easily implement new behaviour using and modifying the buffers exposed by the simulation code.

Plugins are available to wrap various general-purpose I/O libraries (HDF5, SIONlib, NetCDF, ...) and specific-purpose I/O libraries (FTI for checkpointing, ...) The interest of PDI is that once the code has been annotated with PDI enough to support data I/O through these plugins, adding in-situ data transformations usually requires no additional modification to the code. The annotation of the ParFlow and SHEMAT SC applications in EoCoE-II make this transition possible, and the specific case of in-situ treatment is a work in progress and high-level priority for the GyselaX SC code (diagnostics support).

The four following sections describe four approaches developed in the EoCoE-II project to support in-situ data transformations in context of PDI plugins. Section 5 focuses on the `User-code` and `Pycall` PDI plugins for process-local in-situ data processing. Section 6 then presents the FlowVR framework and associated `FlowVR` PDI plugin for in-situ and in-transit data analytics. Section 7 presents work to support in-situ data compression for HDF5 and NetCDF accessible through the `Decl'HDF5` and `Decl'NetCDF` PDI plugins. Section 8 presents work to support in-situ vizualization through the SENSEI interface using a dedicated adaptor for the pycall plugin.

5 Process-local in-situ data processing in PDI

Process-local in-situ data processing is supported in PDI by the `User-code` and `Pycall` plugins. They offer very similar features, enabling PDI users to execute code to transform the data shared with PDI, but the `User-code` plugin supports codes written using compiled languages (C, C++, Fortran) while the `Pycall` plugin supports the Python language. In both cases, the language of the code called from PDI does not have to match the language of the application code.

```
double *buffer = malloc(sizeof(double) * 1000 * 1000 * 1000);
for ( int iteration=0; iteration<1000; ++iteration ) {
    PDI_expose("iteration", iteration, PDI_INOUT);
    work_with(buffer);
    PDI_expose("buffer", buffer, PDI_INOUT);
}
```

Listing 13: Example code using PDI to expose a 3D buffer of size 1000^3

For example, in an application exposing a buffer as illustrated in Listing 13, one might want to write the buffer content directly to disk using HDF5 every ten iterations. This can be achieved using the `Decl'HDF5` plugin as simply as illustrated in Listing 14.

```
metadata: { iteration: int }
data: { buffer: { type: array, subtype: double, size: [ 1000, 1000, 1000 ] } }
plugins:
  hdf5:
    file: "data_$(iteration / 10).h5"
    write: buffer
    when: $(iteration % 10 = 0)
```

Listing 14: Example specification tree to write the buffer of Listing 13 to a different HDF5 file every 10 iterations

Here the code uses a strategy known as time sampling. Since writing the data to disk at every iteration might result in files too large for the file-system or writing times that exceed the computation time, the data is only written every N iterations (with $N = 10$ in this example). While perfectly sensible in some cases, this might be a limitation in others.

For example, the main piece of data manipulated by the GyselaX scientific challenge (SC) code (the distribution function) fills a large part of the memory; typically this represents between a quarter and a third of the total memory of the supercomputer partition used for a single distributed 5D array. In that case, the size on disk and the time that would be required to write this data to disk regularly would be prohibitive; its copy to disk is limited to checkpoints. Instead, the community using large-scale gyro-kinetic codes has chosen to work with what they call diagnostics; smaller (0D to 3D) values derived from the main 5D data.

This second approach where data is locally reduced on the processes executing the main simulation code is much more efficient in that case. It is a specific use-case for what is known as process-local in-transit data processing. With no specific care, this approach can however become a maintenance nightmare. By entangling the simulation and post-processing concerns in a single code-base, it becomes difficult to switch to a different approach for post-processing, such as going back to writing the full data and using post hoc post-processing for small debug cases or switching to in-transit processing on dedicated nodes for example. In addition, the requirement to use the same language for the simulation and data processing might be an issue as in the case of GyselaX where the preferred language for the simulation is Fortran while the one for data processing is

python.

5.1 The pycall plugin

The `Pycall` PDI plugin support writing python code directly in the specification tree to transform the data shared with PDI. For example, in order to write the mean value in the array at every iteration instead of writing the full array every N iteration, one might use the specification tree illustrated in Listing 15. In this example, the buffer exposed to PDI is passed to python code as a numpy array and its mean value is computed before it is re-exposed to PDI. The `Decl'HDF5` plugin is used to write the processed value instead of the original one.

```
metadata: { iteration: int }
data: { buffer: { type: array, subtype: double, size: [ 1000, 1000, 1000 ] } }
plugins:
  pycall:
    on_data:
      buffer:
        with: { C_buffer: $buffer }
        exec: | # the pipe character introduces a multi-line string
              buffer_mean = C_buffer.sum() / C_buffer.size()
              pdi.expose('buffer_mean', buffer_mean, pdi.INOUT)
  hdf5:
    file: "data_${iteration}.h5"
    write: buffer_mean
```

Listing 15: Example specification tree to compute the mean value from the buffer of Listing 13 and write it to a different HDF5 file at each iteration

The first level in the sub-tree taken by the plugin supports two keys: `on_data` or `on_event` (only `on_data` is present in Listing 15). Each of these contain a mapping whose keys define the event when to execute the provided code, a named event for `on_event` and a data-share event for `on_data`. In Listing 15, the code is executed when a data called `buffer` is shared by the application. The `with` sub-key specifies the variables to make available to python when executing this code and their value. In Listing 15, a variable called `C_buffer` is passed to python whose value is the content of the `buffer` data as specified by a `$-expression`. The `exec` sub-key is a string containing the python code to execute. In Listing 15, the code is short and written directly in-line, but in a real-world case, this would likely be limited to a minimum and an external python module with functions specified in a side-file would instead be imported and used using the usual `import module` python syntax.

From a technical point of view, this is implemented by instantiating a python interpreter in the process using PDI. The python code is executed synchronously in this interpreter by the thread of execution that triggered the event (typically the user code via `PDI_share` or `PDI_event`).

The advantage of this approach is that one can very easily switch from one strategy to the other with only minimal changes. The code doesn't have to be recompiled at all and only minimal changes are applied to the specification tree between Listing 14 and 15. It also makes it especially easy to interface the code with any library offering a python interface as in the case of SENSEI presented in Section 8.

5.2 The User-code plugin

In some cases, the performance offered by Python/numpy might not be sufficient. One might also want to interface with a library for which no dedicated PDI plugin exists and that does not offer a python interface. In that case, the `User-code` plugin can be used to call compiled code from

the specification tree. The mean computation of the previous example can be rewritten in C as illustrated in Listing 16 and called from the specification tree as illustrated in Listing 17.

```c
void compute_mean() {
    double* C_buffer; PDI_access("C_buffer", (void**)C_buffer, PDI_IN);
    double buffer_mean = 0;
    for ( int i=0; i<(1000*1000*1000); ++i ) buffer_mean += C_buffer[i];
    buffer_mean /= (1000*1000*1000);
    PDI_expose("buffer_mean", &buffer_mean, PDI_OUT);
    PDI_release("C_buffer")
}
```

Listing 16: Example C code to compute the mean value from the buffer of Listing 13

The function illustrated in Listing 16 does not receive any parameter nor returns any value. Instead, it uses the `PDI_access`/`PDI_release` functions couple to access values passed by PDI and uses `PDI_expose` to share data back to PDI. Apart from this specificity the content of the function is typical C code that can call other functions ar libraries as usual. This function can be compiled together with the main application, but that somehow defeats the purpose of requiring no code recompilation. It can also be compiled in a dynamic library that must then preloaded using the `LD_PRELOAD` environment variable at execution.

```yaml
metadata: { iteration: int }
data:
  buffer: { type: array, subtype: double, size: [ 1000, 1000, 1000 ] }
  buffer_mean: double
plugins:
  user_code:
    on_data:
      buffer:
        compute_mean: { C_buffer: $buffer }
  hdf5:
    file: "data_${iteration}.h5"
    write: buffer_mean
```

Listing 17: Example specification tree to compute the mean value using the C function from Listing 16 and write it to a different HDF5 file at each iteration

The specification sub-tree passed to the `User-code` plugin is very similar to that passed to the `pycall` one. At the top level, it supports two keys: `on_data` or `on_event` (only `on_data` is present in Listing 17). Each of these contain a mapping whose keys define the event when to execute the provided code, a named event for `on_event` and a data-share event for `on_data`. In Listing 17, the code is executed when a data called `buffer` is shared by the application. This then contains another mapping where the key is the name of the function symbol to call and the value a mapping similar to that passed in the `with` section of the `pycall` plugin. It specifies the variables to make available to the code when executing it and their value. In Listing 17, a variable called `C_buffer` is passed whose value is the content of the `buffer` data as specified by a `$-expression`.

In this example, C code is used, but since no parameter is passed, the language function calling convention does not matter and any language for which the symbol name associated to a function can be predicted can be used. Fortran or C++ can easily be used that way too.

5.2.1 Next steps

These two plugins offer very similar features and APIs and can be used in a chain of data manipulations where the data exposed by the code has one treatment applied to it before being exposed to PDI again to be transformed multiple times until the last treatment in the chain writes the data to permanent storage. This approach makes it easy not only to insert new treatments in-situ but also to start developing these treatments offline before integrating them in-situ. It is indeed very simple by using PDI to move from a workflow where data is written to disk by the simulation code, then read transformed and written again by the post-processing tool to another one where the data does not have to go through disk just by changing a few lines in the specification tree. This enables application developers to focus on the core application development while supporting efficient in situ data processing without having to embed the complexity in the simulation code.

With these technical foundations now in place, the focus now moves to the use of this approach in production settings. A new engineer has just been recruited starting on January 2021 to implement this approach in GyselaX. The goal is to replace as much of the Fortran data processing code embedded in GyselaX by python code called through the `pycall` plugin. Some performance-critical operations might need to remain in Fortran or be rewritten in C++ and called by the `User-code` plugin. Discussions are ongoing with the other SC codes to determine whether this would be pertinent in their case.

6 In-transit data analytics with FlowVR

6.1 FlowVR library

FlowVR is a library that works in a distributed environment. It creates a network of processes linked to each other to send and receive data, which allows writing complicated simulation applications.

Each process represents a FlowVR module, which has input and output ports and main loop. In each loop data is received from input port and send by output port only once, when the `wait` signal is called. Modules on the same node will transfer data through the shared memory, which means no data is copied. Communication between modules is done by messages, which consist of a payload and stamps. A payload is a data that user wants to transfer and stamp is a metadata that can describe the payload. Messages in input port are stored in a FIFO queue, where module has access only to the first element. By default, all input ports are blocking, that means that module waits if no message is in the queue.

FlowVR network is created by connecting output ports with input ports of modules to work as a one application that is capable of processing data. At runtime all messages sent and delivered are managed by FlowVR daemon, the supervisor process that runs on each node, that manage shared memory and connections with other daemons. Above description represents figure 2.

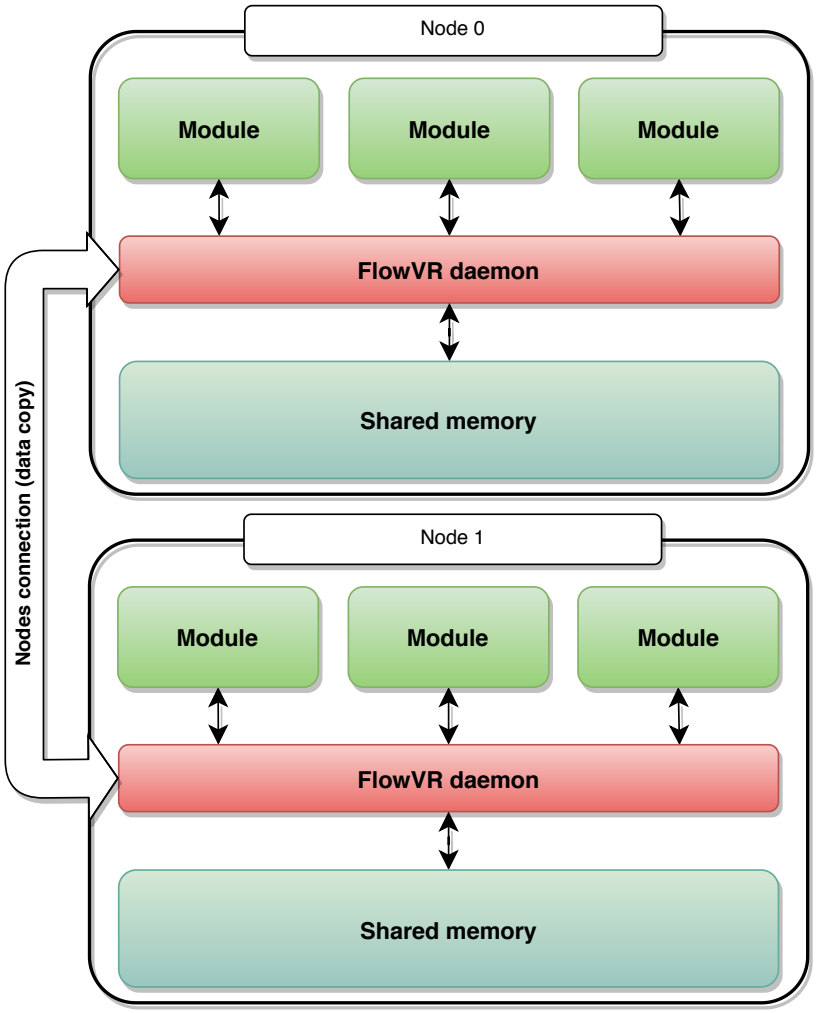

Figure 2: FlowVR modules connection

6.2 PDI flowvr plugin

The Flowvr plugin was made to support in-transit processing. It creates an abstract layer between FlowVR and the user application and provides a full feature set of underlying library, including support for shared memory communication. The PDI library alongside with the plugin is placed between the FlowVR module and daemon, which is shown in figure 3.

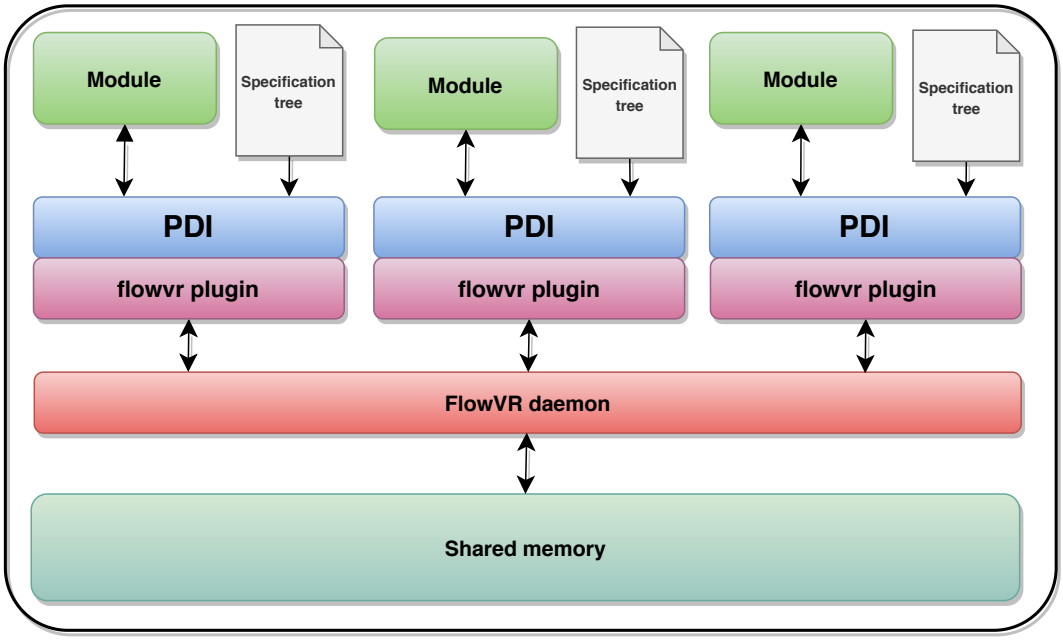

Figure 3: PDI flowvr plugin connection

6.2.1 Specification tree

In the plugins specification tree, `wait` signal, input and output ports are defined. Each time `wait` data is shared with PDI, the FlowVR plugin will call the `wait` signal and write back the status. The port tree consists of the payload and stamps subtree, which declares names of shared data on which payload and stamps will be written to a message in case of the output port and read from a message in case of the input port.

Listing 18 shows a simple example of a specification tree, which defines a module with one input port named `x_port` and an output port named `y_port`. Variables types defined in the data subtree inform of the type of data in message received and sent. This specification tree could be used to implement a module that makes in-transit conversion from float to integer values.

6.2.2 Plugin's features

As mentioned in the introduction, the plugin covers all FlowVR module's feature set, which means that modules created with PDI are indistinguishable from modules written using FlowVR API. Compatibility was proven in four FlowVR example applications, where each FlowVR module can be replaced by the one created with PDI.

The plugin supports calling `wait` signal on data share or when an event in PDI is called. The first one is preferred, because it returns a status value to the user. The status of the module also can be checked at anytime. To abort whole FlowVR application, which means stopping all the modules in the network, the event in PDI can be called. The plugin provides also the possibility to

```yaml
data:
  wait: int
  x: float
  y: int
plugins:
  flowvr:
    wait_on_data: wait
    input_ports:
      x_port:
        data: x
    output_ports:
      y_port:
        data: y
```

Listing 18: flowvr plugin specification tree example

send an abort signal on PDI finalization. This is very useful if the receiver finished the algorithm and the whole application should be stopped.

Messages received or sent can have one payload and multiple stamps. In the port subtree, the user defines data to put to payload and creates stamps, by giving a name and value to use. The stamp value can be either data name, where the data is found, or it can be an expression to evaluate. All cases are presented in example in listing 19.

```yaml
data:
  wait: int
  array_data: {type: array, subtype: int, size: '${array_size}'}
  array_size: int
  array_type: {type: array, subtype: char, size: 4}
flowvr:
  wait_on_data: wait
  output_ports:
    some_port_name:
      data: array_data
      stamps:
        payload_size: '${array_size} * 4'
        payload_type: array_type
```

Listing 19: flowvr plugin payload and stamps example

The FlowVR application can be run in parallel using MPI. In this case, when the module initializes, the MPI rank of process and the number of all processes must be known. These values can be passed from the source code or from the FlowVR's Python configuration file. The plugin supports both cases. The first one requires to create a module after rank and number of processes values are shared with PDI. This leads to initialize the plugin on the event feature, which must be called after both values are shared. An example specification tree can be seen in listing 20. For the second case, the user defines in specification tree data name where to put those values read from FlowVR library. Example specification tree can be seen in listing 21.

The Flowvr plugin provides a Python module which creates FlowVR configuration object by reading the specification tree file, which validates yaml file, eliminates mistakes and typos in modules and ports names and removes the redundancy of module definition. Also it reduces the size of the FlowVR configuration script and allows the user to care only about the connections of the network.

```yaml
data:
  wait: int
  mpi_rank: int
  mpi_size: int
plugins:
  flowvr:
    wait_on_data: wait
    init_on: "init_event"
    parallel:
      set_rank: $mpi_rank
      set_size: $mpi_size
```

Listing 20: flowvr plugin sets MPI rank and size example

```yaml
data:
  wait: int
  mpi_rank: int
  mpi_size: int
plugins:
  flowvr:
    wait_on_data: wait
    parallel:
      get_rank: mpi_rank
      get_size: mpi_size
```

Listing 21: flowvr plugin gets MPI rank and size example

6.3 FlowVR plugin evaluation

Benchmarks were designed to simulate real world application scenarios. The overhead of using PDI in single send and receive operation was tested. Benchmarks were run on the Eagle cluster in PSNC using Intel Xeon E5-2697 v3 processors. They were compiled with gcc 6.2.0 and with the OpenMPI 2.1.2 library. Every test was made in two versions: first one using only FlowVR API calls and second using only PDI API to execute exactly the same operations. Each test was also split to use shared memory and to copy data to shared memory. Comparing the results gives overhead of using the PDI library in all above cases.

Each module makes 10000 iterations, what means that the same amount of messages will be sent and received. Time was measured only for the main loop, because it scales linearly. Each test was run three times and benchmarks results are an arithmetic average of the timing values.

6.3.1 FIFO benchmark

The FIFO benchmark is the simplest application that can be written in FlowVR. It consist of 2 modules, where one is the message sender, and the second is a receiver. In the main loop only input/output operation was made, so the relative overhead result are pessimistic. In case of using shared memory the time shouldn't depend on payload size and in case of data copy time should increase linearly. Payload sizes from 1kB to 256MB were tested.

As can be seen in figure 4, results confirm assumptions that have been made and execution time is not dependent on payload size in the shared memory case. PDI overhead by average is 33.6%, which seems a lot, but this is the worst-case scenario because no other operations are done in the main loop and there is no data copy. The iteration time consists only of FlowVR library exchanging pointers and PDI overhead is preparing the payloads defined in specification tree. Taking that into consideration, 33.6% is not achievable by any other application.

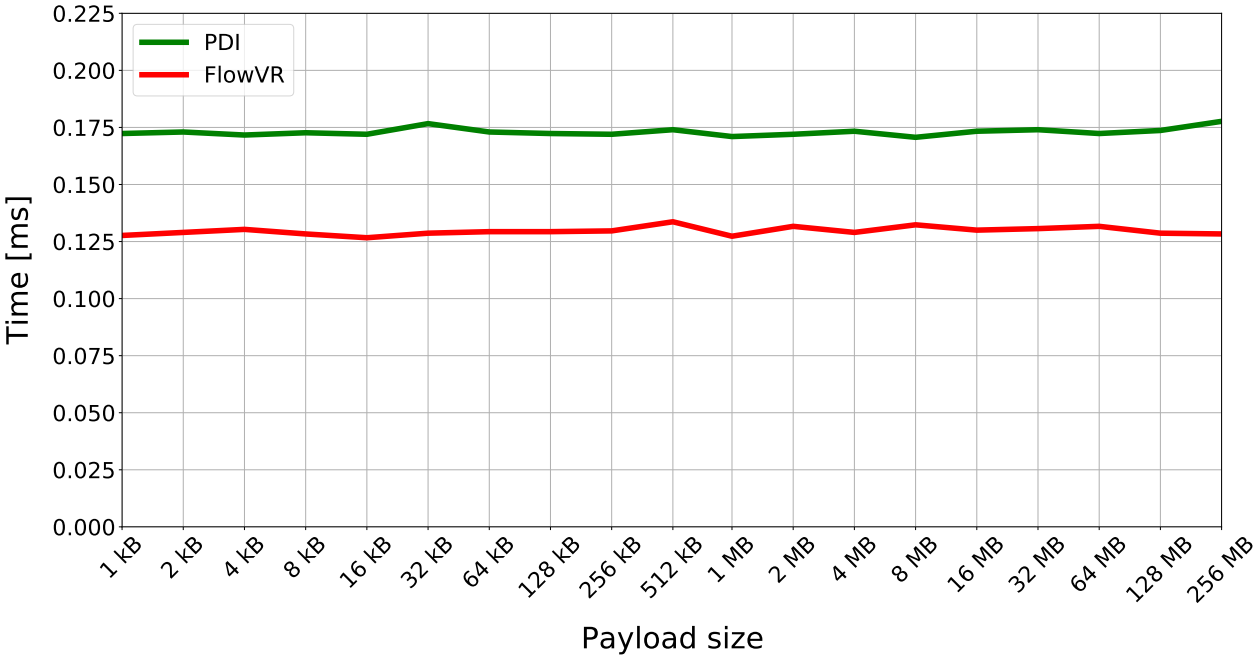

Figure 4: FIFO shared memory benchmark result

In real world applications sometimes data copy to the message is required. That was a case in the next benchmark. Data is copied to output message at the sender and copied from message at receiver side. Copy operation is already expensive enough to reduce PDI overhead to by average 16%, where copy message size of 1kB - 16kB was around 30% (when copy operation cost is close to shared memory), 2MB - 32MB only 2% by average and 64MB - 256MB overhead is a negative value, which can suggest that other factors influenced the reuslt (for example node occupation by other cluster users). Modules that use PDI for big messages are indistinguishable from the one that use FlowVR API.

6.3.2 Greedy benchmark

Connecting modules in the network with other FlowVR components, that are built in the library, allows input module port can take only the newest generated message by the sender. This synchronization method in FlowVR is called Greedy, which can be used for example when simulation generates more frames that display can handle. Thanks to that receiver gets most recent frames. Con of this solution is that the sender is not synchronized in any way. It sends a message immediately when data is ready. This is dangerous, because unread messages can take more memory space than available shared memory on the system. Payload sizes where taken experimentally and they start at 1kB and ends at 8MB. Messages with size 16MB and above caused shared memory overflow.

Similar to FIFO shared memory benchmark, in figure 6 can also be seen, that PDI overhead is by average 22.2%. It is caused by the reasons mentioned in previous benchmark and only confirms the worst scenario cost of using PDI won't be over 33.6%.

Data copy to and from a message is presented in figure 7. In this case PDI has only 11.5% overhead by average. That small value is caused by cost not only of the data copy, but also of the network synchronization. This confirms that the relative cost of using PDI is better when there is more computation time between input/output operations.

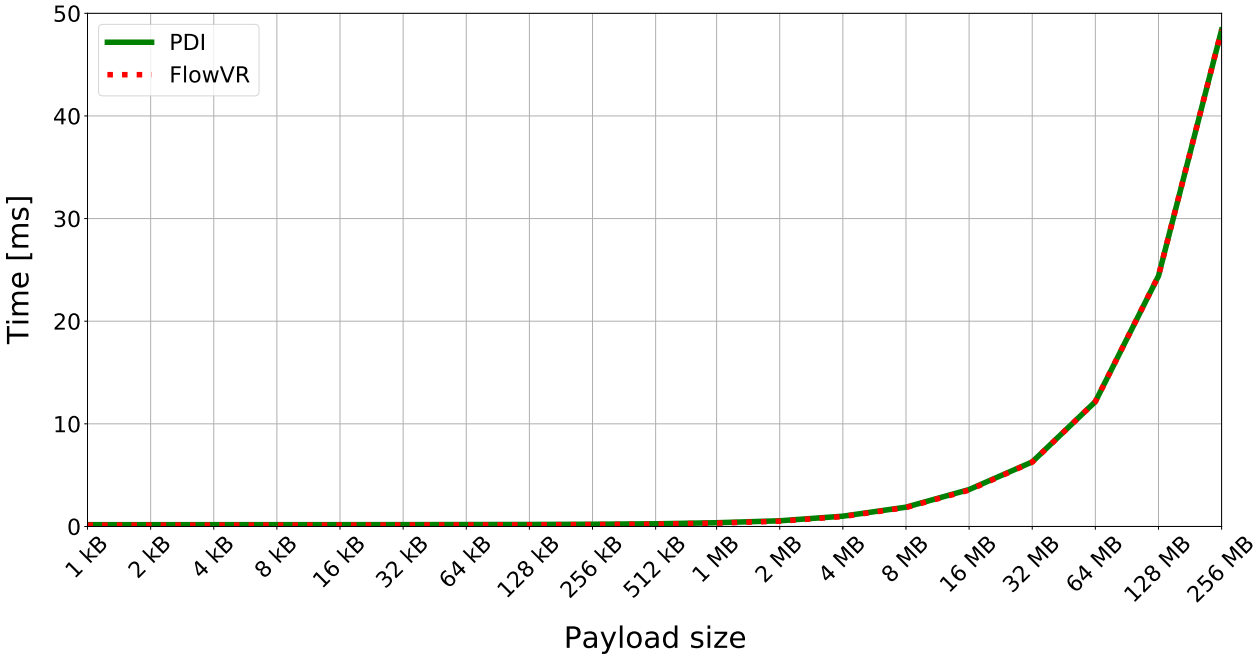

Figure 5: FIFO data copy benchmark result

6.3.3 Gather benchmark - 4 modules

FlowVR modules can be created as MPI processes. Gather benchmark uses that feature to create 4 modules where each creates submatrix, send it to special FlowVR component that merges these submatrices into one big matrix, which is received by one receiving module. The described use case requires modules synchronization and copy of submatrices messages to one big message, which means PDI overhead should be relatively small.

The results of using shared memory and data copy to the shared buffer are very similar. The average PDI overhead is equal 9.9% and 9.6% respectively. For this reason only shared memory is shown in figure 8, where received payload sizes range from 1MB to 16GB.

6.3.4 Gather benchmark - 4kB output message

The last benchmark tests weak scaling of the PDI overhead, starting at 1 MPI processes and ending at 128. Each output module sends a message of size 4kB. Again results of using shared memory and data copy to the shared buffer are very similar, average PDI overhead is equal 5.3% and 7.8% respectively. For this reason only shared memory is shown in figure 9.

6.3.5 Conclusion

The biggest overhead of using PDI is 33.6%, which is not achievable by user application. Adding only data copy operation reduced the relative cost to 11.5% on average. Taking under consideration, that the FlowVR library has low performance impact itself, using it with PDI for a big application will be imperceptible. Over a diverse range of benchmarks it was shown that the PDI overhead is constant on a single interation (send and receive operation), which on the Eagle cluster located in PSNC is equal to 0.05 milliseconds. The total cost of using PDI can be easly calculated by multiplying this time by the number of input/output iterations.

Flowvr plugin hides all FlowVR library API calls, which allows to use it faster and with less knowledge. Another pro of the plugin is that it reduces lines of the source code. Definition of the output port can be reused for the input port of the receiving module, which can be also used to

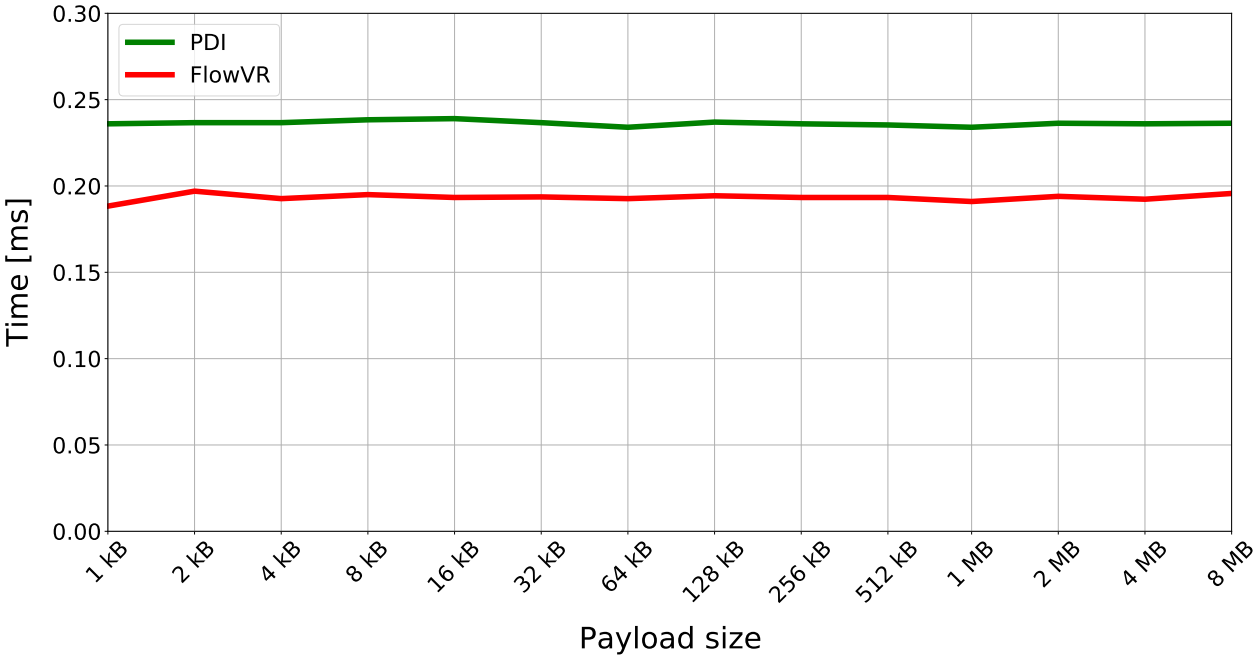

Figure 6: Greedy shared memory benchmark result

create Python script. All above arguments lead to the conclusion that using PDI reduces the risk of making mistakes and creating working applications much faster.

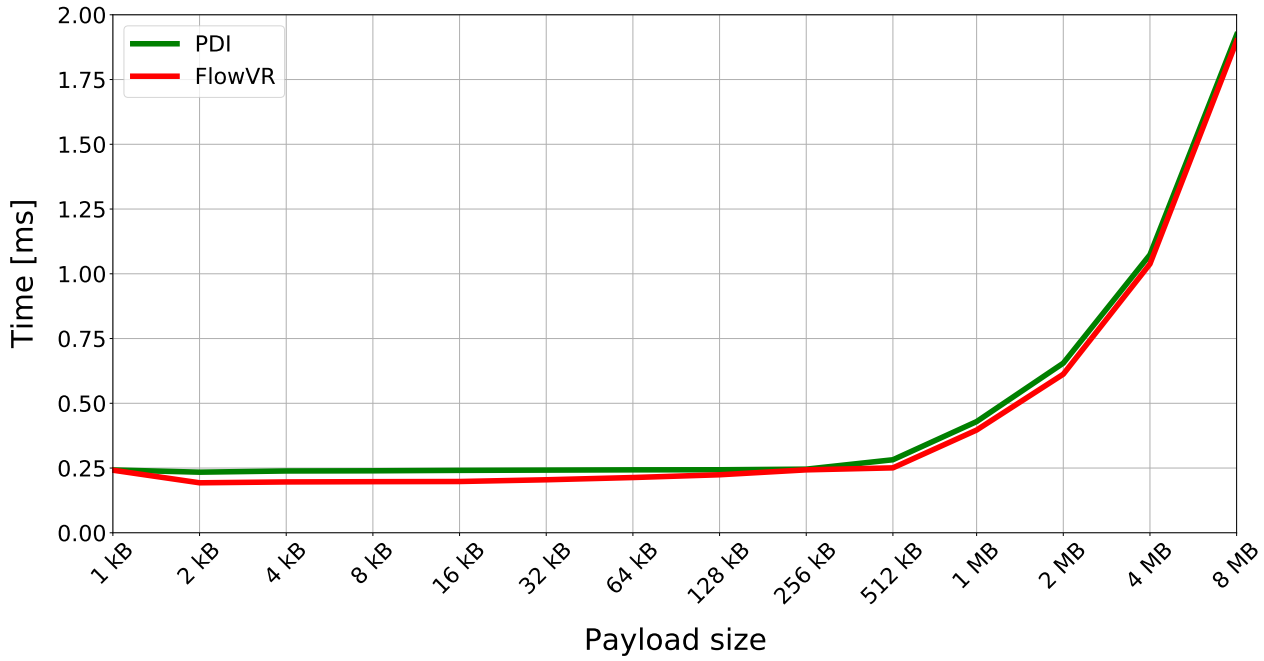

Figure 7: Greedy data copy benchmark result

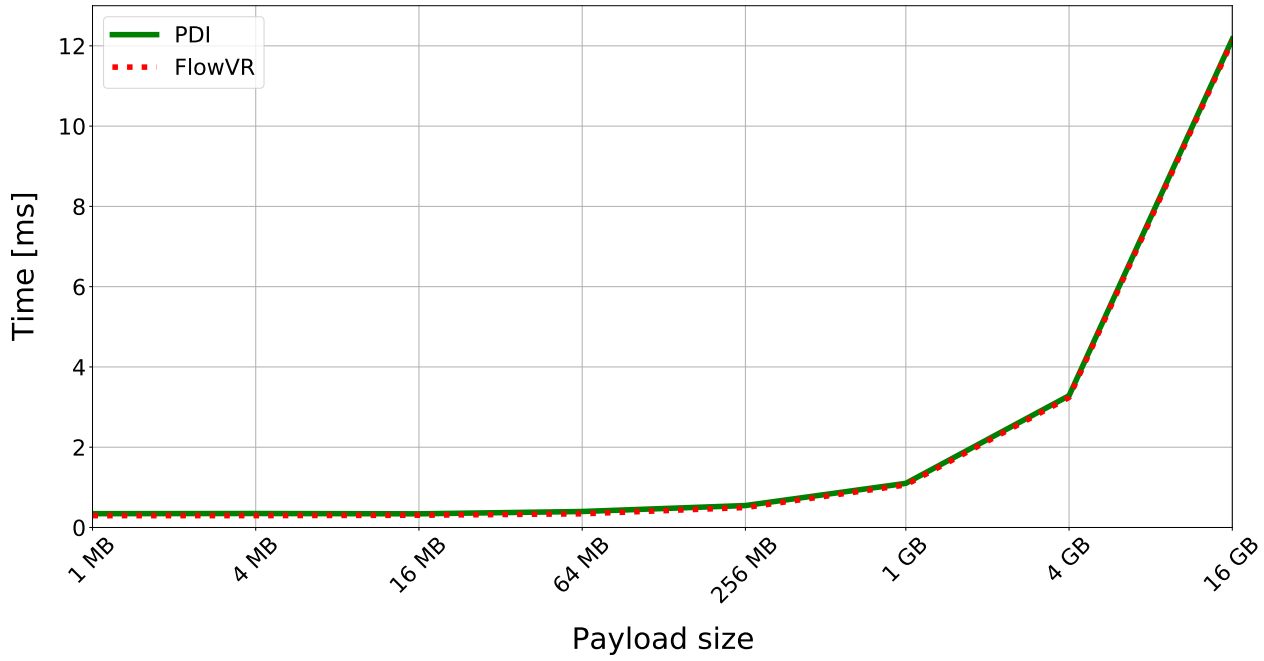

Figure 8: Gather with 4 MPI modules benchmark result

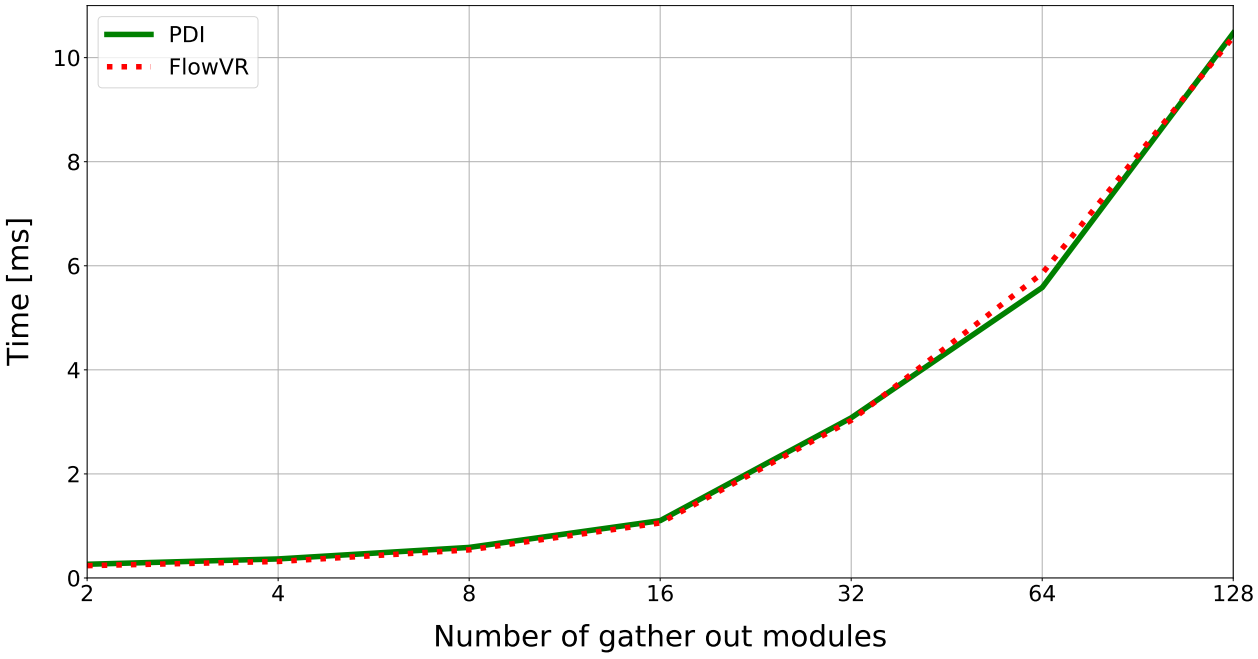

Figure 9: Gather with 4kB output message benchmark result

7 In-situ data compression

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

As explained in Deliverable D4.1, in situ compression compresses the data on-the-fly, which avoids having the total data footprint on the filesystem at any time and can also reduce the involved necessary network transfers towards/from the filesystem. On the other side the compression will increase the necessary computing time, so the approach has to be validated and configured to achieve a significant lower data size while keeping the necessary computing time limited.

Compression itself is typically executed on the global dataset, which makes it more complicated if the data is distributed amongs different tasks, as each task can only compress its local datasets. Due to this reason, the in-situ compression approach of the HDF5 library was selected, which allows to use different compression algorithms while creating a parallel file involving multiple processes. HDF5 supports this mechanic by utilizing the so called chunking capabilities of the library. If chunking is used, the file format switches from a continous byte stream to a set of individual blocks (chunks). Each block itself is a continous stream again. The file access from the application still runs like before, the chunk handling is handeled fully internally as part of the HDF5 library [6]. By combining chunking and compression HDF5 is able to compress each chunk individually. Therefore each process can handle its own chunks and runs a local compression. Of course this local compression does not reach the same quality as a gobal compression, because only the local dataset can be taken into account, but can run significantly faster and in an in-situ manner.

ParFlow, the SC application selected for the compression approach, does not directly support the HDF5 format, but it allows to write data by using the NetCDF4 format, which is also the preferred format by the Water4Energy scientific challenge community for their future ParFlow work. NetCDF4 is directly build on top of HDF5, which allows to utilze all HDF5 capabilities within NetCDF4. The parallel compression capabilities were ported from HDF5 into the NetCDF4 library in release 4.7.4 on March 27, 2020[7].

Beside the compression access, the compression library itself must be prepared and selected. The most easiest library to use here is ZLIB[8] (based on the deflate algorithm), which is typically preinstalled on any major system. It can also be directly linked into HDF5, which allows HDF5 to utilize ZLIB without much coding overhead by the user. This link is typically in place for any HDF5 installation. The deflate compression is losless and performs best for regular data pattern. Because of its simplicity and not introducing any new dependency ZLIB was selected for the compression default to be added to the ParFlow master branch. The `compression_level` can be configured (between 1 and 9), which can increase the compression quality but also the necessary compression runtime overhead.

```
1  nc_def_var_deflate(netCDFID,varID,0,1,compression_level);
```

Listing 22: Enable NetCDF4 deflate compression

A second option directly accessible by HDF5 is SZIP[9]. SZIP is an implementation of the extended-Rice lossless compression algorithm and can sometimes reach slightly better compression rates towards ZLIB and a faster compression time. SZIP can also be linked to HDF5. However this is less often the default compared to ZLIB for the pre-installed HDF5 installations. This is also related to the licensing terms of SZIP, which is only free for non-commercial utilization.

[6] https://support.hdfgroup.org/HDF5/doc/Advanced/Chunking/index.html
[7] https://github.com/Unidata/netcdf-c/blob/master/RELEASE_NOTES.md#474---march-27-2020
[8] https://zlib.net/
[9] https://support.hdfgroup.org/doc_resource/SZIP/

```
int parms[] = {H5_SZIP_EC_OPTION_MASK,16};
nc_def_var_filter(netCDFID,varID,H5Z_FILTER_SZIP, 2, parms);
```

Listing 23: Enable NetCDF4 SZIP compression

The third and last library to be tested was ZFP[10]. In contrast to the two other libraries, ZFP also allows non-lossless compression especially relevant for floating point datasets, which can reduce the datasize even more. HDF5 supports ZFP by using a so called filter[11], which can be applied to the individual data sets. Filter are open a HDF5-API, which can be utilized to couple external libraries during runtime underneath of the individual HDF5 calls. ZFP also supports a loosless mode (called reversible), a fixed precision mode (to control the relative error), a fixed accurancy mode (to control the absolute error) and a fixed rate method which limits the overall data size after compression. For the benchmark tests the reversible and fixed precision mode were implemented as these a typically most relevant in the context of the given simulation datasets.

```
unsigned int cd_values[10];
size_t cd_nelmts = 10;
H5Pset_zfp_precision_cdata(20,cd_nelmts, cd_values);
//H5Pset_zfp_reversible_cdata(cd_nelmts, cd_values);
nc_def_var_filter(netCDFID,varID,H5Z_FILTER_ZFP,cd_nelmts,cd_values);
```

Listing 24: Enable NetCDF4 ZFP compression

7.1 Implementation

The compression algorithm utilization was directly integrated into the main ParFlow code, which provides a NetCDF-4 based backend for writing the main output data to disk, because these datasets contain the main data footprint. For the master branch of ParFlow only the ZLIB based approach was directly included due to its lack of additional new dependencies. In addition to enable the compression for each individual multi dimensional NetCDF variable two new configuration options `NetCDF.Compression` and `NetCDF.CompressionLevel` were added to enable or disable the compression routines and to control the compression level of ZLIB.

Because the compression in performed in parallel utilizing the HDF5 chunking mechanic, chunking must be enabled for obvious reasons. However this is automatically done implicitly by NetCDF when using parallel compression on a non-chunked dataset. ParFlow also allows manual chunking, which is still possible like before. Another necessary aspect to allow parallel compression is the utilization of collective read/write operations for all compressed datasets. This was already the case for ParFlow so no additional steps were necessary.

The changes were accepted for the ParFlow master branch an integrated into the main ParFlow repository[12].

7.2 Compression runtime tests

To test the different compression approaches a scalable ParFlow input setup for a simple constant geometry was utilized. This approach allowed an easy weak scaling by increasing the data size written to disk for a increasing number of utilized nodes. All tests were performed on the

[10] https://github.com/LLNL/H5Z-ZFP
[11] https://support.hdfgroup.org/services/filters.html
[12] https://github.com/parflow/parflow/pull/267

JUWELS cluster system at JSC[13] and its main IBM Spectrum Scale (GPFS) based SCRATCH filesystem. To avoid spending too much benchmark runtime the largest parametrisation on 3072 tasks was only executed for the compressed setup. All benchmarks were executed within the JUBE bechnmarking environment.

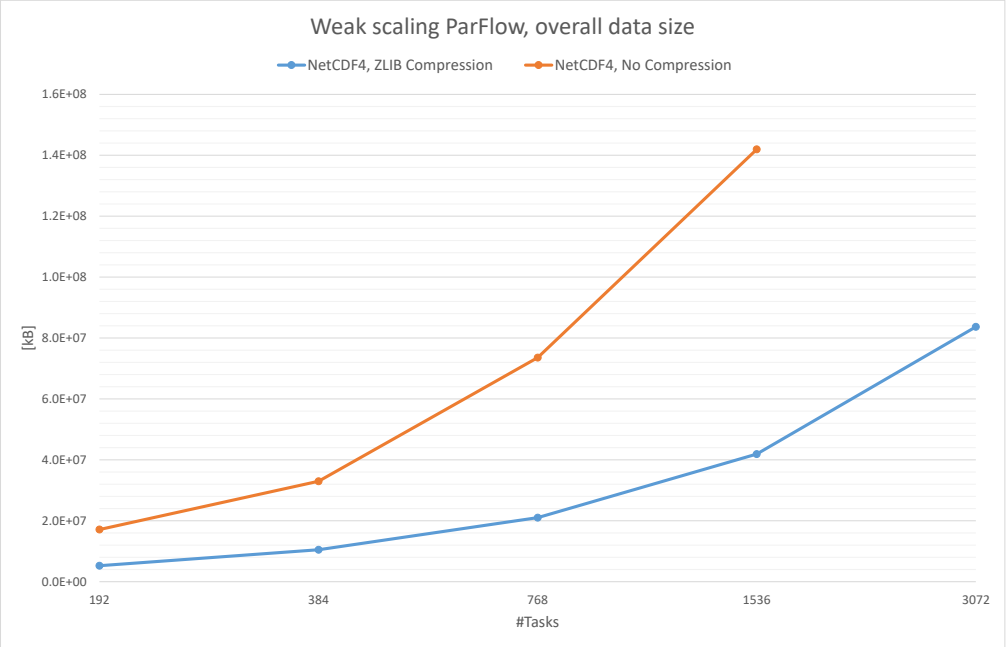

Figure 10: Overall datasize stored on disk storage after running a weak scaling ParFlow ZLIB compression benchmark on JUWELS.

In Figure 10 and Figure 11 ZLIB compression (using the lowest compression level) was compared towards non compression. The overall datasize for this lossless compression could be lowered by a factor of three. This compression factor highly depends on the data layout, therefore it can perform better within such a regular benchmark pattern compared to a real geometry. Beside the the reduction of the datasize the runtime could also be lowerd by a factor of two to three. Of course for the benchmark setup the portion of I/O as part of the overall runtime is significantly larger compared to a real setup, therefore this benefit only affects the I/O portion of the code. However this benchmark already proves the aspect, that moving compression into the application itself (by slightly increasing the computational runtime) helps to reduce the impcat of a slower filesystem on the other side. The benefit gets even larger on larger computing scales when the filesystem is more saturated. Similar to the compression quality the individual sweet spot between additional compression computational runtime and saved I/O runtime depends on the application setup and input dataset.

In a second benchmark setup, the different compression approaches were compared to each other using the same weak scaling approach.

Figure 12 shows that the compression quality is quite similar for the given benchmark setup for all lossless compressions (with SZIP beeing the worst). Only the non-lossless fixed precision compression of ZFP allows even better compression, but of course produceses a (controlable) error towards the original simulated datasets. The error for ZFP is controlled by fixing the number of used bit planes within the compression algorithm. For the benchmark setup 18 bit planes were used, which resulted in a relative error smaller than $1e^{-3}$. Increasing the number of used bit planes allows to lower the relative error even further but increases the data size again on the other side.

[13] `https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/Configuration/Configuration_node.html`

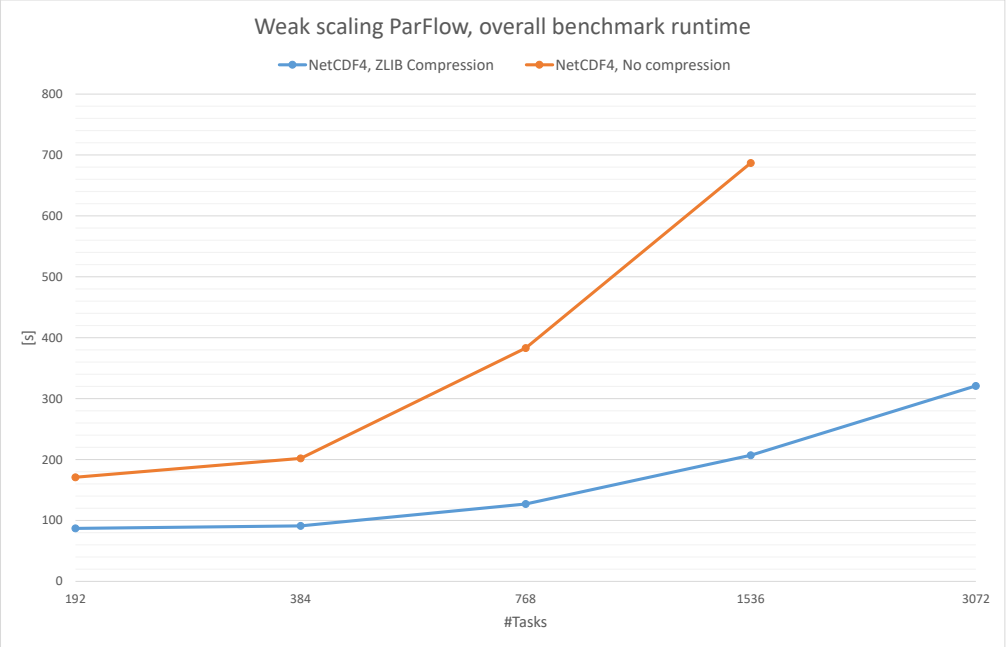

Figure 11: Overall ParFlow ZLIB compression benchmark runtime for a weak scaling run on JUWELS.

Figure 13 shows that the compression time is also rather similar. Even ZFP in fixed precision mode does not run significantly faster (due the the algorithm complexity) but starts to outperform ZLIB on larger scales due to the huge data reduction which compensate the additional computational costs.

Based on these benchmark results ZLIB already provides a good compression rate together with a small computational overhead. As it also needs the smallest amount of additional dependecies it is a good default value to be used within ParFlow. ZFP non-lossless compression can even outperform ZLIB on larger scales and might be relevant for output files which can be limited to a specific relative error rate.

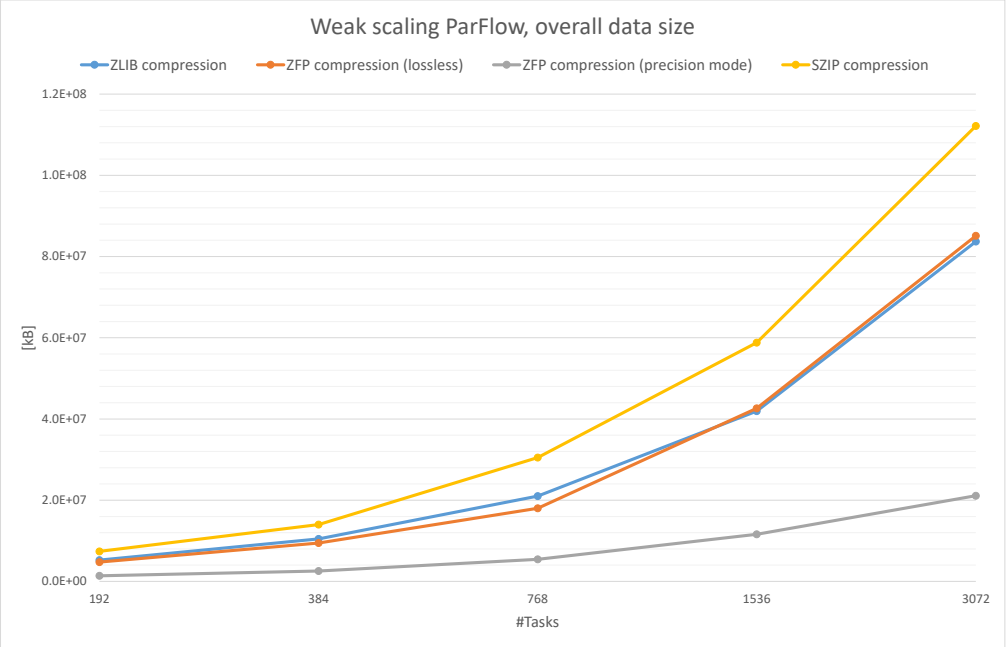

Figure 12: Overall datasize stored on disk storage after running a weak scaling ParFlow compression comparison benchmark on JUWELS.

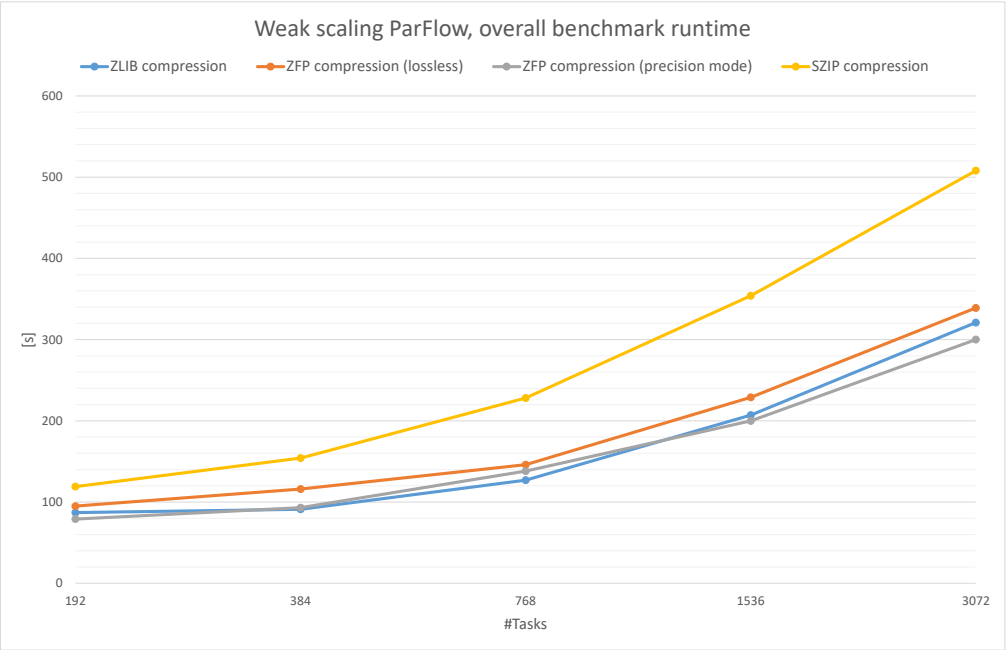

Figure 13: Overall ParFlow compression comparison benchmark runtime for a weak scaling run on JUWELS.

8 In-situ visualization

In situ visualization is a tool that allows us to visualize simulation data directly without having to store the data. Since the hard disk performance increases slower than the computational power, in-situ visualization can generate images with high temporal and spatial resolution. These then require significantly less storage space than storing the entire dataset. This will not replace saving the entire dataset in most cases, but will allow insight into the intermediate steps between saves.

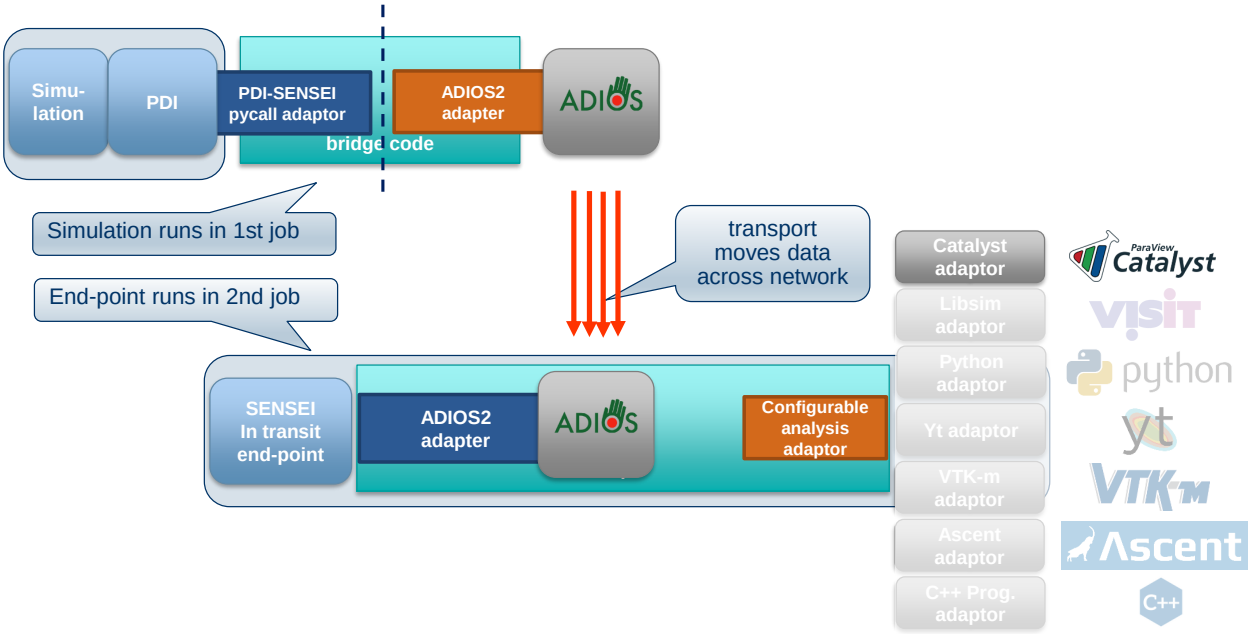

Figure 14: Overview off components used in in-situ visualization with PDI

In figure 14 is an overview of the components we want to use for in-situ visualization from PDI. The simulation transfers the data to be visualized to SENSEI with the help of PDI. This is specified in a yaml file using the pycall plugin of PDI. After all data of a time step is collected, a data transport is started using an event triggered by the simulation. This transport moves the data, with the help of ADIOS2, to a second job where the data is received for visualization. There is a clear separation between the resources used by the simulation and the visualization, so this is a special form of in-situ visualization, also called in-transit. This prevents problems in the visualization from affecting the simulation, and also allows users to assign their own resources to the visualization, which can be adapted to the needs of the visualization. After this transport, an analysis adapter can be chosen for SENSEI to perform the analysis and/or visualization. This can be configured with some choice, but the focus here is on the interface offered by ParaView called Catalyst.

8.1 Sensei

SENSEI[14] is a project that allows a high flexibility to interact with different in-situ software. This allows a very easy interchangeability of the different components used. For example, the ADIOS2 adapter used for data transport is also just an analysis adapter from SENSEI's point of view. This reduces the dependency on other software and also allows an easy exchange for another form of data transport in the future. Likewise, with SENSEI it is possible to use the

[14] https://sensei-insitu.org/

in-situ visualization directly without a data transport, without the need for major changes. Even if this bears risks, because the visualization and the simulation are then very strongly coupled and problems of the visualization could then lead to a crash of the simulation.

At the end-point (see figure 14), an ADIOS adapter is used again, which receives the data and passes it back to SENSEI, in principle like a simulation. By the separation into two different jobs it is also possible to run a slimmed down variant of SENSEI (on the simulation side), because it only needs to know ADIOS2 as an analysis to be able to use the configuration presented here. Only the SENSEI running in the end-point has dependencies to the different visualization and analysis programs. But even here, one is not fixed, since one can use new and additional programs that are integrated in SENSEI here by a simple configuration change.

8.2 Adios2

ADIOS2[15] is a framework that allows to transform and transport self-describing data. The focus is on data streaming, using different transport routes. ADIOS2 allows to specify the data transport via so called engines. The engine we will use here is called SST, for Sustainable Staging Transport. Since it was developed for HPC environments, it can use RDMA networks for communication, but also offers the possibility to be used over standard sockets. It also offers the possibility to have a different number of receivers and senders. This allows us to use in our end-point more or less processes for visualization than the simulation uses. ADIOS uses the description of the data to distribute it among the receivers.

Even if the ADIOS2 engine used communicates using the network, it needs a common data system between sender and receiver, since it creates a small configuration file itself to exchange the connection information. However, this has the advantage that the configuration of the data exchange is done automatically, and the simulation can start even if the end-point has not yet started and the connection information is exchanged later.

8.3 Catalyst

Catalyst[16] is the in-situ interface of ParaView, a large scientific visualization software. It allows a user to specify one or more desired visualizations before starting the simulation. In addition, there is also the possibility to send the data to ParaView using Catalyst to enable a live interactive visualization of the simulation data.

Catalyst expects at least one script containing instructions for data processing. In these scripts you can use all the visualizations that are available in ParaView, except that these visualizations are executed during simulation runtime. Additionally there is the possibility to save prepared data and images of the visualization. Furthermore, if you have different scripts that have different visualization foci, you can give Catalyst multiple scripts and they will all be executed. This allows to separate different visualizations into different files and still use them together.

ParaView itself allows you to create these scripts automatically to use a visualization you created interactively in the user interface for your next simulation with similar data. Also the interactive live in-situ visualization can be activated in one of these scripts. If you want to use an interactive live visualization, you can start a third job for this, on which the interactive visualization is then carried out. Since you can use the preprocessed data of the catalyst script in the interactive visualization, you may not need much power for the interactive visualization. But if you want to work with all the data of a large simulation, you will need another parallel job for the duration of the interactive visualization.

[15] `https://adios2.readthedocs.io/en/latest/`
[16] `https://www.paraview.org/in-situ/`

8.4 Status of work

Currently there is a working version of this setup for an example test code. This proof-of-concept code runs in parallel and allows to test the different resource allocation to the simulation and visualization possible with ADIOS2. With this it is possible to use the complete connection from a simulation to an interactive in-situ visualization using PDI, Pycall, SENSEI and a data transport using ADIOS2. The next work will focus on a simpler usage and clear interfaces that can then be used by different scientific challange applications.