ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

PSI

In Situ Analysis and Steering for High-Performance Particle Accelerator Simulations

Master Thesis

in Computational Science and Engineering

Department of Mathematics

ETH Zurich

written by

Severin A. T. Klapproth

supervised by

Dr. Andreas Adelmann (ETH)

scientific advisers

Ryan Ammann
Dr. J. H. Göbbert
Dr. S. Muralikrishnan

July 27, 2026

Abstract

As high-performance computing resources scale, traditional post-processing workflows face a significant bottleneck: the inability to write massive datasets to disk efficiently. Consequently, there is growing interest in in situ frameworks that permit data analysis and visualization while a simulation is ongoing and data is still resident in memory, bypassing disk I/O entirely. When coupled with computational steering, in situ processing allows scientists to interactively monitor the simulation and adjust parameters in real time. This thesis focuses on the IPPL library (Independent Parallel Particle Layer)[53], the underlying framework for the OPALX (Object Oriented Particle Accelerator Library for Exascale)[61] particle accelerator code. Drawing inspiration from experimental facilities, where operators dynamically adjust fields and potentials based on live diagnostics, we aim to bring this level of interactivity to Computational Particle Accelerator Physics. Leveraging ParaView Catalyst 2 [12], we developed an easy to use in situ visualization and computational steering interface for the IPPL framework. This extension allows users to perform real-time image extraction and interactive analysis on running simulations. Furthermore, it enables dynamic control of simulation parameters directly through the standard ParaView GUI or via a custom web dashboard-prototype that we created with Trame [32]. By instrumenting IPPL – and by extension, OPALX – with versatile in situ visualization and parameter steering capabilities, this work effectively pushes these simulation frameworks closer to the experimental setups they emulate, to further bridge the gap for a possible "digital twin" accelerator environment.

Contents

List of Figures

List of Tables

List of Listings

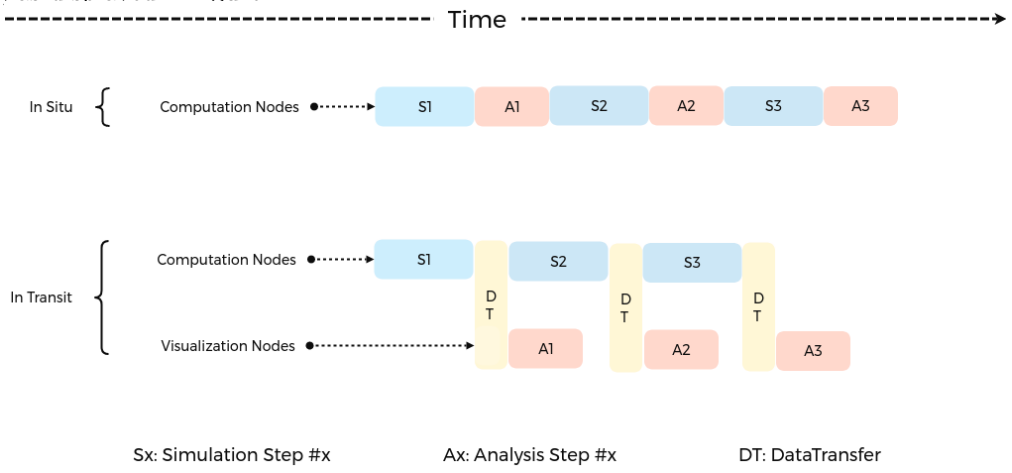
Chapter 1

Introduction

1.1 In Situ Status Quo

In Situ versus In Transit

For a long time the standard paradigm for analyzing high-performance scientific simulations has relied on a post hoc workflow, in which the simulation state is periodically written to persistent storage to only be analyzed after the run is complete. While this decoupling allows for flexible exploration of the saved data, nowadays it faces a critical scalability challenge known as the I/O bottleneck [8]. The gap between computational performance and storage bandwidth is widening, making it prohibitively expensive to save high-fidelity data at the regular temporal resolution required for capturing transient physical phenomena [21]. To circumvent this disparity, in situ and in transit visualization have established themselves as necessary alternatives. While the in situ paradigm tightly couples simulation and data processing, by sharing the simulation's memory and compute resources, for the in transit visualization, data is asynchronously offloaded over the interconnect to dedicated visualization nodes to minimize impact on the simulation's execution time, as displayed in figure 1.1.

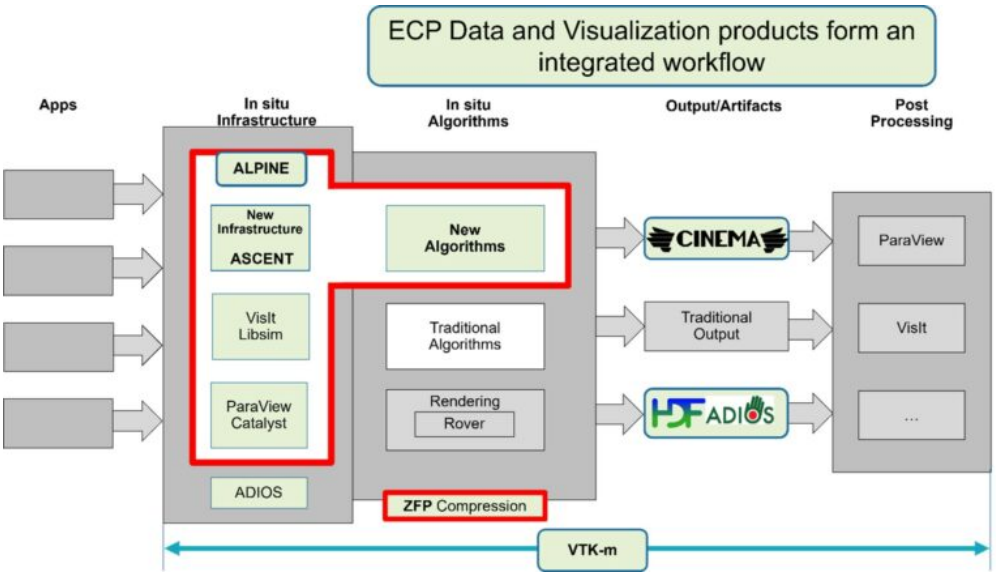


Source: kitware.com/catalyst-adios2-a-new-catalyst2-implementation-for-in-transit-analysis , accessed 03.01.2026

Figure 1.1: *Execution pipeline for in situ and in transit visualization paradigms.*

Algorithms and Infrastructure for In Situ Visualization and Analysis

While early implementations of these workflows were often developed ad-hoc and were difficult to maintain [14], the field has recently matured towards standardized, easy-to-use frameworks (eg. VisIt Libsim (2011) [7, 6, 76], ParaView Catalyst (2015) [4, 12, 20, 55, 56], Ascent (2022) [22, 43, 42]). These modern libraries decouple the visualization logic from the simulation code, providing scientists with interfaces that facilitat easy access to in-memory analysis methods without the need for exhaustive custom instrumentation and coding knowledge. A major contributor for this push towards modern visualization for exascale was the United States Department of Energy ECP's (Exascale Computing Project) [50] ALPINE (Algorithms and Infrastructure for In Situ Visualization and Analysis) project [35]. Running from 2016 till 2024 it created or contributed to all of previously named Visualization frameworks [41]. ALPINE focused on a layered architecture centered on Conduit and it's Blueprint protocol [24, 59]. Conduit provides a standardized data interface that allows simulations to describe their in-memory state using simple, hierarchical descriptions without being tightly bound to specific visualization libraries. This abstraction effectively shields the simulation code from the visualization backend. and facilitates zero-copy access for tightly coupled in situ analysis. For in transit scenarios requiring data movement, architectures can utilize ADIOS (The Adaptable Input/Output System [23, 19]) to asynchronously transport data over the interconnect to dedicated staging nodes. By simply publishing data according to the Conduit Blueprint convention, the simulation enables downstream frameworks, such as Ascent or ParaView Catalyst, to automatically map that data into high-performance structures. These frameworks then leverage VTK (The Visualization Toolkit) [5] for robust processing pipelines and Viskores (formerly VTK-m) [13, 29, 77] for portable hardware-accelerated algorithms on GPUs and scalable distributed rendering.



Source: exascaleproject.org/highlight/alpine-zfp-addresses-analysis-visualization-and-data-reduction-needs-for-exascale-science-applications , accessed 03.01.2026

Figure 1.2: *The integrated workflow formed with ECP Data and Visualization products. Red: Products directly supported by ALPINE.*

1.2 Related Work

In Situ Visualization for IPPL

While in situ visualization for IPPL [53] has been explored in unpublished works and preliminary studies, these efforts have primarily focused on feasibility demonstrations rather than generalized framework integration. Most prior attempts involved instrumenting specific simulation setups, such as the PenningTrap mini-application [26] [26], rather than the core IPPL library itself. Notably, ATML's (Algorithms, Tools and Methods Labs) Visualization & Interactive HPC group at the Jülich Supercomputing Center developed a ParaView Catalyst-based setup for the PenningTrap mini-app involving both visualization and steering [52, 54]. Similarly, a proof-of-concept implementation utilizing Ascent and Trame was developed at the Argonne National Laboratory [36, 37, 60]. However, IPPL is designed as a versatile framework enabling scientists to implement a multitude of particle-mesh simulations. A robust in situ solution for IPPL must preserve the framework's flexibility, allowing scientists to extend any IPPL-based code with visualization and steering capabilities with minimal effort and without requiring deep knowledge of the underlying visualization API.

Adoption of Modern In Situ Frameworks

The adoption of in situ visualization has been particularly strong in the Computational Fluid Dynamics (CFD) community, initially relying on legacy frameworks such as ParaView Catalyst 1.x and VisIt Libsim [17, 21, 28]. However the last couple years the landscape has slowly shifted toward the modern, Conduit-based ecosystems of ParaView Catalyst 2.0 and Ascent. The capabilities of the new Catalyst 2.0 workflow were first demonstrated in 2021 through the instrumentation of the LULESH proxy application [20, 9]. This was quickly followed by the development of AdiosCatalyst [27, 34], which additionally enables pure in transit and hybrid in situ/in transit workflows. Simultaneously, the Ascent framework after first demonstrating it's initial exascale capabilities in 2022 [22], has seen further development on expanding it's ecosystem, particularly regarding user interface options and steering capabilities [30, 31, 44], alongside case studies demonstrating its efficacy in real-world scenarios [37, 36]. While formal literature outside of the core developer teams remains sparse, evidence from source code repositories, technical documentation, and community forums indicates that several state-of-the-art simulation codes are actively integrating these newer frameworks. Prominent examples include the integration of Ascent into NekRS [33], Catalyst 2.0 into Code Saturne [75], and the adoption of both frameworks within the WarpX code [78].

Chapter 2

Software Stack

2.1 IPPL and OPALX

"The IPPL (Independent Parallel Particle Layer) library provides performance portable and dimension independent building blocks for scientific simulations requiring particle-mesh methods, with Eulerian (mesh-based) and Lagrangian (particle-based) approaches. IPPL makes use of Kokkos, HeFFTe[51], and MPI (Message Passing Interface)[2] to deliver a portable, massively parallel toolkit for particle-mesh methods." - *IPPL developer team, github.com/IPPL-framework/ippl, accessed at 03.01.2026* [26, 53]. IPPL relies heavily on Kokkos [25] for it's performance portability. Both it's main data classes – fields and attributes of particle bunches – rely on Kokkos for data allocation and manipulation. IPPL itself serves as the underlying framework for OPAL.

"OPALX (Object Oriented Parallel Accelerator Library for Exascale) is a parallel open source tool for charged particle optics in linear accelerators and rings, including 3D space charge. OPAL is built from the ground up as a parallel application exemplifying the fact that high performance computing is the third leg of science, complementing theory and experiment." - *A.Adelmann, OPAL a Versatile Tool for Charged Particle Accelerator Simulations, (p. 1), accessed 03.01.2026* [16, 61]. OPAL being built upon IPPL 2, will now be followed by OPALX currently in development in the push towards an exascale capable Accelerator Simulation framework. To further improve OPAL's usability and to bring Simulation closer to the experiments which we try to mirror, it was decided OPALX should support in situ visualization, live diagnostics and bidirectional steering via. the use of a modern in situ analysis framework. It is the goal of this thesis to instrument IPPL – and by extension OPALX – with these in situ capabilities.

2.2 Kokkos

"Kokkos Core implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms. For that purpose it provides abstractions for both parallel execution of code and data management. Kokkos is designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. It currently can use CUDA, HIP, SYCL, HPX, OpenMP and C++ threads as backend programming models with several other backends development." - *direct quotation; Kokkos developer team, github.com/kokkos/kokkos, accessed at 03.01.2026* [10, 11, 25, 57, 58].

2.3 Conduit

Conduit is an open-source project developed by Lawrence Livermore National Laboratory (LLNL). Its API provides a streamlined method for describing hierarchical scientific data structures, enabling the sharing of data between different scientific libraries and tools by clearly defining the structure and memory location of the data. The **Conduit Mesh Blueprint** [24, 59] establishes a set of conventions for the hierarchical description of scientific data. The term "Mesh" is somewhat distinct in this context, as the Blueprint also accommodates data not traditionally considered mesh-based, such as particle data. Since its introduction, the Blueprint has been incorporated into major tools such as Ascent and Catalyst 2.0. For the purposes of this work, two distinct Blueprint configurations are required to handle the targeted data structures: 1. A **uniform Cartesian mesh** for field data. 2. An **explicit Cartesian coordinate mesh** for particle data.

```
<mesh name>          # <***> marks user defined labels that can vary
  coordsets:
    <coordset_name>:
      type: "uniform"
      dims:
        i: <mesh nodes x dimension>
        j: <mesh nodes y dimension>
        k: <mesh nodes z dimension>
      origin:
        x: <mesh origin for x dimension>
        y: <mesh origin for y dimension>
        z: <mesh origin for z dimension>
      spacing:
        dx: <uniform mesh spacing for x dimension>
        dy: <uniform mesh spacing for y dimension>
        dz: <uniform mesh spacing for z dimension>
  topologies:
    <topology_name>:
      type: "uniform"
      coordset: "<coordset_name>"   # needs to refer to a defined coordset label of this mesh
  fields:
    <field_name_1>:
      association: "element"   # "vertex" for mesh-node data or "element" for mesh-cell data
      topology: "<topology_name>"
      values: [field values for cells or nodes in ??RowMajor/ColMajor]
    <field_name_2>: ...
```

Listing 2.1: *Conduit Blueprint example for a regular uniform cartesian mesh*

```
<mesh_name>:
  coordsets:
    <coordset_name>:
      type: "explicit"
      values:
        x: [<list of n explicit x coordinates>]
        y: [<list of n explicit y coordinates>]
        z: [<list of n explicit z coordinates>]
  topologies:
    <topology_name>:
      type: "unstructured"
      coordset: "<coordset_name>"
      elements:
        shape: "points"
        connectivity: [ <array with ascending integer values from 0 to n-1 >]
  fields:
    <field_name_1>:
      association: "vertex"
      topology: "<topology_name>"
      values:
        x: [<list of n x component field values>]
        y: [<list of n y component field values>]
        z: [<list of n z component field values>]
    <field_name_2>:
```

Listing 2.2: *Conduit Blueprint example for an explicit unstructured cartesian mesh*

2.4 Catalyst 2

In the revised version of Kitware's in situ framework, the Catalyst API [45] has been decoupled from the visualization pipeline backend, adhering to the layered architecture described previously [20]. This decoupling enables simulation developers to implement custom backends or extend the base Catalyst implementation, offering a streamlined approach for switching between different backends. Currently, three Catalyst API implementations are supported. Two are developed by Kitware: ParaView Catalyst [56] for in situ workflows and ADIOS Catalyst [40] for in transit workflows. The third is developed by LLNL to leverage an Ascent-based backend. All Catalyst API implementations support five core function calls, each capable of receiving or returning data via a Conduit Node (a generic container able to store data and memory locations in a hierarchical structure).

```
// Returns information about Catalyst implementation currently being used.
enum catalyst_status catalyst_about(conduit_node* params);

// Receives data for proper initializsation of the Catalyst API
enum catalyst_status catalyst_initialize(const conduit_node* params);

//Receives scientific data via params and triggers in situ analysis.
enum catalyst_status catalyst_execute(const conduit_node* params);

// Returns data from the finished in situ analysis
enum catalyst_status catalyst_results(conduit_node* params);

// Finalizes Catalyst, usually no need to pass/receive any data
enum catalyst_status catalyst_finalize(const conduit_node* params);
```

Listing 2.3: *The five functions defined by the Catalyst API*

2.5 ParaView

"ParaView is an open-source, multi-platform scientific data analysis and visualization tool that enables analysis and visualization of extremely large datasets. ParaView is both a general purpose, end-user application with a distributed architecture that can be seamlessly leveraged by your desktop or other remote parallel computing resources and an extensible framework with a collection of tools and libraries for various applications including scripting (using Python), web visualization (through trame and ParaViewWeb), or in situ analysis (with Catalyst)."
- *direct quotation; ParaView developer team, https://docs.paraview.org/en/latest/UsersGuide, (ch. Introduction), accessed 03.01.2026* [4, 55, 68]. For this thesis, four primary components of the ParaView ecosystem are utilized to enable the in situ workflows: the graphical user interface (GUI), the Python scripting environment (`pvpython`), the server executable (`pvserver`) and the in situ analysis framework (ParaView Catalyst).

The Graphical User Interface (GUI)

The ParaView application provides a graphical interface (via. the paraview executable) that allows users to interactively inspect and manipulate data. The workflow is centered around the *Pipeline Browser*, which visualizes the data flow as a directed acyclic graph. Users load data sources and apply successive filters (e.g. slices, contours, ghosts) to process the data.

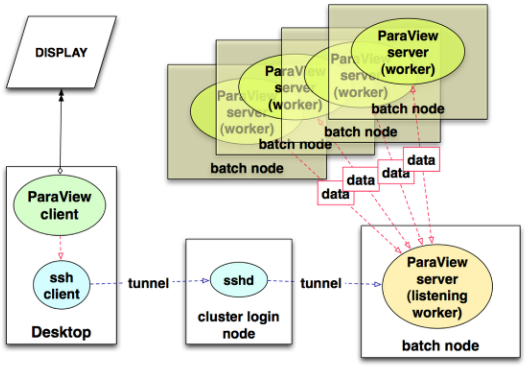
Scripting API: pvpython

`pvpython` is a Python interpreter bundled with ParaView. It essentially serves as an alternative Interface to ParaView. Through different ParaView Python modules it provides access to the

application's functionality via a scripting API. This allows for the automation of visualization pipelines without a graphical interface.

Distributed Rendering: pvserver

To handle data that exceeds the memory or processing capabilities of a single workstation, ParaView utilizes a client-server architecture. The `pvserver` executable runs on the high-performance computing resource (parallelized via MPI), handling the heavy lifting of data processing and rendering. In a standard remote visualization setup, a user connects a local GUI client to the remotely running `pvserver`. The server processes the massive datasets in parallel and sends only the rendered geometry or images back to the client.



Source: hpc.llnl.gov/running-paraview-client-server-mode , accessed 03.01.2026

Figure 2.1: *Visual representation of the client server mode setup for remote rendering of visualization data.*

2.6 ParaView Catalyst

ParaView Catalyst [20, 47, 56] is the primary implementation of the Catalyst API. It empowers users to harness the extensive data analysis and visualization capabilities of ParaView directly within the simulation. Users define their in situ analysis data pipelines using Python files, hereafter referred to as *Catalyst Scripts*. Within these scripts, utilizing the ParaView Python scripting API, one can retrieve data objects published via a Conduit node to Catalyst and perform the following operations:

- **Data Processing:** Apply ParaView's comprehensive filter suite to manipulate data or save it in various file formats.
- **Rendering:** Setup visual representations to generate and export rendered images in situ.
- **Live Connection:** Stream data to a ParaView instance (paraview, pvpython, or pvserver) running in parallel for "live visualization," and create backward data channels for bidirectional steering. Using the connected ParaView client GUI (direct connection or connection via. a pvserver) allows for interactive exploration, enabling the user to apply additional filters and adjust visualizations in real-time.

ParaView-Catalyst Blueprint

To standardize the data handoff from the simulation to Catalyst, ParaView Catalyst defines its own *ParaView-Catalyst Blueprint*. This specification dictates how the overall simulation data package must be hierarchically structured within the Conduit Node to be correctly interpreted. This hierarchical structure adheres to the following simplified setup for the `catalyst_initialize()` call:

```
catalyst_load:
  implementation_name: "name_of_implementation"  # e.g., "paraview", "adios", ...
  implementation_path: "/path/to/ParaView-5.12.0/lib/catalyst"
catalyst:
  mpi_comm: <MPI Communicator Handle>   # needs to be a fortran style integer
  scripts:          # define a List of Catalyst Scripts for the Visualization Pipeline
    <script_name_1>:
      filename: "/path/to/catalyst_script.py"
    <script_name_2>:
      filename: "/path/to/another_script.py"
      args: ["--verbose", "--threshold", "0.5"]
    ...
  proxies:           # define a List of Proxy Files for Steering Configuration
    <proxy_name_1>:
      filename: "/path/to/steering_proxies.xml"
    <proxy_name_2>:
      filename: "/path/to/other_proxies.xml"
    ...
```

Listing 2.4: *Paraview-Catalyst Blueprint for the initialize protocol*

Similarly, the `catalyst_execute()` call requires the data to be structured as follows:

```
catalyst:
  # Global state
  state:
    timestep: <timestep as integer>
    cycle: <cycle as integer>
    time: <time as float>
    parameters: ["...", "..."]

  # Data channels
  channels:
    <channel_name_1>:
      type: "mesh"                              # Protocol type: 'mesh' or 'multimesh'
      data:                                     # according to Mesh Blueprint + additions
        coordsets: ...
        topologies: ...
        fields:
          <field_name>: ...
          <helper_field_name_a>:  ...           # e.g vtkGhostType, globalId, rankId
          <helper_field_name_b>:  ...
        state:
          metadata:
            vtk_fields:
              <helper_field_name>:              # references a registered field name
                attribute_type: <type-name>     # e.g "Ghosts", "GlobalIds", "ProcessIds"
              ...
      state: <like Global State description>    # Optional: Override global state

    <channel_name_2>:
      type: "multimesh"
      data:
        <blockname_2a>:                         # like a mesh layout
            coordsets: ...
            topologies: ...
            fields: ...
            state: ...
        <blockname_2b>: ...                     # like a  mesh layout
        ...
    ...
```

Listing 2.5: *Paraview-Catalyst Blueprint for the execute protocol*

ParaView Catalyst Scripts

Catalyst scripts define the analysis pipeline for the available *in situ* data. These Python scripts follow a standard structure, where specific functions correspond directly to calls made by the simulation via the C++ Catalyst API.

Global Scope Imports necessary modules (typically `paraview.simple`) and statically defines the visualization pipeline components (sources, filters, views, representations, extractors).

`catalyst_initialize` Called once at the start of the simulation. Used for one-time setup tasks that do not need to be defined globally.

`catalyst_execute` The core event loop called at every simulation time step. It updates the pipeline with the current simulation data, triggers rendering, and exports the requested extracts (images or data files).

`catalyst_finalize` Called once after the simulation finishes. Handles cleanup, releases resources, and closes any open files or logs.

The Catalyst framework does not exclusively trigger user-defined functions but also performs a set of standard internal subroutines. For example, once an extractor has been defined in the Global Scope, it is automatically triggered at every time step through the `catalyst_execute` API call, without requiring explicit commands in the user's execute function. When multiple Catalyst scripts are referenced, the adaptor executes them sequentially in the order they are provided. During each phase (initialization, execution, and finalization), the framework iterates through the list of scripts and invokes the corresponding function for every script before proceeding. This allows for modular pipeline designs where different analysis tasks (e.g., steering vs. image extraction) are separated into distinct files without interference.

ParaView Plugins

ParaView was designed so a user can easily add new functionalities. For installations relying on the public binaries it only allows plugins written in xml file format [62]. We need to rely on this plugin functionality tomanage Catalyst's parameter steering. In our plugins we will need the `vtkSteeringDataGenerator` [64] class to define the layout of our steering UI andto create vtk data from the steering UI's current parameter configuration. A Catalyst Python script can then fetch these data structures and forward them to the simulation in Conduit Node form.

2.7 Trame

Trame [32, 73] is an open-source Python framework developed by Kitware that enables the creation of interactive, web-based visual analytics applications. Based on ParaView, VTK, and a set of web utilities, trame is designed to be usable without extensive knowledge of web development [71, 72]. Through a combination of interactive elements, charts, tables, 3D renderings, and more, it serves as a powerful tool for designing frontends for HPC applications. The integration of Trame, ParaView Python, and ParaView Catalyst allows for the creation of custom, lightweight steering and monitoring dashboards, that acts as a connected client to a Catalyst-enabled simulation. It can be optimized for specific simulation frameworks and be streamed from an HPC resource directly to a local web browser. Constructing such a bespoke "Control Room" interface, specific to the simulation's needs rather than exposing the full complexity of a general-purpose visualization tool, lowers the barrier to entry for IPPL/OPALX users with little to no experience with the

ParaView GUI. Furthermore, this approach aligns with the trend in the HPC community towards decoupled, web-accessible interfaces for running and monitoring exascale jobs [37].

2.8 Overview

In short, IPPL and OPALX are particle simulation frameworks that use Kokkos for high-performance data management. In this thesis, we use Conduit's hierarchical data description to map the IPPL data layout and publish it to the Catalyst API. We configure Catalyst to use the ParaView backend to process this data. This allows us to access the simulation's data structures inside a Paraview Catalyst Python script, where we can define visualization and analysis pipelines. Additionally, the data can be passed to a client for live visualization. Either the standard ParaView GUI or a custom Trame-based web application. With the use of ParaView plugins, we can configure either client to also send data back to the simulation, effectively enabling parameter steering.

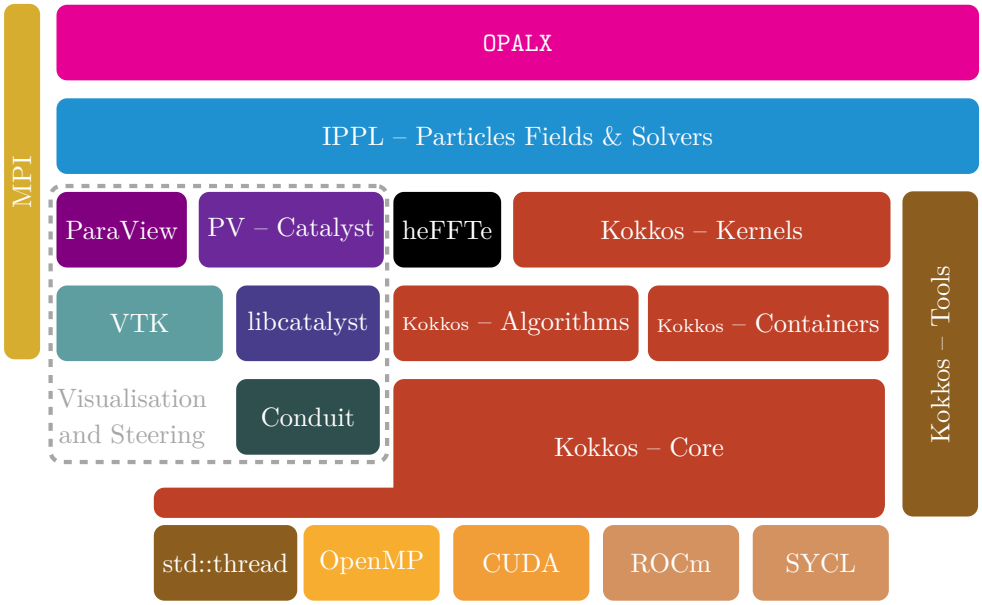


Figure 2.2: *Representation of the OPALX extended software stack (excluding Trame).*

Chapter 3

Theory

3.1 Software Design

3.1.1 Unified Modeling Language (UML)

To visualize software architecture and design, this thesis relies on the Unified Modeling Language (UML) [3]. As UML is a widely adopted industry standard, we omit a detailed introduction here and only give a brief overview of the used notation in Appendix G. Instead readers are referred to standard literature [3, 39] or, for a concise overview, online resources such as Wikipedia [79].

3.1.2 Design Patterns

Software design patterns represent reusable solutions to commonly occurring problems in software design [1]. Below, we provide a brief introduction to the specific patterns referenced in the implementation chapter. For a comprehensive treatment, readers are directed to established web resources [69] or the seminal work by *The Gang of Four* [1].

Builder ([1] Creational, page 97): Facilitates the construction of complex objects that require multiple initialization steps. It separates construction logic from the object being built, implementing the initialization substeps in isolation and providing a method to retrieve the final result.

Factory Method ([1] Creational, page 107): Defines an interface for creating objects, allowing their creation without specifying the exact class. This decouples the client code from the specific object types.

Adaptor ([1] Structural, page 139): It converts the interface of a class into another interface expected by the client, improving accessibility and compatibility between otherwise incompatible structures.

Proxy ([1] Structural, page 207): A proxy class serves as an interface and/or surrogate for a different object. This is useful when one needs to to control or regulate access to the original object.

Command ([1] Behavioral, page 233) Encapsulates a request as an object, containing all necessary information to perform an action or fulfill that request. This decouples the object that invokes the operation from the one that knows how to perform it, allowing commands to be stored, passed between actors, or executed at a later time.

Visitor ([1] Behavioral, page 331): A visitor object represents an operation to be performed on the elements of a complex object structure. By separating the operation from the structure the behavior is applied to, this pattern lets one define new behaviors and algorithms without changing the structure on which they operate.

3.2 In Situ Processing

3.2.1 Semantics: In Situ versus In-Place

The term *in situ* has acquired a dual meaning in computer science literature [18]. Depending on the context, it may refer to the high-level concept of processing newly created data while it still resides in memory (to avoid I/O), or it may refer to the broader algorithmic paradigm where an operation modifies input data directly in memory to produce the result—thereby overwriting the original input and minimizing memory usage.

To avoid ambiguity, this thesis adopts the following distinction: the former concept (processing simulation data without I/O) is referred to as **in situ**, while the latter (overwriting memory buffers) is referred to as **in-place**.

3.2.2 Zero-Copy versus Deep-Copy In Situ

An in situ implementation most faithful to the concept would utilize simulation memory directly for its processing pipeline. This is known as a **zero-copy** approach. The alternative is a **deep-copy** (or staged) approach, where a local, short-term copy of the data is created in memory specifically for analysis. While the latter may initially appear inefficient—potentially doubling the simulation's memory footprint—there are several compelling reasons to utilize data staging:

Safety through Isolation: Exposing raw simulation memory to an in situ pipeline risks data corruption during the analysis process [15]. While modern frameworks are designed to handle data with care, deep-copying ensures strict memory separation. This guarantees that the simulation's state remains untouched and valid, regardless of any errors or accidental writes occurring within the visualization scripts.

Incompatible Data Layouts: Although the Conduit Blueprint offers a flexible means of describing diverse scientific data structures, there is no guarantee that a simulation's internal data layout matches the requirements of the visualization framework. Staging the data via a copy allows for the transformation and "mending" of disparate structures into a layout suitable for the in situ API.

Device-to-Host Staging: In many modern GPU-accelerated HPC simulations, data resides in device memory. While in situ frameworks are increasingly GPU-aware, data staging has traditionally been performed on host (CPU) memory. This is because passing direct references to device memory across framework boundaries has only recently become standardized. For Catalyst feasibility studies were conducted in early 2024 [46], and full integration into ParaView 6.0 was only realized in late 2025 [38].

3.3 High Performance Computing

3.3.1 Ghost Data

In parallel computing, the simulation domain is partitioned into sub-domains, each handled by a distinct MPI rank. Data dependencies across these boundaries are managed using Ghost Cells (or Halo Regions).

In Simulations

Mathematically, solving Partial Differential Equations (PDEs) on a grid often requires stencil operations (e.g., finite difference schemes). Calculating the value of a grid point $u_{i,j}$ often depends on its neighbors ($u_{i+1,j}$, $u_{i-1,j}$, etc.). At the edge of a partition, these neighbors reside in the memory of a different MPI rank. To avoid latency-expensive communication for every single update, a layer of "ghost cells" is added to the boundary of the local domain. These cells are copies of the data from the adjacent rank's internal boundary. They are updated via synchronization steps (using `MPI_Send` / `MPI_Recv`) . This allows the solver to treat the boundary points exactly like interior points during the computation phase.

In Visualization (ParaView)

In visualization, ghost cells serve a similar purpose: ensuring **topological and visual continuity**. Many of ParaView's algorithms (e.g., smooth shading, gradients) require neighborhood data to correctly interpolate values across partition boundaries. Without ghost cells, these calculations fail at the edges, creating visible discontinuities or "seams" where the domain is split. However, while ghost data is necessary for calculation, it must not be directly rendered. Rendering the same physical region twice (once as a real cell, once as a ghost) would result in overlapping geometry and visual artifacts. To resolve this, ParaView uses a **Ghost Mask** (specifically a `vtkGhostType` array) to flag these cells. This allows the engine to use them for computation but automatically hide them during the final rendering process, resulting in a seamless, unified image.

Chapter 4

Implementation

The project's complete source code can be found at github.com/klappi-s/ippl-frk/catalyst-vis_v1.3.3_merged/src/Stream. A detailed guide on how to install and use IPPL in situ features can be found in this document. VISUALISATION.md in the same GitHib repository.

4.1 Project Overview

To integrate in situ capabilities into the IPPL C++ library, we initially identified two primary objectives:

- **Design and Implementation of a CatalystAdaptor:** The adaptor must properly configure, publish, and fetch Conduit nodes via the Catalyst API. Crucially, it must encapsulate this complexity, offering an intuitive interface for users to pass IPPL data objects and simulation steering parameters without being exposed to the underlying Catalyst and Conduit APIs. At its most fundamental level, the Adaptor serves as the interface to trigger the `initialize`, `execute`, and `finalize` Catalyst calls.
- **Development of Robust Catalyst Scripts:** The project requires the design of modular Catalyst Scripts to support flexible in situ workflows. We aim to provide a robust set of scripts that strictly isolates the critical communication logic from the visualization definitions. This allows users to easily adapt or extend the default visualization setup without compromising the integrity of the underlying in situ infrastructure.

The ultimate goal was to validate this framework on IPPL mini-applications [26] and OPALX. For the development and iterative testing of the in situ framework, we primarily utilized IPPL's PenningTrap mini application [53]. Access to a preliminary, *hardcoded* implementation of a CatalystAdaptor [52] for the PenningTrap provided a significant head start for this thesis.

Design Approach

In addition to the `CatalystAdaptor` class, we introduce a helper class that allows the IPPL user to aggregate all objects relevant to the in situ analysis. Once the user has finished constructing this configuration object, it is passed to the `CatalystAdaptor` for initialization. Subsequently, the `CatalystAdaptor` requires only a single function call per time step. This call handles the complex logic of packaging all relevant data into a Conduit Node, triggering the in situ analysis, and writing back any modified values to the simulation's steering parameters.

4.2 Visualisation and Steering Registry

4.2.1 Design

We implemented a configuration approach inspired by the Builder pattern, enabling users to register objects of interest for in situ analysis within a specialized container class designated as the `VisRegistry`. Rather than managing a simple list of pointers to the simulation data, this class serves as a type-erased (programming techniques to remove or hide type information of objects so they can be treated uniformly) container capable of handling the heterogeneous template types found in IPPL. The user is responsible for instantiating two separate objects of this class:

1. A **Visualization Registry** to capture fields and particles intended for export to Catalyst.
2. A **Steering Registry** to capture simulation parameters that can be modified by the analysis.

This class offers a fluent interface to register objects and provide methods to dispatch operations (Visitors) to the underlying entries. Once the user has populated both instances, they are passed to the `CatalystAdaptor` during initialization. This handoff provides the Adaptor with the specific capabilities needed for its distinct tasks, the possibility to dispatch visitors to all entries of the two registries. By utilizing this structure, we provide a flexible API that abstracts the complexity of the underlying system. The IPPL user can enable in situ visualization by simply populating these two registry objects, avoiding direct confrontation with the complex Catalyst and Conduit APIs.

Technical Implementation: The Registry Mechanism

The two primary object types targeted for visualization are Fields and Particle Bunches. In IPPL, these are not defined by a single static class type, which presents a challenge for a generic registry: **Fields** are instantiations of the `ippl::Field` class template (varying by dimension and data type). A field can be realized as a scalar or vector field (via `ippl::Vector`), with dimensions limited only by the underlying Kokkos capabilities (currently supporting up to 8 dimensions).

```
1    template <typename T, unsigned Dim, class Mesh, class Centering, class... ViewArgs>
2    class Field : public BareField<T, Dim, ViewArgs...> { 7* ... */ }
```

Listing 4.1: *The ippl::Field class template*

Similarly, **Particle Bunches** are user-defined classes inheriting from the `ippl::ParticleBase` template:

```
1    template <class PLayout, typename... IDProperties>
2    class ParticleBase{ /*...*/ }
```

Listing 4.2: *The ippl::ParticleBase class template*

Furthermore, for simulation steering, we aimed to support a wide range of types, assuming at minimum support for all basic C++ types. This diversity implies that our Registry must handle an immense variety of distinct types. Storing arbitrary data objects in a unified container while preserving type information – allowing for standard iterator or dictionary access – contradicts C++'s strict static typing and cannot be achieved without significant compromise. To find a feasible solution, we reformulated the requirement. We do not need to *retrieve* a typed reference from the registry; we only need the **ability to call arbitrary functions** on the stored entries from within the Adaptor.

A Visitor-Command Hybrid Pattern

To showcase the workings of our registry we will use a simplified example. The complete self-contained working demonstration of this pattern is provided in Listing A.2.

To overcome this, we implemented a hybrid mechanism using `std::variant` and the **Command pattern**. We encapsulate our distinct visitors (e.g., `VisitorA`, `VisitorB`) within a `std::variant` (see Listing 4.3). Instead of relying on a polymorphic base class, we will show how we implement a type erased container, which instead of saving references or pointers to data objects, captures them inside a generic lambda expression (inside the lambda we simply *visit* a method with the entry-reference as a function argument). We show how the method inside the lambda expression can retroactively be fitted according to our needs.

From the perspective of the `CatalystAdaptor`, the registry must support distinct functions applied to it's entries, behaving differently depending on the distinct entry type. For instance, methods of type "A" during initialization and method of type "B" applied once per timestep. To handle the diverse IPPL data types, the Adaptor defines separate sets of overloaded methods (e.g., `method_a` and `method_b`) specialized for each type family (e.g for IPPL these would be Fields, Particles, Scalars etc.).

```
1
2  template<typename T> concept Arithmetic = std::is_arithmetic_v<T>;
3
4  class Adaptor {  public:
5
6      // --- Method A:  Overloads for a generic void function  (template) ---
7
8      template <std::integral T, std::size_t D>
9      void method_a(std::array<T, D>& a
10     {  std::cout << "  [A] Arr<Int," << D << ">\n"; for(auto& v : a) v += 10; }
11
12     template <std::floating_point T, std::size_t D>
13     void method_a(std::array<T, D>& a)
14     {  std::cout << "  [A] Arr<Flt," << D << ">\n"; for(auto& v : a) v += 1.5; }
15
16     void method_a(Arithmetic auto& e) { std::cout << "  [A] Scalar\n"; e += 5; }
17
18     void method_a(auto&) { std::cout << "  [A] Other\n"; }
19
20     // --- Method B: Overloads  for a generic void function (template) ---
21
22     template <std::integral T, std::size_t D>
23     void method_b(std::array<T, D>& a)
24     {   std::cout << "  [B] Arr<Int," << D << ">\n"; for(auto& v : a) v *= 2;   }
25
26     template <std::floating_point T, std::size_t D>
27     void method_b(std::array<T, D>& a)
28     {   std::cout << "  [B] Arr<Flt," << D << ">\n"; for(auto& v : a) v *= 2.0; }
29
30     void method_b(Arithmetic auto& e) { std::cout << "  [B] Scalar\n"; e *= 2;  }
31
32     void method_b(auto&) { std::cout << "  [B] Other\n"; }
33
34     // --- Visitor Logic ---
35     struct VisitorB{ Adaptor& a;
36                      void operator()(const std::string&, auto& v){ a.method_b(v); }  };
37
38     struct VisitorA{ Adaptor& a;
39                      void operator()(const std::string&, auto& v){ a.method_a(v); } };
40
41     using VisitorVariant = std::variant<VisitorA, VisitorB>;
42 };
```

Listing 4.3: *The simplified concept for the CatalystAdaptor Layout*

Dispatching such functions across a heterogeneous collection is typically achieved using the Visitor pattern. However, the classic inheritance-based Visitor implementation poses a critical limitation: virtual functions cannot be templates. We cannot define a base class for the visitors we need – which have to use function templates. Given that IPPL data structures rely heavily on complex templates, defining a rigid `Visitor` base class with virtual methods for every possible template instantiation is infeasible. To overcome this, we encapsulate our distinct visitors (e.g., `VisitorA`, `VisitorB`) within a `std::variant` (see Listing 4.3), instead of relying on a polymorphic base class.

To properly use the described `Adaptor` layout, each entry added to the `Registry` is stored as an instance of the helper struct `Entry`. When a user adds an object to the registry, the `dispatch` lambda captures the typed reference of that object. This lambda is defined to invoke `std::visit` on a provided `VisitorVariant` instance, effectively applying the active Visitor to the stored data. Essentially, we are not storing the object directly; we are storing a **Command** that knows how to apply a **Visitor** to that specific object type. The Registry then provides two functions employing this Visitor pattern. The `forEach(auto&& vis)` function iterates over all the stored `Entry` commands and executes their `dispatch` function, passing the Visitor `vis` along as function argument. `forOne` does the same thing but only applies the visitor to entries with a specific label. This results in a counter-intuitive but useful inversion of control: The Adaptor never accesses the Registry entries directly. Instead the Registry entries invoke the `() operator` of the Visitor and can pass along the object reference as argument. This Visitor – which was passed to the `forEach` function call with a stored reference to an `Adaptor` instance – can then trigger the appropriate overloaded member method for the underlying type.

This approach provides a "magic container" interface with near-perfect type erasure. It grants the Adaptor versatile access to the entire content of a `RuntimeVisRegistry` instance, allowing us to perform complex, type-specific operations on a heterogeneous collection of IPPL objects without ever exposing their template types to the outer scope. A simple demo code how to use this sample Registry and Adaptor is given in Listing 4.5.

```
template <typename T, size_t N>
void print_array(const std::array<T,N>& a) {for(auto v : a)std::cout<< v << " ";}

int main() {
    // 1. Create adaptor and registry instance
    Adaptor a;
    Registry r;

    // 2. Create Objects of Interest for in "situ analysis"
    int i = 1; double d = 1.0; std::string s = "test";
    std::array<int, 2> ai = {10, 20};
    std::array<double, 2> ad = {1.1, 2.2};

    // 3. Register Objects inside the registy
    r.add("i", i); r.add("d", d); r.add("ai", ai); r.add("ad", ad); r.add("s", s);

    // 4. Logging Helper function: Lets us check if Adaptor methods are properly
    //                             applied by checking values of original objects.
    auto log = [&]() {
      std::cout << " i=" << i << " d=" << d << " ai="; print_array(ai); std::cout << "\n";
    };

    // 4. Pseudo In Situ logic: access and modify Objects via. registry.
    std::cout << "INIT:\n"; log();
    std::cout << "\n forEach(A)       \n"; r.forEach(     Adaptor::VisitorA{a});  log();
    std::cout << "\n forOne('i', B):  \n"; r.forOne("i",  Adaptor::VisitorB{a});  log();
    std::cout << "\n forOne('ai', B)  \n"; r.forOne("ai", Adaptor::VisitorB{a});  log();
}
```

Listing 4.5: *Demo function, showcasing how to use the simplified Registry and Adaptor*

```
class Registry {
  // Every registered Objects will be reresented via a:
  // 1. label
  // 2. void function able to save a typed object indirectly inside a closure (type erasure)
    struct Entry {  std::string label;
                    std::function<void(Adaptor::VisitorVariant&)> dispatch;
    };

    // Member Variable storing a dynamic array of entries.
    std::vector<Entry> entries_;

  public:

    // Method to register new objects
    void add(const std::string& l, auto& v) {
        entries_.push_back({
                l,
                [l, &v](Adaptor::VisitorVariant& var) {
                    std::visit( [&](auto& active){active(l,v);}, var);
                }
        });
    }

    // Method to Visit all registered Objects with a provided Visitor instance.
    void forEach(auto&& vis) {
        Adaptor::VisitorVariant var = std::forward<decltype(vis)>(vis);
        for (auto& e : entries_) e.dispatch(var);
    }

    // Method to Visit a specifically labeled registered Object.
    void forOne(const std::string& target, auto&& vis) {
        Adaptor::VisitorVariant var = std::forward<decltype(vis)>(vis);
        for (auto& e : entries_) {
            if (e.label == target) {
                e.dispatch(var);
                return;
            }
        }
        std::cout << "  [!] Label '" << target << "' not found.\n";
    }
};
```

Listing 4.4: *The simplified concept for the VisRegistry class layout*

4.2.2 Registration of Visualization Objects

The architectural complexity described above is entirely shielded from the user. The primary interface is the `add` method (signature: `void add(const std::string& l, auto& v)`), which allows the user to iteratively register new entries by providing a descriptive label and a reference to the object itself. To further simplify the initialization process, we provide a factory function, `MakeVisRegistry` (Listing A.1.) This helper allows users to instantiate and populate a registry in a single step using a list of label-object pairs. Additionally, the registry design is flexible, accepting either direct object references or object pointers as arguments.

4.2.3 Registration of Advanced Steering Parameters

While the registry itself supports any C++ types, we need to extend extended the framework functionality so the `CatalystAdaptor` can *understand* more complex data structures. By allowing more complex structures for parameter steering we want to enable users to fully leverage ParaView's UI capabilities, such as dropdown menus and dynamic property groups.

Registration of `ippl::Vector`

ParaView's UI supports multi-component widgets, allowing up to three values to be edited side-by-side in a single row. We will implement the *simulation-to-ParaView-client* communication, in a way ensuring this specific UI layout for steering registry entries of type `ippl::Vector`.

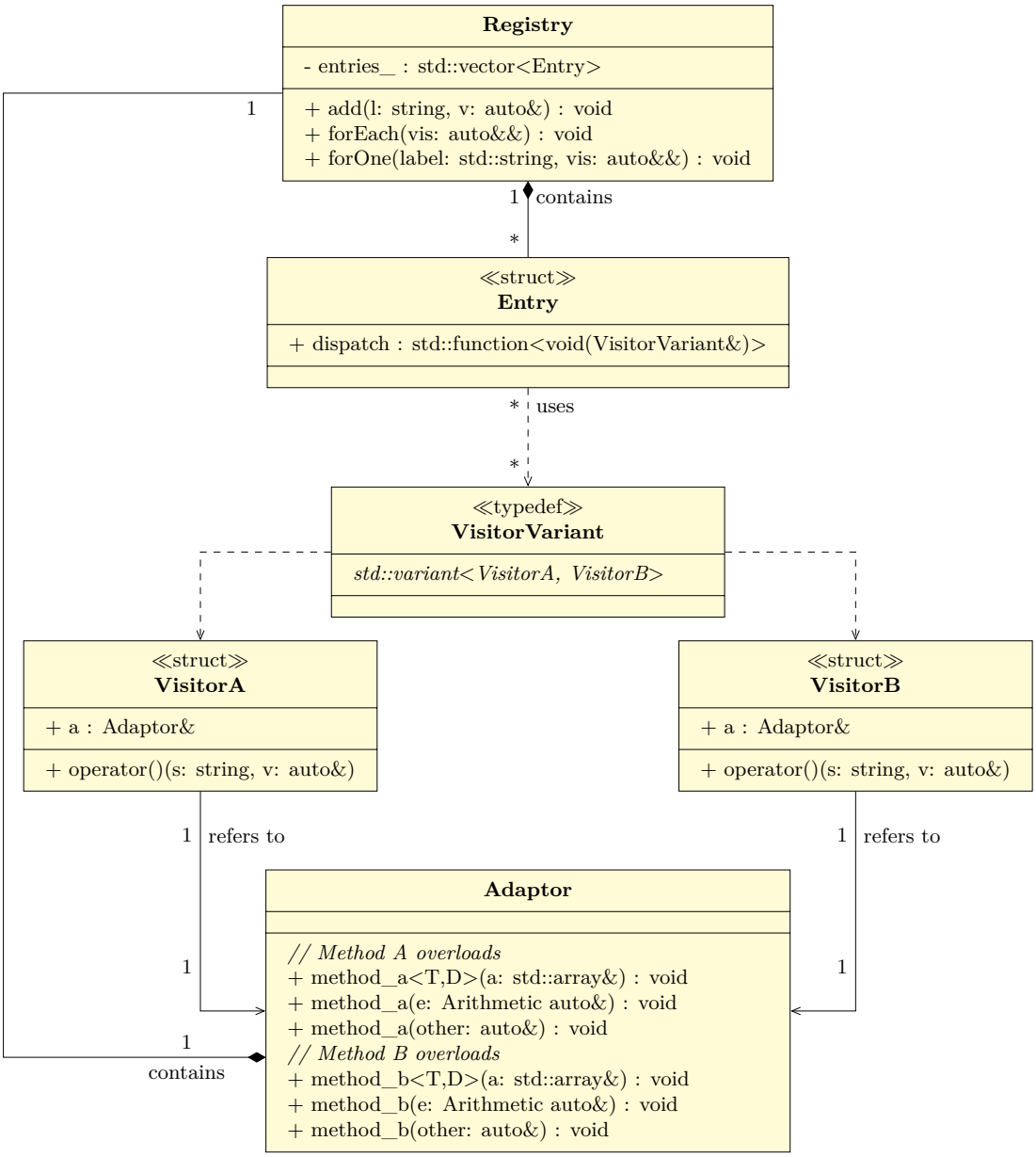


Figure 4.1: *Class diagram representing the demo Adaptor/Registry/Visitor implementation.*

```
std::unordered_map<
            std::type_index,
            std::vector<std::pair<std::string,int>>
                    > enumChoicesByType_m;
```

Listing 4.7: . `enumChoicesByType_m`*: A private member variable of the CatalystAdaptor. Maps a enum type to a vector of pairs representing the enum type's vlaue translations.*

Registration of Enums

We enable dropdown selection lists in the steering UI representing `enum` types in the steering registry. To display the same labels as the source code uses, we must bridge the gap between the underlying integer values of a C++ `enum` and their string representations. Since C++ lacks native reflection for this, we introduced the `RegisterEnumChoicesTyped` API method. This allows the user to explicitly associate string labels with enum values.

```
template<typename E>
requires (std::is_enum_v<std::decay_t<E>>)
void CatalystAdaptor::RegisterEnumChoicesTyped(const std::vector<std::pair<std::string, E>>&
     entries) {
    std::vector<std::pair<std::string,int>> conv;
    conv.reserve(entries.size());
    for (const auto& p : entries) {
        // Tranlsate the enum-values to their representing integers.
        conv.emplace_back(p.first, static_cast<int>(p.second));
    }
    enumChoicesByType_m[std::type_index(typeid(E))] = std::move(conv);
}
```

Listing 4.6: *The* `RegisterEnumChoices` *function*

This function must be invoked during initialization to properly configure the steering interface for a given enum type. It uses a `CatalystAdaptor` member variable, to save a container mapping type identities to vectors of pairs of string and integer values.

```
// 1. Global definition of an enumeration tye
enum class BeamElement {
    Solenoid,
    Dipole,
    Quadrupole
};

// 2. As parrt of the initialization logic, create enum type, Registry, and Adaptor instance
BeamElement element_;
auto steerRegistry = MakeVisRegistry("first_element", element_);
CatalystAdaptor adaptor;

// 3. Inside simulation setup befor the Adaptor object is initialized:
//    Enables adaptor to associate different enum-integer-values with a string.
adaptor.RegisterEnumChoicesTyped<BeamElement>({
    {"Solenoid",   BeamElement::Solenoid},
    {"Dipole",     BeamElement::Dipole},
    {"Quadrupole", BeamElement::Quadrupole}
});
adaptor.Initialize(/*...*/, steerRegistry);
```

Listing 4.8: *Example: Registering Enum Choices*

Registration of Dynamic Arrays

ParaView's steering utilities allow for the manipulation of dynamically sized objects. For seamless integration, we adopted a design convention where registering any `std::vector<T>` automatically triggers this dynamic resizing and editing behavior in the steering UI.

Registration of Custom Structs

When using dynamic arrays for steering, it is often desirable to manage arrays of complex structures (Array of Structs). From a user interface perspective, it is preferable to keep the members of a single structure instance visually bundled together rather than splitting them into separate arrays. ParaView supports this concept of "grouped" steering parameters. To expose this feature through our API, we implemented support for custom structs. While a custom struct instance (or a `std::vector` of them) can be added to the registry like any basic type, IPPL requires explicit insight into the struct's internal memory layout to map it correctly to the UI. Since C++ lacks native reflection [48, 70], this introspection is achieved via a dedicated registration method for the `CatalystAdaptor`. This function associates member pointers with string labels, enabling the Adaptor to serialize and update the struct members correctly.

```
// 1. Function signature; Method declared inside CatalystAdaptor.
template<typename T, typename... Args>
static void RegisterStructMembers(Args&&... args);

// 2. Custom steering structure defined globally
struct LinMap {
    double time;
    ippl::Vector<double, 3> x_row;
    ippl::Vector<double, 3> y_row;
    ippl::Vector<double, 3> z_row;
};

// 3. Struct registration necsssary during simulation setup.
ippl::CatalystAdaptor::RegisterStructMembers<LinMap>(
    "time",  &LinMap::time,
    "x_row", &LinMap::x_row,
    "y_row", &LinMap::y_row,
    "z_row", &LinMap::z_row
);
```

Listing 4.9: *Example: Registering Struct Members*

The internal mechanisms for handling `std::vectors` and custom structs as steering parameters are examined in more detail in Section 4.3.8.

4.3 CatalystAdaptor

This section details the implementation of the `CatalystAdaptor`. This class serves as an adaptor for the Catalyst API, with its core responsibility being the construction of Conduit nodes. It translates the high-level object references stored in the user's registries into the specific Conduit Node hierarchy required by the Catalyst API. The Adaptor must fulfill three distinct tasks:

1. **Initialization:** It builds an *initialization node* by iterating over both the visualization and steering registries, which is then passed to `catalyst_initialize()`.
2. **Execution:** Before triggering a in situ analysis, it constructs an *execution node* via the same iteration process, passing it to `catalyst_execute()`.
3. **Write-Back:** Finally, it iterates once more over the steering registry after the in situ analysis, to process the *results node* returned by `catalyst_results()`. Writing any modified parameter values back to the simulation state.

Given the considerable scope of this class and its dependencies, this section will focus on the architectural logic rather than exhaustively listing implementation details.

4.3.1 Members

Public Methods

At its core – aside from the Registry helpers discussed in Section 4.2.3 – the Adaptor exposes four primary methods to the IPPL user.

The Adapted Catalyst Interface: Following the official Catalyst integration guidelines found in their examples [55], we provide wrappers for the three primary functions of the Catalyst API. This establishes a direct mapping between the IPPL Adaptor methods and the underlying API calls:

`CatalystAdaptor::Initialize()`	→	`catalyst_initialize()`
`CatalystAdaptor::Execute()`	→	`catalyst_execute()`
`CatalystAdaptor::Finalize()`	→	`catalyst_finalize()`

The handling of `catalyst_results()` is integrated automatically within the `Execute` subroutine, simplifying the user's control flow.

`rememberNow():` Since some of IPPL's PDE solvers operate in-place (overwriting input data during computation), the state of a field at the end of a time step may differ from the physical state of interest. This method allows the user to trigger an immediate deep copy of a registered data structure at any point in the loop. When `Execute()` is subsequently called, the Adaptor will visualize this cached "remembered" state instead of the current simulation memory.

Variables

The `CatalystAdaptor` maintains the state of the in situ pipeline through the member variables listed in Listing 4.10. The core logic relies on the registries and Conduit nodes discussed previously. To create a flexible workflow, we implemented an environment-variable-driven configuration system. In `const char*` pointers store keys received from specific environment variables, while corresponding `const bool` flags store their parsed state (enabled/disabled). This allows users to toggle critical features such as live connections, steering capabilities, or data extraction purely via the runtime environment. This eliminates the need to unnecessarily recompile the simulation code repeatedly for identical simulations. Consequently, users and developers can easily switch between settings to run and analyze isolated sub-processes. A full reference for these configuration keys and their usage is provided in Appendix E.

Private Methods and their Visitors

The registry architecture described previously relies on five distinct internal visitors. These Visitor structs are defined within the `CatalystAdaptor` class.They act as lightweight dispatchers that delegate operations to specific overloaded adaptor methods. The first two are designed to work with the visualization registry and the remaining three operate on the steering registry entries. The mapping between Visitors and their corresponding methods is as follows:

`InitVisitor`	→	`InitVizChannel`
`ExecVisitor`	→	`ExecVizChannel`
`SteerInitVisitor`	→	`InitSteerChannel`
`SteerForwardVisitor`	→	`ForwardSteerChannel`
`SteerFetchVisitor`	→	`FetchSteerChannel`

```
    // --- Registries ---
    std::shared_ptr<ippl::VisRegistryRuntime> visRegistry_m;
    std::shared_ptr<ippl::VisRegistryRuntime> steerRegistry_m;

    // --- Conduit Nodes ---
    conduit_cpp::Node node_m;     // Initialization and Execution node
    conduit_cpp::Node results_m;  // Write-back node (steering results)

    // --- Logging ---
    Inform ca_m;
    Inform ca_warn;

    // --- Metadata Storage ---
    // Maps Enum types to their string-integer pairs
    std::unordered_map<std::type_index,
                       std::vector<std::pair<std::string,int>>> enumChoicesByType_m;

    // --- MPI rank ---
    int rank_m;  // = ippl::Comm->rank()
    // --- Runtime Configuration Flags ---

    // Environment variable strings
    const char* catalyst_vis;
    const char* catalyst_live;
    const char* catalyst_steer;
    const char* catalyst_png;
    const char* catalyst_vtk;
    const char* catalyst_ghost_mask;
    const char* proxy_option;

    // Parsed configuration state
    const bool vis_enabled;
    const bool live_enabled;
    const bool steer_enabled;
    const bool png_extracts;
    const bool vtk_extracts;
    const bool use_ghost_masks;

    const std::filesystem::path source_dir;

    // --- Advanced Features ---
    // (Detailed in later sections)
    ViewRegistry viewRegistry;
    ProxyWriter proxyWriter_m;
    std::unordered_map<GhostKey_t, HostMaskView1D_t, GhostKeyHash> ghostMaskCache_m;
```

Listing 4.10: *A list of all* `CatalystAdaptor` *member variables.*

Should the visitor encounter a registry entry that fails to match any of the defined function overloads during iteration, an exception is raised. The detailed handling of such invalid entries will not be discussed further.

4.3.2 Meta-Programming Helpers

To properly leverage function overloading for the diverse entries in the Visualization and Steering registries, we must be able to distinguish between different element types at compile time. This sometimes requires the use of C++20 concepts or standard metaprogramming techniques. However, creating a reliable concept to check if a type is derived from the `ippl::ParticleBase` template proved challenging, as `ParticleBase` is a class template rather than a static type. To resolve this, we introduced a non-templated base class, `ParticleBaseBase` (4.11), from which the `ParticleBase` template inherits. This provides a *stable* type-anchor for our concept checks. With this common ancestor established, defining the necessary meta-concepts for particle containers becomes straightforward. Similarly, concepts for all other supported types are defined by leveraging the standard C++ metaprogramming library [49]. A complete overview of all defined concepts is provided in the Appendix A.3.

```
// Untemplated base class for type detection
class ParticleBaseBase {
    public:
    virtual ~ParticleBaseBase() = default;
};

// Original template now inherits from ParticleBaseBase
template <class PLayout, typename... IDProperties>
class ParticleBase : public ParticleBaseBase {
    /* ... */
};
```

Listing 4.11: *Introduction of the* `ParticleBaseBase` *class as a Parent class to* `ParticleBase`.

4.3.3 Construction

The `CatalystAdaptor` class defines a single default constructor. Its primary task is to initialize the internal state. Starting with the configuration of the console-logging-verbosity by setting the output level of a `ippl::Inform` member objects. The level is determined by querying a dedicated environment variable `IPPL_CATALYST_VERBOSITY`. In the absence of this user configuration, the verbosity falls back to the global IPPL settings, retrieved via `ippl::Info->getOutputLevel()`.

Beyond logging, the constructor is responsible for parsing and caching the full set of runtime configuration flags. This ensures that the environment-variable-driven settings (as discussed in Section E) are readily accessible as member variables throughout the Adaptor's lifecycle.

4.3.4 Initialization

The `CatalystAdaptor` is designed to accept the user's registry instances exactly once. This handoff occurs during the `Initialize` method. The Adaptor then saves smart-pointers to the two registries ensuring it has persistent access to the simulation's data. The primary objective of this method is to construct the Conduit Initialization Node , which is subsequently published to Catalyst. This node acts as the configuration draft for the Catalyst in situ session, specifying the following key components:

Script Configuration The node specifies the paths to all relevant Catalyst Python scripts.

- **The Main Script:** A primary script is always designated. We pass a comprehensive set of command line arguments to this script, including the environment settings (controlling verbosity, live visualization, and steering toggles) and a list of all channel names.

  A "Channel Name" is a generated string that uniquely identifies the label and type of a registered object. Passing these names as arguments simplifies the script's logic, allowing it to easily identify and fetch the published data sources from the execution node later in the pipeline.

- **PNG Extraction Scripts:** For every registered visualization object, we configure a separate entry for a generic PNG extraction script. By passing a different set of command line arguments to the same scripting file, we can instruct Catalyst to generate distinct visualizations for each specific data source.

Steering Proxy Configuration To enable parameter steering, Catalyst requires a description of the "backward channels" (data flowing from ParaView to the simulation). This is achieved via XML proxy files. Similar to Python scripts, the initialization node accepts a list of paths

to these proxy files. To automate this, we designed a helper class, the `ProxyWriter`, which is instantiated as a member of the `CatalystAdaptor`. The workflow is as follows:

1. The Adaptor initiates a `SteerInitVisitor` which iterates over the steering registry.
2. For each entry, the visitor informs the `ProxyWriter` of the required channel parameters.
3. At the end of initialization, the `ProxyWriter` generates a single XML file containing definitions for all backward channels.
4. The path to this newly generated file is added to the Conduit node.

This ensures that Catalyst is explicitly informed of how the simulation expects the backward flow of information to be structured. A detailed discussion on this mechanism follows in Section 4.3.8.

MPI Configuration Finally, the initialization node allows for the specification of the (Fortran-Style) MPI Handle. We convert the current communicator using `MPI_Comm_c2f(MPI_Comm comm)` and attach it to the node. If this is omitted, Catalyst defaults to using `MPI_WORLD`.

Source Code The resulting implementations displayed in the Listing 4.13. An example of the resulting layout of the initialization node is provided in the Appendix in Listing B.1.

```
void CatalystAdaptor::set_node_script(
    conduit_cpp::Node node_path,
    const std::string env_var,
    const std::filesystem::path default_file_path
){
        std::filesystem::path file_path;

        // 1. Tries to access an environment variable
        const char* file_path_env = std::getenv(env_var.c_str());

        // 2. If it exists us the environment configuration.
        if (file_path_env && std::filesystem::exists(file_path_env)) {
           file_path = file_path_env;
        }
        // 3. Else use the default filepath.
        else {
          file_path = default_file_path;
        }
        // 4. Set string value inside the Conduit Node.
        node_path.set(file_path.string());
}
```

Listing 4.12: *Simple helper method to set a file path value inside a Conduit Node. First tries to fetch a string from the Environemnt Configuration and alternatively uses a given default file path.*

```cpp
void CatalystAdaptor::Initialize(    const std::shared_ptr<VisRegistryRuntime>& visReg,
                                     const std::shared_ptr<VisRegistryRuntime>& steerReg)
{
    // 1. Initialize member registries
    visRegistry_m   = visReg;
    steerRegistry_m = steerReg;

    // 2. Conduit Node Setup:
    // 2.1  MPI Communicator Handle
    const int fcomm = MPI_Comm_c2f(MPI_COMM_WORLD);
    const int64_t fcomm64 = static_cast<int64_t>(fcomm);
    node["catalyst/mpi_comm"].set(fcomm64);

    // 2.2 Set main Catalyst Script path.
    //     Use Helper function: Sets a path as string in the Conduit node,
    //                          First tries to fetch Env Variable else use default.
    set_node_script(node["catalyst/scripts/script/filename"],
                    "CATALYST_PIPELINE_PATH",
                    source_dir / "catalyst_scripts" / "pipeline_default.py");

    // 2.3 Set Proxy File path.
    set_node_script(node["catalyst/proxies/proxy_/filename"],
        "CATALYST_PROXYS_PATH",
        proxy_path
    );

    // 3. Pass script arguments to the Main script
    // 3.1 Basic Configuration Arguments
    conduit_cpp::Node args = node["catalyst/scripts/script/args"];
    args.append().set_string("--verbosity");
    args.append().set_string(std::to_string(ca_m.getOutputLevel()));
    args.append().set_string("--VTKextract");
    args.append().set_string(std::string(catalyst_vtk));
    args.append().set_string("--steer");
    args.append().set_string(std::string(catalyst_steer));
    // 3.2 Channel each Viz channel appends it's name during the initialization visit.
    args.append().set_string("--channel_names");
    InitVisitor initV{*this};
    visRegistry->forEach(initV);

    /* --- PROXY WRITER CONFIGURATION AND SETUP --- */
    auto proxy_path = (source_dir / "catalyst_scripts" / "catalyst_proxy.xml").string() ;
    proxyWriter.initialize(proxy_path, cfg_yaml);
    /*
        ...
    */
    // 3.3 Channel each Steer channel appends it's name during the initialization visit.
    args.append().set_string("--steer_channel_names");
    if (steer_enabled ) {
        SteerInitVisitor steerInitV{*this};
        steerRegistry->forEach(steerInitV);
    }

    // 4. Logic if proxy file should be newly created and if Simulation should abort.
    if(std::string(proxy_option) == "PRODUCE_ONLY"){
            proxyWriter.produceUnified("SteerableParameters_SCALARS", "SteerableParameters");
            throw IpplException("proxywriter:PRODUCE_ONLY");
    }else if(std::string(proxy_option) == "OFF"){
    }else{proxyWriter.produceUnified("SteerableParameters_SCALARS", "SteerableParameters");}

    // 5.  CATALYST API CALL
    catalyst_status err = catalyst_initialize(conduit_cpp::c_node(&node));
    if (err != catalyst_status_ok) throw IpplException(/* ... */);
    node.reset();
}
```

Listing 4.13: *Implementation of the* `CatalystAdaptor::Initialize` *method.*

4.3.5 Execution

The core responsibility of the `Execute` subroutine is to orchestrates the entire in situ analysis cycle for a given time step. First, construct the Conduit Execution Node – which represents the current state of the simulation – then pass this node to Catalyst to trigger the in situ analysis pipeline, and at the end fetch and write back steering parameters.

Data Transfer Strategy

Ideally, in situ visualization aims for a "zero-copy" approach, where the analysis tool reads directly from the simulation's memory to minimize overhead. However, our implementation prioritizes a robust "deep copy" strategy to address specific compatibility constraints. Since IPPL is designed to run on CPUs and GPUs, simulation data can reside in device memory. For this thesis we were working with ParaView 5.13, for which staging data directly from device memory is not yet supported. While a zero-copy approach is feasible for data residing strictly in host memory, a deep copy is currently required to ensure reliable operation across all hardware backends. In this thesis, we outline the logic for both approaches, though our current implementation relies on the safer deep-copy method. We utilize `Kokkos` to create deep copies of the data from the internal IPPL fields and particle bunches into contiguous, host-accessible memory buffers. Once the data is secured in host memory, we use Conduit's `set_external` API call. This function takes in a description of the existing buffer by storing the pointer along with metadata regarding the memory layout (size, stride, offset).

```
// Generic signature where 'cname' represents the specific C-type
void set_external(  cname* data,
                    conduit_index_t num_elements = 1,
                    conduit_index_t offset = 0,
                    conduit_index_t stride = sizeof(cname),
                    conduit_index_t element_bytes = sizeof(cname),
                    conduit_index_t endianness = 0
                );
```

Listing 4.14: *Conduit* `set_external` *function signature*

For scalar values or small data arrays (typically used in steering), we utilize Conduit's standard `set` function, which copies the lightweight data directly into the Conduit node structure. The signature for the `set_external` function, which serves as the primary bridge for visualization data is shown in Listing 4.14 (the standard `set` function shares an identical signature):

Distributed Data Handling

Since IPPL can operate in a distributed memory environment using MPI, the Adaptor is only responsible for describing the local data residing on the current MPI rank. When constructing the execution node, we describe the local fields and particle subsets. Paraview Catalyst is capable of ingesting this distributed collection of local descriptions. It translates them into distributed VTK (VTK-m/Viskores) data objects, allowing the in situ pipeline to operate in parallel while still producing a coherent global result.

Source Code

The implementation of the execution logic is shown and neatly explained in Listing 4.15. The specific subroutines invoked here and in the `Initialize` function (see Listing 4.13) are detailed in the subsequent sections.

```cpp
void CatalystAdaptor::Execute(int cycle, double time) {

    // 1. Update Simulation State Metadata
    conduit_cpp::Node& state = node["catalyst/state"];
    state["cycle"].set(cycle);
    state["time"].set(time);
    state["domain_id"].set(rank_m);

    // 2. Populate Visualization Data
    if(vis_enabled){
        ExecVisitor execV{*this};
        visRegistry->forEach(execV);
    }

    // 3. Populate Forward Steering Parameters
    if (steer_enabled) {
        SteerForwardVisitor steerV{*this};
        steerRegistry->forEach(steerV);
    }

    // 4. Trigger Catalyst Execution
    catalyst_status err = catalyst_execute(conduit_cpp::c_node(&node));
    if (err != catalyst_status_ok) {
        std::cerr << "::Execute() Failed to execute Catalyst: "
                  << err << std::endl;
    }

    // 5. Handle Steering Results
    if (steer_enabled) {
        // 5.1 Collect Catalyst steering data inside results node
        catalyst_status err = catalyst_results(conduit_cpp::c_node(&results));
        if (err != catalyst_status_ok){
            std::cerr << "Failed to execute Catalyst-results: " << err << std::endl;
        }
        // 5.2  Write-Back
        SteerFetchVisitor fetchV{*this};
        steerRegistry->forEach(fetchV);
    }

    // 6. Cleanup and Reset for next cycle
    viewRegistry.clear();
    ghostMaskCache.clear();
    node.reset();
    results.reset();
}
```

Listing 4.15: *Method implementation:* `CatalystAdaptor::Execute`

From here on forward, code samples will mostly be listed in the Appendices (A: IPPL Source Code, B: Conduit Nodes, C: Proxy Files, D: Catalyst Scripts).

ViewRegistry

An important talking point in our deep-copy strategy involves memory lifetime management. Because we generate new metadata and create deep copies of simulation data during the execution phase, we instantiate new `Kokkos::View` objects to hold this data. `Kokkos::View` objects function similarly to smart pointers, relying on reference counting to manage memory. However, Conduit's `set_external` API accepts only a raw C-pointer to the data. It does not acquire ownership or increment the View's reference count. Consequently, as soon as the method that allocated the View finishes, the local `Kokkos::View` goes out of scope. Without an external owner, the reference count drops to zero, the memory is deallocated, and the Conduit Node is left pointing to "garbage data" (a dangling pointer). To prevent this, we introduced the `viewRegistry`. Its sole purpose is to extend the lifetime of these temporary Views until the Catalyst execution is complete. The `viewRegistry` acts as a generic ownership anchor, pinning the objects in memory by maintaining a positive reference count. It wraps a `std::unordered_map<std::string, std::any>`,

using `std::any` for type erasure. This allows us to store a copy of any `Kokkos::View` (regardless of its template parameters) or standard smart pointer. Using the `viewRegistry` in our workflow is straightforward:

1. When a new View is created for Conduit, it is immediately added to the `viewRegistry`.
2. The data remains valid throughout the `catalyst_execute` call.
3. Once execution returns, we explicitly call `viewRegistry.clear()`, releasing the references and freeing the memory.

Since we only keep the data alive and never need to retrieve it, the implementation is minimal (see Listing A.4). We use string keys to track already registered views, which becomes relevant for the `rememberNow` functionality discussed in Section 4.3.7.

Note: To reduce clutter in the upcoming source code examples, calls to the `viewRegistry` will be omitted, but it is implied that all newly created Views are registered there.

4.3.6 Visualization Methods (Execution)

In this section, we examine the implementation of the `ExecVizChannel` method, which drives the data publication process during the execution phase. We focus on the two primary overloads: one dedicated to `ParticleContainer` objects and the other to `Field` objects. The approach for scalar and vector fields is functionally identical, allowing us to discuss them as a unified category.

Particle Data

IPPL provides a base class template for defining particle containers. Any custom defined particle container class represents a dynamically sized group of particles.

```
template <class PLayout, typename... IDProperties>
class ParticleBase: public ParticleBaseBase {
    constexpr static bool EnableIDs = sizeof...(IDProperties) > 0;
    /* ... */
}
```

Listing 4.16: *ParticleBase class template head*

A class diagram illustrating the relationships between the for particle-relevant classes can be found in Figure 4.2. Relevant source code can be found in the Appendix A.5 and Conduit Node samples in Appendix B.2.1.

The primary function overload responsible for publishing particle data to the Conduit Execution Node is declared with the signature shown in Listing 4.17. This method utilizes a C++20 constraint for its arguments.

```
template<typename T>
requires (std::derived_from<std::decay_t<T>, ParticleBaseBase>)
void CatalystAdaptor::ExecVizChannel(const T& entry, const std::string label){
```

Listing 4.17: *ExecVizChannel function signature for particles*

This particle bunch maintains a collection of associated attributes. Internally, IPPL organizes the data in a Structure-of-Arrays (SoA) layout: each attribute (e.g., position, velocity, charge) is stored in its own independent `ParticleAttrib` object, which wraps a `Kokkos::View`. Within these attribute arrays, the i-th entry corresponds to the value of that attribute for the i-th particle. Two attributes are fundamental to the IPPL architecture:

- `R`: The position attribute, which is mandatory and always present.
- `ID`: The unique global identity attribute. Its inclusion is configurable via the `IDProperties...` template parameter of the `ParticleBase` class.

Beyond these defaults, users can define arbitrary custom attributes by adding members of type `ParticleAttrib<T, Properties...>` to their custom container class. For every object in the Visualization Registry derived from `ParticleBaseBase`, the Adaptor establishes a dedicated channel in the Conduit Execution Node. The upcoming discussion details the standard setup required for all particle containers, followed by a section on the mechanism for dynamically discovering and publishing these custom attributes.

Conduit Mesh and Topology: To represent particles within the Conduit ecosystem, we utilize the *Explicit Mesh Blueprint* with an *unstructured topology*. In this schema, the particle positions (attribute `R`) are mapped directly to the mesh's `coordsets` (node positions). However, a valid unstructured topology also requires a **connectivity** array, which defines how mesh elements are constructed from these nodes. Since individual particles are conceptually "point elements"—meaning each element corresponds to exactly one node—this connectivity array takes the form of a simple linear sequence (a iota array: $0, 1, 2, \ldots, N_{local} - 1$). This trivial mapping is essential for Conduit to correctly associate "fields" (particle attributes like velocity or ID) with the geometry. By defining this connectivity, we ensure that the i-th entry in any attribute array is correctly mapped to the i-th spatial position in the visualization.

Particle Positions: Now somewhat ironically we must reference the particle position attribute a second time, specifically as a Conduit field. Although the position data has already been used to define the mesh geometry (the `coordset`), Conduit does not automatically expose coordinate data as a variable for visualization filters. To ensure that position data is available for coloring and analysis within ParaView, we explicitly register it as an additional Conduit field attached to the mesh.

Particle Identities Next, we address attributes that play a specific role as VTK metadata. First, we handle particle identities. If the `ParticleContainer` has the `ID` attribute enabled , we registers this data as a scalar Conduit field. To ensure ParaView recognizes these values as unique identifiers rather than generic data, we explicitly mark this field with the `GlobalIds` tag in the Conduit node's state metadata.

Particle Domain: To provide spatial context, we also transmit the particle domain boundaries within the same Conduit channel. This is implemented as a lightweight auxiliary mesh consisting of the 8 corner points of the bounding box, retrieved directly from the `ParticleContainer`'s associated region layout. Since this channel now accommodates two distinct mesh definitions – the particle point cloud and the domain bounding box – it can no longer adhere to the standard single-mesh blueprint. Instead, it must utilize the ParaView Catalyst *multimesh* Blueprint structure (see Listing 2.5), which allows multiple grid topologies to coexist within a single data channel.

MPI Ranks: In addition to the existing particle attributes, we introduce a synthetic Conduit field that is not present in the original simulation data: the `MPI_Rank`. This field assigns the current MPI rank index to every particle in the local container. Like the ID field, this is registered as a standard scalar field but can be marked as VTK metadata with the tag `ProcessIds`.

MPI rank data allows for direct visual inspection of the domain decomposition and load balancing.

It is important to note that, due to the design of the Explicit Mesh Blueprint, the code layout used to describe a single rank's data remains unchanged in a multi-process environment. Since the Adaptor is strictly responsible for describing the local data partition on the current rank, the same logic supports distributed parallel execution without modification.

Custom Particle Attributes IPPL enables users to attach custom attributes to any `ParticleContainer` derived from the `ParticleBase` template. Internally, IPPL manages these attributes by storing them in a list of pointers to the abstract base class, `ParticleAttribBase`. This polymorphic storage strategy is necessary because `ParticleAttrib` is a class template; attributes storing different data types are distinct C++ classes. This generic access via. base pointer allows IPPL to iterate over/access all attributes for key operations with virtual functions declared in the base class and implemented within the specific template instantiations.

This architecture presents a challenge for the Catalyst Adaptor. Accessing an attribute via a `ParticleAttribBase*` pointer effectively erases the specific type information of the `ParticleAttrib` instantiation (the template parameter `T`) for an outside scope. But Conduit requires explicit knowledge of the underlying data type to correctly describe the memory layout in the Execution Node. To resolve this, we adopted a solution that modifies the preexisting IPPL interface. Departing from the Open-Closed Principle (open for extension, but closed for modification), we introduced a new virtual function within the `ParticleAttribBase` class template. The resulting virtual function signature is shown in Listing 4.18:

```
virtual void signConduitNode(
          const size_type Np_local
        , conduit_cpp::Node& node_fields
        , ViewRegistry& viewRegistry
        , Inform& ca_m
        , Inform& ca_warn
        , const bool forceHostCopy
    ) const = 0;
```

Listing 4.18: *The `signConduitNode` function signature. A newly introduced virtual function for the `ParticleAttributeBase` class template. It receives as arguments all necessary Objects to add data to the conduit Node properly.*

The concrete implementation of this function is done within the derived `ParticleAttrib<T>` class template. Because this implementation resides within the templated scope, it has direct access to the specific data type `T`. When invoked via the base pointer, this method executes the necessary type-specific logic to register particle attribute data as a new field in the Conduit Execution Node. The full implementation details are provided in Listing A.8.

Attribute Labels We introduced a new internal string member to the `ParticleAttrib` class template to store a human-readable label. This label is retrieved by the Adaptor during execution and serves as the identifier for the attribute within the in situ analysis pipeline, ensuring that data arrays are correctly named and recognizable in the visualization environment.

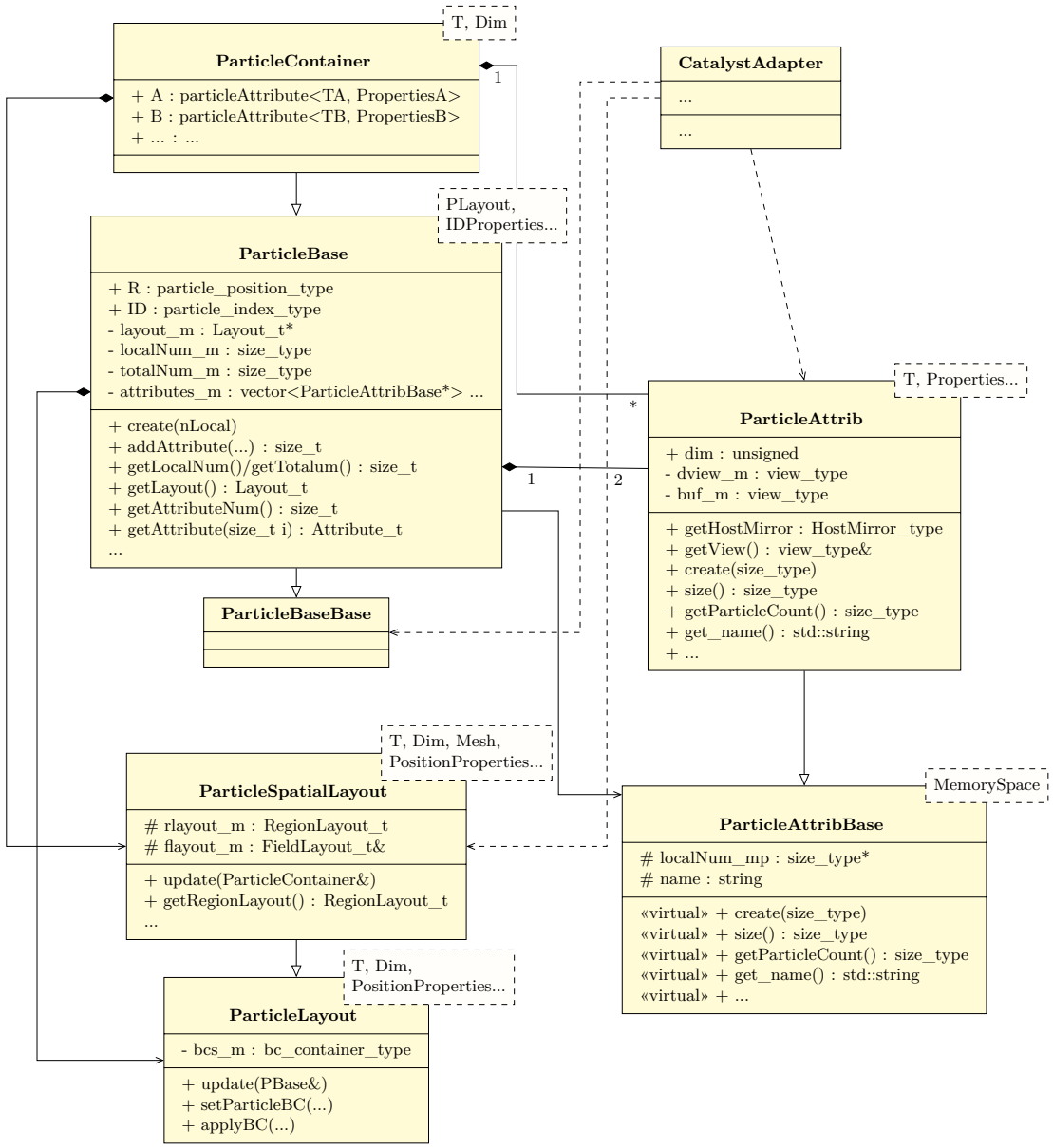


Figure 4.2: *UML Class Diagram for IPPL classes/templates related to a Particle Bunch structure. Members whcih are mainy irrelevant for this thesis were omitted for clarity.*

Fields

We now examine the `ExecVizChannel` function overload designed for instantiations of the IPPL `Field` class template. By templating this function, the signature automatically deduces the field's constituent type parameters. This grants the function body direct access to critical compile-time information, including the underlying data type, and dimensionality.

```
template<typename T, unsigned Dim, class... ViewArgs>
void CatalystAdaptor::ExecVizChannel(   const Field<T, Dim, ViewArgs...>&   entry,
                                        const std::string                   label){ ...
```

Listing 4.19: *`ExecVizChanne` function signature for the `ippl::Field` overload*

A class diagram illustrating the relevant structures for the `ippl::Field` class template is presented in Figure 4.3. The corresponding source code for this section is available in Appendix A.6, while the associated Conduit node structures are detailed in Appendix B.2.2.

Conduit Mesh and Topology: For every registered field, we establish a distinct Conduit data channel within the Execution Node. The handling of scalar fields (where `T` is a standard arithmetic type) and vector fields (where `T` is `ippl::Vector<T_v, Dim_v>`) is functionally equivalent. To represent the underlying grid structure, we utilize Conduit's *Uniform Mesh Blueprint* (specifically with the `uniform` topology). This requires defining the geometry using three fundamental parameters for each dimension:

1. **Grid Origin** (o_x, o_y, o_z): The spatial coordinates of the grid's starting point.
2. **Grid Spacing** (h_x, h_y, h_z): The distance between adjacent grid nodes (cell width).
3. **Grid Dimensions** (n_x, n_y, n_z): The total count of grid nodes along each axis.

The source code for the mesh setup is shown in Listing A.9. We emphasize again that the Adaptor must describe only the local data partition assigned to the current MPI rank, not the global simulation grid. IPPL provides straightforward tools to retrieve local grid information. The precise logic used to determine these values also needs to take into consideration the disussion in the upcoming section (4.3.6.Ghost Cells).

Memory Layout There is a challenge regarding memory layout. ParaView Catalyst expects the Conduit Mesh data to be provided in Column-Major order . In contrast, IPPL delegates the choice of data layout to the user via template parameters. These layouts are managed by Kokkos, which supports both `Kokkos::LayoutLeft` (Column-Major) and `Kokkos::LayoutRight` (Row-Major). Fortunately, by relying on a copy-based in situ architecture, we can resolve this mismatch trivially. Since we are already performing a deep copy of the data from the simulation to the visualization channel, we enforce the specific Kokkos layout required by Catalyst on the destination buffer. This ensures that the data is automatically reordered into the correct format during the copy operation , regardless of the original storage layout used by the simulation (displayed in Listing 4.20).

```
    using HostView_t = Kokkos::View<    typename DeviceView_t::data_type,
                                        Kokkos::LayoutLeft,
                                        Kokkos::HostSpace               >;
    HostView_t hostMirror = HostView_t("hostMirror_LayoutLeft", nx, ny, nz);
    Kokkos::deep_copy(hostMirror, deviceView);
```

Listing 4.20: *Enforcing a data layout during `Kokkos::deep_copy` call*

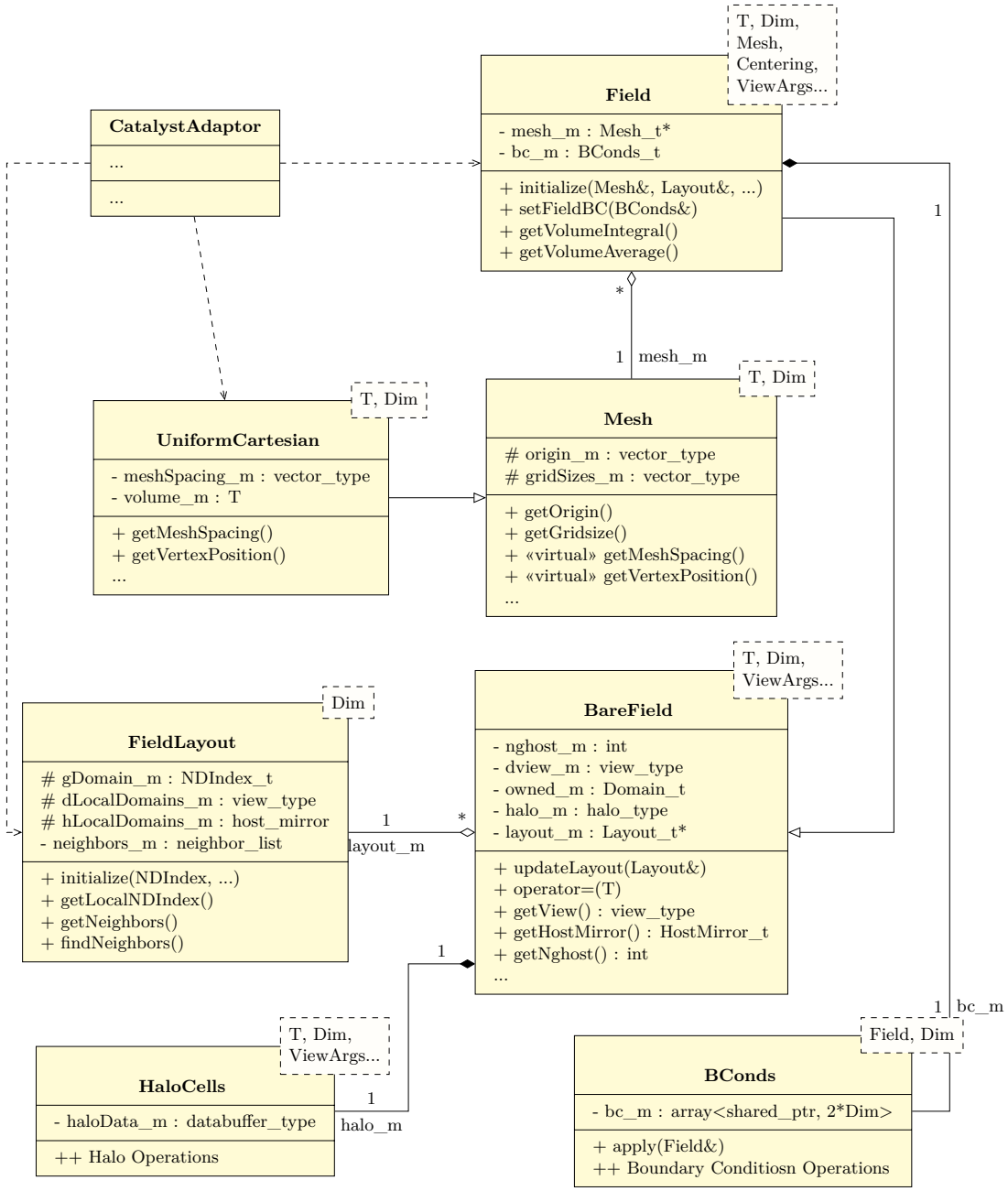


Figure 4.3: *UML Class Diagram for IPPL class templates relevant for field.*

Field and Meta Data A `ippl::Field<T>` encapsulates a single physical quantity determined by the template parameter `T`. To enhance the visualization's diagnostic capabilities, we again introduce the MPI rank as an auxiliary metadata field, similar to the approach taken for particle containers. This allows users to inspect the domain decomposition directly on the grid. The complete implementation is provided in Appendix Listing A.10. For reference, representative examples of the resulting Conduit Node hierarchies for both scalar and vector fields can be found in Appendix B.2.2 (Listings B.4 and B.5).

Zero-Copy Approach Alternatively to implement a zero-copy architecture, one must utilize Conduit's "Strided Structured Field" description. Unlike the standard description which assumes a specific contiguous ordering, this schema allows the user to explicitly define the byte offset (stride) between elements for each dimension. For example, if the source field is stored using `Kokkos::LayoutRight` we can avoid data rearrangement by calculating and specifying the exact strides that describe this layout. This enables Catalyst to read the simulation data directly from its original memory location. The specific Conduit structure required for such a strided field is illustrated in Listing 4.21.

```
...
  channels:
    ippl_sField_<label>:
      data:
        ...
        fields:  # Strided structured field sample: Specifies a reversed order data array
    layout.
          <label>:
            association: "element"
            topology: <topology_string>
            offsets: [0, 0, 0]
            strides: [<nz*ny>, <nz>, 1]   # default access pattern would be [1, <nx>, <nx*ny
    >]
            values: [...]
        ...
```

Listing 4.21: *Hierarchial Layout of strided structured fields. We can use this to specify that Data is stored in Row Major order.*

Ghost Cells IPPL natively supports the use of Ghost Cells (halo regions) for distributed field computations. However, the presence of these duplicated regions requires careful handling within the in situ analysis framework. If ghost data is transmitted to the visualization pipeline without distinction, the backend receives overlapping data definitions, where multiple ranks provide different data for the same physical space. This overlap can lead to rendering artifacts, incorrect analysis, or even crashes in the visualization backend. To resolve this, we consider two distinct approaches:

1. **Exclusion:** Omit ghost cells entirely from the Conduit Node description, transmitting only the rank-owned internal data. In a copy-based architecture, this approach is particularly straightforward to implement. We leverage `Kokkos::subview` to define a view restricted to the process-owned interior domain. This allows us to selectively copy only the unique, non-ghost data into the visualization staging buffer, effectively trimming the halo regions before data publication. This is shown in Listing A.11

2. **Masking:** Transmit the ghost cells but explicitly flag them using a metadata field (specifically the `vtkGhostType` array), instructing the visualization engine to treat them as Ghost data. To pass Ghost meta data we need to create a field mask of type `unsigned char` marking all halo ghost cells with `1` and all internal cell with `0`, and reference this new array to Conduit. The code for this approach is outlined in Listing A.12.

Grid Configuration Computation Correctly defining the mesh configuratios requires adjusting the coordinate origin and grid dimensions to account for the number of Ghost layers included. Listing A.10 demonstrates the correct computation of these local configuration parameters with the use of IPPL.

Zero Copy (continued): To maintain a zero-copy approach when ghost cells are present but need to be excluded, we can still utilize the "strided structured field" Conduit Blueprint. Provided the created Conduit mesh properly defines only the interior layout without the Halo, we can specify memory offsets and strides of our contiguous data array. Conduit then only reads the internal domain, effectively skipping the ghost.

```
channels:
  ippl_sField_<label>:
    data:
      fields:
        <label>:
          association: "element"
          topology: "fmesh_topo"
          offsets: [<nGhosts>, <nGhosts>, <nGhosts>] # nGhosts being the depth of the Halo
          strides: [<nz*ny>, <ny>, 1]
          values: [...]
```

Listing 4.22: *Layout of astrided structured fields, to specify Row Major and cut Ghosts*

Ghost Mask Caching: In many simulations, distinct fields often share the same basic mesh layout. Consequently, generating a separate Ghost Mask for every field is computationally inefficient and memory-intensive, as it results in redundant identical masks. To address this, we introduce a caching mechanism. Before creating a new mask, we check a hash map for an existing mask with the exact same layout. If found, the existing data array is reused. Since the distributed architecture may alter local field sizes between time steps (e.g., due to dynamic load balancing), we clear the cached masks at the end of the `Catalyst` execute call to prevent excessive memory usage (see Listing A.13 for implementation details).

4.3.7 rememberNow() (Execution)

The purpose of `rememberNow(std::string label)` function is to capture the state of a Visualization Registry entry at the moment of the call. The immediate state of the data is preserved for Visualisation. When CatalystAdaptor::Execute() is called the preserved state is referenced in the Conduit Node instead of the original current data. To implement this, the function triggers `ExecVizChannel` early for the specified label. We prevent the later execution phase from overwriting the resulting Conduit node by checking against existing labels in the `viewRegistry`. Additionally, to support future zero-copy implementations, we track which data arrays require enforced copies using the member map: `std::unordered_map<std::string, bool> forceHostCopy_m`. The relevant code is shown in Listings A.14 and A.15.

4.3.8 Steering

For parameter steering with ParaView Catalyst, the forward (simulation → Catalyst) and backward (Catalyst → simulation) information flows must be established separately. The forward flow is implemented identically to the visualization data pipeline: even a single steering parameter must be described within a valid VTK structure in Conduit before being sent to Catalyst. The backward flow, which enables the actual steering, relies on the XML proxy definitions mentioned previously. This proxy file serves a dual purpose: it defines the structure of the data channel sent from Catalyst back to the simulation, and it specifies the layout of the steering controls within

the ParaView GUI. The simulation retrieves these steering values via the `catalyst_results()` API call. The Conduit node passed to this function is populated with the results in a format identical to the forward channel nodes. We then extract these values and update the original parameter references stored in the `CatalystAdaptor`'s steering registry.

Forward Steering Channels through Conduit

To transfer parameter data from the simulation to Conduit, we implement a set of overloads for the function `CatalystAdaptor::AddSteerableChannel`.

Static Parameters: We choose to bundle all non-dynamically sized steerable parameters into a single Conduit channel within the Conduit Execution Node. This consolidation is possible because all scalars technically require only a zero-dimensional mesh to define their structure; thus, each scalar can be treated as a distinct field on this shared mesh. We utilize the simplest possible mesh and topology description available: an explicit unstructured mesh consisting of a single node point (seen in Listing B.6). This layout supports standard arithmetic types, `ippl::Vector`s, enums, and custom structs within a single steering channel. To integrate custom structs into this scheme, the `CatalystAdaptor` "deconstructs" them: each registered struct member is processed as an independent parameter via its respective `AddSteerableChannel` overload. The adaptor preserves their logical grouping by signaling their association to Catalyst via an internal naming convention. For the `LinMap` struct example shown previously in Listing 4.9, this approach results in the Conduit fields displayed in Listing B.7.

Dynamic Parameters For dynamically sized `std::vector`-based steering parameters, the approach differs slightly. Since the lengths of different steering vectors are not guaranteed to be identical, we establish a separate Conduit channel for each dynamic parameter. We again utilize an explicit Conduit Mesh structure defined by node points; however, in this case, we employ a one-dimensional mesh where the number of nodes corresponds directly to the vector length, as demonstrated in Listing B.8. The resulting hierarchical Conduit layout for arrays of structs functions as a logical combination of the two previously described approaches. We establish a single dynamic channel using an explicit one-dimensional unstructured mesh, populated with multiple fields that represent the distinct struct members. This structure is shown in Listing B.9.

Backward Steering Channels through Proxy Files

To properly configure steering for the ParaView client, a ParaView plugin defined via an XML file is required. We have introduced a helper class, `ProxyWriter`, responsible for generating a valid proxy file once during the `CatalystAdaptor::Initialize` subroutine. The following section focuses primarily on the layout of this XML file, rather than the internal implementation details of the `ProxyWriter` class itself. In this file the entire steering proxy definition must be wrapped within a `<ServerManagerConfiguration>` tag. This root element encloses two distinct `ProxyGroup` elements.

`<ProxyGroup name='sources'>`: Within this scope, we define multiple `<SourceProxy>` elements, with the specific attribute `class="vtkSteeringDataGenerator"`. Inside any such source proxy, we define properties using the two tags `IntVectorProperty` or `DoubleVectorProperty`. These vector properties requires a `command` attribute that references a specific member method of ParaView's `vtkSteeringDataGenerator` class [64]. Each `SourceProxy` defined here translates

directly into a single transmitted Conduit channel (a distinct data object sent back to the simulation). Within that channel, the individual vector properties generally correspond to Conduit fields delivered via the results Node. In terms of the ParaView steering interface, each channel appears as a distinct source element in the client's Pipeline Browser, while the vector properties are exposed as editable controls in the *Properties* tab of that source. Additionally, other items defined inside the `SourceProxy` are required to properly encapsulate the data into a valid VTK configuration for the backward channel. Mirroring the strategy used for forward channels, we define a single `SourceProxy` to group all static scalar parameters, and separate `SourceProxy` elements for each dynamic array being sent back. Examples of these proxies can be found in Listings C.4 and C.2. These source proxies can be further customized by including `<Hints>` elements within their scope. Hints allow us to specify several key behaviors:

1. **Value Initialization:** With an element `<CatalystInitializePropertiesWithMesh>`, we can specify fields from incoming forward channels to be used to initialize values of a backward channel. This ensures that the GUI initially displays the correct current values from the simulation, overwriting the static default values usually defined in vector properties.
2. **UI Layout Grouping:** We can organize different properties within the channel's *GUI Properties tab*. This is achieved using `<PropertyGroup>` elements, which explicitly list which steering parameters should be bundled together in the interface.
3. **Widget Customization:** Inside a `<PropertyGroup>` element, we can specify an additional layer of `<Hints>`. Here, we can include a `<PropertyCollectionWidgetPrototype>` element with `group` and `name` attributes referencing an external layout prototype, determining the specific interface widgets used for the elements in that group.

Examples of these hints are provided in Listings C.3. We note that the "prototypes" referenced here must be defined separately in the `misc` proxy group, as described below.

`<ProxyGroup name='misc'>` (miscellaneous): We use this proxy group exclusively to define the UI prototypes. These definitions specify the group label, specific layouts, and widgets to be applied to the elements of any `PropertyGroup` referencing the prototype. Typically, widgets are selected to correspond to the defined value domains of the referenced properties (e.g., checkboxes for a `BooleanDomain`, discrete sliders for an `IntRangeDomain`, etc. [62]). If no specific widget is defined, the default interface renders them as text boxes. Examples of such prototypes are shown in Listing C.5.

The Configuration File: To allow the user to flexibly define ranges for integer and vector domains – ensuring that slider widgets in the generated proxy file specify meaningful bounds – we introduce a configuration mechanism. The `ProxyWriter` can fetch range values from an external YAML file, which the user specifies via the environment variable `IPPL_PROXY_CONFIG_YAML`. The writer then utilizes these specified ranges to generate the proxy file accordingly. A simple example for the layout is shown in Listing C.6.

ProxyWriter: The `ProxyWriter` class manages a collection of parametrized strings. Based on the steering parameter types requested by the `CatalystAdaptor` via. calls to the `InitSteerableChannel` functions, the writer bundles and stacks the corresponding string fragments into internal string streams. Once all relevant information has been aggregated, the `CatalystAdaptor` triggers these streams to be written out to generate the final proxy file.

4.4 Catalyst Scripts

As previously mentioned, rather than consolidating all in situ analysis into a single file, we adopt a modular approach by splitting ParaView Catalyst tasks into four distinct scripts:

Main Script: This script serves as the core of the in situ analysis logic. It is primarily responsible for configuring Catalyst for Live Visualization and steering, as well as defining basic filters and VTK file extractors.

PNG Extraction Scripts: Three specialized scripts, each configuring a rendered scene for image extraction. These are dedicated to particle data, scalar field data, and vector field data, respectively.

Regarding the PNG extractors, we will not delve deeply into specific visualization designs. Instead, we focus on how these scripts establish the foundation required to properly fetch IPPL data. This enables users to adapt the final visualization logic to their specific needs. We will mainly use paraview's `catalyst` [65] and `simple.catalyst` [67] modules. Simplified Code samples for the scripts can be found in Appendix D.

The primary task shared by all scripts is the configuration of Catalyst options to control their own basic behaviors (shown in Listing D.6). For the image extractor scripts, `EnableCatalystLive` is strictly disabled (set to 0), since enabling live when not needed can impact performance.

4.4.1 Data Analysis

Fetching Data Source Proxies

Accessing Catalyst data sources requires their unique registration names, which are passed as runtime arguments. We retrieve these using `catalyst.get_args()` and parse them via Python's `argparse` library (see Listing D.5). This process provides the main script with all available channel names and configuration settings, while each extractor script receives only its specific target-channel's name. In the main script, we iterate over the channel name list, and by using `PVTrivialProducer`, we fetch the data channels *source proxies* from Catalyst and save them inside a dictionary.

```
_sources = {}
for cname in parsed.channel_names:
    _sources[cname] = paraview.simple.PVTrivialProducer(registrationName=cname)
```

Listing 4.23: *Fetching all Catalyst data sources via. the availlable channel names*

Enabling Live Visualisation

Filters: Importing raw field and particle sources is often insufficient for effective analysis. Therefore, we establish a basic processing pipeline using ParaView filter proxies. We restrict this in-script processing to filters that are essential across most scenarios. Similar to the source proxies, we collect these filter objects in a dictionary named `_filters`. Many filters introduced here are primarily for convenience, most could technically be applied later within the ParaView GUI. However, there are critical exceptions where applying a filter within the Catalyst script yields different behavior compared to applying the same filter interactively in the client. It remains unclear to us whether this functional discrepancy is an intentional design choice or a bug within ParaView. A notable exception is the handling of distributed mesh data. In our tests, fetching distributed IPPL field data (VTK Image Data) sources during a Live session

resulted in improper aggregation, where only the rank-0 data was displayed. To resolve this, we enforce the combination of multi-rank data early in the pipeline using ParaView's `MergeBlocks` filter, which ensures the distributed data blocks are correctly unified. Critically, this filter is only effective if applied directly within the Catalyst script. However, the output of `MergeBlocks` is an unstructured grid, which lacks the performance and rendering benefits of the original format. Therefore, we apply an additional `ResampleToImage` filter to restore the data to the VTK Image Data structure. For convenience, we also apply the `Glyph` filter to visualize vector fields and the `ExtractBlock` filters to separate particle data from auxiliary geometry (bounding box corners). To ensure these filters are applied only to the appropriate data types, we leverage the naming convention introduced on the simulation side. IPPL prepends specific prefixes – `ippl_particles_`, `ippl_sField_`, or `ippl_vField_` – to the channel names. By checking if a channel name starts with the corresponding substring, we can identify the underlying data type and apply the relevant filters.

Pipeline Synchronization: Making these sources and filters available to a connected ParaView client is straightforward. We parse the command-line arguments to determine if Live Visualization is requested and pass this configuration to `catalyst.Options().EnableCatalystLive`. If set to `1`, any instantiated sources or filters automatically appear in the client's Pipeline Browser, provided the client connects to the port specified in `catalyst.Options().CatalystLiveURL`. To ensure data consistency, the visualization pipelines must be synchronized with the simulation at every time step. This is handled by iterating through all registered sources and filters within the `catalyst_execute` subroutine and triggering an update:

```
def catalyst_execute(info):
    global _sources
    global _filters

    for name_, source in _sources.items():
        source.UpdatePipeline()
    for name_, filter in _filters.items():
            filter.UpdatePipeline()
```

Listing 4.24: *Updating pipelines during execution*

VTK File Extraction

Setting up VTK file extraction is similarly straightforward. We instantiate extractor objects within the script's global scope and collect them in a dictionary named `_extractors`. For channels utilizing a simple mesh structure (e.g., Fields), we employ the `VTPD` extractor:

```
paraview.simple.catalyst.CreateExtractor('VTPD', <source>, <extractorRegistrationName>)
```

Conversely, for sources relying on a multi-mesh structure (e.g., Particles), the `VTPC` extractor is required:

```
paraview.simple.catalyst.CreateExtractor('VTPC', <source>, <extractorRegistrationName>)
```

These commands alone are sufficient to configure the automatic generation of VTK files for a specified source during execution.

PNG Extraction

Data Retrieval: For the PNG extractor scripts, we begin by retrieving the relevant data proxy using `PVTrivialProducer`. It is worth noting that for VTK Image Data, fetching the original source directly is sufficient for PNG extraction; the specific aggregation filters required for Live

Visualization are not needed here. Instead Ghost data can cause problems for PNG rendering and need to manually be excluded with a `Threshhold` filter if present.

Rendered Views: Once the appropriate data producer is retrieved, we follow standard ParaView scripting procedures to configure a rendered scene (a `RenderView`). Using this view object, we instantiate an image extractor:

```
paraview.simple.catalyst.CreateExtractor('PNG', <renderView>, <extractorRegistrationName>)
```

The specific details of configuring `RenderViews` and extractors are well-documented in the ParaView literature and are outside the scope of this thesis. However, simplified sample code demonstrating our default PNG extraction setup is provided in the Appendix in Listings D.9, D.10, and D.12.

4.4.2 Parameter Steering

For forward steering channels, no additional scripting is required; the configured XML proxy files automatically utilize this data to initialize the steering UI controls within the ParaView GUI. However, to ensure steering data is effectively returned to the simulation (the backward channel), we must explicitly activate the steerable parameters in the script. For the static scalar channel, we can do this with a function call shown in listing 4.25.

```
pvs.catalyst.CreateSteerableParameters(
    steerable_proxy_type_name         = "SteerableParameters_SCALARS",
    steerable_proxy_registration_name = "SteeringParameters_SCALARS",
    result_mesh_name                  = "steerable_channel_backward_all"
)
```

Listing 4.25: *Using `CreateSteerableParameters` inside a Catalyst Script to activate a backward steeering channel. `steerable_proxy_type_name` needs to reference a predfined proxy loaded inside a ParaView client plugin.*

This function call needs to directly reference a proxy definition within the XML proxy file. Consequently, execution will only succeed if the corresponding XML plugin has been properly loaded in a connected ParaView client beforehand. The configuration for dynamically sized array steering channels follows an identical pattern. The requirement for this function call to succeed is, that the argument `steerable_proxy_type_name` needs to match the specific `SourceProxy` definitions in the XML file. Since the steering channel names are also parsed from the script's arguments at the beginning of the script's global scope, we can determine all available proxy names using our established naming conventions. Collectively, these calls to `CreateSteerableParameters` are sufficient to enable full steering capabilities within the ParaView GUI.

4.5 Trame-based Web Application

As established in previous sections, ParaView functionality is not limited to its standard GUI; it exposes a comprehensive Python interface (`pvpython`). By leveraging this interface in conjunction with the **Trame** library we can construct custom client applications that connect directly to the simulation and visualize the data received from Catalyst. We developed a prototype web application serving as a simplified frontend alternative to the standard ParaView GUI. While the reliability of the application is currently somewhat inconsistent due to the limited development timeframe, it successfully functions as a user-friendly surrogate for the full client. This interface is particularly accessible to users unfamiliar with the complexities of the standard ParaView environment. A key advantage of this web-based approach is, when running simulations on HPC resources, the user's local machine no longer requires a separate ParaView installation. Access requires only an SSH tunnel to the resource and a standard web browser.

4.5.1 Functionality

We have implemented the following core features in the application:

- **Session Controls:** Easily accessible buttons to connect to a running Catalyst instance, control the timeline (Play/Pause), and capture screenshots.
- **Data Inspection:** A dynamic dropdown list displaying all IPPL data objects available for visualization.
- **Automated Visualization Setup:** Upon selecting a source, the application automatically configures a default `RenderView` similar to the PNG extractor logic. It intelligently selects the appropriate proxy object (e.g., using the Glyph filter for vector fields rather than the raw source).
- **Visualization Configuration:** Users can customize extracted data proxies via a pop-up configuration window. The available options represent a simplified subset of the standard GUI capabilities:
  - Selection of color schemes from a list of default color maps.
  - Modification of opacity maps.
  - Selection of different field or particle attributes for coloring and opacity.
  - Toggling of data representation styles (e.g., Surface, Wireframe).
- **Curated Filtering:** The configuration window also offers a selection of relevant filters to apply directly:
  - For scalar fields: Slice and Ghost Cell extraction.
  - For vector fields: Component extraction and Ghost Cell extraction.

4.5.2 Implementation

We will not delve into the specific design details of our prototype application; readers are better served by consulting the official Trame documentation for general application structure [74, 72]. Instead, we focus on the critical mechanisms required to effectively utilize Catalyst data within a Trame environment. This is essential because standard Trame examples typically rely on static VTK data objects saved on disk , rather than dynamic live streams. Our implementation primarily relies on the `paraview.live` module [66], which provides several essential API calls for managing the connection:

```
def ConnectToCatalyst(ds_host='localhost', ds_port=22222):
    """Creates a connection to a catalyst simulation.
       Returns a LiveInsituLink object."""
# ------------------------------------------------------------------------
def ExtractCatalystData(link, name):
    """Extract data called 'name' from simulation managed by the LiveInsituLink.
       Registers it in the displaySession."""
# ------------------------------------------------------------------------
def PauseCatalyst(link, pause=True):
    """Pause / Unpause the Catalyst simulation managed by the LiveInsituLink."""
# ------------------------------------------------------------------------
% def ProcessServerNotifications():
    """Processes pending events from server."""
```

Listing 4.26: *Key functions in the paraview.live module*

Catalyst Data Proxies: Establishing a connection to a Catalyst-enabled simulation is straightforward using `ConnectToCatalyst`. Subsequently, we want to identify the names of available sources, so we can extract them into the application's scope. The returned `LiveInsituLink` object is not clearly documented in the Python API. To understand its capabilities, one must refer to the C++ documentation for the underlying `vtkSMLiveInsituLinkProxy` [63] class. Through this reference, we determined that the list of active sources in the connected session can be retrieved by accessing the link's associated `vtkSMSessionProxyManager`, available via the `GetInsituProxyManager()` method (see Listing 4.27).

```
def get_available_extract_names(catalyst_link):
    if not catalyst_link:
        return []

    # 1. Retrieve the In Situ Proxy Manager.
    pm = catalyst_link.GetInsituProxyManager()
    if pm is None:
        return []

    # 2. Get the total amount of proxies saved inside the 'sources' group in the Manager.
    #    This is the amount of proxies availlabe for "Catalyst Extraction".
    num_proxies = pm.GetNumberOfProxies("sources")

    # 3. Fetch all Names of the availlable source proxies and save them inside a list.
    all_names = []
    for idx in range(num_proxies):
        name = pm.GetProxyName("sources", idx)
        if name:
            all_names.append(name)

    return all_names
```

Listing 4.27: *Fetching Catalyst source proxy registration names within a Trame application. These names are possible keys we can use for the* `paraview.live.ExtractCatalystData` *function call.*

This returned list contains the names of all data proxy objects (sources and filters) defined in our Catalyst scripts. Crucially, these names adhere to the strict naming conventions established earlier:

1. In the C++ Conduit channel definitions (for visualization and forward steering).
2. When instantiating filters within the Catalyst script itself.

Because these registration names unambiguously identify the type and structure of the underlying objects, we possess all the necessary metadata required to automate the visualization setup.

Steering Proxies: Handling steering proxies differs from visualization data: instead of accessing existing objects, the client must create them. To achieve this, we again utilize the `vtkSMSessionProxyManager`. By parsing and loading the XML configuration file directly into the manager, we effectively instantiate the proxies and so the Catalyt scripts can establish the backward steering channels (see Listings 4.25, 4.28).

```
with open(xml_path, 'r') as f:
    xml_content = f.read()
    # pm is the vtkSMSessionProxyManager instance
    pm.LoadConfigurationXML(xml_content)
```

Listing 4.28: *Creating the steering channel data proxies for a Trame application.*

Catalyst Polling Function: The final key distinction between our in situ Trame application and a standard, static one is the integration of a polling loop. While standard apps wait passively for user input, our in situ application must actively maintain constant synchronization with the simulation. This is typically managed via an asynchronous Python function running an infinite background loop, which handles a few critical tasks:

1. **Process Notifications:** It calls `live.ProcessServerNotifications()` from the ParaView Live module. This function fundamentally handles data synchronization, receiving bulk data from Catalyst and transmitting any pending steering parameters.
2. **Update Pipelines:** Analogous to the Catalyst script, we must call `UpdatePipeline()` on our active sources to properly process the newly received bulk data within the application's proxy context.
3. **Dynamic Configuration:** It adjusts any data-dependent visualization settings that cannot be fully automated.
4. **Render Trigger:** Finally, it invokes `simple.Render()` to ensure the Trame display is refreshed to reflect the updated views.

Trame UI Having established the steering channels, a synchronized polling loop, and access to all relevant data proxies, the backend requirements are fully met. This allows us to build the frontend using standard Trame methodologies, treating the live data objects similarly to static VTK inputs. While this infrastructure enables the creation of a highly customized user interface, the specific design and implementation details of the our UI Protoype itself fall outside the scope of this thesis.

Chapter 5

Results

5.1 Tests

5.1.1 Example: IPPL Penning Trap with Image extraction

Instrumentation of the IPPL PenningTrap mini-application was straightforward. In the Visualization Registry, we registered the Particle Container, the Electric Field, and the Scalar Field. Notably, the Scalar Field was registered twice under distinct channel names ("density" and "potential") to capture both states. For steering, we introduced a new `double` member to the simulation's manager class to control the scaling of the electric potential. This variable was added to a new Steering Registry during initialization. The adaptor's `Initialize` method is called once all other simulation initialization steps are complete. During the Leapfrog time-stepping loop, the timing of data capture is critical. We invoke `rememberNow("density")` immediately before the FFT Poisson solver is executed, as the solver operates in-place and overwrites the density field with the potential solution. Once the time integration step concludes, we trigger the adaptor's `Execute` method to process the data. Finally, the `Finalize` method is called upon simulation completion. With these minimal changes we enabled full in situ visualization and live steering of the trap's *static* field. By setting appropriate environment flags we can activate the preconfigured PNG extraction, where our default extractor scripts automatically generate images for all registered objects. Figure 5.1 illustrates the results for the three visualized data types.

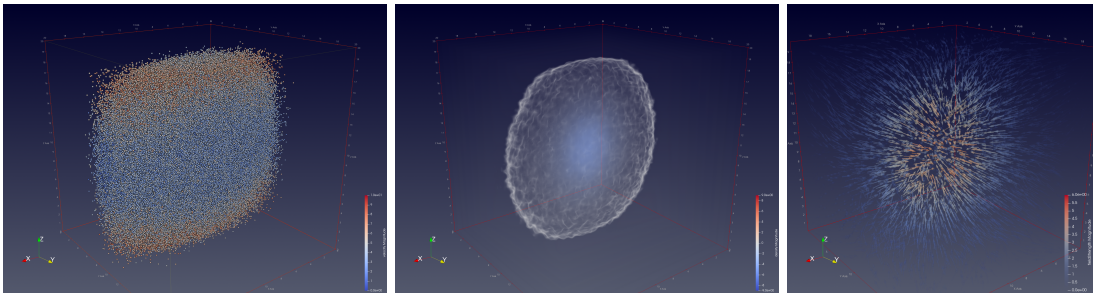


Figure 5.1: *PenningTrap Visualization. Images (PNG) rendered and extracted with ParaView-Catalyst scripts (Left: Particles, Center: Density Field, Right: Electric Field).*

5.1.2 Example: OPALX FODO Lattice with Live Visualization

As a primary feasibility experiment for OPALX, we tested the integration of visualization and steering within the main OPALX executable using a simple beamline scenario. In this example, a bunch of electrons traverses a lattice of one hundred FODO cells. Our steering goal was to control the focusing strength of all quadrupole magnets in the beamline simultaneously using a single global steering parameter.

The integration was implemented within `parallelTracker.cpp`, specifically inside the `ParallelTracker::execute()` method, which governs the main time-integration loop based on the input configuration. We instantiated the `CatalystAdaptor` and created a Visualization Registry referencing the tracked particle bunch. However, accessing beamline parameters directly within the `ParallelTracker` context is non-trivial. To circumvent this, we instantiated a local helper variable, `double K`, to serve as a steering proxy. This variable is registered in the Steering Registry. The standard `Initialize`, `Execute`, and `Finalize` hooks were built into execution method. Crucially, after each in-situ execution step, the updated value stored in the proxy `K` is propagated to all quadrupole elements via respective parameter setter functions. Using environment variables, we enabled both Live Visualization and Steering. By connecting a ParaView client to the running simulation, we successfully modified the focusing parameter K in real-time. This allowed us to manipulate the beam dynamics dynamically, causing particles – which typically remain confined near the axis – to spread significantly beyond the intended domain boundaries, as shown in Figure 5.2.

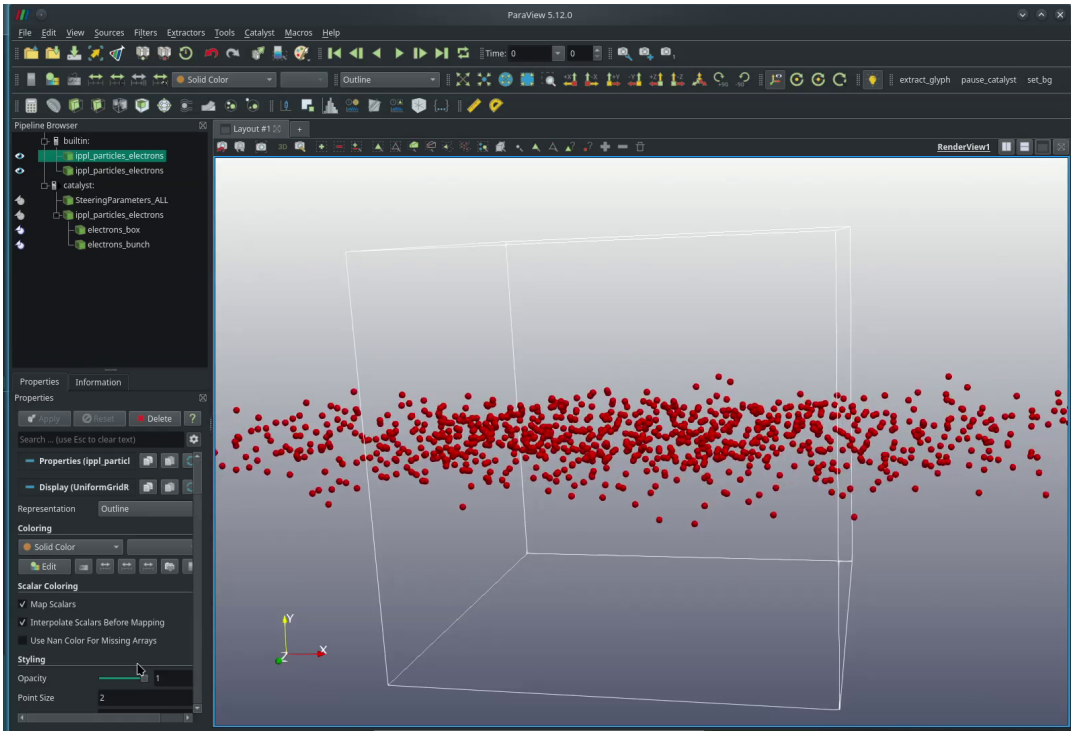


Figure 5.2: *ParaView GUI visualizing electrons traversing a FODO Lattice beamline. The beam is spreading excessively along the x-axis due to live modification of the focusing parameters.*

5.2 Performance

While the primary focus of this thesis was the functional implementation of the in situ framework, evaluating its performance impact is essential for determining its practical viability. To this end, we conducted a series of benchmarks designed to address five key inquiries:

1. **Overhead:** What is the temporal overhead introduced by ParaView Catalyst on the main time-integration loop?
2. **Scalability:** How does the framework scale with increased computational resources and parallelization (speedup analysis/strong scaling)?
3. **Data Scaling (Complexity):** How does the visualization overhead correlate with increasing data volume (algorithmic complexity)?
4. **Modality Comparison:** How does the performance of direct image rendering compare to full raw data extraction?
5. **Live Visualization Cost:** How does the "best-case" Live Visualization scenario compare against the baselines of file extraction and image rendering?

These metrics provide a basis for assessing usability and identifying necessary future optimizations.

5.2.1 Test Environment

All performance tests were conducted on an NVIDIA DGX A100 320GB system. The relevant hardware specifications for this environment are listed below:

- **Processors:** Dual AMD EPYC 7742 64-Core Processors
- **GPUs:** Up to 8 × NVIDIA A100-SXM4-40GB
- **Interface:** PCIe Gen 4.0 x16 (CPU ↔ GPU: 64 GB/s)
- **Interconnect:** InfiniBand HDR (up to 200 GB/s) based on Mellanox ConnectX-6
- **System Memory:** 1 TB
- **Storage (Scratch):** 4 × Samsung PM1733 NVMe SSDs (3.84 TB each).
  - Rated Sequential Read: 7.0 GB/s
  - Rated Sequential Write: 3.8 GB/s
- **RAID Configuration:** RAID 5

To establish a baseline for storage performance, we conducted a simple write speed test using `dd` in parallel across multiple MPI ranks. The command targets the scratch directory, creating distinct files for each rank to simulate distributed writing:

```
mpirun -np <n> \
    bash -c "dd if=/dev/zero of=/scratch/test_rank_${OMPI_COMM_WORLD_RANK}.img \
    bs=4M count=1280 oflag=direct"
```

Listing 5.1: *Storage write speed benchmark command*

Based on this benchmark, the effective aggregate write throughput was observed to be approximately **800 MB/s**, a figure that remained consistent across varying rank counts (n).

For the simulation benchmarks, we adopted a hybrid resource allocation strategy. The IPPL simulation logic is executed entirely on the CPUs sequentially with distributed memory parallelism (no shared memory parallelism), while the visualization workload is offloaded to the GPUs. This was achieved by utilizing a custom build of ParaView linked against EGL (Embedded-System Graphics Library). This configuration allows for hardware-accelerated off-screen rendering without the overhead or requirement of a windowing system (X server).

5.2.2 Input/Output Analysis for the Penning Trap application

Determining the performance "break-even" point for in situ visualization is highly problem-dependent. To establish a baseline for our framework, we analyzed the **IPPL Penning Trap** mini-application. By calculating the theoretical data volume per timestep and applying the known write throughput of our test environment (approx. 800 MB/s, see Section 5.2.1), we can predict the minimum I/O cost (throttling) that traditional file-based extraction would impose.

We assume the following data structures per grid cell. Ghost data is omitted from these calculations as its extraction can be suppressed during output.

Scalar Field:

- 1 × Scalar: `double` (8 B)
- 1 × Rank: `int32` (4 B)
- **Total: 12 bytes/cell**

Vector Field:

- 3 × Comp.: `double` (24 B)
- 1 × Rank: `int32` (4 B)
- **Total: 28 bytes/cell**

Particles

- 1 × ID: `int64` (8 B)
- 3 × Pos: `double` (24 B)
- 3 × Vel: `double` (24 B)
- 3 × E-field: `double` (24 B)
- 1 × Charge: `double` (8 B)
- 1 × Rank: `int32` (4 B)
- 1 × Connectivity: `int64` (8 B)
- **Total: 100 bytes/particle**

Following the IPPL scaling studies by Muralikrishnan et al. [26], we assume a conservative density of 8 particles per grid cell. For small grid sizes, the overhead of in situ visualization may exceed the cost of simply writing files to disk. Therefore, we focus our analysis on moderately large domains, starting with 128^3 nodes. As shown in Table 5.1, particle data dominates the storage requirements. A single timestep for a 128^3 grid generates approximately 1.6 GB of particle data. Given our write throughput of 800 MB/s, this imposes a minimum stall of 2 seconds per timestep. As resolution increases, this I/O bottleneck becomes severe. For a 1024^3 grid, a single timestep would require nearly 1 TB of storage and over 15 minutes to write, rendering high-frequency output impossible.

		Scalar Field		Vector Field		Particles	
Grid (N^3)	**Nodes**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**
128^3	$\approx 2 \times 10^6$	25.2 MB	≈ 0.03 s	58.7 MB	≈ 0.07 s	1.68 GB	≈ 2.1 s
256^3	$\approx 1.6 \times 10^7$	201.3 MB	≈ 0.25 s	469.8 MB	≈ 0.59 s	13.4 GB	≈ 16.8 s
512^3	$\approx 1.3 \times 10^8$	1.61 GB	≈ 2.01 s	3.76 GB	≈ 4.70 s	107.4 GB	≈ 2.2 min
1024^3	$\approx 1.1 \times 10^9$	13.0 GB	≈ 16.3 s	30.4 GB	≈ 38.0 s	859.1 GB	≈ 18 min

Table 5.1: *Projected I/O write times for the PenningTrap example. Times assume an aggregate write speed of 800 MB/s. Particle data assumes 8 particles per cell.*

Experimental Limits: In our benchmarks, the maximum feasible grid size was 256^3, as larger configurations exceeded the available memory on the DGX A100. In general resulting VTK data files tended to be slightly larger than the here calculated sizes.

Particles vs. Fields: The comparison in Table 5.1 highlights the stark contrast between field data and particle data. While saving a scalar field is relatively inexpensive (approx. 200 MB for 256^3), full particle extraction provides maximum fidelity but is computationally prohibitive at scale. Traditional workflows often sacrifice this fidelity (e.g., by only saving scalar reductions) to maintain performance. In situ visualization offers a crucial alternative trade-off: it allows for the analysis of full-fidelity particle data without incurring the massive I/O penalty.

5.2.3 ParaView Catalyst Performance Analysis

In this section, we analyze the performance characteristics of ParaView Catalyst, specifically focusing on the processing and storage of IPPL particle data

VTK File Extraction

We first analyze the scenario where Catalyst extracts and writes full VTK particle data. Given the large data volumes calculated in Section 5.2.2, we expect I/O throttling to be clearly visible.

Figure 5.3 (left) displays the execution time IPPL spends within the internal `catalyst_execute()` call. Figure 5.3 (right) translates this time into effective write throughput based on the known data size. We observe distinct behavior across scales:

- **Small Grid Sizes:** ParaView's internal overhead dominates, significantly diminishing the effective write speed.
- **Large Grid Sizes:** The write speed improves but eventually stalls, flattening out after reaching approximately 400 MB/s, roughly half of the practical maximum disk bandwidth (800 MB/s) established in our benchmarks. We are unsure what exactly causes this threshold.

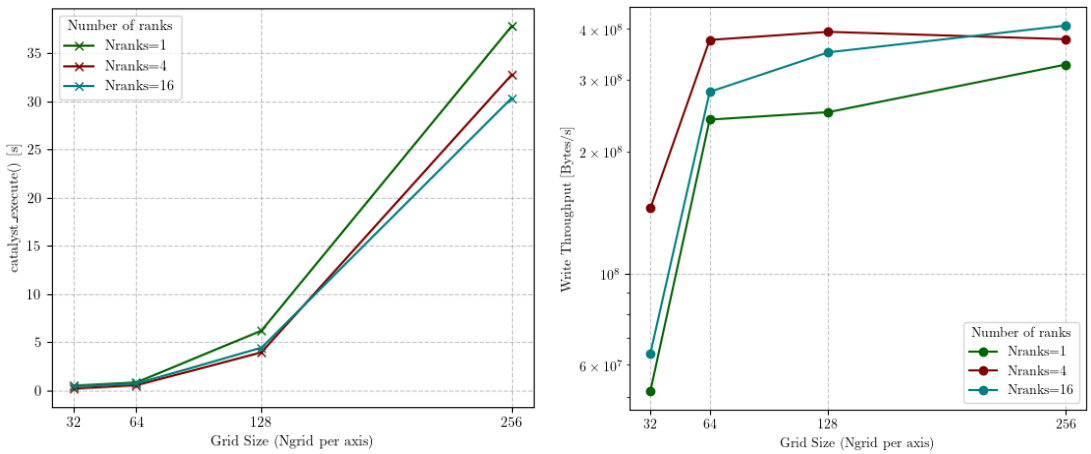


Figure 5.3: *Catalyst Performance during VTK extraction. Left: Execution time of the* `catalyst_execute()` *call. Right: Effective write throughput to disk.*

In Figure 5.4, we examine the time breakdown between the simulation (Green) and the visualization/I/O (Blue). It is evident that storing raw data massively impacts the total runtime across all grid sizes. Furthermore, the I/O component scales poorly compared to the simulation itself. As the system becomes more parallelized (higher rank counts), the simulation computation accelerates significantly, whereas the distributed writing process achieves a much lower speedup. Consequently, for parallel runs, I/O occupies an increasingly disproportionate share of the total runtime. We can also note that our C++ Adaptor implementation (Red and Teal) has a negligible impact on the overall time integrator execution compared to the heavy `catalyst_execute()` cost.

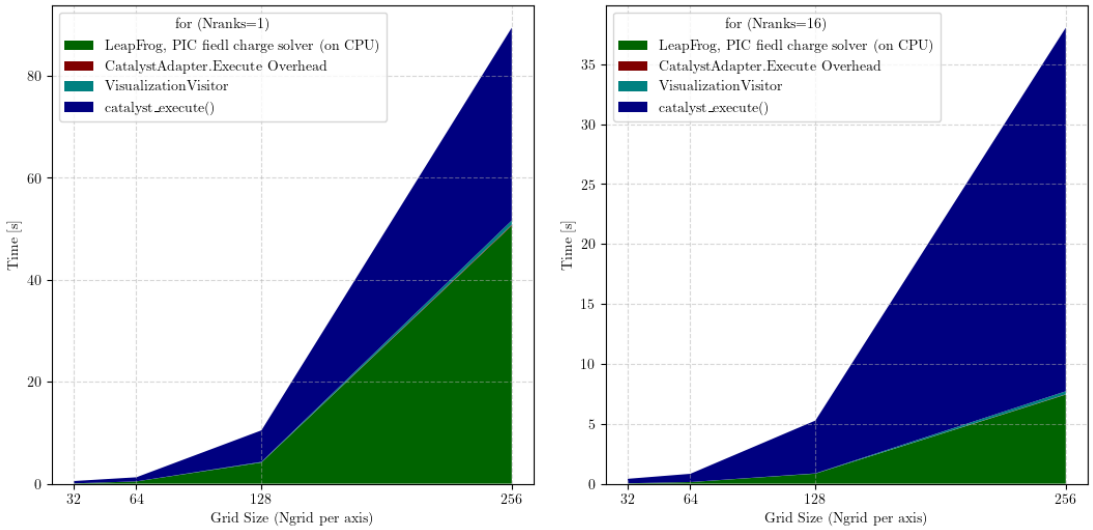


Figure 5.4: *Time breakdown for a time step advance with VTK extraction (Left: 1 Rank, Right: 16 Ranks). The blue area represents the massive cost of I/O.*

Comparative Analysis

Next, Figure 5.6 presents a broad comparison of execution times (left) and corresponding data throughput (right) for MPI ranks 1, 4, and 16. We compare the following distinct configurations:

No Catalyst Actions: Catalyst is enabled, but no pipelines (PNG, VTK, Live, or Steering) are active. This isolates the baseline overhead of the Catalyst API call passing particle data.

LIVE, no Viz: Live visualization is enabled, but no client is connected. This represents the technical "ready-state" overhead, showing the upper performance limit for utilizing live visualization or steering when no data transfer is actively requested.

PNG, 1 GPU: We allocate a single GPU for ParaView Catalyst to render particle data and save a PNG image.

PNG, 4 GPU: Identical to the above, but allocating 4 GPUs for rendering to test rendering scalability.

VTK Data: We extract raw VTK data and write it to disk, serving as the I/O-bound baseline discussed previously.

Performance Discrepancy in Image Extraction

The results presented in Figure 5.6 reveal an unexpected performance bottleneck in the PNG extraction pipeline. While the baseline Catalyst API overhead ("No Catalyst Actions") is two orders of magnitude lower than the active pipelines, the PNG extraction configuration performs surprisingly poorly, usually worse than direct disk I/O ("VTK Data").

Several key observations suggest that this is not a simple resource limitation:

- **Problem Sizes is large enough:** The performance trend for PNG extraction flattens out for large grid sizes in a manner strikingly similar to the disk I/O. The trends in these plots

seem don't indicate in any way that PNG extraction could out scale VTK extraction for larger problem sizes, suggesting so a data throughput limit due to bandwith or resource limitation.

- **Lack of GPU Scaling:** However, shown in Figure 5.5, increasing the rendering resources from 1 to 4 GPUs yields no performance benefit. If the bottleneck were rendering speed or single-GPU memory bandwidth, distributing the workload across four devices should have improved throughput. Also the unidirectional PCIe (Peripheral Component Interconnect Express) transfer rates of 32 GB/s, should easily accommodate the particle data. This absence of scaling indicates the bottleneck lies elsewhere.
- **GPU Idle Time:** To investigate further, we monitored GPU utilization using `nvtop` (see Figure F.1). We observed that during the `catalyst_execute` call, the GPUs remain idle for a significant duration. This is followed by a very brief spike (< 1s) where the image is actually generated.

The extended idle period combined with the lack of multi-GPU scaling strongly suggests that Catalyst is performing extensive CPU-side preprocessing or data marshalling before the data is ever sent to the GPU. We hypothesize that an inefficient internal data conversion or synchronization step within the Catalyst/VTK pipeline is acting as a serial bottleneck.

Future Optimization Strategy: To resolve this, future work must focus on:

1. **Configuration Review:** Re-examining the Conduit Node descriptions and Catalyst script settings to ensure we are not inadvertently triggering expensive data duplication, format conversion, or similar.
2. **Profiling:** Utilizing tools like NVIDIA Nsight Systems to trace execution flow and pinpoint the specific processes responsible for the latency.
3. **Consultation:** Engaging with Kitware support to determine if this behavior stems from a known issue or configuration anti-pattern.

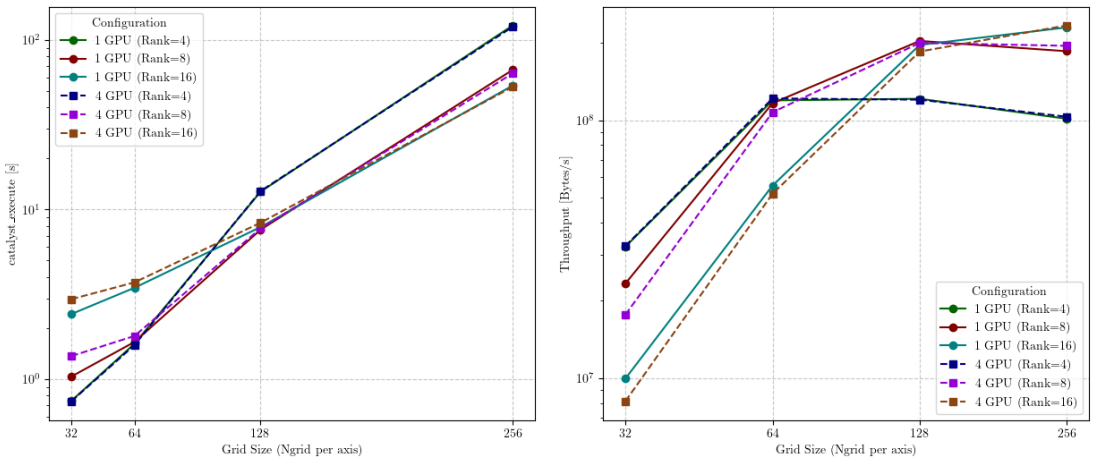


Figure 5.5: *Impact of GPU allocation on visualization performance. The complete overlap of lines indicates that allocating more GPU resources (4 vs 1) has no impact on current performance.*

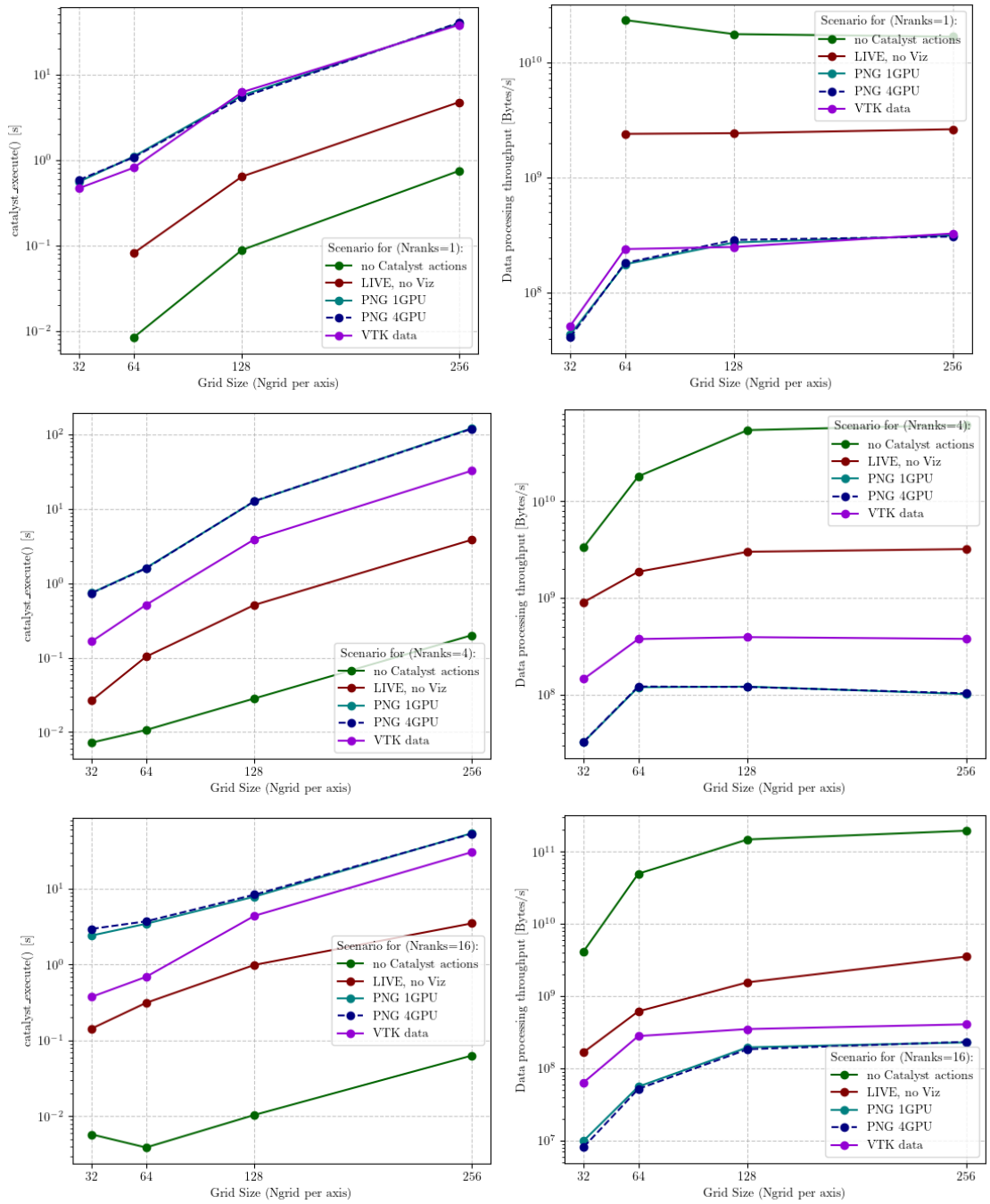


Figure 5.6: *Performance comparison of different ParaView Catalyst configurations for particle data processing (Rows: 1, 4, and 16 Ranks).*

Chapter 6

Conclusion and Future Work

In this thesis, we successfully developed a flexible and user-friendly in situ analysis framework for IPPL capable of image extraction, Live Visualization, and parameter steering. The resulting architectural structure is illustrated in Figure 6.1. Despite these achievements, the current implementation is not without limitations and requires further optimization to reach full production quality. However, we envision this work serving as a foundational stepping stone towards a complete, efficient, and capable in situ infrastructure. The establishment of this core architecture ensures that subsequent feature integration and functional expansions are possible with a reduced development cost. The remainder of this chapter provides an overview of necessary improvements and discusses potential extensions to the system.

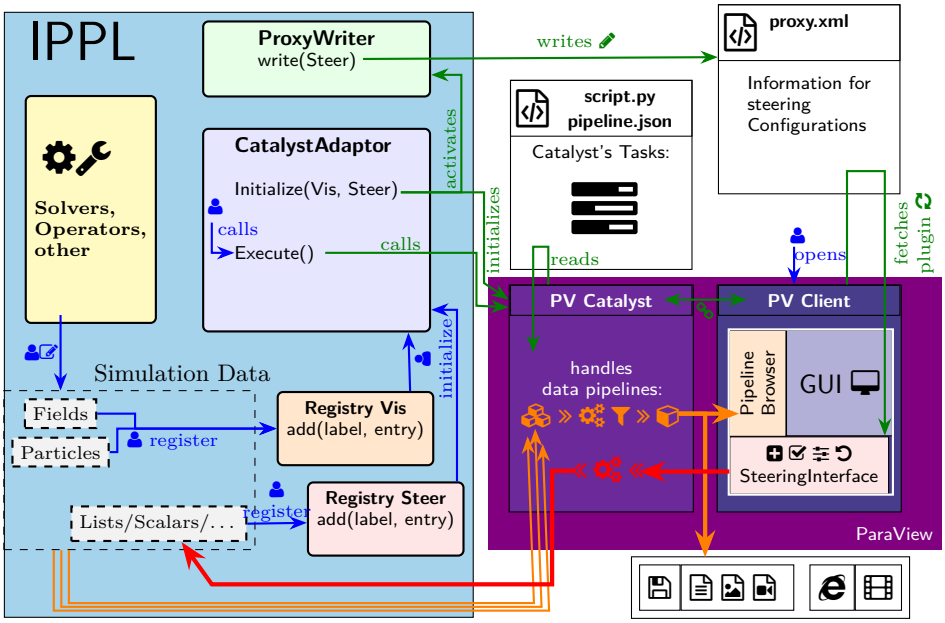


Figure 6.1: *The flow of control and information of the IPPL in situ framework. The figure displays needed user input/implementation (blue) internal function calls and methods (green) and the flow of data (orange/red).*

6.1 The Immediate Future for IPPL In Situ

Performance Optimization

As discussed in Section 5.2.3, the current implementation of the ParaView Catalyst backend introduces excessive latency, stalling the simulation significantly during image extraction. A detailed analysis to identify and rectify the source of this bottleneck is a priority.

Unit Testing

Unit testing is a cornerstone of robust software development. While in situ workflows present unique challenges for for defining meaningful tests, ensuring data integrity is critical. We propose implementing a test suite that explicitly leverages ParaView Catalyst's native verification capabilities. Specifically, we can integrate the conduits built-in verify functions (checking te validity of node structures) into our unit tests. This allows us to automatically validate that every Conduit Node produced by the `CatalystAdaptor` strictly conforms to the official Catalyst Blueprint protocol. Such a test suite would ensure that even when the adaptor is fed randomized or edge-case simulation objects, the resulting output remains schema-compliant and valid.

Refactoring the Registry Class

The current approach, which utilizes two identically typed `Registry` instances for distinct purposes (Visualization and Steering), is architecturally clumsy. To improve code clarity and safety, this design should be refactored. Potential solutions include:

- **Unification:** Merging the functionality into a single class that handles both domains internally.
- **Inheritance:** Deriving distinct `SteeringRegistry` and `VisualizationRegistry` subclasses to enforce type safety and separation of concerns.

Dimension Independence

A core feature of IPPL is its dimension-agnostic design, supporting 1D, 2D, and 3D domains for many of it's features. The `CatalystAdaptor` was implemented with this flexibility in mind, attempting to support arbitrary dimensions generically. However, validation has thus far been limited to 3D scenarios. Future work must verify and ensure full functionality for 1D and 2D test cases.

Parameter Tracking

Macro-State Tracking

Most simulations compute global scalar quantities (macro-states) such as total energy, emittance, or particle count—that characterize the system's behavior over time. Additionally to visualizing 1,2, and 3 dimensional data, we could further allow in situ analysis of scalar data, allowing transmission of these scalars to the visualization pipeline, to plot the temporal evolution of these states in a live graph or diagram, providing immediate feedback on the simulation's stability and progress.

Steering Parameter Tracking

Currently, parameter changes made during a steering session are not recorded in any way. This creates a reproducibility gap: if a user creates a unique result by dynamically altering parameters, there is no record of the specific values used to generate this specific outcome. To address this, it is essential to implement automatic tracking of steering parameter history. This mechanism should function transparently, without requiring manual intervention from the IPPL user. Furthermore, in a Live Visualization session, these steering parameters could be plotted alongside the macro-state data.

Trame UI

As previously noted, the current Trame-based Web UI functions as a proof-of-concept prototype and lacks robustness. Further effort is still required to transform this interface into a more functional stable, and reliable service. Developing a robust, optimized, and user-friendly GUI is a critical step, as the accessibility of the interface can contribute to the general interest in the in-situ framework's capabilities.

Native IPPL Labeling

Currently, the registration process requires users to manually assign string names to objects. However, IPPL is expected to evolve to support native labels for high-level structures like `ParticleContainer` and `Field` objects, similar to the labeling system recently introduced for Particle Attributes. Once these internal labels are standardized, the `VisualizationRegistry` can be refactored to rely on these internal labels instead.

Advanced Runtime Configuration

We must further develop the runtime configuration system to enable comprehensive customization via a run script. In an ideal scenario, an IPPL user should need to instrument their simulation only once, registering all potential candidates for visualization and steering. The actual utilization of these parameters—such as specific channel activation, PNG extraction, VTK output, or passing data to Conduit—should then be fully configurable at runtime. While environment variables provide a straightforward and easily extensible mechanism for this purpose, they may become unwieldy as complexity grows. Before fully committing to this approach for the production interface, we must critically evaluate whether a more structured configuration method (e.g., JSON or YAML configuration files) would offer a superior user experience.

6.2 OPALX In-Situ Instrumentation

Runtime configuration is particularly critical for OPALX, as the software is architected to be compiled once and then deployed across a wide variety of simulation scenarios using input files. We must devise a methodology to integrate IPPL's visualization and steering utilities into the OPAL core that respects this workflow. This will likely require either extending the existing beamline input file specification to support visualization and steering directives or establishing an OPALX specific extension to the environment configuration already used in IPPL to ensure flexibility without requiring recompilation.

6.3 Alternative Processing Paradigms

Zero-Copy In Situ

In this thesis, we have outlined the theoretical basis for a zero-copy implementation. To enhance versatility, in the the framework should support dynamic switching between deep-copy and zero-copy modes. A critical area for investigation is the handling of zero-copy visualization on GPU devices, specifically regarding the exposure of device pointers directly to the visualization engine without intermediate host staging. A successful implementation of this approach is expected to yield the highest possible performance by eliminating redundant memory transfers.

Alternative In Situ Framework: Ascent

Since our implementation already utilizes the Conduit Mesh Blueprint for data description, the groundwork for integrating Lawrence Livermore National Laboratory's Ascent framework is largely complete. Ascent, like Catalyst, natively supports Conduit-based data sharing. From an architectural standpoint, we propose introducing an abstract base class (following a Strategy pattern) for the Visualization Adaptor. This would decouple the specific backend implementation from the IPPL core, allowing users to switch transparently between ParaView Catalyst, Ascent, or other in situ backends while maintaining a unified interface.

In Transit and Hybrid Visualization

Kitware is actively developing ADIOS Catalyst, which extends the Catalyst API to support in transit and hybrid workflows via the ADIOS2 library. Since ADIOS Catalyst is built directly upon the ParaView Catalyst infrastructure, transitioning our current in situ implementation to an in transit setup would require minimal refactoring. Hybrid workflows offer significant advantages, most notably the decoupling of simulation and visualization resources. This allows users to allocate specific computational resources (e.g., dedicated visualization nodes) independently of the simulation scaling, providing fine-grained control over the performance balance between computation and analysis.

6.4 IPPL and Machine Learning

In the future an extended In-Situ framework might help overcome an I/O bottleneck for machine learning based physics models, transforming simulations into real-time, in-memory data generators. Leveraging HPC resources, it enables the continuous online retraining of neural surrogates (e.g., CNNs, Invertible NNs) directly from complex physics kernels whose output is too grand to serialize. This also facilitates "real-time evolution" of models and Bayesian Model Calibration, allowing Digital Twins to dynamically align with experimental data during runtime.

Appendix A

IPPL Source Code

A.1 VisRegistry

Discussed in Section 4.2.2.

```
1
2  namespace detail {
3      inline void add_pairs(VisRegistryRuntime&) {}
4      template<class L, class V, class... Rest>
5      void add_pairs(VisRegistryRuntime& r, L&& label, V&& value, Rest&&... rest) {
6          std::string lbl{std::forward<L>(label)};
7          if constexpr (std::is_lvalue_reference_v<V&&>) {
8              r.add(lbl, value);
9          }
10         if constexpr (sizeof...(Rest) > 0) {
11             static_assert(  sizeof...(Rest) % 2 == 0,
12                             "Factory arguments must be label-value pairs");
13             add_pairs(r, std::forward<Rest>(rest)...);
14         }
15     }
16 } // namespace detail
17
18 template<class... Args>
19 VisRegistryRuntime MakeVisRegistry(Args&&... args) {
20     static_assert(  sizeof...(Args) % 2 == 0,
21                     "MakeVisRegistry requires label-value pairs");
22     VisRegistryRuntime reg;
23     if constexpr (sizeof...(Args) > 0) {
24         detail::add_pairs(reg, std::forward<Args>(args)...);
25     }
26     return reg;
27 }
28
29 template<class... Args>
30 std::shared_ptr<VisRegistryRuntime> MakeVisRegistryPtr(Args&&... args) {
31     static_assert(  sizeof...(Args) % 2 == 0,
32                     "MakeVisRegistryPtr requires label-value pairs");
33     auto reg = std::make_shared<VisRegistryRuntime>();
34     if constexpr (sizeof...(Args) > 0) {
35         detail::add_pairs(*reg, std::forward<Args>(args)...);
36     }
37     return reg;
```

Listing A.1: *The* `ippl::VisRegistry` *factory function*

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <variant>
#include <concepts>
#include <array>
#include<functional>
template<typename T> concept Arithmetic = std::is_arithmetic_v<T>;

class Adaptor {public:
    // --- Method A ---
    template <std::integral T, std::size_t D>
    void method_a(std::array<T, D>& a
    {  std::cout << "  [A] Arr<Int," << D << ">\n"; for(auto& v : a) v += 10; }
    template <std::floating_point T, std::size_t D>
    void method_a(std::array<T, D>& a)
    {  std::cout << "  [A] Arr<Flt," << D << ">\n"; for(auto& v : a) v += 1.5; }
    void method_a(Arithmetic auto& e) { std::cout << "  [A] Scalar\n"; e += 5; }
    void method_a(auto&) { std::cout << "  [A] Other\n"; }
    // --- Method B ---
    template <std::integral T, std::size_t D>
    void method_b(std::array<T, D>& a)
    {   std::cout << "  [B] Arr<Int," << D << ">\n"; for(auto& v : a) v *= 2;   }
    template <std::floating_point T, std::size_t D>
    void method_b(std::array<T, D>& a)
    {   std::cout << "  [B] Arr<Flt," << D << ">\n"; for(auto& v : a) v *= 2.0; }
    void method_b(Arithmetic auto& e) { std::cout << "  [B] Scalar\n"; e *= 2;  }
    void method_b(auto&) { std::cout << "  [B] Other\n"; }

    struct VisitorB{ Adaptor& a;
                     void operator()(const std::string&, auto& v){ a.method_b(v); }  };
    struct VisitorA{ Adaptor& a;
                     void operator()(const std::string&, auto& v){ a.method_a(v); } };
    using VisitorVariant = std::variant<VisitorA, VisitorB>;
};
class Registry {public:
    struct Entry {  std::string label;
                    std::function<void(Adaptor::VisitorVariant&)> dispatch;
    };
    std::vector<Entry> entries_;
    void add(const std::string& l, auto& v) {
        entries_.push_back({ l, [l, &v](Adaptor::VisitorVariant& var) {
            std::visit( [&](auto& active){active(l,v);}, var);
    }});}
    void forEach(auto&& vis) {
        Adaptor::VisitorVariant var = std::forward<decltype(vis)>(vis);
        for (auto& e : entries_) e.dispatch(var);
    }
    void forOne(const std::string& target, auto&& vis) {
        Adaptor::VisitorVariant var = std::forward<decltype(vis)>(vis);
        for (auto& e : entries_) {
            if (e.label == target) {
                e.dispatch(var);
                return;
        }}  std::cout << "  [!] Label '" << target << "' not found.\n";  }
};

template <typename T, size_t N>
void print_array(const std::array<T,N>& a) {for(auto v : a)std::cout<< v << " ";}

int main() {
    Adaptor a; Registry r;
    int i = 1; double d = 1.0; std::string s = "test";
    std::array<int, 2> ai = {10, 20};
    std::array<double, 2> ad = {1.1, 2.2};

    r.add("i", i); r.add("d", d); r.add("ai", ai); r.add("ad", ad); r.add("s", s);

    auto log = [&]() {
      std::cout << " i=" << i << " d=" << d << " ai="; print_array(ai); std::cout << "\n";
    };
    std::cout << "INIT:\n"; log();
    std::cout << "\n forEach(A)       \n"; r.forEach(     Adaptor::VisitorA{a});  log();
    std::cout << "\n forOne('i', B):  \n"; r.forOne("i",  Adaptor::VisitorB{a});  log();
    std::cout << "\n forOne('ai', B)  \n"; r.forOne("ai", Adaptor::VisitorB{a});  log();
}
```

Listing A.2: *Self contained Demo, showcasing Registry and Adapter interaction*

A.2 Meta Programming Helpers

```cpp
// Check if T inherits from the IPPL particle base class
template<class T>
inline constexpr bool is_particle_v = std::is_base_of<ippl::ParticleBaseBase, typename std::
    decay<T>::type>::value;

// Type traits to identify IPPL Field instantiations
template<class T>
struct is_field : std::false_type {};

template<class V, unsigned Dim, class... Rest>
struct is_field<ippl::Field<V, Dim, Rest...>> : std::true_type {};

template<class T>
inline constexpr bool is_field_v = is_field<std::decay_t<T>>::value;
// Type traits to identify IPPL Vector (mathematical small vector)
template<typename U>
struct is_ippl_vector : std::false_type {};
template<typename V, unsigned Dim>
struct is_ippl_vector<ippl::Vector<V, Dim>> : std::true_type {};
template<typename T>
inline constexpr bool is_ippl_vector_v = is_ippl_vector<std::decay_t<T>>::value;
// Helpers to validate the contents of std::vector containers
// (Scalar types, IPPL Vectors, or generic vectors for struct arrays)
template<typename U>
struct is_std_vector_of_allowed_scalar : std::false_type {};

template<typename U>
struct is_std_vector_of_allowed_scalar<std::vector<U>> : std::bool_constant<
    std::is_arithmetic_v<std::decay_t<U>>
    || std::is_enum_v<std::decay_t<U>>
    || std::is_same_v<std::decay_t<U>, bool>
    || std::is_same_v<std::decay_t<U>, Button>> {};

template<typename U>
struct is_std_vector_of_ippl_vector : std::false_type {};
template<typename U>
struct is_std_vector_of_ippl_vector<std::vector<U>> : std::bool_constant<is_ippl_vector_v<U>>
    {};

template<typename U>
struct is_std_vector_any : std::false_type {};
template<typename U>
struct is_std_vector_any<std::vector<U>> : std::true_type {};

// Whitelist for types that can be modified (steered) via the UI
template<class T>
inline constexpr bool AllowedSteerType_v =
    std::is_arithmetic_v<std::decay_t<T>>
    || std::is_enum_v<std::decay_t<T>>
    || std::is_same_v<std::decay_t<T>, Button>
    || is_ippl_vector_v<T>
    || is_std_vector_of_allowed_scalar<std::decay_t<T>>::value
    || is_std_vector_of_ippl_vector<std::decay_t<T>>::value;

// Whitelist for heavy data types supported by the visualization pipeline
template<class T>
inline constexpr bool AllowedVisType_v = is_particle_v<T> || is_field_v<T>;

// Unified constraint for all types capable of being registered
template<class T>
inline constexpr bool AllowedRegistryType_v =  AllowedVisType_v<T>
                                            || AllowedSteerType_v<T>
                                            || is_std_vector_any<std::decay_t<T>>::value;
// Traits to support shared_ptr wrappers around any valid registry type
template<class T>
struct is_allowed_shared_ptr : std::false_type {};

template<class U>
struct is_allowed_shared_ptr<std::shared_ptr<U>> : std::bool_constant<AllowedRegistryType_v<U
    >> {};

template<class T>
inline constexpr bool AllowedRegistryTypeOrShared_v =
    AllowedRegistryType_v<T> || is_allowed_shared_ptr<std::decay_t<T>>::value;
```

Listing A.3: *Fundamental type traits for detecting IPPL Objects and allowed registry entries.*

A.3 The View Registry

Discussed in Section 4.3.5.

```
namespace ippl{ class ViewRegistry {
private:
    std::unordered_map<std::string, std::any> m_storage;
    size_t m_unnamed_counter = 0;
public:
    template<typename T>
    void set(const std::string& name, T object);

    template<typename T>
    std::string set(T object);

    template<typename T>
    std::shared_ptr<T> get(const std::string& name) const;

    void unset(const std::string& name);
    void clear();
    size_t size() const {     return m_storage.size();    }
    bool contains(const std::string& key) const {    return m_storage.contains( key );    }
};}
```

Listing A.4: *The* `ViewRegistry` *implementation*

A.4 Initialization and Execution

The source code for the Catalysts's Adaptirs Initialization and Execution method were discussed and previously shown in Section 4.3.4 and 4.3.5.

A.5 Data Publishing for Particle Containers

Discussed in section 4.3.6.Particle Data.

A.5.1 Multimesh Setup and Position/Coorindate Data

```
template<typename T>
requires (std::derived_from<std::decay_t<T>, ParticleBaseBase>)
void CatalystAdaptor::ExecVizChannel(const T& entry, const std::string label)
{
    // SETTING UP MULTIMESH STRUCTURE
    const std::string channelName = "ippl_particles_" + label;
    auto particleContainer = &entry;
    auto channel = node["catalyst/channels/"+ channelName];
    channel["type"].set_string("multimesh");

    auto data =     channel["data/block_main"];
    auto data_help =     channel["data/block_help"];
    channel["assembly/main"] = "block_main";
    channel["assembly/help"] = "block_help";

    // SETUP MESH STRUCTURE OF MAIN BLOCK
    auto fields = data["fields"];
    data["type"].set_string("mesh");

    // STAGING DEEP COPIED POSITION DATA FOR VISUALIZATION
    using RAttrib_t  = std::remove_reference_t<decltype(particleContainer->R)>;
    using hostMirror_R_t  = typename RAttrib_t::HostMirror;
    hostMirror_R_t  R_hostMirror = particleContainer->R.getHostMirror();
    Kokkos::deep_copy(R_hostMirror ,  particleContainer->R.getView());
    viewRegistry.set(R_hostMirror);

    using R_elem_t = std::remove_pointer_t<decltype(R_hostMirror.data())>;
    static constexpr size_t R_stride_bytes = sizeof(R_elem_t);

    /* EXPLICIT COORDINATES DEFINITION */
    data["coordsets/p_explicit_coords/type"].set_string("explicit");
    auto cvalues = data["coordsets/p_explicit_coords/values"]
    cvalues["x"].set_external(  &R_hostMirror.data()[0][0],
                                particleContainer->getLocalNum(),
                                0, R_stride_bytes);
    cvalues["y"].set_external(  &R_hostMirror.data()[0][1],
                                particleContainer->getLocalNum(),
                                0, R_stride_bytes);
    cvalues["z"].set_external(  &R_hostMirror.data()[0][2],
                                particleContainer->getLocalNum(),
                                0, R_stride_bytes);
    /* POSITION ATTRIBUTE */
    auto R_field = fields["position"];
    R_field["association"].set_string("vertex");
    R_field["topology"].set_string("p_unstructured_topo"); // defined later on
    R_field["volume_dependent"].set_string("false");
    R_field["values/x"].set_external(&R_hostMirror.data()[0][0],
                                     particleContainer->getLocalNum(),
                                     0, R_stride_bytes);
    R_field["values/y"].set_external(&R_hostMirror.data()[0][1],
                                           particleContainer->getLocalNum(),
                                           0, R_stride_bytes);
    R_field["values/z"].set_external(&R_hostMirror.data()[0][2],
                                           particleContainer->getLocalNum(),
                                           0, R_stride_bytes);
    /* ITERATION OVER CUSTOM ATTRIBUTES */
    const size_t n_attributes =  entry.getAttributeNum();
    for(size_t i = 2; i < n_attributes; ++i){
        const auto attribute = entry.getAttribute(i);
        attribute->signConduitNode( localNum, fields, viewRegistry,
                                    ca_m, ca_warn, forceHostCopy[label]);
    } ...
```

Listing A.5: *Publishing Particle position data to the Conduit Node and setting up Conduit's explicit mesh structure (simplified)*

A.5.2 Meta Data Attributes and Topology

```
...

        /* Global ID ATTRIBUTE: Check existence */
        if( particleContainer->EnableIDs)
        {
            using IDAttrib_t = std::remove_reference_t<decltype(particleContainer->ID)>;
            hostMirror_ID_t ID_hostMirror ;
            using hostMirror_ID_t = typename IDAttrib_t::HostMirror;
            ID_hostMirror = particleContainer->ID.getHostMirror();
            Kokkos::deep_copy(ID_hostMirror,  particleContainer->ID.getView());
            viewRegistry.set(label, ID_hostMirror);

            auto id_field = fields["ParticleIDs"];
            id_field["association"].set_string("vertex");
            id_field["topology"].set_string("p_unstructured_topo");
            id_field["volume_dependent"].set_string("false");
            id_field["values"].set_external(ID_hostMirror.data(), localNum);
            data["metadata/vtk_fields/ParticleIDs/attribute_type"].set_string("GlobalIds");
        }

        // CREATING DATA FOR PROCESS ID AND IOTA ARRAY FOR TOPOLOGY

        using HostExecSpace = Kokkos::DefaultHostExecutionSpace;
        const size_t localNum = particleContainer->getLocalNum();
        Kokkos::RangePolicy<HostExecSpace> host_policy(0, localNum);


         /* UNSTRUCTURED TOPOLOGY RELYING ON LOCAL PARTICLe ASSIGNMENT */
        using IotaView_t = Kokkos::View<int64_t*, Kokkos::HostSpace>;
        IotaView_t iota_view("iota", localNum);
        Kokkos::parallel_for("fill_iota_host", host_policy,
          KOKKOS_LAMBDA(const int64_t i) {
            iota_view(i) = i;
        });
        viewRegistry.set(label + "_iota", iota_view);
         auto topo = data["topologies/p_unstructured_topo"]
        topo["coordset"].set_string("p_explicit_coords");
        topo["type"].set_string("unstructured");
        topo["elements/shape"].set_string("point");
        topo["elements/connectivity"].set_external( iota_view.data(),localNum);


        /* Process ID ATTRIBUTE */
        const int rank = ippl::Comm->rank();
        using RankView_t = Kokkos::View<int*, Kokkos::HostSpace>;
        RankView_t rank_id_view("rank_id_view", localNum);
        Kokkos::parallel_for("fill_rank_ids", host_policy,
          KOKKOS_LAMBDA(const int64_t i) {
            rank_id_view(i) = rank;
        });
        auto rank_field = fields["RankID"]; // You can name this anything
        rank_field["association"].set_string("vertex");
        rank_field["topology"].set_string("p_unstructured_topo");
        rank_field["volume_dependent"].set_string("false");
        rank_field["values"].set_external(rank_id_view.data(), localNum);
        data["metadata/vtk_fields/RankID/attribute_type"].set_string("ProcessIds");
        viewRegistry.set(label + "_rank_id", rank_id_view);


...
```

Listing A.6: *Publishing particle meta data to the Conduit Node and defining Conduits unstructured topology (simplified)*

A.5.3 Domain Box

```
...
    // SETUP MESH STRUCTURE OF HELPER BLOCK
    using PLayout_t = T::Layout_t;
    // 1. CHECK IF PLAYOUT IS A SPATIAL LAYOUT (else no spatially defined bounding box)
    if constexpr (has_getRegionLayout_v<PLayout_t>){
    // 2. RETRIEVE LAYOUT INFORMATION
        using RLayout_t  = PLayout_t::RegionLayout_t;
        using NDRegion_t = RLayout_t::NDRegion_t;
        constexpr unsigned dim_ = PLayout_t::dim;
        const NDRegion_t ndr = particleContainer->getLayout().getRegionLayout().getDomain();

    // 3. CREATE COORDINATE SET TO PASS THE BOUNDING BOX IN VTK FORMAT
        data_help["coordsets/bound_helper_coords/type"].set_string("uniform");
        data_help["topologies/bound_helper_topo/coordset"].set_string("bound_helper_coords");
        data_help["topologies/bound_helper_topo/type"].set_string("uniform");

    // 4. CREATE UNIFROM COORDINATE MESH ONLY CONSISTING OF THE CORNER POINTS OF THE DOMAIN
        auto helper = data_help["coordsets/bound_helper_coords"]
        {
            helper["dims/i"].set(2);
            helper["spacing/dx"].set( ndr[0].max()  - ndr[0].min() );
            helper["origin/x"].set(   ndr[0].min() );
        }
        if constexpr(dim_ >= 2){
            helper["dims/j"].set(2);
            helper["spacing/dy"].set( ndr[1].max()- ndr[1].min() );
            helper["origin/y"].set(   ndr[1].min()             );
        }
        if constexpr(dim_ >= 3){
            helper["dims/k"].set(2);
            helper["spacing/dz"].set( ndr[2].max()- ndr[1].min() );
            helper["origin/z"].set(   ndr[2].min()             );
        }

} //end of EcevVizChannel
```

Listing A.7: *Publishing particle domain information to the conduit Node*

A.5.4 Custom Particle Attributes

Discussed in Paragraph 4.3.6.ParticleAttributes

```
template <typename T, class... Properties>
void ParticleAttrib<T, Properties...>::signConduitNode(
        const size_type Np_local,
        conduit_cpp::Node& node_fields,
        ViewRegistry& viewRegistry,
        Inform& ca_m,
        Inform& ca_warn,
        const bool forceHostCopy
    )  const {
        HostMirror  hostMirror;
        if(forceHostCopy){
            hostMirror  = this->getHostMirror();
            Kokkos::deep_copy(hostMirror ,  this->getView());
        } else{
            hostMirror =   Kokkos::create_mirror_view_and_copy(Kokkos::HostSpace(), this->
    getView());
        }
        auto field = node_fields[this->name];
        field["association"].set_string("vertex");
        field["topology"].set_string("p_unstructured_topo");
        field["volume_dependent"].set_string("false");

    if constexpr (std::is_scalar_v<T>) {
    // --- SCALAR CASE ---
        field["values"].set_external(hostMirror.data(), Np_local);
    }
    else if constexpr (is_vector_v<T>) {
    // --- VECTOR CASE ---
        using elem_t = std::remove_pointer_t<decltype(hostMirror.data())>;
        const size_t stride_bytes = sizeof(elem_t);

        field["values/x"].set_external( &hostMirror.data()[0][0],
                                        Np_local,
                                        0,
                                        stride_bytes);
        if constexpr (T::dim>=2){
            field["values/y"].set_external( &hostMirror.data()[0][1],
                                        Np_local,
                                        0,
                                        stride_bytes);
        }
        if constexpr (T::dim>=3) {
            field["values/z"].set_external( &hostMirror.data()[0][2],
                                        Np_local,
                                        0,
                                        stride_bytes);
        }

    } else { /* --- INVALID CASE --- -> issue warning or throw exception */}
    viewRegistry.set(hostMirror);
} // signConduitNode
#endif
```

Listing A.8: `ParticleAttrib::signConduitNode` *implementation (simplified)*

A.6 Data Publishing for Fields

A.6.1 Field Mesh setup

Discussed in Paragraph 4.3.6.Fields.ConduitMeshAndTopology.

```
void CatalystAdaptor::ExecVizChannel(   const Field<T, Dim, ViewArgs...>& entry,
                                        const std::string label)

  // 1. Retrieve IPPL field information
  typename Field_type::Layout_t& Layout_ = field->getLayout();
  typename Field_type::Mesh_t&   Mesh_   = field->get_mesh();
  const auto LocalNDIndex_  = Layout_.getLocalNDIndex();
  const auto Origin_        = Mesh_.getOrigin();
  const auto h_             = Mesh_.getMeshSpacing();
  const size_t nGhost       = field->getNghost();

  // 2. Compute Origin and dimensionatlity depending on our Ghost Handling startegy
  const size_t index_offset = (use_ghost_masks) ?    size_t(nGhost)   :  0 ;
  const size_t extra        = 2*index_offset ;
  const double ox=Origin_[0]+(double(int(LocalNDIndex_[0].first())-int(index_offset)))*h_[0];
  const double oy=Origin_[1]+(double(int(LocalNDIndex_[1].first())-int(index_offset)))*h_[1];
  const double oz=Origin_[2]+(double(int(LocalNDIndex_[2].first())-int(index_offset)))*h_[2];
  const size_t nx =               LocalNDIndex_[0].length() + extra + 1    ;
  const size_t ny = (Dim >= 2) ?  LocalNDIndex_[1].length() + extra + 1 : 1;
  const size_t nz = (Dim >= 3) ?  LocalNDIndex_[2].length() + extra + 1 : 1;

  // 3. Build conduit mesh configuration
  auto fields = data["fields"];
  auto field_node = fields[label];
  data["topologies/fmesh_topo/type"].set_string("uniform");
  data["topologies/fmesh_topo/coordset"].set_string("cart_uniform_coords");
  data["coordsets/cart_uniform_coords/type"].set_string("uniform");
  data["topologies/fmesh_topo/type"].set_string("uniform");
  data["topologies/fmesh_topo/coordset"].set_string("cart_uniform_coords");
  data["coordsets/cart_uniform_coords/type"].set_string("uniform");
    {
        data["coordsets/cart_uniform_coords/dims/i"].set(    nx   );
        data["coordsets/cart_uniform_coords/spacing/dx"].set(h_[0]);
        data["coordsets/cart_uniform_coords/origin/x"].set(  ox   );
                data["topologies/fmesh_topo/origin/x"].set(  ox   );
    }
    if constexpr(Dim >= 2){
        data["coordsets/cart_uniform_coords/dims/j"].set(    ny   );
        data["coordsets/cart_uniform_coords/spacing/dy"].set(h_[1]);
        data["coordsets/cart_uniform_coords/origin/y"].set(  oy   );
        data["topologies/fmesh_topo/origin/y"].set(          oy   );
    }
    if constexpr(Dim >= 3){
        data["coordsets/cart_uniform_coords/dims/k"].set(    nz   );
        data["coordsets/cart_uniform_coords/spacing/dz"].set(h_[0]);
        data["coordsets/cart_uniform_coords/origin/z"].set(  oz   );
        data["topologies/fmesh_topo/origin/z"].set(          oz   );
    }
```

Listing A.9: *Setting up a Conduit Mesh for a* `ippl::field`

A.6.2 Fields and Meta Data

Discussed in Paragraph 4.3.6.FieldAndMetaData.

```

  /*  === Referencing field data === */
  /* In case Kokkos uses padding: Use elem_t from Kokkos information directly instead of <T>.
      Guarantees to handle padding properly. */

    using elem_t =  std::remove_pointer_t<decltype(hostMirror.data())>;
    const auto n_elems = hostMirror.size();
    static constexpr size_t stride_bytes = sizeof(elem_t);
    const size_t offset = 0;


    field_node["association"].set_string("element");
    field_node["topology"].set_string("fmesh_topo");
    field_node["volume_dependent"].set_string("false");

    // --- SCALAR FIELD CASE ---
    if constexpr (std::is_arithmetic_v<T>) {
        field_node["values"].set_external(hostMirror.data(), n_elems);
    }
    // --- VECTOR FIELD CASE ---
    else if constexpr (is_vector_v<T>) {

        field_node["values/x"].set_external(
            &hostMirror.data()[0][0], n_elems, offset,stride_bytes);

      if constexpr (T::dim>=2)
        field_node["values/y"].set_external(
            &hostMirror.data()[0][1], n_elems, offset,stride_bytes);

      if constexpr (T::dim>=3)
        field_node["values/z"].set_external(
            &hostMirror.data()[0][2], n_elems, offset,stride_bytes);
    }



    /*  === creating and referencing field rank meta data === */
    using RankViewCells_t = Kokkos::View<int***, Kokkos::HostSpace>;
    using HostExecSpace = Kokkos::DefaultHostExecutionSpace;
    RankViewCells_t rank_id_view_cells("rank_id_view_cells_3D", nx, ny, nz);

    Kokkos::MDRangePolicy<HostExecSpace, Kokkos::Rank<3>> host_policy(
        {0, 0, 0},
        {nx, ny, nz}
    );

    Kokkos::parallel_for("fill_rank_ids_3D", host_policy,
        KOKKOS_LAMBDA(const int i, const int j, const int k) {
        rank_id_view_cells(i, j, k) = rank;
    });


    conduit_cpp::Node rank_field = fields["RankID"];
    rank_field["association"].set_string("element");
    rank_field["topology"].set_string("fmesh_topo");
    rank_field["volume_dependent"].set_string("false");
    rank_field["values"].set_external(rank_id_view_cells.data(), localNumCells);
    data["metadata/vtk_fields/RankID/attribute_type"].set_string("ProcessIds");

```

Listing A.10: *Publishing* `ippl::field` *data and meta data to the Conduit node*

A.6.3 Cutting Ghost Cells

Discussed in Paragraph 4.3.6.Fields.GhostCells.Exclusion.

```
    using Field_type = Field<T, Dim, ViewArgs...>;
    typename Field_type::Layout_t& Layout_ = field->getLayout();
    typename Field_type::Mesh_t&   Mesh_   = field->get_mesh();
    const auto LocalNDIndex_  = Layout_.getLocalNDIndex();
    const auto Origin_        = Mesh_.getOrigin();
    const auto h_       = Mesh_.getMeshSpacing();
    const size_t nGhost       = field->getNghost();

    auto r0 =
            Kokkos::make_pair(nGhost, nGhost + nx);
    auto r1 = (Dim >= 2)
            ? Kokkos::make_pair(nGhost, nGhost + ny)
            : Kokkos::make_pair(size_t(0), fullDeviceView.extent(1));
    auto r2 = (Dim >= 3)
            ? Kokkos::make_pair(nGhost, nGhost + nz)
            : Kokkos::make_pair(size_t(0), fullDeviceView.extent(2));

    auto deviceSubView = Kokkos::subview(fullDeviceView, r0, r1, r2);
    HostView_t hostMirror = HostView_t("hostMirrorNoGhosts_LayoutLeft", nx, ny, nz);
    Kokkos::deep_copy(hostMirror, deviceSubView);
```

Listing A.11: *Cutting Ghost Cells out from data array during* `Kokkos::deep_copy`

A.6.4 Masking Ghost Cells

Discussed in Paragraph 4.3.6.Fields.GhostCells.Masking.

```
    using m_t = unsigned char;
    using DeviceMaskView_t = typename Field<m_t, Dim, ViewArgs...>::view_type;
    // STORAGE
    using HostMaskView1D_t = Kokkos::View<m_t*, Kokkos::LayoutLeft, Kokkos::HostSpace>;
    // WRAPPER
    using HostMaskView_t = Kokkos::View<
        typename DeviceMaskView_t::data_type, // e.g., unsigned char***
        Kokkos::LayoutLeft,
        Kokkos::HostSpace
    >;

    Field<m_t, Dim, ViewArgs...> ghostMaskField(Mesh_, Layout_, static_cast<int>(nGhost));
    ghostMaskField = static_cast<m_t>(1);
    DeviceMaskView_t deviceMaskView   = ghostMaskField.getView();

    auto interior   = ghostMaskField.template getFieldRangePolicy<>();
    if constexpr (Dim == 1) {
        Kokkos::parallel_for("ZeroOwnedMask1D", interior,
          KOKKOS_LAMBDA(const int i) {
            deviceMaskView(i) = static_cast<m_t>(0);
        });
    } else if constexpr (Dim == 2) {
        Kokkos::parallel_for("ZeroOwnedMask2D", interior,
          KOKKOS_LAMBDA(const int i, const int j) {
            deviceMaskView(i,j) = static_cast<m_t>(0);
        });
    } else if ( Dim == 3){
        Kokkos::parallel_for("ZeroOwnedMask3D", interior,
          KOKKOS_LAMBDA(const int i, const int j, const int k) {
            deviceMaskView(i,j,k) = static_cast<m_t>(0);
        });
    } else{
        throw IpplException(/* ... */);
    }
    hostMaskView1D = HostMaskView1D_t("hostGhostMask_1D", deviceMaskView.size());
    HostMaskView_t hostMaskView_N_Rank(
        hostMaskView1D.data(),
        deviceMaskView.extent(0),
        deviceMaskView.extent(1),
        deviceMaskView.extent(2),
        deviceMaskView.extent(3),
        deviceMaskView.extent(4),
        deviceMaskView.extent(5),
        deviceMaskView.extent(6),
        deviceMaskView.extent(7) // Good/Flexible for up to 8D
    );
    Kokkos::deep_copy(hostMaskView_N_Rank, deviceMaskView);

    conduit_cpp::Node ghostMask_field_meta  =  data["metadata/vtk_fields/vtkGhostType"];
    conduit_cpp::Node ghostMask_field_node  =  fields["vtkGhostType"];
    ghostMask_field_node["association"].set_string(associate); // vs vertex ...
    ghostMask_field_node["topology"].set_string("fmesh_topo");
    ghostMask_field_node["volume_dependent"].set_string("false");
    ghostMask_field_node["values"].set_external(hostMaskView1D.data(),hostMaskView1D.size());
```

Listing A.12: *Creation of a Ghost Mask View*

A.6.5 Ghost Mask Caching

Discussed in Paragraph 4.3.6.GhostMaskCaching.

```cpp
/* in ippl namespace */
using GhostKey_t = std::tuple<const void*, const void*, size_t>;
struct GhostKeyHash {
    std::size_t operator()(const GhostKey_t& k) const {
        // Get the hash for each element in the tuple
        auto h1 = std::hash<const void*>{}(std::get<0>(k));
        auto h2 = std::hash<const void*>{}(std::get<1>(k));
        auto h3 = std::hash<size_t>{}(std::get<2>(k));

        // Combine the hashes. This is a common pattern (based on boost::hash_combine)
        // It xors and bit-shifts to mix the bits well.
        h1 ^= h2 + 0x9e3779b9 + (h1 << 6) + (h1 >> 2);
        h1 ^= h3 + 0x9e3779b9 + (h1 << 6) + (h1 >> 2);
        return h1;
    }
};

////////////////////////////////////////////////
/* defined inside the CatalystAdapter class: */

std::unordered_map<GhostKey_t, HostMaskView1D_t, GhostKeyHash> ghostMaskCache;
const void* meshKey   = static_cast<const void*>(&Mesh_);
const void* layoutKey = static_cast<const void*>(&Layout_);
auto ghostKey = GhostKey_t{meshKey, layoutKey, nGhost};



//////////////////////////////////////////
/* during ExecVizChannel for ippl::Field */

    using m_t = unsigned char;
    using DeviceMaskView_t = typename Field<m_t, Dim, ViewArgs...>::view_type;
    /* STORAGE */
    using HostMaskView1D_t = Kokkos::View<m_t*, Kokkos::LayoutLeft, Kokkos::HostSpace>;
    /* WRAPPER */
    using HostMaskView_t = Kokkos::View<
        typename DeviceMaskView_t::data_type, // e.g., unsigned char***
        Kokkos::LayoutLeft,
        Kokkos::HostSpace
    >;

    HostMaskView1D_t hostMaskView1D; // Declare view


    auto it = ghostMaskCache.find(ghostKey);
    if (it != ghostMaskCache.end())
    {
        /* --- CACHE HIT --- */
        hostMaskView1D = it->second;
    }
    else
    {
        /* --- CACHE MISS --- */
        /*
        Create Ghost Mask as shown in previousy Listing
        */

        ghostMaskCache[ghostKey] = hostMaskView1D;
    }
```

Listing A.13: *Caching logic for created Ghost masks*

A.6.6 Remember Now

Discussed in Section 4.3.7.

```
void CatalystAdaptor::RememberNow(const std::string label){

    auto it  = forceHostCopy.find(label);
    if (it == forceHostCopy.end() || !visRegistry)   throw IpplException(/* ... */);

    bool tmp = it->second;
    forceHostCopy[label] = true;
    ExecuteVisitor execV{*this};
    visRegistry->forOne(label, execV);
    forceHostCopy[label] = tmp;
}
```

Listing A.14: *`RememberNow` function implementation*

```

template</* ... */ >
requires(/* ... */)
void CatalystAdaptor::ExecVizChannel(const T& entry, const std::string label)
{
    if(viewRegistry.contains(label)) return;
    /* ... */

}
```

Listing A.15: *Necessary addition to all `ExecVizChannel` methods*

Appendix B

Conduit Nodes

This appendix demonstrates the hierarchical structure of the Conduit Nodes produced by the current IPPL implementation in yaml format. We employ a combination of generic schema and explicit samples to demonstrate these structures effectively.

For visualization objects, we present representative examples for a Particle Container, a Scalar Field, and a Vector Field. For steering parameters, we illustrate three distinct cases: a simple double-precision scalar using its original label, an instance of the `LinMaps` structure (referencing Listing 4.9), and an integer array.

B.1 Initialization Node

Discussed in Section 4.3.4

```
catalyst:
 mpi_comm: 1140850688
 scripts:
   script:
     filename: "/path/to/pipeline_default.py"
     args:
       - "--channel_names"
       - "ippl_sField_density"
       - "ippl_particles_ions"
       - "ippl_vField_electrostatic"
       - "ippl_sField_density"
       - "--verbosity"
       - "5"
       - "--VTKextract"
       - "ON"
       - "--live"
       - "ON"
       - "--steer"
       - "ON"
       - "--steer_channel_names"
       - "electric_scale"
       - "<LinMapLabel>.time"
       - "<LinMapLabel>.x_row"
       - "<LinMapLabel>.y_row"
       - "<LinMapLabel>.z_row"
       - "array:<intArrayLabel>"
   density:                                    # scalar field with <label> = density
     filename: "/path/to/png_ext_sfield.py"
     args:
       - "--channel_name"
       - "ippl_sField_density"
       - "--label"
       - "density"
       - "--verbosity"
       - "5"
   ions:                                       # Particle Container with <label> = ions
     filename: "/path/to/png_ext_particle.py"
     args:
       - "--channel_name"
       - "ippl_particles_ions"
       - "--label"
       - "ions"
       - "--verbosity"
       - "5"
   potential:                                  # scalar field with <label> = potential
     filename: "/path/to/png_ext_sfield.py"
     args:
       - "--channel_name"
       - "ippl_sField_potential"
       - "--label"
       - "potential"
       - "--verbosity"
       - "5"
   electrostatic:                              # scalar field with <label> = electtrostatic
     filename: "/path/to/png_ext_vfield.py"
     args:
       - "--channel_name"
       - "ippl_vField_electrostatic"
       - "--label"
       - "electrostatic"
       - "--experiment_name"
       - "AlpineSight"
       - "--verbosity"
       - "5"
 proxies:
   proxy_:
     filename: "/path/to/catalyst_proxy.xml"
```

Listing B.1: *Example for a Conduit Initialization Node for an IPPL application*

B.2 Execution Node, Conduit Visualization Channels):

B.2.1 Particles Conduit Channels

Discussed in section 4.3.6 Particle Data. We split the multimesh structure sample up into two Listings.

Main Block

```
ippl_particles_<label>:
  type: "multimesh"
  data:
    block_main:
      type: "mesh"
      fields:
        RankID:
          association: "vertex"
          topology: "p_unstructured_topo"
          volume_dependent: "false"
          values: [1, 1, 1, ..., 1, 1]
        ParticleIDs:
          association: "vertex"
          topology: "p_unstructured_topo"
          volume_dependent: "false"
          values: [...]
        position:
          association: "vertex"
          topology: "p_unstructured_topo"
          volume_dependent: "false"
          values:
            x: [...]
            y: [...]
            z: [...]
        <custom_attribute>:
          association: "vertex"
          topology: "p_unstructured_topo"
          ...
      coordsets:
        p_explicit_coords:
          type: "explicit"
          values:
            x: [...]
            y: [...]
            z: [...]
      topologies:
        p_unstructured_topo:
          coordset: "p_explicit_coords"
          type: "unstructured"
          elements:
            shape: "point"
            connectivity: [0, 1, 2, ..., 2086, 2087]
      metadata:
        vtk_fields:
          RankID:
            attribute_type: "ProcessIds"
          ParticleIDs:
            attribute_type: "GlobalIds"
    block_help:
        <see next Listing>
  assembly:
    main: "block_main"
    help: "block_help"
```

Listing B.2: *Conduit multi mesh channel definition for particle cotnainers. Here displayed is the main block defining particle positions and attributes.*

Helper Block

```
ippl_particles_<label>:
  type: "multimesh"
  data:
    block_main:

        <see previous Listing>

    block_help:
      coordsets:
        bound_helper_coords:
          type: "uniform"
          dims:
            i: 2
            j: 2
            k: 2
          spacing:
            dx: <x_max>
            dy: <y_max>
            dz: <z_max>
          origin:
            x: <x_min>
            y: <y_min>
            z: <u_min>
      topologies:
        bound_helper_topo:
          coordset: "bound_helper_coords"
          type: "uniform"
  assembly:
    main: "block_main"
    help: "block_help"
```

Listing B.3: *Conduit multi mesh channel definition for particle cotnainers. Here displayed is the helper block defining the particle domain bounds.*

B.2.2 Field Conduit Channels

Discussed in section 4.3.6 FieldAndMetaData.

Conduit Channel for a Scalar Fields

```
ippl_sField_<label>:
  type: "mesh"
  data:
    fields:
      <label>:
        association: "element"
        topology: "fmesh_topo"
        volume_dependent: "false"
        values: [...]
      RankID:
        association: "element"
        topology: "fmesh_topo"
        volume_dependent: "false"
        values: [...]
      vtkGhostType:
        association: "element"
        topology: "fmesh_topo"
        volume_dependent: "false"
        values: [...]
    topologies:
        fmesh_topo:
            type: "uniform"
            coordset: "cart_uniform_coords"
            origin:
                x: 7.5
                y: -2.5
                z: -2.5
    coordsets:
        cart_uniform_coords:
            type: "uniform"
            dims:
              i: 7
              j: 11
              k: 11
            spacing:
              dx: 2.5
              dy: 2.5
              dz: 2.5
            origin:
              x: 7.5
              y: -2.5
              z: -2.5
    metadata:
          vtk_fields:
            RankID:
              attribute_type: "ProcessIds"
            vtkGhostType:
              attribute_type: "Ghosts"
```

Listing B.4: *Conduit Node channel for a scalar field*

Conduit Channel for a Vector Field

```
ippl_vField_<label>:
  state:
  type: "mesh"
  data:
    fields:
      <label>:
        association: "element"
        topology: "fmesh_topo"
        volume_dependent: "false"
        values:
          x: [...]
          y: [...]
          z: [...]
      RankID:
        association: "element"
        topology: "fmesh_topo"
        volume_dependent: "false"
        values: [1, 1, 1, ..., 1, 1]
      vtkGhostType:
        association: "element"
        topology: "fmesh_topo"
        volume_dependent: "false"
        values: [1, 1, 1, ..., 1, 1]
    topologies:
      fmesh_topo:
        type: "uniform"
        coordset: "cart_uniform_coords"
        origin:
          x: 7.5
          y: -2.5
          z: -2.5
    coordsets:
      cart_uniform_coords:
        type: "uniform"
        dims:
          i: 7
          j: 11
          k: 11
        spacing:
          dx: 2.5
          dy: 2.5
          dz: 2.5
        origin:
          x: 7.5
          y: -2.5
          z: -2.5
    metadata:
      vtk_fields:
        RankID:
          attribute_type: "ProcessIds"
        vtkGhostType:
          attribute_type: "Ghosts"
```

Listing B.5: *Conduit Node channel for vector field*

B.3 Execution Node, Conduit Steering Channels

Discussed in Section 4.3.8 ForwardSteering

B.3.1 Simple Parameter

```
steerable_channel_0D_mesh:
  type: "mesh"
  data:
    coordsets:
      coords:
        type: "explicit"
        values:
          x: 0
    topologies:
      scalar_Mesh_topo:
        type: "unstructured"
        coordset: "coords"
        elements:
          shape: "point"
          connectivity: 0
    fields:
        steerable_field_f_<label_0>:
            association: "vertex"
            topology: "scalar_Mesh_topo"
            volume_dependent: "false"
            values:
                x: <parameter_value>
                ...
        steerable_field_f_<label_1>:
            ...
```

Listing B.6: *Scalar channel for steerable conduit-fields in the execution Conduit node*

B.3.2 Custom Structs

```
    fields:
        steerable_field_f_<LinMapLabel>.time:
            topology: "scalar_Mesh_topo"
            association: "vertex"
            volume_dependent: "false"
            values: <Vt>
      steerable_field_f_<LinMapLabel>.x_row:
            ... <same as for standard steering fields> ...
            values:
                x: <Vxx>
                y: <Vyx>
                z: <Vyx>
      steerable_field_f_<LinMapLabel>.y_row:
        ...
      steerable_field_f_<LinMapLabel>.z_row:
        ...
```

Listing B.7: *Fields in the Execution Conduit Node for a LinMap steering structure*

B.3.3 Simple Arrays

```
...
    steerable_channel_1D_mesh_array:<label_0>:      # Separate channel for one scalar array
          type: "mesh"
          data:
            coordsets:
              coords:
                type: "explicit"
                values:
                  x: [0, 1, ..., n]
            topologies:
              array_Mesh_topo:
                type: "unstructured"
                coordset: "coords"
                elements:
                  shape: "point"
                  connectivity: [0, 1, ..., n]
            fields:
              steerable_field_f_array:<label_0>:
                association: "vertex"
                topology: "array_Mesh_topo"
                volume_dependent: "false"
                values:
                    x: [<Vx0>, <Vx1>, ..., <Vxn>]
                    y: [<Vv0>, <Vy1>, ..., <Vyn>]
                    ...
    steerable_channel_1D_mesh_array:<label_1>:   # Separate channel for another scalar array
        ...
```

Listing B.8: *Steering forward channel Conduit layout for dynamically sized steering array*

B.3.4 Array of Structs

```
...
    steerable_channel_1D_mesh_array:array:<LinMapsLabel>:   # separate channel for one array
    of structs
      type: "mesh"
      data:
        ... <like scalar array example> ...
        fields:
          steerable_field_f_array:array:<LinMapsLabel>.time:
            association: "vertex"
            topology: "array_Mesh_topo"
            volume_dependent: "false"
            values: [<Vt0>, <Vt1>, ..., <Vtn>]
          steerable_field_f_array:array:<LinMapsLabel>.x_row:
            association: "vertex"
            topology: "array_Mesh_topo"
            volume_dependent: "false"
            values:
              x: [<Vxx0>, <Vxx1>, ..., <Vxxn>]
              y: [<Vyx0>, <Vyx1>, ..., <Vyxn>]
              z: [<Vzx0>, <Vzx1>, ..., <Vzxn>]
          steerable_field_f_array:array:<LinMapsLabel>.y_row:
            ...
          steerable_field_f_array:array:<LinMapsLabel>.z_row:
            ...
```

Listing B.9: *Steering Conduit-field for array of Structs*

Appendix C

Proxy Files

The code samples fro the upcoming Appendix were discussed in Section 4.3.8 BackwardSteering

C.1 ProxyGroup Sources

Discussed in Paragraph 4.3.8 BackwardSteering.ProxyGroupSources.

C.1.1 Source Preamble

```
<ServerManagerConfiguration>
    <ProxyGroup name='sources'>
        <SourceProxy    class='vtkSteeringDataGenerator'
                        name='SteerableParameters_ ... '>

<!--   ================================================================================ -->
<!-- 'Preambel' always needed  for the vtkSteeringDataGnerator, not further relevant to us-->
            <IntVectorProperty  name='PartitionType'
                                command='SetPartitionType'
                                number_of_elements='1'
                                default_values='1'
                                panel_visibility='never'>
            </IntVectorProperty>
            <IntVectorProperty  name='FieldAssociation'
                                command='SetFieldAssociation'
                                number_of_elements='1'
                                default_values='0'
                                panel_visibility='never'>
            </IntVectorProperty>
<!--   ================================================================================ -->
        ...

        </SourceProxy>

        ...
    </ProxyGroup>
    ...
</ServerManagerConfiguration>
```

Listing C.1: *Preamble needed in every SourceProxy definition*

C.1.2 Collective SourceProxy for non Array Parameters

Steering Channel Definitions

```
1
2  <ServerManagerConfiguration>
3      <ProxyGroup name='sources'>
4
5          ...
6
7          <SourceProxy     class='vtkSteeringDataGenerator'
8                           name='SteerableParameters_SCALARS'>
9
10             <!--    ======================== -->
11             <!--         Preamble            -->
12             <!--    ======================== -->
13
14             <!--    ====== Channel Definitions========== -->
15             <DoubleVectorProperty name='electric'
16                                   label='electric'
17                                   command='SetTuple1Double'
18                                   clean_command='Clear'
19                                   use_index='1'
20                                   initial_string='steerable_field_b_electric'
21                                   default_values='30'
22                                   number_of_elements_per_command='1'
23                                   repeat_command='1'
24                                   panel_widget='DoubleRange'
25                                   >
26             </DoubleVectorProperty>
27
28             <DoubleVectorProperty name='LinMapLabel.time'
29                                   label='LinMapLabel.time'
30                                   command='SetTuple1Double'
31                                   clean_command='Clear'
32                                   use_index='1'
33                                   initial_string='steerable_field_b_LinMapLabel.time'
34                                   default_values='0.75'
35                                   number_of_elements_per_command='1'
36                                   repeat_command='1'
37                                   panel_widget='DoubleRange'
38                                   >
39             </DoubleVectorProperty>
40
41             <DoubleVectorProperty name='LinMapLabel.x_row'
42                                   label='LinMapLabel.x_row'
43                                   command='SetTuple3Double'
44                                   use_index='1'
45                                   clean_command='Clear'
46                                   initial_string='steerable_field_b_LinMapLabel.x_row'
47                                   default_values='0.000000 0.000000 0.000000'
48                                   number_of_elements='3'
49                                   number_of_elements_per_command='3'
50                                   repeat_command='1'>
51             </DoubleVectorProperty>
52
53             <DoubleVectorProperty name='LinMapLabel.y_row'  ...
54             </DoubleVectorProperty>
55
56             <DoubleVectorProperty name='LinMapLabel.z_row'  ...
57             </DoubleVectorProperty>
58
59             ...
60
61
62         </SourceProxy>
63     </ProxyGroup>
64     ...
65 </ServerManagerConfiguration>
```

Listing C.2: *Channel definition for the "static channel" inside the 'sources' ProxyGroup*

Hints and Layout

```
<ServerManagerConfiguration>
    <ProxyGroup name='sources'>
        <SourceProxy class='vtkSteeringDataGenerator' name='SteerableParameters_SCALARS'>

<!--   ============================================================================== -->
<!--   ====== Hints =========== -->
<!-- 'Specifying correspondance to forward channel and field                          -->
             <Hints>
              <CatalystInitializePropertiesWithMesh mesh='steerable_channel_0D_mesh'>
                <Property   name='electric'
                            association='point'
                            array='steerable_field_f_electric' />
                <Property   name='LinMapLabel.time'
                            association='point'
                            array='steerable_field_f_LinMapLabel.time' />
                <Property   name='LinMapLabel.x_row'
                            association='point'
                            array='steerable_field_f_LinMapLabel.x_row' />
                <Property   name='LinMapLabel.y_row'
                            association='point'
                            array='steerable_field_f_LinMapLabel.y_row' />
                <Property   name='LinMapLabel.z_row'
                            association='point'
                            array='steerable_field_f_LinMapLabel.z_row' />
              </CatalystInitializePropertiesWithMesh>
            </Hints>

<!--   ============================================================================== -->
<!--   ====== Layout =========== -->
<!-- 'Specifies strucutural Layout in the ParaView GUI                              -->

            <PropertyGroup label='LinMapLabel' panel_widget='PropertyCollection'>
                <Property   name='LinMapLabel.time'
                            function='LinMapLabel:time'
                            label='LinMapLabel.time'/>
                <Property   name='LinMapLabel.x_row'
                            function='LinMapLabel:x_row'
                            label='LinMapLabel.x_row'/>
                <Property   name='LinMapLabel.y_row'
                            function='LinMapLabel:y_row'
                            label='LinMapLabel.y_row'/>
                <Property   name='LinMapLabel.z_row'
                            function='LinMapLabel:z_row'
                            label='LinMapLabel.z_row'/>
                <Hints>
                  <PropertyCollectionWidgetPrototype    group='misc'
                                                        name='LinMapLabelPrototype' />
                </Hints>
            </PropertyGroup>

            <PropertyGroup  label='Numerics' panel_widget='PropertyCollection'>
                <Property   name='electric'
                            function='scaleFactor_electric'
                            label='electric'/>
                <Hints>
                  <PropertyCollectionWidgetPrototype    group='misc'
                                                        name='SteerableNumericsPrototype' />
                </Hints>
            </PropertyGroup>
            ...
        </SourceProxy>
    ...
    </ProxyGroup>
</ServerManagerConfiguration>
```

Listing C.3: *Defined Hints and PropertyGroups for the "static channel" inside the 'sources' ProxyGroup*

C.1.3 SourceProxy for Scalar Arrays

```
<ServerManagerConfiguration>
    <ProxyGroup name='sources'>
        <SourceProxy    class='vtkSteeringDataGenerator'
                        name='SteerableParameters_intArrayLabel'>

            <!--   ======================== -->
            <!--        Preamble            -->
            <!--   ======================== -->



            <!--   ====== Channel Definitions=========== -->
            <IntVectorProperty  name='intArrayLabel'
                                label='intArrayLabel'
                                command='SetTuple1Int'
                                clean_command='Clear'
                                use_index='1'
                                initial_string='steerable_field_b_array:intArrayLabel'
                                default_values='1 1 1'
                                number_of_elements_per_command='1'
                                repeat_command='1'>
            </IntVectorProperty>


            <!--   ====== Layout =========== -->
            <PropertyGroup  label='intArrayLabel'
                            panel_widget='PropertyCollection'>
                <Hints>
                  <PropertyCollectionWidgetPrototype    group='misc'
                                                        name='intArrayLabelPrototype' />
                </Hints>
                <Property   name='intArrayLabel'
                            function='intArrayLabel'
                            label='intArrayLabel'/>
            </PropertyGroup>

            <!--   ====== Hints =========== -->
            <Hints>
              <CatalystInitializePropertiesWithMesh
                            mesh='steerable_channel_1D_mesh_array:intArrayLabel'>
                <Property   name='intArrayLabel'
                            association='point'
                            array='steerable_field_f_array:intArrayLabel' />
              </CatalystInitializePropertiesWithMesh>
            </Hints>
        </SourceProxy>
    </ProxyGroup>
    ...
</ServerManagerConfiguration>
```

Listing C.4: *Channel definitions, Hints, and PropertyGroups for a dynamic array channel inside the 'sources' ProxyGroup*

C.2 ProxyGroup misc

Discussed in Paragraph 4.3.8 BackwardSteering.ProxyGroupMISC.

C.2.1 Protoype Definitions

```
<ServerManagerConfiguration>
    ...
    <ProxyGroup name='misc'>
      <Proxy name='SteerableNumericsPrototype' label=' Numerics-Collective-Prototype'>
        <DoubleVectorProperty   name='scaleFactor_electric'
                                label='electric'
                                number_of_elements='1'
                                default_values='0'>
          <DoubleRangeDomain name='range' min='0' max='50'/>
        </DoubleVectorProperty>
      </Proxy>
      <Proxy name='LinMapLabelPrototype' label='LinMapLabel'>
        <DoubleVectorProperty   name='LinMapLabel:time'
                                label='time'
                                number_of_elements='1' default_values='0.75'>
        </DoubleVectorProperty>
        <DoubleVectorProperty   name='LinMapLabel:x_row'
                                label='x_row'
                                number_of_elements='3'
                                default_values='0.0 0.0 0.0'>
        </DoubleVectorProperty>
        <DoubleVectorProperty   name='LinMapLabel:y_row'
                                label='y_row'
                                number_of_elements='3'
                                default_values='0.0 0.0 0.0'>
        </DoubleVectorProperty>
        <DoubleVectorProperty   name='LinMapLabel:z_row'
                                label='z_row'
                                number_of_elements='3'
                                default_values='0.0 0.0 0.0'>
        </DoubleVectorProperty>
      </Proxy>
      <Proxy name='intArrayLabelPrototype' label='intArrayLabel'>
        <IntVectorProperty  name='intArrayLabel'
                            label='intArrayLabel'
                            number_of_elements='1'
                            default_values='1'>
          <IntRangeDomain name='range' min='-10' max='10'/>
        </IntVectorProperty>
      </Proxy>
    </ProxyGroup>
</ServerManagerConfiguration>
```

Listing C.5: *The 'misc' Proxy Group defining the Protopye Layout for previously defined Property Collection widgets.*

C.3 Configuration File; Example for Steering UI Layout

Discussed in Paragraph 4.3.8 BackwardSteering.ConfigurationFile

```
ranges:
  electric:
    min: 0.0
    max: 50.0
  intArrayLabel:
    min: -10.0
    max: 10.0
```

Listing C.6: *Example: Specify proxy Group prototype configuration via. yaml configuration file.*

Appendix D

Catalyst Scripts

D.1 Helpers

D.1.1 Get Global Bounds and Extent

In the Catalyst script the retrieving of a proxy objects global properties was ofetn not directly feasible. We most received only rank 0 data. We therefore introduced helper functions to retriv gloabl qunatities given the local quantities, with rank reduction calls. The use of a Rank reduction for such a minor task is some what inefficient. For future implementations, since this data is often easily retrievable in the IPPL scope, we should try to pass Global Field information in a separate VTk mesh (like we did for the Particle Domain), or via arguments that can be directly passed to the Catalyst scripts execute function via. the Conduit execution node (`paraview.catalyst.get_execute_params()`.

```python
def get_global_range(local_min, local_max):
    controller = vtkMultiProcessController.GetGlobalController()
    if not controller or controller.GetNumberOfProcesses() == 1:
        return local_min, local_max
    g_min = [0.0]
    g_max = [0.0]
    controller.AllReduce([local_min], g_min, 1, vtkCommunicator.MIN_OP)
    controller.AllReduce([local_max], g_max, 1, vtkCommunicator.MAX_OP)
    return g_min[0], g_max[0]

def get_global_spatial_bounds(local_bounds):
    controller = vtkMultiProcessController.GetGlobalController()
    if not controller or controller.GetNumberOfProcesses() == 1:
        return local_bounds
    gb = [0.0] * 6
    for i in (0, 2, 4):
        send = array.array('d', [float(local_bounds[i])])
        recv = array.array('d', [0.0])
        controller.AllReduce(send, recv, 1, vtkCommunicator.MIN_OP)
        gb[i] = float(recv[0])
    for i in (1, 3, 5):
        send = array.array('d', [float(local_bounds[i])])
        recv = array.array('d', [0.0])
        controller.AllReduce(send, recv, 1, vtkCommunicator.MAX_OP)
        gb[i] = float(recv[0])
    return tuple(gb)
```

Listing D.1: *These helper functions can be used to get the global bounds of a vtk object from the local bounds. This might be needed to optimize the application for certain paraview filters.*

```python
def get_global_extent(field_info, global_bounds)
    ghost_info = field_info.GetArrayInformation("vtkGhostType") if field_info else None
    local_bounds = field_info.GetBounds()
    local_extent = field_info.GetExtent()

    #local amount of cells
    nx = (local_extent[1] - local_extent[0] )
    ny = (local_extent[3] - local_extent[2] )
    nz = (local_extent[5] - local_extent[4] )
    # local domain width
    lx = (local_bounds[1] - local_bounds[0])
    ly = (local_bounds[3] - local_bounds[2])
    lz = (local_bounds[5] - local_bounds[4])
    #local spacing (since uniform grid) -> global spacing
    dx = lx / nx
    dy = ly / ny
    dz = lz / nz
    # !! currently assumes only max 1 Ghost Layer
    if ghost_info:
        cut_layers = 1
    else:
        cut_layers = 0
    # Cut global bounds if we have an additional row of ghost cells
    global_bounds = (
        global_bounds[0] + cut_layers * dx,
        global_bounds[1] - cut_layers * dx,
        global_bounds[2] + cut_layers * dy,
        global_bounds[3] - cut_layers * dy,
        global_bounds[4] + cut_layers * dz,
        global_bounds[5] - cut_layers * dz,
    )
    # Compute global extent form glabal bounds and uniform spacing
    dim_x = int(round((global_bounds[1] - global_bounds[0]) / dx))
    dim_y = int(round((global_bounds[3] - global_bounds[2]) / dy))
    dim_z = int(round((global_bounds[5] - global_bounds[4]) / dz))
    global_extent = [dim_x, dim_y, dim_z]
    return global_extent
```

Listing D.2: *This method allows one to retrieve the global extent (mesh dimension) without andother MPI reduce call, if the Global Bounds have already been retrieved. This might be needed to optimize the application for certain paraview filters.*

D.1.2 Hide Proxies from the ParaView GUI

```python
    from paraview import servermanager

    def hide_source_from_gui(proxy):
        pxm = servermanager.ProxyManager()
        name = pxm.GetProxyName("sources", proxy.SMProxy)
        if name:
            pxm.UnRegisterProxy("sources", name, proxy.SMProxy)
```

Listing D.3: *Proxy objects created in the Catalyst script will be displayed in the ParaView GUI. If these proxies are necessary for PNG extraction, interacting with them in the GUI can crash the Extractor or even Catalyst as a whole. We need a simple method to hide these sources form the GUI's pipeline browser.*

D.1.3 VTK extractor helpers

```
def create_VTPD_extractor(name, object, fr = 10):
    """Use for: scalar fields, vector fields (Conduit type: mesh)
    Output: .vtpd files (XML Partitioned Dataset)
    """
    vTPD = CreateExtractor('VTPD', object, registrationName='VTPD_'+ name)
    vTPD.Trigger.Frequency = fr
    vTPD.Writer.FileName = 'ippl_'+name+'_{timestep:06d}.vtpd'
    return vTPD
def create_VTM_extractor(name, object, fr = 10):
    """Use for: particle data with multiple block types (Conduit type: multimesh)
    Output: .vtpc files (XML Partitioned Dataset Collection)
    """
    vTPC = CreateExtractor('VTPC', object, registrationName='VTPC_'+ name)
    vTPC.Trigger.Frequency = fr
    vTPC.Writer.FileName = 'ippl_'+name+'_{timestep:06d}.vtpc'
    return vTPC
```

Listing D.4: *Helper methods for VTK file extraction*

D.2 Main Pipeline:

D.2.1 Parsing Script Arguments

```
import argparse

arg_list = paraview.catalyst.get_args()
parser = argparse.ArgumentParser()
parser.add_argument("--channel_names",          nargs="*")
parser.add_argument("--steer_channel_names",    nargs="*")
parser.add_argument("--VTKextract",             default="OFF")
parser.add_argument("--live",                   default="OFF")
parser.add_argument("--steer",                  default="OFF")
parser.add_argument("--show_forward_channels",  default="OFF")
parser.add_argument("--experiment_name",        default="_")
parser.add_argument("--verbosity",              default="1",    type=int)

parsed = parser.parse_args(arg_list)
```

Listing D.5: *How we parse the script's "Initiliazation Arguments" published inside the conduit node*

D.2.2 Setting Basic Catalyst Options

```
import paraview as pv
from paraview import simple as pvs
from paraview import catalyst
# pv.catalyst is distinct from pvs.catalyst

options = catalyst.Options()

if parsed.live == "ON":
    options.EnableCatalystLive = 1
    options.CatalystLiveTrigger = 'Time Step'
    options.CatalystLiveURL = 'localhost:22222'
else:
    options.EnableCatalystLive = 0

options.GlobalTrigger = 'Time Step'
options.ExtractsOutputDirectory = 'data_vtk_extracts_' + parsed.exp_string
```

Listing D.6: *How we configure basic Catalyst options in our main Catalyst script*

D.2.3 Visualization

```
_extractors = {}
_sources = {}
_filters = {}
pm = servermanager.ProxyManager()

for cname in parsed.channel_names:

    proxy = PVTrivialProducer(registrationName=cname)
    proxy.UpdatePipeline()
    _sources[cname] = proxy

    if "particles" in cname:
        if options.EnableCatalystLive:
            particles = ExtractBlock(
                registrationName=f"{cname}.bunch",
                Input=proxy,
                Selectors=['//main', '//block_main']
            )
            helper = ExtractBlock(
                registrationName=f"{cname}.box",
                Input=proxy,
                Selectors=['//help', '//block_help']
            )
            particles.UpdatePipeline()
            helper.UpdatePipeline()
            _filters[cname+"_main"] = particles
            _filters[cname+"_help"] = helper
            Show(particles)
            Show(helper)
        if parsed.VTKextract == "ON":
            _extractors[cname] = create_VTM_extractor(cname, proxy, 1)

    if "sField" in cname:
        if options.EnableCatalystLive:
            merged = MergeBlocks(registrationName=cname +'.MergedBlocks',
                                 Input=proxy)
            merged.MergePartitionsOnly = 1
            Show(merged)
            info = proxy.GetDataInformation()
            local_bounds = info.GetBounds()
            local_extent = info.GetExtent()
            global_bounds = get_global_spatial_bounds(local_bounds)
            global_extent = get_global_extent(info, global_bounds)
            resample = ResampleToImage( registrationName=cname+'.ResampleToImage',
                                        Input=merged)
            resample.UseInputBounds = 1
            resample.SamplingDimensions = global_extent
            Show(resample)
            _filters[cname[12:]+"_merge"]    = merged
            _filters[cname[12:]+"_resample"] = resample
        if parsed.VTKextract == "ON":
            _extractors[cname] = create_VTPD_extractor(cname, proxy, 1)

    if "vField" in cname:
        if options.EnableCatalystLive:
            merged = MergeBlocks(registrationName=cname+'.MergedBlocks',Input=proxy)
            merged.MergePartitionsOnly = 1
            Show(merged)
            glyph = Glyph(registrationName=cname +'.Glyph', Input=merged, GlyphType='Arrow')
            glyph.OrientationArray = ['CELLS', cname[12:]]
            Show(glyph)
            _filters[cname + "_merged"] = merged
            _filters[cname + "_glyph"]  = glyph
        if parsed.VTKextract == "ON":
            _extractors[cname] = create_VTPD_extractor(cname, proxy, 1)
```

Listing D.7: *How we set up live visulization and VTK file extraction inside the main Catalyst script*

D.2.4 Steering

```
if parsed.steer and parsed.steer_channel_names :

    # Unified sender: one proxy carrying one property per channel, single result mesh
    sender_all = CreateSteerableParameters(
        steerable_proxy_type_name           = "SteerableParameters_SCALARS",
        steerable_proxy_registration_name   = "SteeringParameters_SCALARS",
        result_mesh_name                    = "steerable_channel_backward_all"
    )


    # Detect struct-array namespaces from steer_channel_names with prefix 'array:'
    # Example entries: 'array:LinMaps.time', 'array:IntArrayLabel'
    struct_array_senders = {}
    detected_namespaces = set()
    try:
        for entry in (parsed.steer_channel_names or []):
            if not isinstance(entry, str):
                continue
            if not entry.startswith("array:"):
                continue
            # Extract namespace between ':' and first '.'; fallback to full tail if no '.'
            tail = entry.split(":", 1)[1]
            ns = tail.split(".", 1)[0] if "." in tail else tail
            if ns:
                detected_namespaces.add(ns)
    except Exception as e:
        print_info(f"[DEBUG][WARN] Failed to parse struct-array namespaces: {e}")

    # Create one sender per detected namespace (e.g., 'LinMaps', 'simpVec')
    for ns in sorted(detected_namespaces):
        sender = CreateSteerableParameters(
            steerable_proxy_type_name         = f"SteerableParameters_{ns}",
            steerable_proxy_registration_name = f"Steering_{ns}",
            result_mesh_name                  = "steerable_channel_backward_all"
        )
        struct_array_senders[ns] = sender
```

Listing D.8: *How we activate Steerable Channels inside our main Catalyst script*

D.3 PNG Extractor Script for Particles

```
from paraview import catalyst
from paraview.simple import *
# Parse arguments received via conduit node------------------------
arg_list = paraview.catalyst.get_args()
parser = argparse.ArgumentParser()
parser.add_argument("--channel_name",       default="def")
parser.add_argument("--label",              default="def")
parser.add_argument("--experiment_name",    default="_")
parser.add_argument("--verbosity",          default="1", type=int,)
parsed = parser.parse_args(arg_list)
# Get Data sources ------------------------------------------------
ippl_particle_p = PVTrivialProducer(registrationName = parsed.channel_name)
ippl_particle_bunch = ExtractBlock(
                registrationName=f"{parsed.label}_bunch_png_ext",
                Input=ippl_particle_p,
                Assembly = 'Hierarchy',
                Selectors=['//block_main']
            )
ippl_particle_box = ExtractBlock(
                registrationName=f"{parsed.label}_box_png_ext",
                Input=ippl_particle_p,
                Assembly = 'Hierarchy',
                Selectors=['//block_help']
            )
hide_source_from_gui(ippl_particle_bunch)
hide_source_from_gui(ippl_particle_bunch)
# Visualization and Rendering-------------------------------------
view_name = f"View_{parsed.channel_name}"
renderView1 = CreateView('RenderView', registrationName=view_name)
# ----------------------------------------------------------------
# <setup visualisation  inside render View> ...
# ----------------------------------------------------------------

# Choose proxies to display in renderView -------------------------
SetActiveView(renderView1)
particleDisplay     = Show(ippl_particle_bunch, renderView1, 'UnstructuredGridRepresentation'
    )
particle_boxDisplay = Show(ippl_particle_box, renderView1, 'GeometryRepresentation')

# ----------------------------------------------------------------
# <setup visualisation for particles and boundary domain (box)> ...
# ----------------------------------------------------------------

#  Create PNG Extractor --------_--------------------------------
pNG1 = CreateExtractor('PNG', renderView1, registrationName='PNG_'+ cname)
pNG1.Trigger = 'Time Step'
pNG1.Writer.FileName = parsed.label+'_Particles_{timestep:06d}{camera}.png'
pNG1.Writer.Format = 'PNG'
pNG1.Writer.ImageResolution = [2000, 1500]
pNG1.Writer.TransparentBackground = 0
SetActiveSource(pNG1)
# Catalyst options------------------------------------------------------------
options = catalyst.Options()
options.GlobalTrigger = 'Time Step'
options.EnableCatalystLive = 0
options.ExtractsOutputDirectory = 'data_png_extracts_' + exp_string
# ----------------------------------------------------------------------------
if __name__ == '__main__':
    SaveExtractsUsingCatalystOptions(options)
# ----------------------------------------------------------------------------
def catalyst_execute(info):
    global ippl_particle_bunch
    global ippl_particle_box
    ippl_particle_bunch.UpdatePipeline()
    ippl_particle_box.UpdatePipeline()
```

Listing D.9: *Simplified script framework enabling PNG extraction for particle buch data*

D.4 PNG Extractor Script for Scalar Fields

```python
from paraview import catalyst
from paraview.simple import *
# Parse arguments received via conduit node------------------------
arg_list = paraview.catalyst.get_args()
parser   =  # ... <like particle png extractor> ...#
parsed   = parser.parse_args(arg_list)
# Get Data sources ------------------------------------------------
ippl_scalar_p = PVTrivialProducer(registrationName=cname)
scalar_info = ippl_scalar_p.GetDataInformation()

# -----------------------------------------------------------------
# <get global bounds and extent via helper fuction>
# -----------------------------------------------------------------

# Check for ghost data --------------------------------------------
cinfo = scalar_info.GetCellDataInformation()
ghost_info = cinfo.GetArrayInformation("vtkGhostType") if cinfo else None

# -----------------------------------------------------------------
ippl_scalar = # <ghost handling> see next listing for details
# -----------------------------------------------------------------

#  Visualization and Rendering ------------------------------------
view_name = f"View_{parsed.channel_name}"
renderView1 = CreateView('RenderView', registrationName=view_name)

# -----------------------------------------------------------------
# <setup visualization  inside render View>
# -----------------------------------------------------------------

# Choose proxies to display in renderView -------------------------
ippl_scalarDisplay = Show(ippl_scalar, renderView1)

# -----------------------------------------------------------------
# <setup scalar field visualization>
# -----------------------------------------------------------------

#  Create PNG Extractor --------_-----------------------------------
pNG1 = CreateExtractor('PNG', renderView1, registrationName='PNG_'+cname)
pNG1.Trigger = 'Time Step'
pNG1.Trigger.Frequency = 1
pNG1.Writer.FileName = label+'_ScalarField_{timestep:06d}{camera}.png'
pNG1.Writer.ImageResolution = [2000, 1500]
pNG1.Writer.Format = 'PNG'

# -----------------------------------------------------------------
# Catalyst options------------------------------------------------
options = catalyst.Options()
options.GlobalTrigger = 'Time Step'
options.EnableCatalystLive = 0
options.ExtractsOutputDirectory = 'data_png_extracts_' + exp_string
# ------------------------------------------------------------------------------
if __name__ == '__main__':
    SaveExtractsUsingCatalystOptions(options)
# -----------------------------------------------------------------
def catalyst_execute(info):
    global ippl_scalar_p
    ippl_scalar_p.UpdatePipeline()
```

Listing D.10: *Simplified script framework enabling PNG extraction for scalar field data.*

```
if ghost_info:
    # 2. Ghost Handling & Restore Image data (Source -> Threshold -> Resample)
    # ------------------------------------------------------------------
    # A. Cut out Ghost Cells of Visualization with a Threshold filter
    ippl_scalar_t = Threshold(registrationName='RemoveGhosts', Input=ippl_scalar_p)
    hide_source_from_gui(ippl_scalar_t)
    ippl_scalar_t.Scalars = ['CELLS', 'vtkGhostType']
    ippl_scalar_t.AllScalars = 0
    ippl_scalar_t.ThresholdMethod = 'Between'
    ippl_scalar_t.LowerThreshold = 0.0
    ippl_scalar_t.UpperThreshold = 0.0

    # B. Convert BACK to Uniform Grid using ResampleToImage
    ippl_resampled = ResampleToImage(registrationName='ResampleToGrid', Input=ippl_scalar_t)
    hide_source_from_gui(ippl_resampled)
    ippl_resampled.UseInputBounds = 1
    ippl_resampled.SamplingDimensions = global_extent
    ippl_scalar = ippl_resampled
    associate = "POINTS" # ResampleToImage produces Point Data by default
else:
     # if there are no ghost we still resample to image. In both cases we now
     # have the same final filter and data format so we can use the same rendering
     # and display configuration.
     # Mainly necessary since ippl_scalar_p would associate with CELLS not POINTS.

    ippl_resampled = ResampleToImage(registrationName='ResampleToGrid', Input=ippl_scalar_p)
    hide_source_from_gui(ippl_resampled)
    ippl_resampled.UseInputBounds = 0
    ippl_resampled.SamplingBounds = global_bounds
    ippl_resampled.SamplingDimensions = global_extent

    ippl_scalar = ippl_resampled
    associate = "POINTS" # ResampleToImage produces Point Data by default
```

Listing D.11: *We had some difficulties for Image extractors in case of multi rank data with Ghost data. For the image Extraction instead of correctly using the Ghost data (different to live visualization), any spatial point which was somehow associated as a Ghost Cell was not rendered on any rank leading to large Gaps in the final Image. We therefor needed to introduce a filtering systems to remove the Ghost Cells manually. This should be revisted since we are not sure if this is due to our misunderstanding or a current bug in ParaView Catalyst*

D.5 PNG Extractor Script for Vector Fields

```
from paraview import catalyst
from paraview.simple import *

# Parse arguments received via conduit node-------------------------
arg_list = paraview.catalyst.get_args()
parser   =  # ... <like particle png extractor> ...#
parsed   = parser.parse_args(arg_list)

# Get Data sources ------------------------------------------------
ippl_vector_field = PVTrivialProducer(registrationName = parsed.channel_name)
hide_source_from_gui(ippl_vector_field)

# create filter to visualize a Vector Field ------------------------
glyph1 = Glyph(registrationName='Glyph1', Input=ippl_vector_field, GlyphType='Arrow')
hide_source_from_gui(glyph1)

# Visualization and Rendering  -------------------------------------
view_name = f"View_{cname}"
renderView1 = CreateView('RenderView', registrationName=view_name)

# ----------------------------------------------------------------
# <setup visualization inside render View> ...
# ----------------------------------------------------------------

# Choose proxies to display in renderView --------------------------
glyph1Display = Show(glyph1, renderView1, 'GeometryRepresentation')

# ----------------------------------------------------------------
# < setup glyph1 Configuration for visualization> ...
# ----------------------------------------------------------------

# Create PNG Extractor ---------------------------------------------
pNG1 = CreateExtractor('PNG', renderView1, registrationName='pNG_'+cname)
pNG1.Trigger = 'Time Step'
pNG1.Trigger.Frequency = 1
pNG1.Writer.FileName = 'VectorField_{timestep:06d}{camera}.png'
pNG1.Writer.ImageResolution = [2000, 1500]
pNG1.Writer.Format = 'PNG'

# ----------------------------------------------------------------
# Catalyst options-------------------------------------------------
options = catalyst.Options()
options.GlobalTrigger = 'Time Step'
options.EnableCatalystLive = 0
options.ExtractsOutputDirectory = 'data_png_extracts_' + exp_string
# ----------------------------------------------------------------
if __name__ == '__main__':
    SaveExtractsUsingCatalystOptions(options)
# ----------------------------------------------------------------
def catalyst_execute(info):
    global glyph1
    global ippl_vector_field
    ippl_vector_field.UpdatePipeline()
    # glyph1.UpdatePipeline()
```

Listing D.12: *Simplified script framework enabling PNG extraction for vector field data.*

Appendix E

Environment Variables for In Situ Configuration

With these envionment variables ona can control the frameworks behavior.

Core Catalyst Configuration

These are necessary definitions; without these defined, the in situ analysis is guaranteed to crash.

Variable	Description
`CATALYST_IMPLEMENTATION_PATHS`	Path to Catalyst implementation inside the ParaView binary (e.g., `$path/to/ParaView-5.12.0/lib/catalyst`)
`CATALYST_IMPLEMENTATION_NAME`	Implementation name (currently always: `"paraview"`)

Table E.1: *Core Catalyst Environment Configuration*

Main IPPL in situ Switches

Variable	Values	Default	Description
IPPL_CATALYST_VIS	ON/OFF	ON	Enable/disable visualization
IPPL_CATALYST_LIVE	ON/OFF	OFF	Enable/disable live viz
IPPL_CATALYST_STEER	ON/OFF	OFF	Enable/disable steering
IPPL_CATALYST_PNG	ON/OFF	OFF	Enable PNG image extraction
IPPL_CATALYST_VTK	ON/OFF	OFF	Enable VTK file extraction
IPPL_CATALYST_GHOST_MASKS	ON/OFF	OFF	Specifies how to avoid Ghost Cell visualization[1]
IPPL_CATALYST_VERBOSITY	0,...,5	Global IPPL verbosity	Separate verbosity for logging concerning IPPL in situ.

[1]: *ON creates a field mask marking all Ghost Cells which enables ParaView to actively ignore the Ghost data, but allows the user with filters and extraction to visualize the Ghost data in the Trame app or the ParaView client. Uses caching logic to only pass reference to one mask if multiple fields rely on the same Field Layout. OFF the Ghost cells are cut out before the field data is sent from the simulation to Catalyst, so less data has to be sent.*

Table E.2: *Main Switches for the IPPL Catalyst Configuration*

Advanced IPPL Catalyst Configuration Options

With these a user can change paths at which IPPL tries to access Catalyst Python scripts and proxy files.

Variable	Values	Default	Description
IPPL_CATALYST_PROXY_OPTION	ON/OFF/ PRODUCE_ONLY	ON	Configures Proxy Writer Class[1]
IPPL_PROXY_CONFIG_YAML	<path>	[2]	Path to custom Proxy config YAML file
CATALYST_PROXYS_PATH	<path>	[2]	Proxy XML for magnetic field steering
CATALYST_PIPELINE_PATH	<path>	[2]	Path to custom Catalyst Python script
CATALYST_EXTRACTOR_SCRIPT_<label>	<path>	[2]	Path to custom PNG extractor script for specific registry entry with <label>

[1]: *ON/OFF runs simulation normally and provides a switch if you write the catalyst_proxy.xml file (at path src/Stream/InSitu/catalyst_scripts/). PRODUCE_ONLY writes the catalyst_proxy.xml file and aborts simulation run.*

[2]: *The default scripts that are used and would be overwritten with these variables can be found in the folder: ${IPPL_DIR}/src/Stream/InSitu/catalyst_scripts/...*

Table E.3: *Advanced IPPL Catalyst environment Configuration*

Appendix F

Tests and Performance

F.1 NVTOP GPU monitoring

Figure F.1: *GPU utilization monitored with* `nvtop` *during the in situ visualization process. The timeline shows GPUs are idle for the majority of the* `catalyst_execute` *call, indicating a CPU-side bottleneck.*

Appendix G

Unified Modeling Language Notation Guide

This appendix provides a brief overview of the Unified Modeling Language (UML) class diagram notation used throughout this report. The diagrams are generated using the `tikz-uml` LaTeX package.

G.1 Classes and Members

A class is represented by a rectangle divided into three distinct compartments: the top compartment displays the class name, the middle contains its attributes (variables), and the bottom contains its operations (methods). For class templates a dashed box in the upper right corner indicates the template parameters. The visibility of attributes and operations is denoted by the following symbols:

- `+` Public (accessible from anywhere)
- `-` Private (accessible only within the class)
- `#` Protected (accessible within the class and its subclasses)

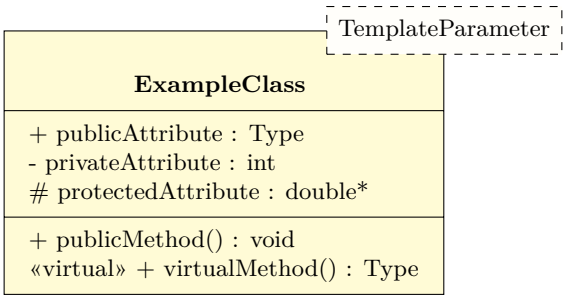


Figure G.1: *Example of a class template representation with template parameters, varying visibilities, and a virtual method.*

G.2 Class Relationships

The structural relationships between classes are represented by different types of arrows.

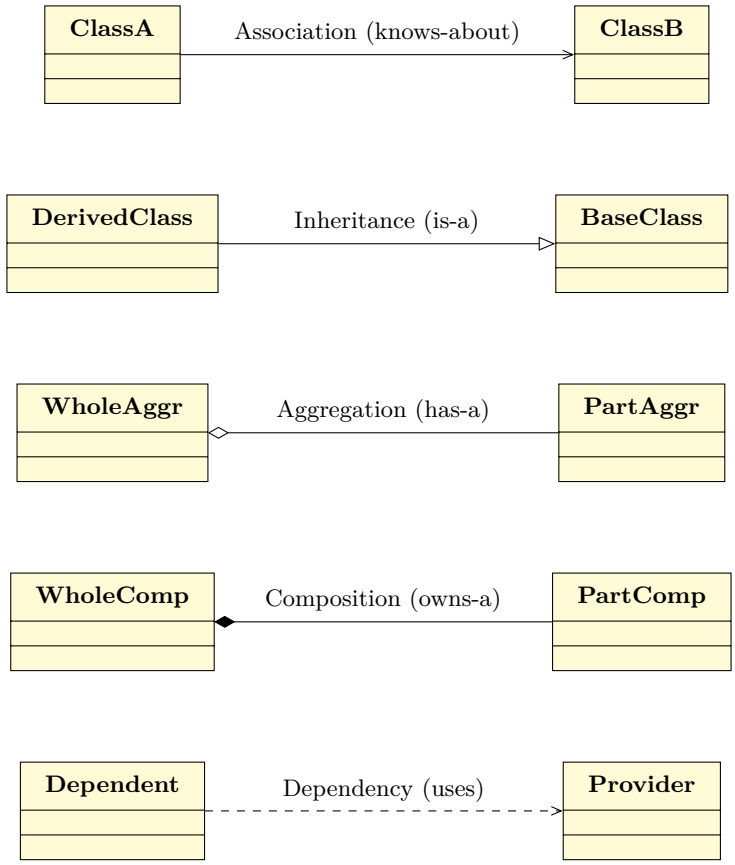


Figure G.2: *Overview of UML relationship notations used in this report*

Multiplicity (Cardinality)

Relationships can include numbers or asterisks near the ends of the relationship lines to indicate multiplicity (how many instances of one class relate to one instance of another):

- `1` : Exactly one
- `0,1` : Zero or one
- `*` : Many (zero or more)
- `1..*` : At least one

Acknowledgements

I would like to express my gratitude to Dr. Andreas Adelmann, for suggesting and supervising this Project. I further want to thank Ryan Ammann and Dr. Sri Muralikrishnan for assisting me during this project as well as the rest of the (PSI-LSM)-AMAS group and the IPPL-team for advise given during weakly meetings. I also extend my thanks to Dr. Jens Henrik Göbbert and Jonathan Windgassen for their guidance regarding ParaView, Catalyst, and Trame, and to the PSI Merlin Administrators for enabling support for the necessary in situ tools on the Merlin cluster.

Declaration of the use of AI-based tools

AI Tool	**Use Case**	**Remarks**
Gemini 2 Pro	Brainstorming and Suggestions for Registry design	
GitHub Copilot (via Claude/Gemini/GPT) Gemini 2 Pro Gemini 3 Pro	Coding support for debugging, refactoring, and implementing the CatalystAdaptor	Initial Code for recursive handling of complex steering structure were largely AI generated based on our previous implementation of simpler steering parameters and provided implementation suggestions Initial ProxyWriter class was AI generated by providing large samples of self written XML file examples. Initial Doxygen documentation was AI generated before manual review
Gemini 3 Pro	Revision of texts (spell-checking, grammar, style, and conciseness)	

Bibliography

Literature

[1] Erich Gamma et al. Design patterns: elements of reusable object-oriented software. USA: Addison-Wesley Longman Publishing Co., Inc., 1994. 395 pp. ISBN: 978-0-201-63361-0.

[2] Message P Forum. MPI: A Message-Passing Interface Standard. Technical Report. USA: University of Tennessee, 1994-03.

[3] James Rumbaugh, Ivar Jacobson, and Grady Booch. Unified Modeling Language Reference Manual, 2nd edition. Pearson Higher Education, 2004-06. ISBN: 978-0-321-24562-5.

[4] James Ahrens, Berk Geveci, and Charles Law. "ParaView: An End-User Tool for Large-Data Visualization". In: Elsevier, 2005, pp. 717–731. ISBN: 978-0-12-387582-2. DOI: 10.1016/B978-012387582-2/50038-1.

[5] Will Schroeder, Ken Martin, and Bill Lorensen. Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 4th Edition. Clifton Park, NY: Kitware, 2006. 528 pp. ISBN: 978-1-930934-19-1. URL: https://raw.githubusercontent.com/lorensen/VTKExamples/master/src/VTKBookLaTeX/VTKTextBook.pdf.

[6] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. The Eurographics Association, 2011. ISBN: 978-3-905674-32-3. URL: https://doi.org/10.2312/EGPGV/EGPGV11/101-109.

[7] Hank Childs. "VisIt: An End-User Tool for Visualizing and Analyzing Very". In: High Performance Visualization. Chapman and Hall/CRC, 2012.

[8] E. Wes Bethel, Hank Childs, and Charles Hansen, eds. High Performance Visualization: Enabling Extreme-Scale Scientific Insight. New York: Chapman and Hall/CRC, 2012-10-25. 520 pp. ISBN: 978-0-429-10535-7. DOI: 10.1201/b12985.

[9] I. Karlin, J. Keasler, and J. R. Neely. LULESH 2.0 Updates and Changes. LLNL-TR–641973. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), 2013-07-22. DOI: 10.2172/1090032.

[10] H. Carter Edwards and Christian R. Trott. "Kokkos: Enabling Performance Portability Across Manycore Architectures". In: 2013 Extreme Scaling Workshop (xsw 2013). 2013 Extreme Scaling Workshop (xsw 2013). 2013-08, pp. 18–24. DOI: 10.1109/XSW.2013.7.

[11] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: Journal of Parallel and Distributed Computing 74.12 (2014-07-22). ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003.

[12] Utkarsh Ayachit et al. "ParaView Catalyst: Enabling In Situ Data Analysis and Visualization". In: In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization. ISAV2015. New York, NY, USA: Association for Computing Machinery, 2015-11-15, pp. 25–29. ISBN: 978-1-4503-4003-8. DOI: 10.1145/2828612.2828624.

[13] Kenneth Moreland et al. "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures". In: IEEE Computer Graphics and Applications 36.3 (2016-05), pp. 48–58. ISSN: 1558-1756. DOI: 10.1109/MCG.2016.48.

[14] E. W. Bethel et al. "In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report". In: Computer Graphics Forum (2016-06-10). URL: https://escholarship.org/uc/item/93q94959.

[15] Kress James. "In Situ Visualization Techniques for High Performance Computing". In: 2017. URL: https://www.semanticscholar.org/paper/In-Situ-Visualization-Techniques-for-High-Computing-Kepler-Aries/94d940a13ca155b4f8a40fca425d8baf31f19646.

[16] Andreas Adelmann et al. OPAL a Versatile Tool for Charged Particle Accelerator Simulations. 2019-05-16. DOI: 10.48550/arXiv.1905.06654. arXiv: 1905.06654[physics].

[17] Rigel F. C. Alves and Andreas Knüpfer. "In Situ Visualization of Performance-Related Data in Parallel CFD Applications". In: Euro-Par 2019: Parallel Processing Workshops. Ed. by Ulrich Schwardmann et al. Cham: Springer International Publishing, 2020, pp. 400–412. ISBN: 978-3-030-48340-1. DOI: 10.1007/978-3-030-48340-1_31.

[18] Hank Childs et al. "A terminology for in situ visualization and analysis systems". In: The International Journal of High Performance Computing Applications 34.6 (2020-11-01), pp. 676–691. ISSN: 1094-3420. DOI: 10.1177/1094342020935991.

[19] William F. Godoy et al. "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management". In: SoftwareX 12 (2020-07-01), p. 100561. ISSN: 2352-7110. DOI: 10.1016/j.softx.2020.100561.

[20] Utkarsh Ayachit et al. "Catalyst Revised: Rethinking the ParaView in Situ Analysis and Visualization API". In: High Performance Computing: ISC High Performance Digital 2021. Berlin, Heidelberg: Springer-Verlag, 2021-06-24, pp. 484–494. ISBN: 978-3-030-90538-5. DOI: 10.1007/978-3-030-90539-2_33.

[21] Hank Childs, Janine C. Bennett, and Christoph Garth, eds. In Situ Visualization for Computational Science. Mathematics and Visualization. Cham: Springer International Publishing, 2022. ISBN: 978-3-030-81626-1 978-3-030-81627-8. DOI: 10.1007/978-3-030-81627-8.

[22] Matthew Larsen et al. "Ascent: A Flyweight In Situ Library for Exascale Simulations". In: In Situ Visualization for Computational Science. Ed. by Hank Childs, Janine C. Bennett, and Christoph Garth. Cham: Springer International Publishing, 2022, pp. 255–279. ISBN: 978-3-030-81627-8. DOI: 10.1007/978-3-030-81627-8_12.

[23] David Pugmire et al. “The Adaptable IO System (ADIOS)”. In: In Situ Visualization for Computational Science. Ed. by Hank Childs, Janine C. Bennett, and Christoph Garth. Cham: Springer International Publishing, 2022, pp. 233–254. ISBN: 978-3-030-81627-8. DOI: 10.1007/978-3-030-81627-8_11.

[24] Cyrus Harrison et al. “Conduit: A Successful Strategy for Describing and Sharing Data In Situ”. In: 2022 IEEE/ACM International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV). 2022-11, pp. 1–6. DOI: 10.1109/ISAV56555.2022.00006.

[25] Christian R. Trott et al. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: IEEE Transactions on Parallel and Distributed Systems 33.4 (2022-04), pp. 805–817. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3097283.

[26] Sriramkrishnan Muralikrishnan et al. “Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures”. In: (2022-05-23). DOI: 10.48550/arXiv.2205.11052.

[27] François Mazen, Lucas Givord, and Charles Gueunet. “Catalyst-ADIOS2: In Transit Analysis for Numerical Simulations Using Catalyst 2 API”. In: High Performance Computing. Ed. by Amanda Bienz et al. Cham: Springer Nature Switzerland, 2023, pp. 269–276. ISBN: 978-3-031-40843-4. DOI: 10.1007/978-3-031-40843-4_20.

[28] S. Bnà et al. “In situ visualization for high-fidelity CFD—Case studies”. In: Computers & Fluids 267 (2023-12-15), p. 106066. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2023.106066.

[29] Kenneth Moreland et al. “Visualization at exascale: Making it all work with VTK-m”. In: The International Journal of High Performance Computing Applications 38.5 (2024), pp. 508–526. ISSN: 1094-3420. DOI: 10.1177/10943420241270969.

[30] Colleen Heinemann et al. “Graphical Representation Through a User Interface for In Situ Scientific Visualization with Ascent”. In: 2024 IEEE 14th Symposium on Large Data Analysis and Visualization (LDAV). 2024 IEEE 14th Symposium on Large Data Analysis and Visualization (LDAV). 2024-10, pp. 71–72. DOI: 10.1109/LDAV64567.2024.00017.

[31] Andres Sewell et al. “Bridging Gaps in Simulation Analysis through a General Purpose, Bidirectional Steering Interface with Ascent”. In: SC24-W: Workshops of the International Conf. for HPC, Networking, Storage and Analysis. SC24-W: Workshops of the International Conf. for HPC, Networking, Storage and Analysis. 2024-11, pp. 841–846. DOI: 10.1109/SCW63240.2024.00119.

[32] Patrick Avery, Sebastien Jourdain, and Patrick O’Leary. “trame: an Open Source Framework for Efficiently Building Interactive Visualization and Analysis Applications”. In: Microscopy and Microanalysis 30 (Supplement_1 2024-07-01), ozae044.1100. ISSN: 1431-9276. DOI: 10.1093/mam/ozae044.1100.

[33] Jens Göbbert et al. “In-Situ Visualization With Ascent and NekRS for Large-Scale CFD Problems on GPUs”. In: Proceedings of the 35th Parallel CFD International Conference 2024 (2025). DOI: https://doi.org/10.3929/ethz-b-000715064.

[34] François Mazen et al. “In Situ in Transit Hybrid Analysis with Catalyst-ADIOS2”. In: High Performance Computing. ISC High Performance 2024 International Workshops. Ed. by Michèle Weiland et al. Cham: Springer Nature Switzerland, 2025, pp. 482–489. ISBN: 978-3-031-73716-9. DOI: 10.1007/978-3-031-73716-9_34.

[35] James Ahrens et al. “ALPINE The ECP project: In situ and post hoc visualization infrastructure and analysis capabilities for exascale”. In: The International Journal of High Performance Computing Applications 39.1 (2025-01-01), pp. 32–51. ISSN: 1094-3420. DOI: 10.1177/10943420241286521.

[36] Thomas Marrinan et al. “Real-time Scientific Visualization and Interactive Steering for High-Performance Computing Simulations”. In: Practice and Experience in Advanced Research Computing 2025. PEARC ’25. New York, NY, USA: Association for Computing Machinery, 2025-07-18, pp. 1–4. ISBN: 979-8-4007-1398-9. DOI: 10.1145/3708035.3736031.

[37] Thomas Marrinan et al. “Intuitive Computational Steering Using Ascent and Trame”. In: 2025 IEEE International Conference on eScience (eScience). 2025 IEEE International Conference on eScience (eScience). 2025-09, pp. 10–19. DOI: 10.1109/eScience65000.2025.00011.

[38] Jeff Lee et al. “Catalyst:Expanding In Situ Analysis and Visualization Workflows forExascale Computing”. In: AIAA SCITECH 2026 Forum. AIAA SciTech Forum. American Institute of Aeronautics and Astronautics, 2026-01-08. DOI: 10.2514/6.2026-0766.

[39] Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language. ISBN: 978-0-321-19368-1. URL: https://www.cs.uah.edu/~rcoleman/CS307/Announcements/UML%20Distilled.pdf.

Software, Documentation, Blogs, and Online Resources

[40] AdiosCatalyst · GitLab. GitLab. URL: https://gitlab.kitware.com/paraview/adioscatalyst (visited on 2026-01-03).

[41] ALPINE - documentation - What is ALPINE. URL: https://alpine-dav.readthedocs.io/en/latest/introduction.html (visited on 2026-01-02).

[42] ALPINE In Situ Infrastructure, Enabling Extreme-Scale Analysis and Visualization. ACM Conferences. DOI: 10.1145/3144769.3144778. (Visited on 2026-01-02).

[43] Alpine-DAV/ascent. URL: https://github.com/Alpine-DAV/ascent (visited on 2026-01-02).

[44] argonne-lcf/ascent-trame. URL: https://github.com/argonne-lcf/ascent-trame (visited on 2026-01-18).

[45] Catalyst · GitLab. GitLab. URL: https://gitlab.kitware.com/paraview/catalyst (visited on 2026-01-02).

[46] Panchumrti Jaswant and Geveci Berg. Catalyst2:GPU resident workflows. URL: https://www.kitware.com/catalyst2-gpu-resident-workflows/ (visited on 2026-01-14).

[47] CatalystParaViewDoc. URL: https://docs.paraview.org/en/latest/Catalyst/index.html (visited on 2026-01-04).

[48] Andrei Alexandrescu. CppCon2022-Reflection in C++ - Past, Present, and Hopeful Future. URL: https://isocpp.org/blog/2023/06/cppcon-2022-reflection-in-cpp-past-present-and-hopeful-future-andrei-alexan (visited on 2026-01-25).

[49] cppref:meta programming library. URL: https://en.cppreference.com/w/cpp/meta.html (visited on 2026-01-15).

[50] ECP Home Page. Exascale Computing Project. URL: https://www.exascaleproject.org/ (visited on 2025-12-31).

[51] heffte-github. URL: https://github.com/icl-utk-edu/heffte (visited on 2026-01-04).

[52] Jens Henrik Göbbert. InSitu-Vis with IPPLParaView - HedgeDoc. URL: https://gitlab.jsc.fz-juelich.de/hedgedoc/BUMCeHfkR46KuqQnjV3pHQ?view (visited on 2026-01-02).

[53] IPPL-framework/ippl. URL: https://github.com/IPPL-framework/ippl (visited on 2026-01-02).

[54] Jens Henrik Göbbert. jhgoebbert/ippl at in-situ-new. GitHub. URL: https://github.com/jhgoebbert/ippl (visited on 2026-01-02).

[55] Kitware/ParaView. URL: https://github.com/Kitware/ParaView (visited on 2026-01-02).

[56] Kitware/paraview-catalyst. URL: https://github.com/Kitware/paraview-catalyst (visited on 2026-01-02).

[57] Kokkos 3: GitHub. URL: https://github.com/kokkos/kokkos (visited on 2026-01-04).

[58] Kokkos documentation. URL: https://kokkos.org/kokkos-core-wiki/index.html (visited on 2026-01-04).

[59] llnl/conduit. URL: https://github.com/llnl/conduit (visited on 2026-01-02).

[60] Victor A Mateevitsi and Thomas Marrinan. mvictoras/ippl at tm_ascent_trame. GitHub. URL: https://github.com/mvictoras/ippl (visited on 2026-01-02).

[61] OPALX-project/OPALX. URL: https://github.com/OPALX-project/OPALX (visited on 2026-01-02).

[62] ParaView:Plugin Howto. URL: https://www.paraview.org/paraview-docs/nightly/cxx/PluginHowto.html (visited on 2026-01-13).

[63] ParaView:vtkSMLiveInsituLinkProxy Class. URL: https://www.paraview.org/paraview-docs/v5.13.3/cxx/classvtkSMLiveInsituLinkProxy.html (visited on 2026-01-19).

[64] ParaView:vtkSteeringDataGenerator Class Reference. URL: https://www.paraview.org/paraview-docs/latest/cxx/classvtkSteeringDataGenerator.html#a530f6e5400713c358d4c52decbccabb3 (visited on 2026-01-14).

[65] paraview.catalyst Python module. URL: https://www.paraview.org/paraview-docs/nightly/python/paraview.catalyst.html (visited on 2026-01-18).

[66] paraview.live Module documentation. URL: https://www.paraview.org/paraview-docs/latest/python/paraview.live.html (visited on 2026-01-19).

[67] paraview.simple.catalyst Python Module. URL: https://www.paraview.org/paraview-docs/nightly/python/paraview.simple.catalyst.html (visited on 2026-01-14).

[68] ParaViewUserGuide. URL: https://docs.paraview.org/en/latest/UsersGuide/ (visited on 2026-01-04).

[69] Refactoring and Design Patterns. URL: https://refactoring.guru/ (visited on 2026-01-06).

[70] Wyatt Childers. Reflection for C++26. URL: https://isocpp.org/files/papers/P2996R13.html#notable-additions-to-p1240 (visited on 2026-01-25).

[71] trameArchitectureAndCapabilities. URL: https://www.kitware.com/trame-architecture-and-capabilities/ (visited on 2026-01-05).

[72] trameDocumentation. URL: https://trame.readthedocs.io/en/latest/index.html (visited on 2026-01-05).

[73] trameGitHub. URL: https://github.com/Kitware/trame (visited on 2026-01-05).

[74] trameTutorial. URL: https://kitware.github.io/trame/guide/tutorial/ (visited on 2026-01-05).

[75] Various questions, issues, and feedback migrating from legacy Catalyst to Catalyst2. ParaView. URL: https://discourse.paraview.org/t/various-questions-issues-and-feedback-migrating-from-legacy-catalyst-to-catalyst2/17171 (visited on 2026-01-02).

[76] VisIt Home. VisIt Home. URL: https://visit-dav.github.io/visit-website/index.html (visited on 2026-01-02).

[77] Viskores/viskores. URL: https://github.com/Viskores/viskores (visited on 2026-01-02).

[78] WarpX-documentation-In Situ Capabilities. URL: https://warpx.readthedocs.io/en/latest/dataanalysis/insitu.html (visited on 2026-01-02).

[79] WikipediaUML. In: Wikipedia. URL: https://en.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=1331316054 (visited on 2026-01-06).

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☐ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☒ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

In Situ Analysis and Steering for High-Performance Particle Accelerator Simulations

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Klapproth	Severin Andrey Thomas

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Zürich, 26.01.2026	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).