UNIVERSITÀ DEGLI STUDI DI CATANIA

DIPARTIMENTO DI MATEMATICA E INFORMATICA

DOTTORATO DI RICERCA IN INFORMATICA XXXVIII CICLO

Nicola Tuccari di San Carlo

High-Performance Visualization for Astronomy and Cosmology: Refactoring and Parallelization of VisIVO Server for Exascale Systems

PH.D. THESIS

University Supervisor: Prof. Emiliano Tramontana

INAF Supervisor: Dr. Eva Sciacca

Anno Accademico 2024 - 2025

Abstract

The exponential growth of data volumes in astronomy and cosmology field poses significant challenges for data visualization and analysis. This thesis addresses these challenges by evolving VisIVO Server, a scientific software suite for multi-dimensional astrophysical data visualization, in multiple ways, to handle these data volumes in high-performance computing (HPC) environments. This work presents multiple major contributions. First, code analysis using tools such as CCCC, CppDepend and Doxygen, and then refactoring to achieve maintainability improvements, with cyclomatic complexity reductions of up to 94% in critical functions. Then, one of VisIVO's importers was parallelized using a hybrid MPI/OpenMP approach, achieving satisfying speedups, this was tested on 512 GB GADGET simulation datasets. Additionally, StreamFlow workflow management system was integrated with VisIVO to enable portable execution across heterogeneous computing platforms. Furthermore, in-situ visualization capability was implemented through integration with the Hecuba distributed toolset. This approach eliminates I/O bottlenecks by processing data directly in memory during simulation execution. For the last contribution, VisIVO was integrated into Cineca's InterActive Computing service by developing Python wrappers that allow users to access visualization capabilities on Jupyter notebooks while keeping HPC access. These contributions enable the astronomy and cosmology communities to better exploit modern HPC infrastructures for large-scale data analysis, supporting critical research into cosmic structure formation and fundamental astrophysical phenomena.

Acknowledgements

I thank C. Carbone for providing the DEMNUni simulations, which were carried out in the framework of “The Dark Energy and Massive-Neutrino Universe" project, using the Tier-0 IBM BG/Q Fermi machine and the Tier-0 Intel OmniPath Cluster Marconi-A1 of the Centro Interuniversitario del Nord-Est per il Calcolo Elettronico (CINECA). C. Carbone was granted generous CPU and storage allocations by the Italian Super-Computing Resource Allocation (ISCRA) as well as from the coordination of the “Accordo Quadro MoU per lo svolgimento di attività congiunta di ricerca Nuove frontiere in Astrofisica: HPC e Data Exploration di nuova generazione”, together with storage from INFN-CNAF and INAF-IA2.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With the advent of next-generation astrophysical instruments, such as the Square Kilometre Array [70] (SKA), and the availability of increasingly accurate cosmological simulation codes (such as GADGET [109] and ChaNGa [81]), the volume of data describing astronomical phenomena is growing at an unprecedented rate. For this reason, there is a fundamental requirement for efficient software tools that can assist in the automated analysis of these massive datasets.

SKA will be the largest radio telescope infrastructure ever built, with a total aperture of one square kilometer, utilizing thousands of radio telescopes in two different locations: Australia and South Africa. The expected data throughput generated by SKA will be approximately hundreds of terabytes per data cube (multidimensional data structure), which would result in up to 3 exabytes of processed data per year. Clearly, it is not feasible to store such enormous volumes of data, making it essential to develop new and advanced analysis methods that can process these vast amounts of data more quickly, accurately, and reliably than currently available solutions. At the same time, modern cosmological simulation codes like GADGET (GAlaxies with Dark matter and Gas intEracT) generate massive datasets simulations running on HPC infrastructures. These simulations produce outputs that can reach terabyte or petabyte scales, which present significant challenges for storage, processing, and visualization.

Current tools and systems exhibit processing limitations as the dimension

of data grows, particularly in the visualization and analysis phases of the obtained products. Hence, the following requirements need to be addressed when working on new software to handle these data.

- Innovative models based on next-generation accelerators for data processing algorithms, going beyond current CPU based approaches, achieving higher throughput using GPUs
- Co-design software development methodology combining the expertise of scientists, HPC experts, hardware and software developers, to aligh architectural and algorithmic design choices with domain-specific needs
- Effective integration of HPC resources with existing Cloud resources (related to the European Open Science Cloud platform), allowing workflows to exploit seamlessly both HPC and cloud platforms
- Efficient I/O systems for fast data access in interactive visual exploration, addressing current single-node bottlenecks (ranging from a few hundred MB/s up to 1.3 GB/s on parallel file systems [87]), and aiming to scale toward sustained tens of GB/s on modern HPC systems
- Scalable visualization solutions capable of handling exascale data volumes, supporting interactive rendering and analysis of datasets in the terabyte–petabyte range

VisIVO [12] (Visualization Interface for the Virtual Observatory) Server, while providing powerful capabilities for astronomical data visualization and analysis, faces these same scalability and performance challenges. With the advent of these new astrophysical instruments and the evolution of simulation software tools VisIVO requires fundamental improvements in its architecture, parallelization strategies, workflow integration, and data handling approaches.

1.2 Objectives

This thesis addresses the challenges described in the previous section by evolving VisIVO Server to make it capable to run on modern heterogeneous high-performance computing environments, ensuring that it can effectively process and visualize massive datasets that may reach petabyte scale.

VisIVO Server codebase is large and composed of many classes, at the moment it is in a status that satisfies old requirements, but it also must be updates, though trying to add functionalities is a nontrivial task and may require modularity. Adapting scientific applications for exascale systems is not achievable through hardware upgrades alone, as stated in other studies [46]. Hence, a major effort is needed to refactor and redevelop the legacy software to add new functionalities. For this reason, the first objective would be to perform a static analysis of the code to understand how difficult it would be to perform the required changes and apply refactoring techniques in most critical classes.

Then, another objective will be to address the limitations imposed by storage. As noted in recent studies, the growth of computational performance in the exascale era has far outpaced improvements in storage bandwidth, making I/O one of the major bottlenecks for scientific applications [53]. The first approach will be to parallelize the importer classes, which rely heavily on I/O operations. The parallelization of these kind of operations is a nontrivial task but thanks to modern distributed file systems it should be possible to achieve considerable speedup. The second approach will be to completely bypass storage using in-situ approaches; these types of approach have already demonstrated to be particularly useful in the context of scientific visualization to mitigate I/O bandwidth limitations [62]. This will be done with Hecuba[1], a set of tools that allows easy interaction with non-relational databases and streaming of data.

An additional objective is to enhance portability and reproducibility, this is has been widely recognized as a key challenge for scientific applications. Previous studies have shown that containerization combined with workflow management systems can enable reproducible and portable workflows across multiple environments [116]. We aim to do this by exploiting workflow management systems, in particular, by using Streamflow[28], VisIVO pipelines can

[1]Hecuba Tool repository, https://github.com/bsc-dd/hecuba

be modeled using the Common Workflow Language (CWL) and orchestrate the multiple steps of the pipelines; together the containerization of VisIVO, this enables seamless execution across heterogeneous HPC and cloud infrastructure. The adoption of CWL has emerged as a widely accepted standard for defining reproducible, portable, and environment-agnostic workflows [31].

In parallel, we also target to improve ease of use on HPC platforms by integrating VisIVO within CINECA's InterActive Computing platform. This aligns with the broader adoption of notebook-based interactive computing in supercomputing centers [85], lowering the entry barrier for researchers while still enabling access to powerful HPC resources. Our integration was achieved by developing Python wrappers allowing interactive, notebook-driven execution of the workflows while supporting multinode execution with MPI and multicore execution with OpenMP, which can run on CINECA's HPC platforms.

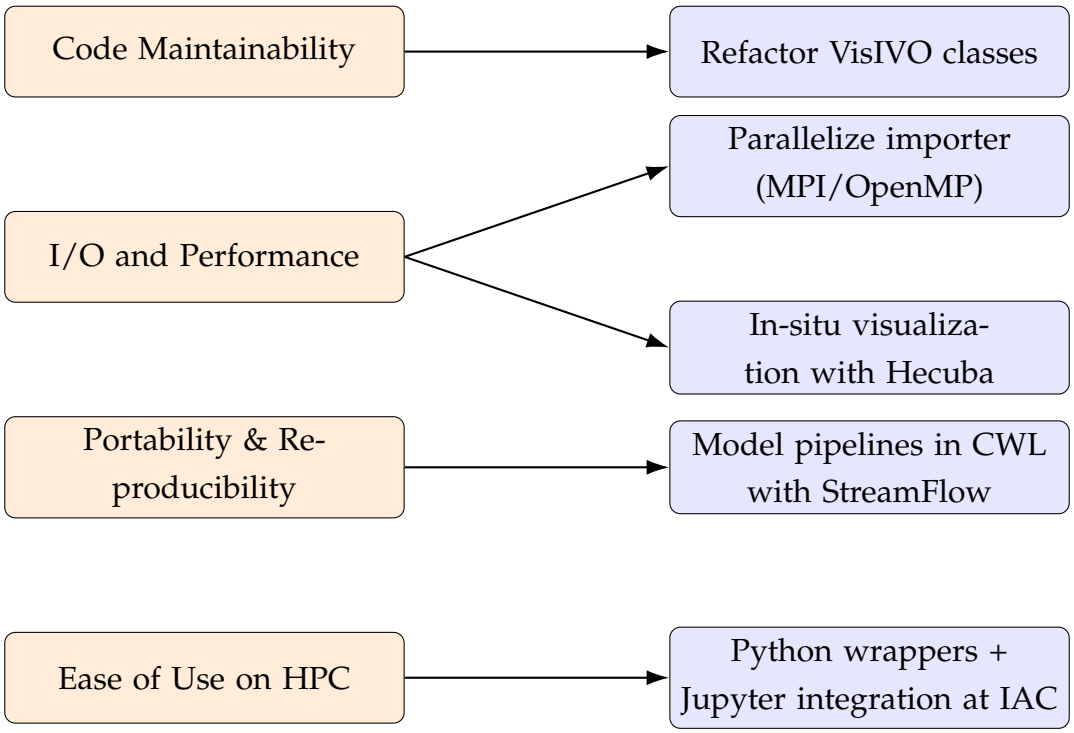

FIGURE 1.1: Main challenges addressed (left) and corresponding objectives (right).

1.3 Structure of the thesis

This thesis is organized into six main chapters that systematically present the evolution and enhancement of VisIVO Server for high-performance computing environments.

Chapter 2 provides background information on VisIVO and the tools, technologies, and related works that are the basis of this research work. It introduces the VisIVO modular applications, relevant simulation codes, relevant code analysis tools, parallel programming frameworks, workflow management system and interactive computing infrastructures.

Chapter 3 presents a comprehensive static analysis of the VisIVO Server codebase using multiple tools (CCCC, CppDepend, Doxygen) to evaluate code quality, maintainability, and complexity. The chapter also shows systematic refactoring techniques applied to improve the code structures and reduce cyclomatic complexity over multiple classes.

Chapter 4 focuses on the parallelization of VisIVO's importer module to exploit high-performance computing resources. It presents implementation details of a hybrid MPI/OpenMP approach for converting astronomical datasets into VisIVO internal data format and shows significant performance improvements on HPC systems.

Chapter 5 explores the integration of VisIVO with StreamFlow, a workflow management system that enables portable execution across heterogeneous computing platforms. It also presents the development of multiple workflows representing common visualization pipelines using VisIVO.

Chapter 6 introduces an in-situ visualization approach using Hecuba, a toolset to easily interact with non-relational technologies, enabling real-time data processing and visualization during simulation runtime. This approach eliminates traditional I/O bottlenecks by processing data directly from memory or by using streams.

Chapter 7 shows the development of an interactive high-performance visualization environment by integrating VisIVO and a Jupyter-based computing platform. It presents the creation of Python wrappers around VisIVO's functionalities and their deployment within the InterActive Computing(IAC)

services at CINECA, enabling users to access HPC visualization capabilities through web browsers.

1.4 Published Papers and talks

These are the papers published during my Ph.D. period:

- Sciacca, E., Tuccari, N., Vitello, & Cesare, V. **"High performance visualization for Astronomy & Cosmology: the VisIVO's pathway toward Exascale systems"** in Astronomical Data Analysis Software & Systems XXXIII, Tucson, Arizona, USA.

- Sciacca, E., Cesare, V., Tuccari, N., Vitello, F., Colonnelli, I., Carbone, C., Tramontana, E., & Becciani, U. **"Toward a Portable and Reproducible High-performance Visualization for Astronomy & Cosmology."** in 16th International Workshop on Science Gateways (IWSG2024), Toulouse, France.

- Tuccari, N., Sciacca, E., Becerra, Y., Sosa Cintero, E., Wissing, R., Shen, S., & Tramontana, E. **"In-Situ High Performance Visualization for Astronomy & Cosmology"** in Astronomical Data Analysis Software and Systems XXXIV, Valletta, Malta.

- Tuccari, N., Sciacca, E., Vitello, F., Colonnelli, I., Becerra, Y., Sosa Cintero, E., Marin, G., Jaros, M., Riha, L., Strakos, P., Trujillo-Gomez, S., Tramontana, E., & Wissing, R. **"High Performance Visualization for Astrophysics and Cosmology."** in 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turin, Italy.

- Sciacca, E., Tuccari, N., Arshad,U., Pitari, F., Muscianisi, G.,& Tramontana, E. **"Interactive High-Performance Visualization for Astronomy and Cosmology"** 17th International Workshop on Science Gateways (IWSG2025), London, United Kingdom.

- Tuccari, N., Sciacca, E., Becerra, Y., Sosa Cintero, E.,& Tramontana, E. **"Exascale In-situ visualization for Astronomy & Cosmology"** The 4th Italian Conference on Big Data and Data Science (ITADATA2025), Turin, Italy.

During the Ph.D. period, I also presented these talks to conferences and workshops:

- **"High Performance Visualization in Peta-scale Astronomy & Cosmology"** in BigHPC2023: Special Track on Big Data and High-Performance Computing, ITADATA 2023, Sept 11 – 13, Naples, Italy.
- **"High performance visualization for Astronomy & Cosmology: the VisIVO's pathway toward Exascale systems"** in Astronomical Data Analysis Software & Systems XXXIII, Tucson, Arizona, USA.
- **"High Performance Visualization in the SPACE Center of Excellence"** in HiPEAC 2024, Munich, Germany.
- **"Toward a Portable and Reproducible High-performance Visualization for Astronomy & Cosmology."** in 16th International Workshop on Science Gateways (IWSG2024), Toulouse, France.
- **"In-Situ High Performance Visualization for Astronomy & Cosmology"** in Astronomical Data Analysis Software and Systems XXXIV, Valletta, Malta.
- **"High Performance Visualization for Astrophysics and Cosmology."** in 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turin, Italy.
- **"High Performance Visualization with VisIVO"** in HPC Visualization Workshop, 13-15 May 2025, Barcelona, Spain.
- **"Exascale In-situ visualization for Astronomy & Cosmology"** The 4th Italian Conference on Big Data and Data Science (ITADATA2025), Turin, Italy.

1.5 Related Projects

This Ph.D. thesis work was carried out in the scientific and technological context of two HPC projects, one national and one international. These two projects are briefly presented below.

"FutureHPC & BigData'' of the ICSC - Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing[2]. The "National Centre for High-Performance Computing, Big Data and Quantum Computing" aims to maintain and upgrade the Italian HPC and Big Data infrastructures and it also aims to develop advanced methods, numerical applications and software tools to integrate computing, simulation, collection, and analysis of data of interest for research manufacturing and society. "FutureHPC & BigData" Spoke 1 is ICSC's technology pillar mainly focused on developing high-performance computing hardware and software for supercomputers. It addresses the challenges of modern computing by applying new technologies into high volume sectors such as cloud and edge computing, IoT connectivity, autonomous vehicles and AI.

SPACE - Scalable Parallel Astrophysical Codes for Exascale[3]. SPACE aims to re-engineer a selection of astrophysics and cosmology codes for heterogeneous exascale systems, enabling advanced simulations, high-performance data analysis, and discovery through machine-learning techniques, while fostering code and data sharing, interoperability and collaboration across European astrophysical communities. SPACE WP3 "Extreme data processing and analysis" focuses on developing prototype frameworks based on ML, visualization tools and work-flow engines for the exascale applications.

1.6 Funding

This Ph.D. is funded by the spoke "FutureHPC & BigData" of the ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing – and hosting entity, funded by European Union – NextGenerationEU" and it is supported by the European High Performance Computing Joint Undertaking (JU) and Belgium, Czech Republic, France, Germany, Greece, Italy, Norway, and Spain under grant agreement No 101093441.

[2]https://www.supercomputing-icsc.it/spoke-1-future-hpc-big-data/
[3]https://www.space-coe.eu/

Chapter 2

Background and Related Works

In this chapter, we present background information and talk about the relevant tools and works that form the foundation of this thesis. We start with an introduction to VisIVO, the main visualization software on which we are focusing our research. Then, we present the tools, codes, and technologies that have been crucial in the evolution of VisIVO, including simulation codes, code analysis tools, parallel programming frameworks, workflow management systems, and data management tools.

2.1 VisIVO

Visualization is the process of representing data graphically, both with 2D and 3D images. Modern computing systems, with new software solutions, allow for interaction with the represented data, enabling users to gain insights or new information from the data being displayed.

VisIVO[11] is a collection of graphical applications that enable data visualization and exploration, as well as providing techniques for analytical visualization. One of these applications is VisIVO Server, which is a platform for visualizing large datasets that can be installed on any server and can work in conjunction with VisIVO Desktop, another application in the suite that can be installed on standard PCs. VisIVO Server allows the creation of custom 3D rendering visualizations using data tables, without any fixed dimensional limits as input. VisIVO Server consists of three main components: VisIVO Importer, VisIVO Filter, and VisIVO Viewer.

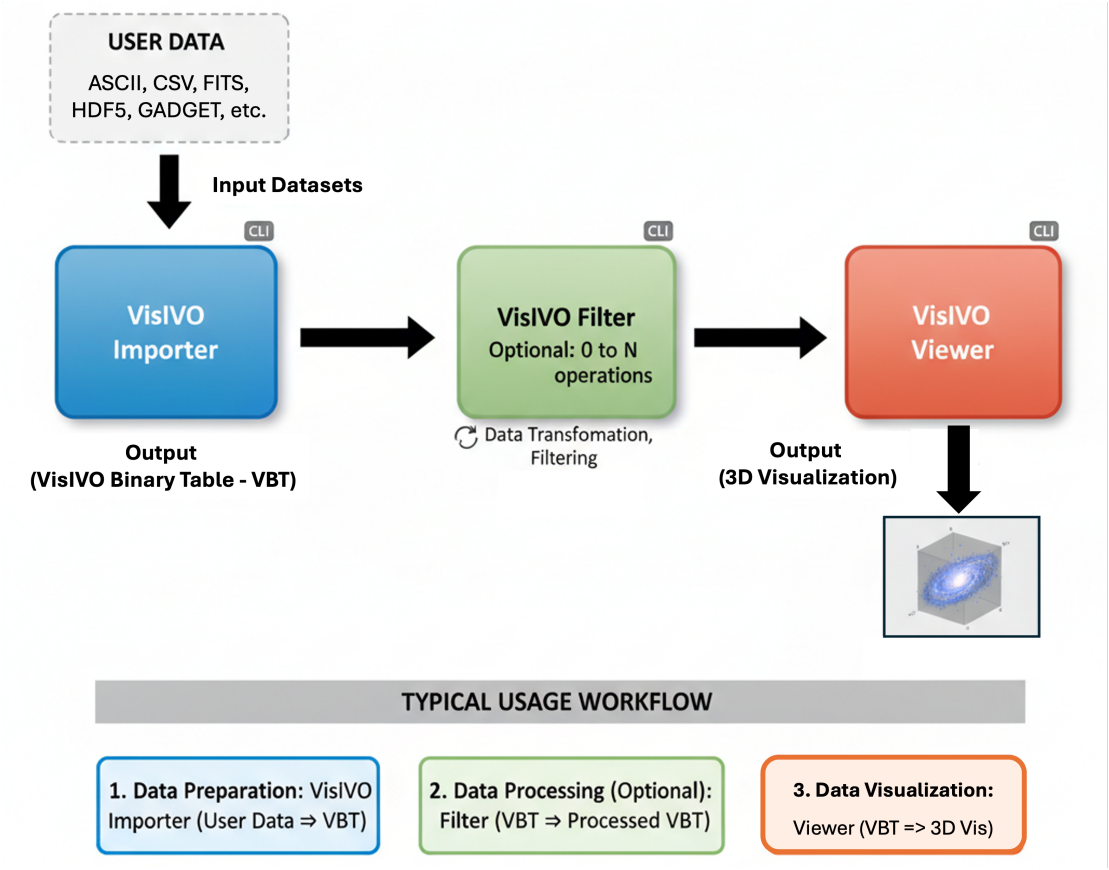

FIGURE 2.1: Typical VisIVO Server components usage

To understand how the different VisIVO components interact, it is useful to describe the logical architecture of the VisIVO system, as illustrated in Figure 2.1. VisIVO Server is designed as a pipeline in which each component performs a specific transformation on the data and passes its output to the next stage. This modular design allows users to compose complex workflows starting from raw scientific data up to interactive 3D renderings.

At the beginning of the workflow, the VisIVO Importer acts as the entry point of the system. It reads datasets in various scientific formats (e.g., FITS[50], CSV, binary simulation data such as GADGET[109] or TIPSY[91]) and converts them into the internal VisIVO Binary Table (VBT) representation. VBTs provide a compact, uniform, and high-performance structure that is used by all other VisIVO components, enabling interoperability and efficient I/O on HPC systems. More details about the VBT format are available in section 4.2.2.

Once the data have been imported, the VisIVO Filter modules operate on the generated VBTs. Each filter performs a specific transformation or analysis, such as selection of subsets, computation of derived quantities, point distribution to volumetric grids, or smoothing operations. Multiple filters can be combined sequentially to build customized processing chains, effectively creating a data analysis workflow. Filters are implemented independently, so that new operations can be easily added without modifying the rest of the system.

Finally, VisIVO Viewer uses the Visualization Toolkit[73] library for multidimensional visualization to create 3D images using VBTs.

The final step is handled by the VisIVO Viewer, which generates visual representations of the processed data. Based on the Visualization Toolkit (VTK) library, the Viewer can produce 3D renderings of scalar or vector fields, isosurfaces, or particle distributions.

Together, these components form a loosely coupled architecture in which each stage communicates through the exchange of VBTs.

Another available component is VisIVO Library, which is an application programming interface developed for porting VisIVO Server to the gLite middleware and can be installed on common Distributed Computing Infrastructures (DCI).

The tools provided by VisIVO enable researchers to analyze various scientific problems, such as visual comparison of large-scale cosmological simulations, online visualization of large-scale cosmological simulations, or particle tracking visualization.

2.2 Tools and Codes

In this section, we report all the tools and codes involved in the evolution of VisIVO. In the following chapters, we will explain in detail how each of them was used.

2.2.1 Simulation Codes

VisIVO supports a variety of snapshot types from different simulation codes for the conversion to VBT using VisIVO Importer.

In cosmological simulations, a snapshot represents a data dump capturing the state of the simulated universe at a given simulation time, including various physical properties of all the particles, such as the positions and velocities.

In the following chapters, we will present multiple works that involved the importers for two main simulation code snapshots: GADGET and ChaNGa.

GAlaxies with Dark matter and Gas intEracT: GADGET

GAlaxies with Dark matter and Gas intEracT(GADGET) is a code for cosmological N-body/SPH simulations on massively parallel computers with distributed memory[109]. Gadget is one of the most important software for cosmological simulation used by the scientific community. It uses an explicit communication model that is implemented with the standardized MPI communication interface. It makes use of a hierarchical tree algorithm for the gravitational forces and represents fluids by means of smoothed particle hydrodynamics.

The high volume of data generated by these simulations can be useful for the design and development of efficient and parallel models needed for advanced visualization techniques and big data analysis.

GADGET stores its simulation outputs as binary dumps containing the complete state of all simulated particles at a given simulation time. Each snapshot typically includes particle positions, velocities, identifiers, and physical quantities such as density, internal energy, and smoothing length. The data are organized by particle type (gas, dark matter, stars) and may be distributed across multiple files when simulations are run in parallel. For further details about GADGET format read chapter 4 section 4.2.1.

ChaNGa

ChaNGa[81][1] (Charm N-body GrAvity solver) is a code that performs collisionless N-Body simulations. It supports cosmological simulations with periodic boundary conditions in comoving coordinates, as well as simulations of isolated stellar systems. In addition, it can incorporate hydrodynamics using the Smooth Particle Hydrodynamics (SPH) method. Gravity calculations are performed using a Barnes-Hut tree algorithm, which includes hexadecapole expansion of nodes and Ewald summation for handling periodic forces. The code employs a leapfrog integrator with individual timesteps assigned to each particle for time integration. ChaNGa stores data using Tipsy, a binary file format used primarily in astrophysical simulations to store snapshot data of N-body systems, it can store three types of particles: gas, dark matter, and star.

2.2.2 Code Analysis Tools

2.2.3 Software Quality and Refactoring Principles

The analysis and restructuring of large scientific codebases require not only domain knowledge but also solid software engineering principles. As systems evolve over long periods, they tend to accumulate architectural degradation and "code aging" [82], a situation where structural quality decreases due to continuous patch-based updates rather than comprehensive redesign. Maintaining code quality is essential to ensure long-term sustainability, reproducibility of results, and the ability to extend the software to new architectures or scientific features [125, 51].

[1] https://github.com/N-BodyShop/changa

Software quality in scientific computing is typically understood through core attributes such as reliability, maintainability, portability, and performance efficiency, which have long been recognized as essential for the sustainability of scientific software [51, 125]. These attributes influence how easily new features can be implemented, how confidently results can be reproduced, and how effectively the software can adapt to new computing architectures.

To measure these attributes, developers rely on software metrics, which provide quantitative indicators of structural quality. Metrics are widely used both in industrial and research settings and have been shown to correlate with maintainability, defect probability, and comprehension effort [39, 65]. They can be grouped into several categories: In this context, metrics-based static analysis and dynamic profiling tools play a central role. Software quality is typically evaluated across several dimensions, including:

- **Size metrics** such as lines of code (LOC) or number of classes, which are simple but useful for estimating maintenance effort [103, 75]. While coarse-grained, they are commonly used as normalization factors for more advanced metrics;

- **Complexity metrics**, including cyclomatic complexity [79], Halstead's software science metrics [49], and cognitive complexity [21]. Numerous empirical studies show that high complexity is associated with increased defect rates and lower maintainability [130, 99];

- **Coupling and cohesion metrics**, with widely used suites such as Chidamber and Kemerer (CK) [25] and MOOD metrics[1]. Excessive coupling reduces modularity and hinders reuse, especially in HPC codes where portability and optimization for new architectures depend on well-defined module boundaries [60, 56]. High cohesion, conversely, is strongly associated with better maintainability and understandability[17];

- **Maintainability metrics** such as the Maintainability Index (MI) [93], which integrate size, complexity, and comment density to approximate the effort required to modify the system. MI has been successfully applied to large-scale scientific software to identify erosion and aging [82, 78].

Architectural aspects also influence software quality in scientific applications. Metrics measuring module dependencies, cyclic structures, and architectural conformance [64, 86] help identify large-scale design issues that are not visible at the function or file level. In HPC contexts, such analyses have been used to study communication patterns and data-flow structures in complex applications [20].

Considering that scientific software evolves incrementally, these metrics are effective when tracked over time, revealing trends such as increasing coupling or "architectural drift" [58, 96]. Modern continuous-integration workflows further reinforce quality by using automated analysis to detect regressions earlier[55, 104].

Metrics produced by code analysis are therefore valuable indicators for identifying modules that may require attention or redesign. Similarly, automatic documentation tools help ensure that the system remains understandable as it evolves.

Refactoring

Refactoring[63] is the process of restructuring existing code without altering its external behavior. Its primary goal is to improve internal attributes of the software, such as clarity, modularity, or extensibility, while preserving the functionality. In scientific software, refactoring is particularly important because codebases often evolve organically over years or decades, incorporating contributions from multiple developers with heterogeneous backgrounds[51, 125]. Without systematic refactoring, such codes tend to accumulate technical debt, including duplicated code, tightly coupled modules, and ad-hoc modifications, which hinder maintenance, testing, and reproducibility of results [78, 23].

Common refactoring tasks include decomposing large functions, isolating platform-dependent components, reducing code duplication, improving naming conventions, or introducing modular interfaces. Empirical studies show that such improvements not only reduce the risk of defects but also enhance code readability and support the implementation of new features [100, 127].

Moreover, in high-performance computing applications, refactoring can be critical for adapting legacy codes to modern parallel or heterogeneous architectures without altering numerical correctness [37, 36].

In software development, refactoring contributes to: improving testability, as clearer module boundaries facilitate unit and integration testing; improving readability; improving maintainability, reducing technical debt, which consists of the future effort required to correct suboptimal design and implementation decisions made during development and it accumulates during the rapid prototyping phase typical of research code.

Refactoring must, however, be guided by reliable feedback. For this reason, static analysis tools (such as CppDepend) and documentation tools (such as Doxygen) help identify structural issues, while dynamic analysis tools (such as Valgrind) ensure that no new memory related errors are introduced during the restructuring process. In subsection 2.2.3 is available a comprehensive description of all the tools we considered for the work in this thesis.

The combination of static and dynamic analysis tools forms an essential workflow for safe and effective refactoring. For example:

- **Metric-based tools** such as CCCC and CppDepend highlight modules with excessive complexity or tight coupling, suggesting where refactoring efforts should be prioritised.
- **Documentation tools** such as Doxygen provide a global view of class hierarchies, dependencies, and data structures, which is indispensable when modifying legacy code.
- **Dynamic analysis tools** such as Valgrind to detect memory leaks, invalid memory accesses, and abnormal memory growth, ensuring that behavioural correctness is maintained after modifications.

Taken together, these tools enable a systematic approach to improving software quality. They support informed decision-making during the refactoring process, reduce the likelihood of introducing regressions, and ultimately contribute to the reliability and maintainability of the software systems used in scientific computing.

CCCC

CCCC[2] (C and C++ Code Counter) is a tool designed for analyzing C, C++, Java, and Ada source code to provide software metrics reports. It generates HTML and XML reports that include key metrics, such as lines of code and cyclomatic complexity, to assess the quality and complexity of code. CCCC can help developers to understand their code's structure and identify potential issues, with thresholds set to flag areas that may require attention, making it useful for both code review and maintenance.

CppDepend

CppDepend[3] is a static analysis tool designed for C/C++ code. It offers extensive support for various code metrics and provides visualization features, such as directed graphs and dependency matrices, to analyze dependencies. The tool enables comparison between snapshots of the codebase and validates architectural and quality rules. Users can define custom rules using LINQ queries, a feature known as CQLinq, alongside a comprehensive set of predefined CQLinq code rules included with the tool.

Doxygen

Doxygen[4]is an open source tool designed to generate documentation directly from source code in a variety of languages, including C++, Java, Python, and more. It extracts comments formatted in a specific way within the code, enabling developers to automatically create HTML, LaTeX, or PDF documentation. By cross-referencing elements within the code, Doxygen builds an interconnected documentation structure, which includes class diagrams, inheritance hierarchies, and other visual aids. This functionality is especially beneficial for large projects, where maintaining up-to-date documentation is challenging, as it ensures that the documentation stays close to the source code itself.

[2] https://sarnold.github.io/cccc/CCCC_ToolUser_Guide.html
[3] https://www.cppdepend.com/
[4] https://www.doxygen.nl/index.html

Valgrind

Valgrind[5] is a widely used dynamic analysis framework designed to detect memory-related issues during program execution. It consists of multiple tools, the main one is memcheck, which identifies common memory errors such as invalid reads and writes, use of uninitialized values, incorrect deallocations, and memory leaks. Valgrind executes the program in a controlled environment through dynamic instrumentation, enabling precise identification of problematic code paths without requiring changes to the source code. It produces detailed reports that highlight the origin and nature of each issue, providing valuable insight into the runtime behavior of applications. In the context of large C++ projects, Valgrind is particularly useful for validating memory safety, ensuring the correctness of refactoring steps, and diagnosing subtle bugs that are difficult to detect through static analysis alone.

Another important tool within the Valgrind suite is Massif, which focuses on analyzing heap memory usage over time. It tracks allocations, deallocations and overall memory growth, producing a detailed report that highlights which parts of the program are responsible for significant memory consumption. The generated output is visualizable and shows memory usage snapshots and call stacks, enabling developers to identify memory bottlenecks and unnecessary allocations.

2.2.4 Parallel programming frameworks: Message Passing Interface and OpenMP

MPI is a standard for distributed-memory parallel programming defined by the MPI forum[6]. In an MPI program, each process(also called a rank) has its own private address space; data movement and synchronization are explicit and occurs via function calls. Communication can occur both within shared memory nodes(intra-node) and between nodes(inter-node).

An MPI job typically follows the SPMD (Single Program, Multiple Data) model, in which many identical processes are launched at program start, and

[5] https://valgrind.org/
[6] https://www.mpi-forum.org/

each process is assigned an integer rank within a communicator. An MPI communicator groups a given number of processes to allow them to communicate with each other and provides a unique communication context that isolates their messages from those of other groups. The default communicator MPI_COMM_WORLD, contains all processes started together, and it is possible to construct subsets using MPI_Comm_split to obtain additional communicators with their own process groups and communication context.

A typical MPI program has a logical structure composed of three main stages:

1. **Initialization:** all processes call `MPI_Init`, establishing the communication environment.
2. **Parallel computation:** each process obtains its rank with `MPI_Comm_rank`, the total number of processes with `MPI_Comm_size`, and exchanges data through point-to-point or collective communications.
3. **Finalization:** all processes call `MPI_Finalize` to terminate the MPI environment.

A minimal MPI program in C++ is shown below, in this program, an array is divided among processes, each of which computes a partial sum of its portion. The partial results are then combined into a global sum using `MPI_Reduce`:

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <numeric> // for std::accumulate

int main(int argc, char** argv) {
MPI_Init(&argc, &argv); // Initialize MPI
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

const int N = 16; // total number of elements
```

```
const int local_N = N / size; // per process
std::vector<int> local_data(local_N);

// Each process initializes its portion
for (int i = 0; i < local_N; i++)
    local_data[i] = rank * local_N + i + 1;

// Compute partial sum locally
int local_sum = std::accumulate(local_data.begin(),
                                local_data.end(), 0);

int global_sum = 0;

// Combine partial results into a single value on rank 0
MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0,
           MPI_COMM_WORLD);

if (rank == 0)
    std::cout << "Global sum = " << global_sum << std::endl;

MPI_Finalize();
return 0;
}
```

In this example, each process computes a portion of the total sum independently, and the `MPI_Reduce` collective operation gathers all partial sums into process 0, which prints the final result. This illustrates both distributed data decomposition and the use of collective communication, which are central to real-world MPI programs.

In large-scale applications, communication efficiency is crucial. MPI provides both *point-to-point* communications (`MPI_Send`, `MPI_Recv`) and *collective* operations that involve all processes in a communicator. For example:

- `MPI_Bcast` distributes the same data from one process to all others (broadcast).

- `MPI_Reduce` combines data from all processes (e.g., computing a global sum, as in the example).
- `MPI_Gather` collects data from all processes into a single array on the root process.

Of course communication should be handled efficiently; generally, sending fewer but larger messages is more efficient than sending many small ones, because sending each message individually bears a fixed latency cost that accumulates with each message. Instead, when grouping many requests together latency will affect communication only once. Moreover, using nonblocking communication calls (such as MPI_Isend and MPI_Irecv) allows computation to proceed while data transfers occur in the background, effectively overlapping communication with computation and reducing idle time among processes.

MPI is implemented as a collection of libraries, header files, and compiler options that extend standard programming languages to support distributed-memory parallelism.

Overall, MPI is a fundamental technology for exploiting HPC infrastructures, as it allows efficient multi-node execution, where multiple compute nodes cooperate on the same problem through coordinated message passing.

OpenMP[33], on the other hand, is an API which provides support for parallel programming in shared-memory environments. A shared-memory environment consists of a computing architecture in which multiple processing cores access the same physical memory space, allowing threads to communicate by directly reading and writing shared variables instead of exchanging messages. It consists of a set of compiler directives with a lightweight syntax, library routines, and environment variables that influence run-time behavior.

The key concept is that code regions can be marked as parallel using annotations (#pragma directives in C/C++), allowing the compiler to automatically generate the necessary thread management logic.

For example, the directive `#pragma omp parallel for` splits the iterations of a loop among multiple threads:

```
#include <omp.h>
```

```
#include <iostream>

int main() {
const int N = 8;
int data[N];
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    data[i] = i * i;
    printf("Thread %d computed data[%d] = %d\n",
           omp_get_thread_num(), i, data[i]);
}
return 0;

}
```

This program automatically distributes the iterations across available CPU cores.

Other useful directives include:

- `omp parallel`: creates a parallel region.
- `omp for / omp do`: splits loops between threads.
- `omp critical`: defines a mutually exclusive region.
- `omp barrier`: synchronizes all threads.

OpenMP allows flexible configuration of thread count at runtime (via `OMP_NUM_THREADS`) and supports nested parallelism. It is especially suitable for parallelizing compute-intensive loops or modules, such as the data import and processing steps of VisIVO modules, where shared-memory execution avoids interprocess communication overhead.

There are multiple other multithreading approaches such as POSIX Threads (Pthreads)[57] and Intel Threading Building Blocks (TBB)[30]. POSIX Threads

provide a low-level interface that requires explicit thread management, synchronization, and workload division, which can lead to complex and error-prone code. Intel TBB offers higher-level constructs for task parallelism but still requires a dedicated library and explicit use of its API.

OpenMP grants these main advantages over other solutions:

- cross-platform and cross-compiler solutions
- flexible and convenient to modify the number of threads (with environment variables)
- no entry functions, code within the same function or loop will be decomposed into multiple threads

OpenMP is supported by many compilers and is typically available in every machine. In particular, it is available for C++ so it can be useful also for the parallelization of VisIVO modules.

2.2.5 Workflows

Workflows can be defined as a powerful abstraction for designing complex applications and executing[26] them on large-scale distributed architectures, which consist of heterogeneous computing environments where multiple infrastructures, such as HPC clusters, cloud systems, and data services, collaborate to execute distinct tasks within the same workflow. Some examples of large-scale distributed architecture that were considered during our work are Cineca's Galileo100[7] and Leonardo[8], or HPC4AI[9] by University of Turin.

Workflows were also defined by the Workflow Management Coalition[121] as the automation of a business process, in whole or part, during which document, information, or tasks are passed from one participant to another for an action according to a set of procedural rules.

In a more generic term, workflows can also be defined as a directed bipartite graph[71], where the nodes represent either computational steps or the ports

[7] https://www.hpc.cineca.it/systems/hardware/galileo100/
[8] https://www.hpc.cineca.it/systems/hardware/leonardo/
[9] https://hpc4ai.unito.it/

through which they communicate. The edges represent the dependency relations between steps, linking steps to ports and ports to subsequent steps.

2.2.6 Workflow Management Systems

Workflow Management Systems (WMSs)[128] provide the coordination semantics required to structure, orchestrate, and monitor scientific applications composed of multiple computational stages. Their main goal is to enable users to describe, execute, and reproduce complex pipelines in a scalable and transparent manner. Although the general idea is common across systems, WMSs differ significantly in their execution models, architectural assumptions, and target computing environments.

A first distinction concerns the *workflow execution model*. Traditional systems implement a *static* model, where the workflow is defined as a fixed sequence or a Directed Acyclic Graph (DAG). This is well suited for batch pipelines but becomes limiting for emerging applications that require *dynamic* behaviour, including runtime loops, adaptive branching, iterative refinement, and interactive steering. Only a subset of WMSs can naturally express these patterns, highlighting a gap between classical workflow engines and next-generation scientific workflows.

Another important dimension is the *target execution environment*. Some WMSs prioritize usability and accessibility, offering high-level abstractions or graphical interfaces intended for domain scientists. Others focus on large-scale, heterogeneous, or distributed infrastructures, integrating with HPC schedulers, cloud orchestration frameworks, or container ecosystems. These design choices lead to systems optimized for very different usage scenarios.

Data management models also vary substantially. File-based communication remains common, but the increasing relevance of in situ and in transit processing has motivated the adoption of streaming abstractions, memory-based data exchange, and data-aware scheduling. These capabilities are critical in workflows where intermediate data cannot be written to disk due to performance or scalability constraints.

A scientific WMS refers to a framework specifically designed to support the automation, execution, and monitoring of computational or data-driven experiments in scientific research and provide features such as provenance tracking, reproducibility, and support for heterogeneous computing environments.[6]

For a more detailed survey of scientific workflow management systems and representative examples, we refer the reader to the dedicated Related Work section 2.3.3.

Streamflow

StreamFlow[29] is a WMS designed to bridge cloud and High-Performance Computing environments. The primary aim of StreamFlow is to enable the execution of complex workflows across heterogeneous computing platforms, such as Kubernetes clusters and traditional batch-based HPC systems, without relying on shared data spaces.

It addresses the need to improve portability of applications by decoupling multiple components from platform-specific dependencies. This separation enables the workflows to be executed across different High-Performance Computing (HPC) infrastructures without relying on a specific underlying environment. By abstracting execution environments from the application logic, the system ensures that workflows can be transferred and executed on heterogeneous systems with little changes.

In addition to enhancing portability, this work raises the level of abstraction by using StreamFlow to define workflows that streamline complex task orchestration. StreamFlow allows tasks to be expressed in a high-level, platform-independent manner, facilitating the seamless management of dependencies and execution across various environments. This level of abstraction is critical for ensuring the scalability and adaptability of workflows, enabling them to run efficiently on both local clusters and cloud-based platforms.

StreamFlow was selected for the work presented in this thesis for several reasons that align directly with the requirements of our workflow scenarios:

- **Container-based execution for portability and reproducibility.** StreamFlow integrates seamlessly with Docker and Singularity, ensuring consistent

execution across heterogeneous environments and simplifying software deployment for complex scientific applications.

- **Suitability for hybrid HPC-cloud workflows.** The system natively supports the execution of tasks across multiple backends (e.g., Kubernetes, SLURM), making it well suited for use cases requiring elastic, distributed, or hybrid infrastructures.
- **Support for streaming and data-aware execution.** StreamFlow can manage data exchanges efficiently and supports streaming communication between tasks.

These characteristics make StreamFlow a strong match for the workflow orchestration challenges addressed in this thesis, providing both the flexibility and the portability required to support complex scientific pipelines across heterogeneous computing environments.

2.2.7 Hecuba

Hecuba[10] is a set of tools and interfaces that aims to facilitate the management and utilization of persistent data for Big Data applications. Hecuba implements an Object Mapper for Apache Cassandra [69] (a recognized NoSQL database) and thus allows programmers to use a common interface to access data as regular in-memory objects, regardless they are persistent (stored in disk) or they are in-memory data. Currently Hecuba implements an interface for Python and C++ programming languages.

Hecuba is currently being extended to also support a lambda architecture [77] which is a data-processing architecture defined to speed up online analysis for Big Data applications, without losing the ability to persist the output data of the applications. It also supports Apache Kafka[68] for streaming data. Thus, Hecuba is able to support both off-line and on-line data analysis. The programmer can use the same interface to access the data whether it is in memory, in the storage or it is coming through a stream: he only needs to modify the class definition of the object to indicate which kind of object it is.

[10] https://github.com/bsc-dd/hecuba

Several other tools in the literature have proposed similar solutions to both facilitate management and utilization of persistent data in big data applications. For example, Legion [10] is a data-centric programming model and runtime systems which decouplse logical data descriptions from their physical layout and automatically manage data placement across distributed memory hierarchies. Although Legion provides transparency in terms of memory locality and movement, it does not support persistent data storage. Another group of tools focuses on abstracting I/O backends, the first one is ADIOS2 [45] which enables scientific applications to write and read data through a unified API while switching between file-based, in-memory, or streaming backends without modifying application code. Similarly, the HDF5 Virtual Object Layer (VOL) [90] allows existing HDF5 applications to operate on different storage systems transparently, although the abstraction is expressed in terms of datasets and attributes rather than language-level objects.

Frameworks designed for distributed in-memory data, such as Dask [102] or Ray [89], also provide location transparency by allowing applications to manipulate distributed data structures without explicitly managing their placement. However, these systems primarily target in-memory analytics and do not support persistent NoSQL databases or streaming sources. Other persistent data abstractions, such as object-relational mappers (e.g., SQLAlchemy [11]), simplify access to traditional databases while preserving object-oriented programming models, but they are not designed for HPC environments, lack data-locality optimizations, and do not unify in-memory, database, and streaming backends.

Compared to these approaches, Hecuba supports Python and C++ applications an API that transparently maps objects to in-memory data, persistent database stores using Cassandra, and streaming systems such as Kafka. This combination of features for both offline and online data analysis is not provided by other explored tools.

[11] https://www.sqlalchemy.org/

Apache Cassandra

Apache Cassandra is a distributed NoSQL database originally developed by Facebook in order to effectively manage massive amounts of unstructured data efficiently. Since Cassandra does not impose a strict table schema like conventional relational databases do, each row can contain a different number or a set of columns. It is designed to operate seamlessly across clusters of multiple nodes, and it is also built to tolerate hardware failures without disrupting operations, ensuring high availability and fault tolerance. Its architecture grants robustness, allowing it to work reliably even with inexpensive hardware, and it offers strong scalability, giving users control over data configuration and system expansion.

Apache Kafka

Apache Kafka is an open-source platform designed for real-time data stream processing and originally developed by LinkedIn to serve as a high-throughput messaging queue, it has since evolved into a comprehensive solution focusing on managing and analyzing event streams in real time.

At a high level, Kafka is based on a publish–subscribe communication model[92]. Producers publish messages to topics, which represent named categories of data. Each topic is divided into partitions, allowing parallel writing and reading across multiple brokers (the servers that form a Kafka cluster). Messages are appended sequentially to partitions and replicated across brokers to ensure fault tolerance and high availability[120]. Consumers subscribe to topics and read the data at their own pace, maintaining an offset that indicates the last processed message.

This architecture decouples data producers and consumers, allowing asynchronous communication and enabling multiple independent applications to process the same data stream simultaneously.[66] Kafka ensures durability by persisting data on disk and provides scalability by distributing partitions across nodes, making it suitable for large-scale deployments.

On top of its core messaging layer, Kafka provides several key components:

- **Kafka Streams**: a lightweight library for building stream-processing applications directly on top of Kafka topics, supporting real-time operations such as filtering, joining, and aggregation.
- **Kafka Connect**: a framework for integrating Kafka with external systems (databases, file systems, analytics tools) through reusable connectors.

Through these components, Kafka enables real-time analytics pipelines capable of ingesting, processing, and distributing data continuously across distributed environments, making it a fundamental technology for modern event-driven architectures.

2.2.8 InterActive Computing service (IAC)

The increasing complexity and interactivity of modern scientific research workflows require compute resources that extend beyond traditional batch processing. In response to this demand, the Fenix[12] infrastructure, which is a European research initiative that connects multiple high-performance computing and data centers to provide advanced computational and data services for scientific communities, introduces InterActive Compute services (IAC) as a core component of its federated service portfolio, targeting the specific needs of the neuroscience community within the Human Brain Project[13] (HBP) and its successor, EBRAINS[14]. The IAC implementation in Fenix was managed by Cineca[15], a non-profit inter-university consortium and Italy's largest supercomputing center, which plays a leading role in national and European HPC initiatives. The project was carried out in collaboration with E4 Computer Engineering (E4), an Italian company specializing in high-performance computing solutions. The system is based on the ICE4HPC suite[16] developed by E4, and it is currently up and running on the Galileo100 cluster at Cineca.

The main benefits for an interactive approach to HPC resources are mainly two.

[12]Fenix, https://fenix-ri.eu/
[13]Human Brain Project, https://humanbrainproject.eu/
[14]EBRAINS, https://ebrains.eu/
[15]Cineca, https://www.cineca.it/it
[16]ICE4HPC, https://www.e4company.com/en/ice4hpc/

1. Users can access directly to the compute node of an HPC system using a web browser interface (available at `https://jupyter.g100.cineca.it`). This is a fundamentally different approach in comparison to the traditional user experience on HPC clusters, which traditionally involves *ssh* access to a single shared login node, and a batch system allowing job submissions in a queue system. The traditional approach, relying only on command line, inhibits any graphical visualization of results, which is on the contrary very smooth on the interactive web interface.

2. HPC resources are guaranteed in a near-instantaneous access to allow an immediate interaction via web browser. This represents a different philosophy with respect to the above-mentioned batch approach (where the job start time is unpredictable by the user). This approach adds the possibility of an on-the-fly interaction with the workflow while the system is running [61], a real-time monitoring of the resource usage and visualization of the intermediate results to make decisions on the following steps of the workflow.

The general implementation of the service is depicted in Figure 2.2. The framework is composed of three main parts:

1. a frontend virtual machine (VM), which exposes the website to the world wide web (it is not exposed from the cluster login nodes for security reasons). The interface asks for user credentials and Two-Factor authentication, then it shows a form to be filled with drop-down menus to request resources to be allocated on the HPC cluster

2. such requested resources are used to submit a job on the cluster on a dedicated partition, as such a VM is able, exceptionally, to communicate directly with the cluster Slurm controller. The job will run a Jupyter-based [67] server instance (detailed later) which is tunneled to the VM to be exposed externally

3. the near-istantaneous access is guaranteed since the dedicated Slurm partition ("backend nodes" in Figure 2.2) is provided in oversubscription, thus, in the unlucky hypothesis of a fully allocated partition, multiple users would share the same CPU. Oversubscription is not applied to GPUs, which are, on the contrary, allocated exclusively.

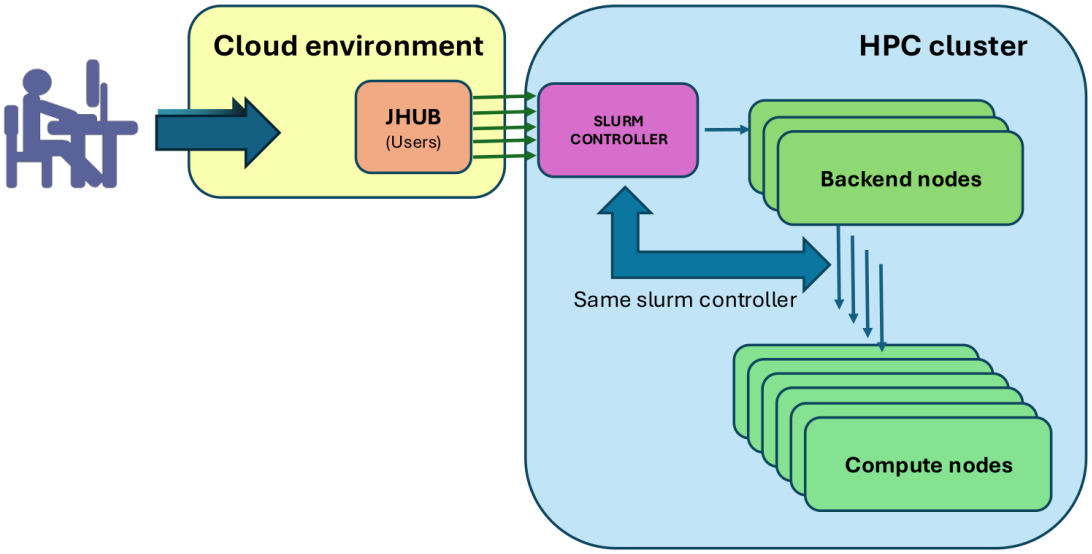

FIGURE 2.2: InterActive Computing service implementation at Cineca

The choice of a Jupyter-based framework for the IAC implementation brought several advantages. Firstly, it already includes a server-client approach (Jupyter Single-user) among its possible implementations. Secondly, it includes a plugin system which allowed us to extend the interface to better address the HPC capabilities (e.g. monitoring tools, jupyter-server-proxy plugin to proxy additional http-based servers through Jupyter dashboard). Lastly, it is a well-known environment for many users who approach an HPC environment for the first time, allowing a more user-friendly impact with the infrastructure.

The security of the service is strongly improved by the fact that all the operations which might be critical to run on the HPC cluster are delegated to a VM running on Cineca cloud infrastructure. The backend part running on the HPC cluster does not need root privileges, and it works like an ordinary SLURM job in user space. In addition, user login is enforced via 2FA authentication, and constant monitoring of potential security issues, as well as periodic vulnerability assessments and penetration tests (VAPT), are currently performed.

2.3 Related Works

This section presents the related works we studied and that influenced our research.

2.3.1 Code Analysis and Refactoring

Code analysis tools have been widely adopted in modern software engineering because they provide quantitative and qualitative insights into the structure and quality of source code. Rather than relying solely on manual inspection, developers use these tools to compute software metrics that capture key properties such as size, complexity[79], cohesion, and coupling[25]. Tools such as *CCCC*, *Doxygen*, and *CppDepend*, introduced in the previous section 2.2.2, generate metrics that help developers understand how code is organized and where potential problems may lie. For instance, *CCCC* has been used to analyze LOC, cyclomatic complexity, and CK metrics in software quality studies [94], *CppDepend* has been applied to assess static quality metrics in multilingual codebases [129], and *Doxygen* has been used extensively to generate documentation for scientific software systems [7].

Commonly used software metrics include Lines of Code (LOC) [75], Cyclomatic Complexity[79], Coupling Between Objects (CBO), and Lack of Cohesion of Methods (LCOM)[25]. These metrics support the identification of typical code problems, such as overly large functions, deeply nested control structures, or modules that depend excessively on one another. When interpreted correctly, metric trends can reveal design degradation, the presence of technical debt, or regions of the codebase that may hinder maintainability and future evolution.

Although metrics are useful for evaluating existing code, they are equally important in guiding code refactoring activities. A growing body of research investigates how software metrics guide developers in refactoring decisions. For example, empirical studies[63] show that developers often apply classical refactoring techniques specifically to improve internal code metrics. Operations like "extract method" or "simplify conditional expressions" are frequently

motivated by the goal of reducing cyclomatic complexity or increasing cohesion. Similarly, large-scale analysis of open-source repositories [4] showed that developers' operations of cleaning up or improving quality of a project reliably align with measurable improvements in structural metrics.

These findings indicate that code metrics are not just evaluative tools; they actively influence how developers reason about code quality and shape their refactoring strategies. As a result, automated code analysis and dynamic analysis tools, such as those discussed in the previous section—serve as essential components to maintain long-term software quality, particularly in large and evolving scientific applications.

2.3.2 Parallel I/O

Parallel I/O is a fundamental research topic primarily used in scientific computing because the performance of many research applications is limited by the I/O system. Modern large-scale simulations and data-analysis pipelines often produce terabytes or even petabytes of data, requiring efficient transfer between memory and storage systems. To achieve optimal performance on HPC platforms, it is therefore essential to understand the fundamental advancements and details, at both software and hardware levels, in systems adopting parallel I/O.

At a high level, parallel I/O allows multiple processes to read from and write to files concurrently. Two primary approaches exist: individual I/O and collective I/O [111]. The individual approach is the simpler one: each process performs independent read or write operations, typically through POSIX calls or through the basic MPI-IO library [19]. Although this is quite flexible, it can lead to uncoordinated access patterns, excessive metadata operations, and poor performance when many processes contend for the same files.

In contrast, collective I/O involves coordination between processes. During an I/O operation, processes or threads pause until each thread reaches the same I/O instruction. When processes access contiguous or overlapping regions of a shared file, collective I/O can combine small requests into a large, efficient operation. This approach is particularly useful when each thread's I/O instruction requests only a small amount of data. Notably, research shows

that implementations of MPI-IO (such as ROMIO) perform optimisations like two-phase I/O (data redistribution + large contiguous I/Os) to improve performance for non-contiguous access patterns[114].

Collective I/O calls, such as those on MPI-IO expect to work only on a single file, so patterns involving multiple files are not best suited with this types of approaches. In the context of snapshots from cosmological codes, they usually involve multiple types of particles coming from a single file and we require to write the data on multiple VBTs, thus on multiple files.

However, parallel I/O is not limited to libraries such as MPI-IO. At the system level, several distributed parallel file systems have been developed to provide scalable and reliable access to shared data on large HPC infrastructures. Among the most widely adopted are Lustre, BeeGFS, and GPFS (IBM Spectrum Scale), which distribute files across multiple storage servers to achieve high throughput and parallelism [123, 32, 117].

These file systems share a common architectural model composed of *metadata servers (MDS/MDT)*, responsible for managing the filesystem namespace and stripe configuration [76], and *object storage servers (OSS/OST)*, which store the actual data distributed across disks or nodes. Files are typically divided into fixed-size chunks, or *stripes*, distributed among storage targets to enable concurrent read and write operations by many processes. This design allows the aggregate bandwidth to scale with the number of clients, while replication mechanisms enhance reliability and fault tolerance in case of server failures [117].

Recent studies and HPC infrastructure documentation have emphasized the importance of the interaction between I/O patterns at the application level and the underlying distributed storage architecture. Well-structured access patterns., such as large, contiguous, and properly aligned operations, can fully exploit the parallelism offered by the storage system, but small or unaligned writes often lead to metadata congestion and performance degradation [115]. Benchmarking analyses also show that design choices such as file striping strategies, buffer aggregation, and the one-file-per-process versus shared-file paradigms have a significant impact on scalability in scientific workloads [72].

In the context of cosmological simulations and large-scale data visualization workflows, these distributed I/O technologies provide the backbone for

efficient handling of particle-based datasets and the generation of derived data like VisIVO Binary Tables (VBTs).

2.3.3 Integration of scientific application with Workflow Management Systems

Workflow Management Systems (WMS) play a crucial role in scientific computing by automating the execution of complex, multi-stage pipelines that combine simulation, analysis, and data-intensive processing. As scientific applications grow in scale and complexity, WMSs help researchers manage task dependencies, handle data movement efficiently, and ensure reproducibility across heterogeneous computing environments. Multiple studies have been conducted on the state of the art available WMS and their characteristics[108], in particular studying workflow execution models, data access methods, and support for heterogeneous and extreme-scale environments.

A fundamental distinction between various WMS concerns the workflow execution models. Traditional pipelines rely on sequential or concurrent dataflow, patterns supported by systems such as Pegasus[34], Askalon[38], Kepler[9], Swift/T[124], and Makeflow[3]. More advanced scientific workflows incorporate iterative structures for convergence-driven or uncertainty quantification studies, as well as external steering and tightly coupled execution typical of multi-physics simulations or interactive in situ analysis. Only a subset of tools, such as Moteur[44], natively support steering, revealing limitations of current WMS for emerging extreme-scale use cases.

Equally important is the ability of WMS to operate across heterogeneous computing environments. Systems such as Pegasus, Swift/T, Makeflow, FireWorks[59], and dispel4py[40] target HPC clusters, Grids, and Cloud deployments, often integrating with batch schedulers or distributed execution engines. Other systems, such as Galaxy[15], Taverna[126], and Triana[112], prioritize user accessibility through graphical interfaces, making them particularly suitable for bioinformatics or domain scientists with limited programming background. At the same time, modern workflow engines such as Nextflow[35]

emphasize portability and reproducibility through containerization technologies (Docker, Singularity), which is increasingly relevant for hybrid HPC–cloud scientific environments.

Data access models also vary substantially across WMS. Traditional workflow engines primarily rely on local or shared file systems, while emerging systems integrate memory-based data sharing, streaming abstractions, or object stores. Some WMS such as ADIOS[74] and DataSpaces were designed specifically for in situ and in transit data movement, supporting tightly coupled simulation-analysis workflows. dispel4py, for instance, targets high-throughput data streaming, making it suitable for geoscience and sensor-driven applications. Makeflow, paired with Work Queue[18], is commonly used for scientific workflow ensembles and high-throughput campaigns.

In astronomy and astrophysics, WMS such as Pegasus have been adopted to orchestrate large-scale image processing pipelines, including the Montage mosaicking workflow[13]. These examples demonstrate how WMS enhance reproducibility and scalability for data-intensive astrophysical workflows, mirroring needs found in cosmological simulation post-processing, spectral analysis workflows, or ML-assisted pipelines.

Overall, the literature shows that Workflow Management Systems are indispensable for managing scientific pipelines at scale. They abstract away low-level execution concerns, improve reproducibility, and allow workflows to run efficiently across heterogeneous resources. However, significant challenges remain for next-generation, extreme-scale workflows—particularly regarding in situ processing, data-aware scheduling, scalable provenance capture, and hybrid workflows combining simulation, analytics, and machine learning.

2.3.4 In-situ visualization

In-situ visualization is an approach that is becoming essential in the context of high-performance computing, because the volume of data produced by simulations is becoming massive. In the literature there are available various surveys[101] focusing on multiple in-situ visualization techniques and tools, emphasizing the importance in scientific workflows that require real-time analysis and reduced I/O overheads.

There are three in-situ methods main categories:

- Tightly coupled: embed visualization routines within the simulation code, offering direct memory access and low latency but at the cost of synchronous execution and increased memory utilization.
- Loosely coupled: decouple computation and visualization processes, allowing them to run on separate resources and communicate via memory or network interfaces.
- Hybrid methods: combine both approaches, typically performing data reduction in-situ and forwarding results to external resources for further analysis.

There are multiple visualization frameworks that support in-situ visualization, one of them is the VisIt Libsim[122]. It is a library that exemplifies a tightly coupled approach by enabling simulation codes to act as a compute engine directly connected to VisIt.

Another example is ParaView Catalyst[8], another widely adopted solution, which offers flexible integration within simulation codes and supports both tightly and loosely coupled approaches.

These frameworks have demonstrated the feasibility and benefits of in-situ workflows, but multiple challenges remains, in particular the need of code instrumentation and resource contention.

2.3.5 Interactive computing infrastructures

Interactive computing is becoming a key capability in modern infrastructure for supporting scientific research. Unlike traditional batch processing, which is optimized for long-running simulations and queue jobs, interactive computing is a more immediate approach which allows the user to analyze data dynamically, prototype code, and control workflows in real-time. This is particularly valuable in scientific domains involving iterative refinement, visualization, and exploratory analysis.

There have been multiple attempts to integrate interactive computing with high-performance and distributed environments, one example is the Minnesota Supercomputing Institute[83] that has implemented Jupyterhub and

the Jupyter notebook server to perform traditional job-oriented HPC operations, while offering significant advantages for ease-of-use, data exploration and workflow development.

These types of platforms also help lower the entry barrier for users by providing ready-to-use environments with pre-installed visualization tools. This is particularly important for first-time users or those lacking access to high-performance local machines, as many scientific visualization applications require significant computational resources to run effectively. For example, ParaviewWeb[106] enables users to interactively visualize large-scale datasets via the web by performing the rendering on a remote server, streaming results to the client.

Chapter 3

VisIVO Server Code Analysis and Refactoring

3.1 Introduction

As scientific software evolves to meet increasingly complex research demands, maintaining a high-quality codebase becomes essential. VisIVO Server has been developed for a long time and, as it evolves, the addition of features such as in-situ visualization requires substantial refactoring.

Before introducing new features, it is important to understand the current state of the codebase. This chapter presents the results of a static analysis of the VisIVO Server source code to evaluate its maintainability, complexity, and extensibility. The analysis identifies potential bottlenecks and code smells and highlights opportunities for improvement, ensuring the software remains robust and efficient for current and future requirements.

By using industry-standard tools such as CCCC, CppDepend, and Doxygen, we collected key metrics on code size, control-flow complexity, and object-oriented design. These insights will help to perform refactoring to help VisIVO become better suited for integrating in-situ processing capabilities.

The primary motivation for this analysis is to improve the overall quality, maintainability, and extensibility of the VisIVO Server codebase. As software requirements evolve, it is essential to ensure the codebase remains maintainable, readable, and robust, enabling the seamless addition of new features and/or functionalities. Some of the aspects important in code analysis are:

- **Maintainability**: A maintainable codebase is essential for addressing bugs, incorporating updates, and supporting smooth collaboration among developers. By identifying areas for refactoring, such as large or complex methods, the project aims to simplify future maintenance efforts[52].
- **Readability**: Readable code fosters quicker onboarding of new developers and minimizes errors. Code analysis identifies areas with unclear structures or excessive complexity to improve code clarity and organization.
- **Addressing Code Smells**: Code smells, such as functions with high cyclomatic complexity, excessive dependencies, or large classes, can reduce software quality and increase technical debt.By leveraging static analysis tools, developers can try to mitigate these issues, promoting cleaner and more efficient code.
- **Facilitating Code Evolution**: The ability to evolve the codebase by seamlessly adding new functionalities is crucial for tools to meet growing research demands.

3.2 Methodology

To evaluate the VisIVO Server codebase, we employed three static analysis tools: CCCC, CppDepend, and Doxygen. Each tool offers different perspectives on the structure, complexity and documentation of the code.

3.2.1 Target Metrics

The analysis focused on a set of well-established software metrics commonly used in code quality evaluations. These metrics guided the interpretation of tool output and helped prioritize which classes or methods to refactor.

Lines of Code and Comments

Lines of code(LOC) metric measures the number of non-blank, non-comment source code lines within a function (LOCf), module (LOCm), or project (LOCp).

As one of the earliest code metrics, it became widely used due to its simplicity and ease of measurement.

The metric is directly linked to the size or complexity of the code and can be calibrated to predict maintenance effort. However, it has faced criticism when used as a measure of programmer productivity, as this may promote verbose coding practices and discourage the simplification of code.

Lines of Comments metric measures the number of comment lines within the project or a specific function, the comments may be useful for generating documentation with tools like Doxygen.

Cyclomatic Complexity

Cyclomatic complexity is a software metric used to measure the complexity of a program's control flow. It calculates the number of linearly independent paths through the program, providing an indicator of its structural complexity. This is determined by analyzing the control flow graph of the program, with the complexity increasing as the number of branches and decision points grows. A lower cyclomatic complexity generally indicates simpler, more maintainable code, while higher values may suggest increased difficulty in testing, debugging, and understanding the software. It is often used as a guideline to improve code quality and ensure adequate test coverage.

Object Oriented Metrics

There are multiple object oriented metrics[107] such as Depth of inheritance tree(DIT), which shows us the length of the longest path from a given class to the root class in the inheritance hierarchy, a higher value indicates that the class is deeper in the hierarchy, inheriting more methods and properties.

Number of Children (NOC), which measures the count of direct subclasses that inherit from a specific class and it does not include subclasses of those children, it represents the extent of reuse of a class functionality and if it has a high value it means that there is an higher potential to affect many other parts of the system.

Coupling Between Objects (CBO) is a software metric used in object-oriented programming to measure the level of interdependence between classes.

It quantifies how many other classes a particular class is directly dependent on, or how many other classes depend on it.

3.2.2 Doxygen analysis

Doxygen was used to generate documentation based on the current comments of the project. Although the comments were not designed to generate comprehensive documentation with Doxygen, this can be useful to start working to update it to obtain better documentation.

At the same time, it was also possible to generate multiple graphs such as inheritance graphs, which are very useful to understand the structures of the three modules: Importer, Filter, Viewer.

Inheritance Graphs

The inheritance graph of all the modules are very similar, there is a main parent class and the various specific components derive from this parent class.

For instance, Figure 3.1 shows the inheritance hierarchy of the `Importer` module, where a common base class defines the core interface and multiple subclasses implement format-specific behavior for the various type of supported snapshots (e.g., `GadgetSource`, `RamsesSource`, `HecubaSource`). Such visualizations help to quickly understand, and reason on, the organization of the codebase and highlight the modularity of software components.

All the images of the inheritance graph for the other modules are available in the appendix.

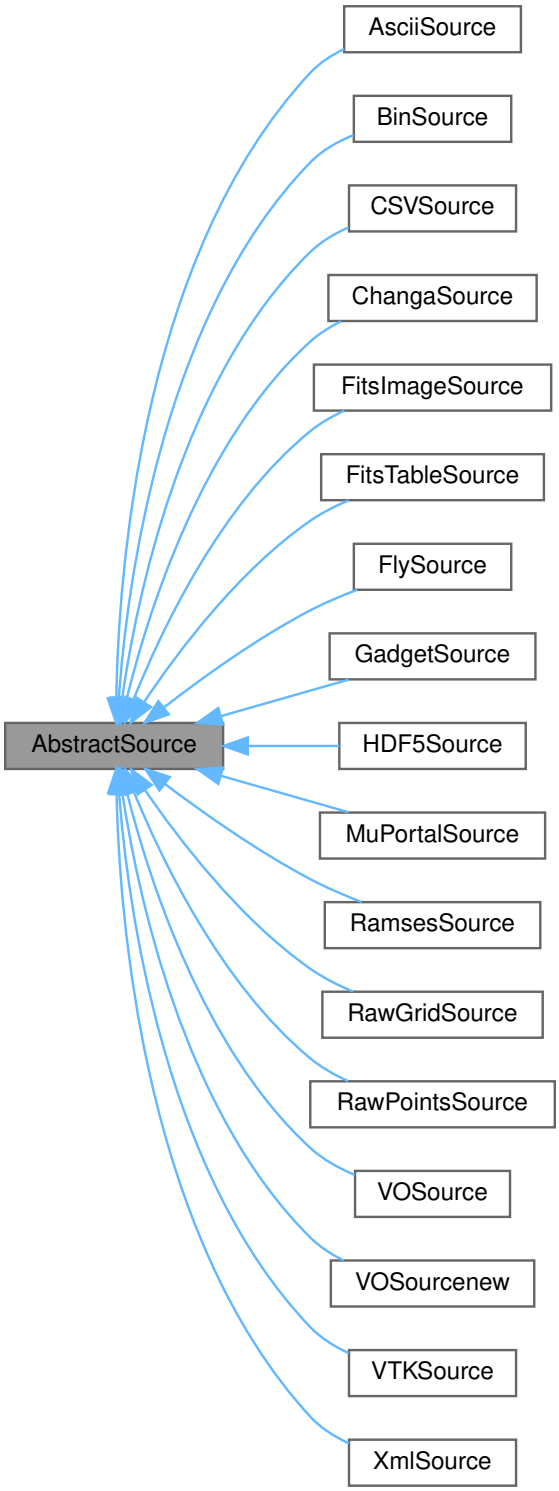

FIGURE 3.1: Inheritance Graph Importer Module

3.2.3 CCCC analysis

Importer analysis

CCCC generates a report using .HTML and .XML files. It produces a main summary HTML report in the file cccc.html, which displays the following metrics: Lines of Code (LOC), the sum of McCabe's Cyclomatic Complexity (MVG) for all the functions in the class, and Lines of Comments (COM).

The metrics for the classes in the importer module are presented in Table 3.1.

As an initial result of the analysis, the tool highlights the large size of the lines of code in the following classes: FitsTableSource, GadgetSource, and XMLSource. However, these data alone are insufficient for a complete analysis. It is necessary to examine the specific metrics for each function within these classes. These detailed metrics are also computed by CCCC, which generates a report for each class.

As an example, we consider the metrics for the GadgetSource class, shown in Table 3.2. The tool indicates that one of its functions, readData, is very big, with LOC of 361 and MVG of 78. This suggests that refactoring this function may be necessary to improve readability and maintainability [24].

Class Name	**LOC**	**MVG**	**COM**
AbstractSource	107	6	65
AsciiSource	242	60	71
BinSource	203	79	63
CSVSource	163	41	56
ChangaSource	172	39	70
FitsImageSource	177	32	16
FitsTableSource	553	114	77
FlySource	389	95	83
GadgetSource	557	116	88
HDF5Source	394	104	106
HecubaSource	146	15	25
MuPortalSource	213	54	69
RamsesSource	269	49	141

Class Name	**LOC**	**MVG**	**COM**
RawGridSource	236	55	64
RawPointsSource	359	76	82
VoSource	233	48	45
VoSourcenew	443	116	46
VTKSource	165	31	12
XmlSource	654	160	101

TABLE 3.1: General Procedural Metrics — Importer Classes

Function Name	**LOC**	**MVG**	**COM**
checkMultipleFiles	14	4	0
readData	361	78	71
readHeader	83	23	8
readMultipleHeaders	16	3	1
swapHeaderType1	29	4	3
swapHeaderType2	37	4	4
updateNpart2	1	0	0

TABLE 3.2: Procedural Metrics — `GadgetSource` Class

As a second result, we analyzed object-oriented metrics (DIT, NOC, CBO).

It is evident that AbstractSource is the parent class of all the other classes, which explains its high number of children. This hierarchy is also visually represented in the graph generated by Doxygen. The CBO for AbstractSource matches its number of children because it is not coupled with any other object.

Some of the child classes have a CBO value greater than 1. However, they do not communicate with each other, they interact with external classes, typically used to facilitate reading datasets, for example the ChangaSource relies on a specific library to read data in the XDR format.

The full per-class table is provided in Appendix A, Table A.8.

Filters analysis

In the filter module, there are numerous classes, 39 in total, and their procedural metrics—LOC, MVG, and COM—are reported in Table 3.3, due to the high number of classes we reported here only the ones with the highest MVG, the full table is available in Appendix A.

The tool issues warnings in the project summary section regarding the Lines of Code (LOC) for several classes: VsExtractListRowsOp, VSGrid2PointDistr, VSInterpolateOp, and VSPointDistributeOp. Additionally, it highlights some concerns about the MVG for the classes VSGrid2PointDistr and VSPointDistributeOp.

Examining the specific metrics for the VSPointDistributeOp class, as shown in Table 3.4, we can see that multiple functions are marked with CCCC warnings. For instance, the allocateArray(int) function is notably large, with 192 lines of code and an MVG of 50. The execute function is even more problematic because it contains 1,152 lines of code and has a high MVG of 386, indicating a critical need for refactoring to improve maintainability and readability.

Class Name	**LOC**	**MVG**	**COM**
VSPointDistributeOp	901	267	126
VSGrid2PointDistr	681	214	71
VSExtractListRowsOp	643	149	33
VSInterpolateOp	593	129	41
VSStatisticOp	294	121	40
VSSigmaContoursOp	375	112	53
VSCoarseVolumeOp	463	104	55
VSPointPropertyOp	447	102	55
VSClusterCellOp	475	102	55
VSSelectFieldOp	469	102	47

TABLE 3.3: General Procedural Metrics — Filter Classes (top by MVG).

Function Name	**LOC**	**MVG**	**COM**
VSPointDistributeOp	54	0	4
allocateArray(int)	192	50	6
allocateArray(int,bool)	2	0	0
computeModelBounds	94	30	20
execute	1152	386	160
executeArray	1	0	0
getOrigin	20	6	4
getSpacing	20	6	4
printHelp	52	2	4
setArray	1	0	0
setOrigin	40	14	2
setSpacing	94	34	2
~VSPointDistributeOp	24	6	4

TABLE 3.4: Procedural Metrics — VSPointDistributeOp Class

The object-oriented metrics, available in Appendix A, Table A.9, presents a situation similar to that of the Importer module. A single class, `VSTableOp` is the common base (DIT=1) and exposes a shared interface; it has NOC=38 and CBO=44 due to being referenced widely across the module. Child filters remain at low DIT (2 at most) and low coupling (CBO 1–4).

Viewer analysis

In the Viewer module, as shown by the data in Table 3.5 there are some larger classes, such as Pipe, PointsPipe and VolumePipe.

The procedural metrics for the Pipe class, as reported in Table 3.6, reveal a couple of functions with high LOC and MVG. The first, saveImageAsPng, is 181 lines long with an MVG of 46. The second, setCamera(SplotchCamera*), has 287 lines of code and an MVG of 92.

Similarly, the metrics for the VolumePipe class, shown in Table 3.7, highlight two large functions with high MVG. The first, setLookupTable, is 127 lines long with an MVG of 33, while the second, createPipe, is 515 lines long with a cyclomatic complexity of 95.

Class Name	**LOC**	**MVG**	**COM**
Pipe	816	177	256
IsosurfacePipe	200	36	87
PointsPipe	507	140	160
PointsSmoothPipe	298	55	75
SlicerPipe	258	42	121
SplotchPipe	31	3	27
SplotchPipeCamera	398	69	88
VolumePipe	862	129	321
VolumeSlicer	668	137	85
VtkImagePipe	328	70	78

TABLE 3.5: General Procedural Metrics — Viewer Classes

Function Name	**LOC**	**MVG**	**COM**
colorBar	45	3	6
constructVTK	21	0	51
createPipe	15	3	6
destroyVTK	30	9	6
getCamera	14	3	6
readData	14	3	6
saveImageAsPng	181	46	26
setAxes	74	3	11
setBoundingBox	96	15	19
setCamera()	1	0	0
setCamera(...)	2	0	0
setCamera(SplotchCamera*)	287	92	119

TABLE 3.6: Procedural Metrics — Pipe Class

Function Name	**LOC**	**MVG**	**COM**
VolumePipe	62	0	89
colorBar	15	1	2

Function Name	**LOC**	**MVG**	**COM**
createPipe	515	95	186
destroyAll	53	0	11
setLookupTable	127	33	18
~VolumePipe	57	0	11

TABLE 3.7: Procedural Metrics — VolumePipe Class

For the object-oriented metrics, also available in Appendix A, Table A.9, `Pipe` is the base class (NOC=9, CBO=15), the other classes have DIT=1 with moderate coupling (CBO up to 12).

Analysis of the obtained results

Overall, the CCCC analysis shows clear patterns across the three main modules.

The *Importer* module presents a few large and complex classes—such as `GadgetSource` and `XmlSource`, whose high LOC and MVG values suggest that refactoring specific functions, such as `readData`, could significantly improve readability and maintainability. In the *Filters* module, several operations exhibit high MVG, particularly within `VSPointDistributeOp` and `VSGrid2PointDistr`, indicating that these functions implement extensive procedural logic that may benefit from decomposition into smaller subroutines. Finally, in the *Viewer* module, the analysis confirms the same problems are also present in the visualization pipeline classes, such as `Pipe` and `VolumePipe`.

These findings provide a quantitative foundation for prioritizing future refactoring efforts. Functions with particularly high MVG values, such as `execute()` in `VSPointDistributeOp` and `createPipe()` in `PointsPipe`, represent critical areas where modularization could reduce technical debt and facilitate future maintenance.

3.2.4 CppDepend analysis

CppDepend analysis provides some insights about some possible problems of some components of code by checking a set of rules. It was also useful to generate the call graphs for each module.

CppDepend Rules

CppDepend considers a set of rules and checks statically if the code follows them. Some of these are classic code smells, such as too many methods in a single class, too many fields, high cyclomatic complexity, too many parameters, or too many local variables, then we have some other rules related to object-oriented design such as avoiding too deep inheritance tree, avoiding to assign static fields from instance methods, or avoiding abstract classes with too many methods. It also has some useful rules that allow us to easily find dead methods or dead fields and checking possible memory-related problems.

The main issues found by CppDepend were related to these rules: Avoid too big types, avoid Methods too big and complex, and avoid methods with too many parameters.

Some of the main classes with these problems were as follows: OptionsSetter, startFilter, and VSTable. OptionsSetter and startFilter have in common the fact that they are the classes called by the main functions for Viewer and Filter modules, respectively, they have a lot of big methods related to handling the various options given as input by the users and handling the data retrieval from the VBTs. VSTable is the class that handles most of the operations related to reading and writing the VBTs.

3.2.5 Refactoring

Based on the results obtained with CCCC's static code analysis (refer to section 3.2.3 for details), which identified multiple functions with high MVG, exceeding the recommended thresholds, we selected three representative classes from the VisIVO Server codebase for refactoring, one for each module:

- GadgetSource: This is an importer class and it is responsible for the conversion of GADGET snapshots into VBTs.

- VsPointDistributeOp: This is a filter class and it is capable of generating a volume VBT using a VBT representing point dataset containing the positions of the particles.
- PointsPipe: This is a viewer class and it is responsible of performing point rendering.

In the following figures we show a simplified version of the UML diagrams for these three classes:

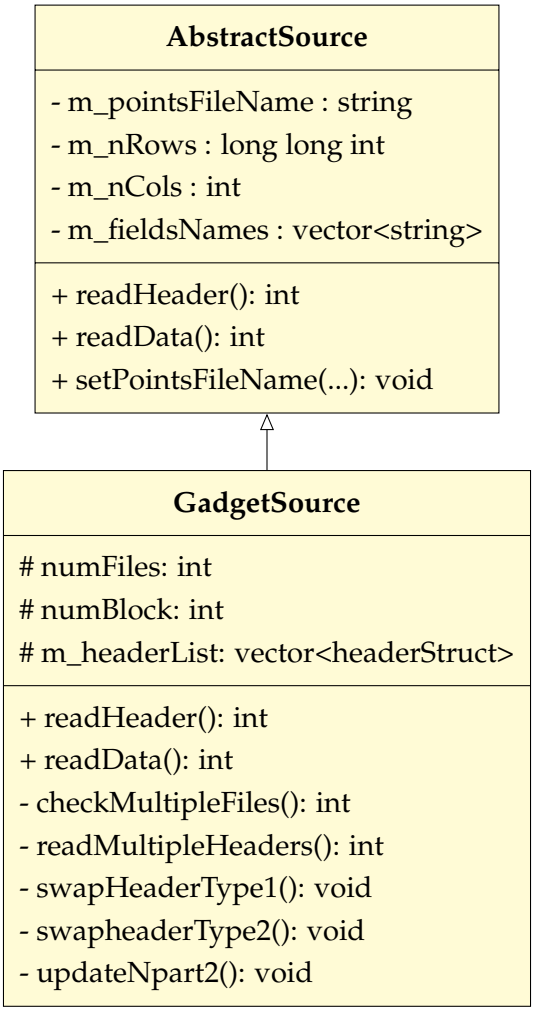

FIGURE 3.2: GadgetSource UML Diagram.

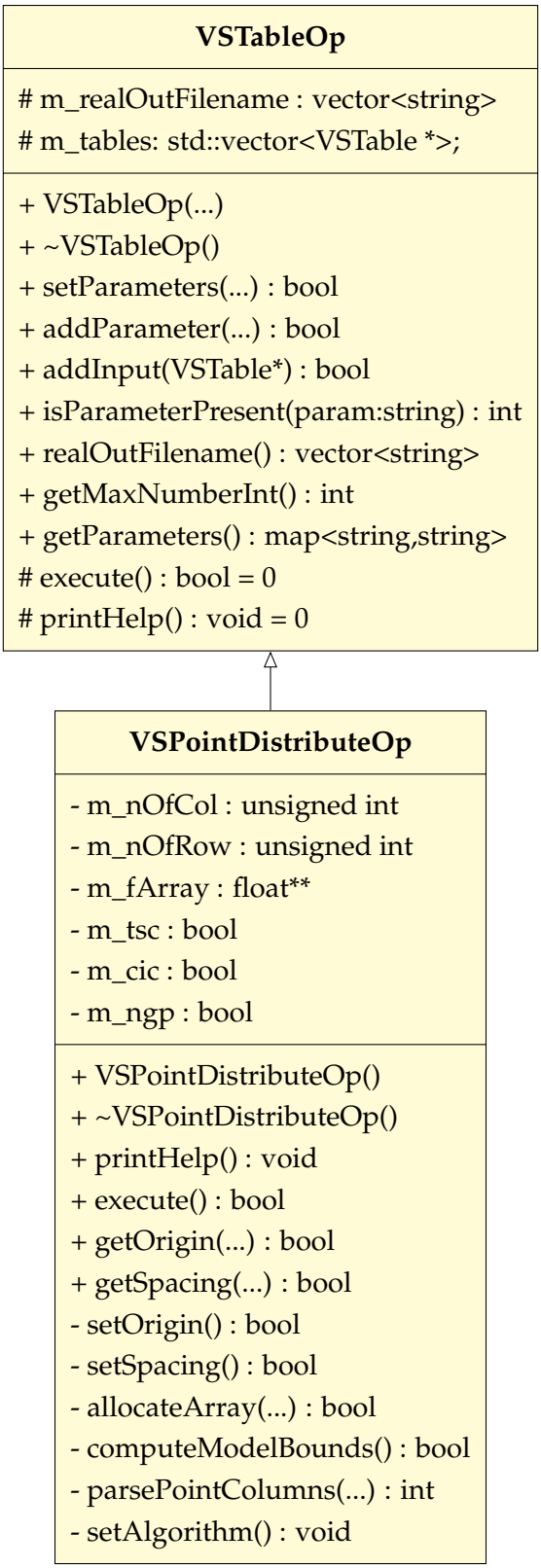

FIGURE 3.3: VSPointDistributeOp UML Diagram.

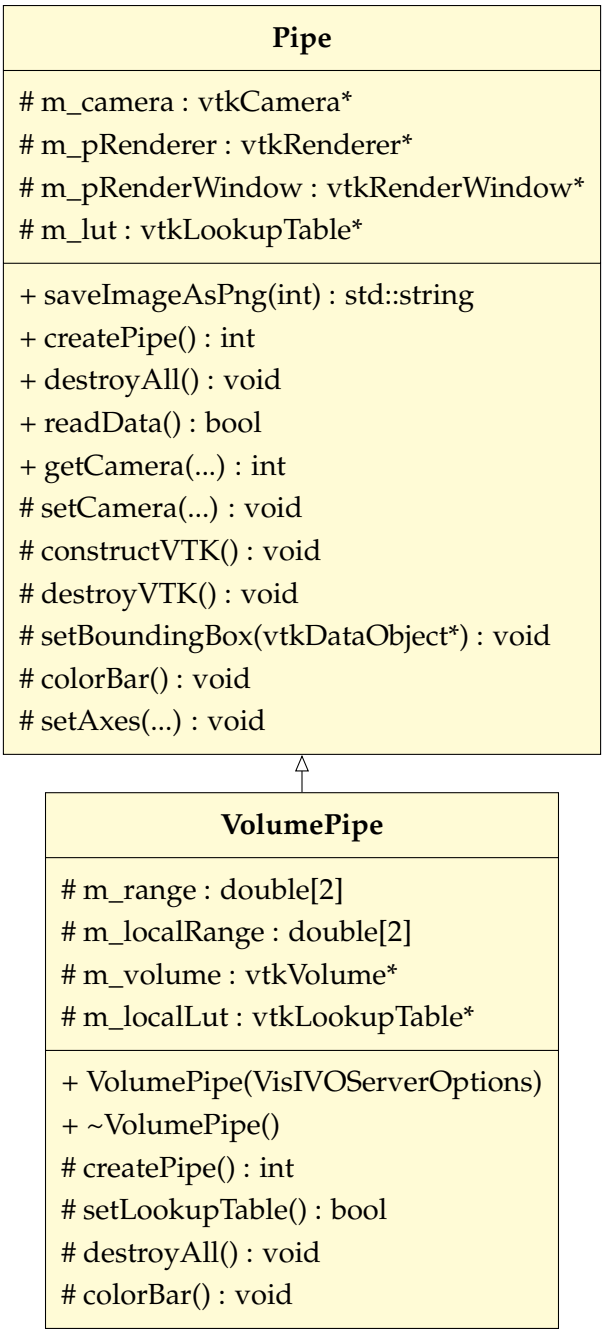

FIGURE 3.4: VolumePipe UML Diagram.

These simplified UML diagrams, hiding some parameters and functions for the sake of readability, show that each of the modules follow a clear inheritance relationship, with a base class providing a generic interface and a derived class implementing specific logic. For example, `AbstractSource` defines a generic

data importer interface, while `GadgetSource` extends it with specific routines for the GADGET format. In particular, the `readHeader` and `readData` methods are specialized to parse the binary snapshot structure and extract particle information according to the GADGET data structure. Similarly, `VSTableOp` acts as a basis for data transformation operations, and `VSPointDistributeOp` specializes in spatial distribution tasks. In this case, it provides a specialized implementation of the `execute` method, which implements an algorithm to generate a volume from particle positions. Finally, in the Viewer module, `Pipe` encapsulates rendering functionality, and `VolumePipe` extends it for 3D volumetric visualization. In this case, the specialization focuses on the construction of the VTK pipeline, where the volumetric data is mapped to generate a rendering representation using volume rendering techniques.

Refactoring is useful to improve the internal structure, readability, maintainability, and extensibility of software systems while preserving their functionality. In particular the cyclomatic complexity of the functions in these classes was very high, generally it is suggested that testing becomes increasingly difficult when its value exceeds 10[79] and in the literature it is usually suggested to use a maximum value of 15 as a threshold[80].

In this subsection we will present the common refactoring techniques[41] that we applied to these three classes and an example on how they were applied.

Extract Method

Extract Method is the most common refactoring operation that has been used to improve the code of VisIVO's classes. It consists of grouping a code fragment together in a new method. The main benefits are that it reduces the size of the original method, it fosters reusability, and it decreases the code duplication.

An example in the GadgetSource code was the extraction of the logic for buffer allocation into a method called allocateBuffers(). This method encapsulates the process of initializing a std::vector of float pointers with dynamically allocated arrays of appropriate size. This method improves both reusability and readability.

```
void GadgetSource::allocateBuffers(int blockSize,
                                   unsigned long long chunk,
                                   std::vector<float*>& buffers)
{
    buffers.resize(blockSize, nullptr);
    for (int dim = 0; dim < blockSize; ++dim) {
        buffers[dim] = new float[chunk];
    }
}
```

Grouping Related Data into Structs

Due to having multiple variables related to each other, parameter lists in the new extracted methods started becoming too big, to reduce it and improve clarity and readability we grouped related fields into dedicated structs. This change also improves maintainability, reducing eventual duplication and it makes function signatures simpler.

An example is the BlockData structure that just groups several fields that are needed to the methods to refer to the right positions, types, and sizes of the blocks.

```
struct BlockData
{
    std::vector<std::string> listOfBlocks;
    std::unordered_map<std::string, int> mapBlockNamesToFields;
    std::unordered_map<std::string, int> mapBlockSize;
    std::vector<std::vector<int>> typePosition;
    std::vector<std::vector<long long>> fileStartPosition;
};
```

Replace Magic Numbers with Named Constants

Another simple refactoring method is Replace Magic Numbers with Named Constants. It is used to improve readability and clarity of the code. In our case an example was in the GadgetSource class, considering that the number of type

of particles in GADGET snapshot is 6, it was recurrent in the code, so we just defined the constant NUM_TYPES.

Replace Temp with Query

This refactoring method just turns a temporary variable, which was the result of a simple expression, into a simple method that will be called every time it is necessary. An example of this refactoring method was executed on the PointsPipe class on a boolean expression.

```
bool PointsPipe::shouldLoadRadiusScalars()
{
  return m_visOpt.radiusscalar != "none" &&
         m_visOpt.scaleGlyphs != "none" &&
         m_visOpt.nGlyphs != 0;
}
```

Introduce Explaining Variable

Introduce Explaining Variable is useful when having a complicated expression and it is necessary to make some operations more readable. It consists of extracting part of an expression into temporary variables with meaningful names to make the code easier to understand.

It is a widely used refactoring method, and it was applied to this snippet of code as an example:

```
blockData.fileStartPosition[type][file] =
    blockData.fileStartPosition[type][file - 1] +
    static_cast<unsigned long long>(m_pHeaderType2[file - 1].npart[type]);
```

It was then updated into a more readable way like this:

```
auto previousOffset = blockData.fileStartPosition[type][file - 1];
auto particlesInPreviousFile = static_cast<unsigned long long>
                               (m_pHeaderType2[file - 1].npart[type]);

blockData.fileStartPosition[type][file] = previousOffset +
                                          particlesInPreviousFile;
```

Remove Dead Code

Another simple refactoring method applied is Remove Dead Code, the name itself is self-explanatory. In our case the removal consisted mainly on removing dead comments, which served a temporary purpose during the migration from an older VTK version to newer one, and the removal of redundant or unused variables.

3.3 Results

The refactoring of the code gave measurable improvements across all the target classes, demonstrating its effectiveness in improving the code quality. These results provide concrete evidence of enhanced maintainability for the codebase.

We will show how different the metrics obtained using CCCC, as they indicate how the maintainability and the readability of the code has improved.

3.3.1 Refactoring results

GadgetSource Metrics

The table 3.8 compares the LOC and MVG of the GadgetSource class from the old implementation to the new one.

	LOC		MVG	
Function	**old**	**new**	**old**	**new**
checkMultipleFiles	14	14	4	4
readData	361	31	78	5
readHeader	83	82	23	23
readMultipleHeaders	16	17	3	3
swapHeaderType1	29	28	4	4
swapHeaderType2	37	36	4	4
allocateBuffers	N/A	6	N/A	1
computeChunkSize	N/A	16	N/A	3
computeFileStartPositions	N/A	11	N/A	2

	LOC		MVG	
Function	**old**	**new**	**old**	**new**
computeTypePositions	N/A	23	N/A	7
determineEndianism	N/A	7	N/A	6
extractBlockList	N/A	30	N/A	8
findBlockOffset	N/A	17	N/A	5
generateMap	N/A	7	N/A	1
initializeParticleCounts	N/A	6	N/A	1
isValidParticleType	N/A	9	N/A	8
openInputFiles	N/A	15	N/A	6
openOutputFiles	N/A	17	N/A	3
processBlocksParallel	N/A	26	N/A	3
processChunk	N/A	12	N/A	3
processFileName	N/A	11	N/A	6
processParticle	N/A	31	N/A	5
writeChunkData	N/A	11	N/A	1

TABLE 3.8: GadgetSource: LOC and MVG by function (old vs. new). N/A indicates functions present only in one version.

The refactoring results obtained show that the refactoring mainly replaced one high complexity function with multiple focused helper functions. We can see that the MVG of the readData function was greatly reduced from 78 to 5. All newly extracted helpers have low complexity (MVG ≤ 8). Across all functions shown, the overall line of code decreased 540 to 463 ($-14.3\,\%$).

The complexity results show lower risk per function with easier testing and better maintainability.

VSPointDistributeOp Metrics

Table 3.9 compares LOC and MVG between the old and new implementations.

	LOC		MVG	
Function	**old**	**new**	**old**	**new**
VSPointDistributeOp()	54	27	0	0
allocateArray(int)	192	96	50	25
allocateArray(int,bool)	2	1	0	0
computeModelBounds()	94	94	30	30
execute()	1152	103	386	38
executeArray()	1	1	0	0
getOrigin(float *)	20	10	6	3
getSpacing(float *)	20	10	6	3
printHelp()	52	26	2	1
setArray()	1	1	0	0
setOrigin()	40	20	14	7
setSpacing()	94	47	34	17
~VSPointDistributeOp()	24	24	6	6
allocateColumnList(int*,vector)	N/A	11	N/A	3
applyPeriodicBoundary_CIC(...)	N/A	14	N/A	4
applyPeriodicBoundary_TSC(...)	N/A	8	N/A	5
computeCICWeights(...)	N/A	17	N/A	0
computeNGPIndex(...)	N/A	16	N/A	5
computeTSCWeight(...)	N/A	10	N/A	7
configureTableGrid(VSTable,vector)	N/A	39	N/A	5
generateOutputFileName()	N/A	22	N/A	4
getLinearizedIndex(int,int,int)	N/A	11	N/A	9
initializeEmptyGrid(VSTable,vector,int)	N/A	25	N/A	6
initializeGrid(vector,int)	N/A	15	N/A	7
parseFieldList(vector)	N/A	25	N/A	10
parsePointColumns(...)	N/A	15	N/A	5
processCIC(VSTable,int*,int,vector,long)	N/A	61	N/A	12
processGridSpacing()	N/A	17	N/A	6
processNGP(VSTable,int*,int,vector,long)	N/A	41	N/A	14
processTSC(VSTable,int*,int,vector,long)	N/A	52	N/A	15
setAlgorithm()	N/A	21	N/A	4
setGridResolution()	N/A	19	N/A	8

	LOC		MVG	
Function	**old**	**new**	**old**	**new**

TABLE 3.9: VSPointDistributeOp: LOC and MVG by function (old vs. new). N/A indicates functions present only in one version.

The refactoring focused mainly on the complex execute function with many helper functions. The MVG is heavily reduced from 386 to 38, which is still very high but significantly improved from the original implementation. Also some other function has been refactored, such as allocateArray(int) and setOrigin, both of them in the newer implementation have halved the MVG. All the new introduced helper functions have complexity below the 15 suggested threshold.

Across the class, LOC has been reduced from 1746 to 899.

This refactoring also extracted the different available algorithms to perform the filter calculations each to a dedicated function, this can be very useful for modularity, and it will make easier to introduce new algorithms just by implementing a new function for the new strategy.

PointsPipe Metrics

Table 3.10 compares LOC and MVG between the old and new implementations.

	LOC		MVG	
Function	**old**	**new**	**old**	**new**
PointsPipe	11	11	0	0
SetXYZ	53	53	15	15
createPipe	277	52	88	23
destroyAll	14	14	5	5
setGlyphs	46	46	7	7
setLookupTable	24	24	5	5
setRadius	22	22	4	4
setResolution	13	13	3	3
setScaling	15	15	8	8
~PointsPipe	15	15	5	5

	LOC		MVG	
Function	**old**	**new**	**old**	**new**
applySingleColor	N/A	14	N/A	7
assignColumnToArray	N/A	35	N/A	15
configureActor	N/A	11	N/A	1
configureGlyphs	N/A	11	N/A	7
configureStereoRender	N/A	26	N/A	3
createAxisArray	N/A	7	N/A	1
finalizeRender	N/A	15	N/A	1
getColorTableIndex	N/A	10	N/A	2
getColumnIndex	N/A	8	N/A	2
loadColorScalars	N/A	13	N/A	4
loadHeightScalars	N/A	9	N/A	3
loadRadiusScalars	N/A	11	N/A	2
setBackgroundColor	N/A	46	N/A	7
setRenderWindow	N/A	8	N/A	2
setupVertexCells	N/A	11	N/A	1
shouldLoadHeightScalars	N/A	7	N/A	4
shouldLoadRadiusScalars	N/A	6	N/A	3

TABLE 3.10: PointsPipe: LOC and MVG by function (old vs. new). N/A indicates functions present only in one version.

Here the refactoring is focused on one main function with many smaller helper functions, similarly to the other previous examples. The function createPipe MVG is reduced from 88 to 23. And also in this case all the new helper functions have MVG lower or equal to 15.

In this case the total number of lines increased from 490 to 512.

3.3.2 Memory Analysis with Valgrind

To complement the refactoring activity and ensure that the modifications introduced did not introduce memory-related issues, we performed a systematic memory analysis using the Valgrind suite. In this work we primarily used `memcheck`, its most widely adopted component for detecting memory errors.

The analysis was conducted on the core modules involved in the import and visualization workflow, including the classes `AbstractSource`, `GadgetSource`. The objective was to verify the correctness of the memory usage after the refactoring.

Valgrind was executed by running the importer on a 4GB dataset and enabling full leak checking through the option `-leak-check=full`. Additional diagnostic flags, such as `-show-leak-kinds=all` and `-track-origins=yes`, were used to identify uninitialized reads and incorrect memory accesses with higher precision.

The results confirmed that the refactoring operations did not introduce memory leaks or invalid memory accesses.

In addition to `memcheck`, we also employed Valgrind's heap profiler `massif` to diagnose anomalous memory usage observed in `VisIVOViewer`. While `memcheck` primarily targets correctness, `massif` provides a detailed temporal view of heap consumption, enabling us to identify which code paths were responsible for unexpected memory growth. By analyzing the `massif` output with a visualizer, we found out that after generating a visualization, if the PointsPipe class was used again for a subsequent visualization the allocated VTK structures used during the first visualization were not released, resulting in a persistent rise in memory usage during multiple visualizations. This analysis was instrumental in identifying and resolving a previously unnoticed memory leak, ultimately improving the memory usage of the viewer during large-scale data visualization.

3.4 Conclusion

This chapter presented a comprehensive static analysis of the VisIVO Server code and demonstrated how refactoring techniques can improve code quality and maintainability. Using multiple code analysis tools we identifies metrics that allowed us to understand what were the requirements for improvement and implemented multiple refactoring strategies.

The applied refactoring techniques allowed us to reduce the cyclomatic complexity for the functions in the chosen classes, achieving improvements

that directly impact software maintainability. The most significant results include a 90% complexity reduction in VSPointDistributeOp's execute function (from MVG 386 to 38) and a 94% reduction in GadgetSource's readData function (from MVG 78 to 5). These improvements, combined with also code size reduction in two of the three refactored class, demonstrate that systematic refactoring can substantially enhance code quality in scientific software systems.

The multi-tool analysis approach using CCCC, CppDepend, and Doxygen proved effective in providing comprehensive coverage of different code quality aspects. The systematic application of established refactoring techniques successfully decomposed monolithic functions into multiple easier to test components.

While we achieved significant improvements, some refactored functions still maintain relatively high complexity, indicating that additional refactoring iterations may be beneficial.

This work provides a basis for continuous code quality improvement across the entire VisIVO Server codebase, and it will be very useful for also improving the other components with high complexity. It has also been of fundamental importance as a basis for some of the updates discussed in the following chapters, such as the Importer parallelization in chapter 4 and the implemented in-situ approach in chapter 6.

Extending this refactoring process to the entire VisIVO Server codebase would represent a substantial engineering effort. The analysis and restructuring of just three classes required on average three weeks of iterative testing for each class. However, each class in VisIVO Server presents a different code structure and level of complexity, meaning that the time and effort required to refactor them may vary significantly. Scaling this work to all the classes across all modules would therefore demand careful prioritization, focusing first on those with the highest cyclomatic complexity or central functional roles. The experience gained during this study provides a methodology that can be systematically applied to the remaining components with manageable effort.

Chapter 4

VisIVO Server Importer Parallelization

4.1 Introduction

With the advent of next-generation astrophysical instruments, such as the Square Kilometer Array, and the availability of increasingly accurate cosmological simulations, like those produced by the OpenGADGET software, the volume of data describing astronomical phenomena is growing at an impressive rate. This represents a great challenge for storing, processing, and analyzing these massive data. To address this challenge, it is necessary to develop software models optimized to exploit the capabilities of modern hardware architectures, including multicore processors, and high-performance distributed storage systems.

These new software models would be fundamental for the evolution of existing visualization and big data analysis, enabling them to fully exploit next-generation HPC systems. Such tools are crucial for the study of sources or the discovery of astrophysical patterns (such as star-forming regions or extended sources like supernova remnants) within astronomical maps. This involves combining information from different wavelengths (from infrared to radio) and cosmological simulations.

In this chapter, we focus on evolving VisIVO's importer module, responsible for converting user-supplied datasets into VisIVO Binary Tables (VBT), to

allow it to take advantage of new HPC facilities. The importer module primarily involves I/O operations, and the parallelization of this kind of application is a non-trivial task.

4.2 Methodology

VisIVO Importer is capable of handling multiple astronomical and cosmological data formats, covering both observational and simulation datasets. Among the supported formats are FITS, commonly used for astronomical images; tools such as Fly[5] and GADGET have their own formats describing particle-based cosmological simulations; and similarly RAMSES[113] is a tool having a grid-based hydrodynamical simulation format. In this work, we mainly focused on the parallelization of the VisIVO Importer module from the GADGET native format to VBT. GADGET generates snapshots, which are dumps representing the state of the system at certain times. The output of the simulation can be distributed in parallel across multiple files, each written by a group of processors.

In the following subsections, we will describe the details of both GADGET native data structure and the VBTs data structure, in order to clearly define the requirements and the modifications needed for the updated importer implementation.

4.2.1 GADGET Data structure

In GADGET simulation snapshots, data are organized in a sequence of binary blocks. Each block stores a specific physical quantity for all particles in the simulation, for example, blocks representing the positions,the velocities, IDs or masses. Blocks are identified by a four-character tag (such as `"HEAD"`, `"POS "`, `"VEL "`, `"ID "`, `"MASS"`, etc.) that describes the type of block. Each block is surrounded by 4-byte integers that specify the size of the block in bytes, allowing the reader to determine where each block begins and ends. This structure provides flexibility and makes it possible to skip or directly access specific blocks. The first block is a special one, the **Header Block**, which contains global metadata about the snapshot. It must always be read before accessing the other blocks. The header stores, the number of particles of each type (`npart`), their

total number across all files (`npartTotal`), cosmological parameters (such as the current simulation time `time` and redshift `redshift`), the mass of each particle type (`mass`), and various flags describing simulation options. Figure 4.1 illustrates the logical structure of the header.

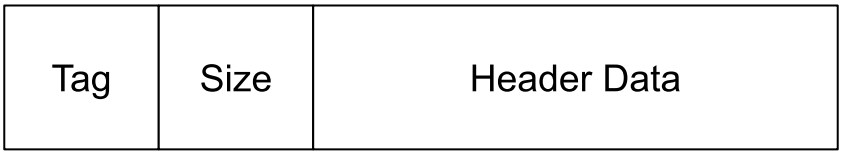

FIGURE 4.1: GADGET Header block data structure.

All the information in the header will be stored in an internal data structure, in particular the number of each type of the particles, which are crucial for the correct handling of the data.

Following the header, the snapshot file contains a sequence of data blocks, typically organized as follows:

`HEAD → POS → VEL → ID → MASS → U → RHO → HSML → ...`

Each of these blocks has the same internal layout: particle data are ordered by particle type, where type 0 corresponds to gas, type 1 to dark matter halos, type 2 to disk particles, and so on. For instance, in the `POS` block, all positions for type 0 particles appear first, followed by those for type 1, etc. This ordering is consistent across all blocks.

As shown in Figure 4.2, every data block begins with a 4-byte integer indicating its total byte size, followed by a 4-character tag, the block data themselves, and a final 4-byte integer repeating the block size for consistency checking. This binary structure allows both sequential reading and random access to individual blocks, which is particularly useful for parallel I/O routines.

FIGURE 4.2: Typical GADGET block data structure.

Multiple file GADGET Snapshots

GADGET allows each snapshot to be written into a single file or into multiple files (e.g., `snap_100.0`, `snap_100.1`,`snap_100.2`, ...). All these files together represent a part of a big snapshot. When the snapshot is split, each file contains: its own copy of the `HEAD` block, only a fraction of the total particles, the same sequence of blocks (`POS`, `VEL`, `ID`, etc.).

The division is purely sequential: file 0 stores the first portion of the particles, file 1 the next portion, and so on.

Each file repeats the same sequence of blocks, but contains a different particle set. During the import phase, the importer reads the header of all files to compute the offset at which each file's data must be placed in the VBT and the starting position from which each thread must read and write.

These offsets allow the parallel reader to map each block of each file into the correct region of the final VBT, reconstructing a single coherent table.

4.2.2 VBT Data structure

A VisIVO Binary Table (VBT) is a efficient data representation used by VisIVO Server internally. It consists of an Ascii header file, using the .bin.head extension, which includes all the required metadata, and a raw data file with the .bin extension that contains the actual data values. This binary organization ensures highly efficient data representation because it minimizes parsing overhead and memory usage, allowing VisIVO Server to read, write, and process large datasets much faster than text-based or self-describing formats such as FITS or CSV. The parsing overhead is minimized because it is easy to retrieve data just knowing the number of rows and the number of the desired field. An example of the improvement of improving memory usage is by choosing to load in memory only the necessary fields when using the Viewer module.

The listing 1 shows an example of a header of a VBT representing particles. Line 1 describes the internal data representation, in this case float; line 2 represents the total number of available field in the VBT, in this case 7; line 3 is useful because it represents both the type of VBT and the number of elements

```
1  float
2  7
3  140005
4  little
5  POS_X
6  POS_Y
7  POS_Z
8  VEL_X
9  VEL_Y
10 VEL_Z
11 ID
```

LISTING 1: Header file examples for a VBT representing particles

for each field, in this case 140006; line 4 the data format identifier(little or big endian) and then from line 5 it starts to list of field names.

```
1  float
2  1
3  56623104 384 384 384 1 1 1
4  little
5  Constant
```

LISTING 2: Header file examples for a VBT representing a volume

Listing 2 shows an example of header for a VBT representing volumes. It is similar to the previous example for the particle dataset, the main difference is in line 3, as we can see it doesn't just show the number of elements for each fields but it also includes information describing the underlying mesh structure, consisting of the grid dimensions of the volume, in our example "384 384 384" and the size of each volumetric cell ("1 1 1" in the example).

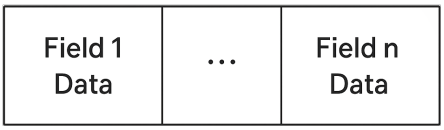

FIGURE 4.3: Visual representation of the VBT binary data.

The raw data file contains the data values for each field consecutively. For example, if each data value is represented as a 4-byte float and there are 100 elements for each field, the first 400 bytes of the file will represent the data for the first field. In figure 4.3 we show a visual representation of a .bin VBT file.

In the context of the GADGET importer, each valid block found is written directly as a VBT field in the .bin.head file. For example, the GADGET block `POS` contains 3 values per particle (x, y, z), which are written sequentially into the VBT fields `POS_X`, `POS_Y` and `POS_Z`. The importer uses the size of the fields specified in the header to compute the exact byte offset at which each array must be written in the `.bin` file. This establishes a direct correspondence between GADGET blocks and VBT columns, enabling the parallel reader to place each block's data at the correct position in the final table.

4.2.3 The Parallel Reader

The main objective of this application is to convert snapshots generated by the GADGET simulation code into VisIVO Binary Tables, exploiting multithreading and multinode approaches and the fact that modern HPC facilities support distributed parallel system that grants faster bandwidth, which cannot usually be saturated by using a single thread. This task primarily involves I/O operations, including multiple reads from multiple files and multiple writes to a single file.

The chosen approach uses individual read/writes rather than using collectives for two main reasons: i) the reading process cannot be parallelized with a collective approach because the data are distributed across multiple files. ii) given the snapshot sizes, each I/O operation made by each thread would be large enough to not require the combination of multiple requests. Therefore, the overhead of collective calls can outweigh their benefits in some specific I/O patterns, as stated in other studies[48].

Thus, we adopted a file-per-process strategy, an approach commonly used in HPC application when data are distributed over multiple files[14], which can provide high throughput, and followed best practices for efficient I/O, such as performing I/O in few large chunks and avoiding unnecessary I/O operations.

The class that needs the update is called GadgetSource. It handles all the operations to read GADGET snapshots and convert them into a VBT.

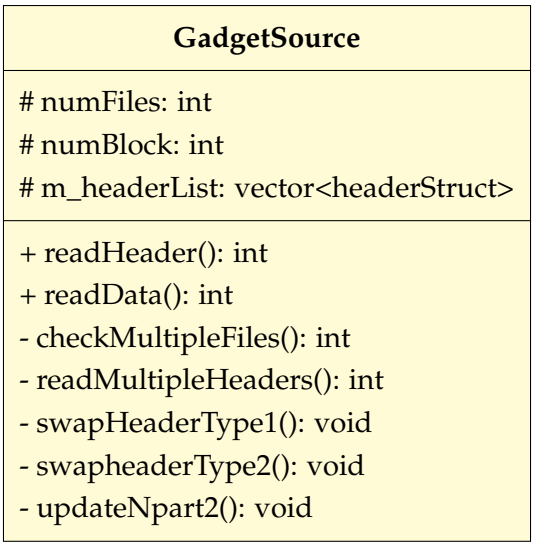

FIGURE 4.4: GadgetSource UML Diagram.

Figure 4.4 show a simplified GadgetSource UML diagram, it includes the most important methods: readHeader() for metadata extraction, using readMultipleHeader() in case the snapshots are distributed over multiple files, and readData() that includes the logic for the effective parallel I/O operations. The private members store essential metadata including particle counts and multiple informations such as number of files and blocks for thread coordination.

The following subsections provide all the details on handling the header and the data.

Reading the header

As a first step, each process needs to read data from the GADGET snapshot header, which is a special block placed at the beginning of the file. This is done by calling the readHeader method. The header contains multiple pieces of information necessary to read the rest of the file correctly, such as the number of elements for each type of particle in each file, the cumulative number for all files, and the number of files into which the snapshot has been divided. Having the number of files, it is possible to call the checkMultipleFiles method, which verifies if all files are available in the same folder as the input file. After this check, it is possible to read all the headers to obtain the information of each file by calling the readMultipleHeaders method.

Reading the data from a snapshot and writing the VBT

After reading the header information, it will be possible to use the readData method. Before starting the process of reading the binary data, each process has to create a list of blocks to read. This operation is necessary because each GADGET block might arrange data into tuples, and during the reading process, these tuples must be correctly arranged before writing them to the VBT file.

For example, the position block is represented by tuples of three elements. A POS block stores 3 values per particle (x, y, z), a VEL block also stores 3 (vx, vy, vz), while an ID block stores a single value. This tuple size information is essential for correctly reconstructing data before writing the VBT. Other block types follow the same logic, with tuple sizes and per-type presence determined from the GADGET format documentation.

Generating the list of blocks to read is facilitated by the fact that, before the beginning of each block, the GADGET snapshots place a tag made of four characters to identify the block name. Next to it, there is an integer indicating how many bytes to skip to reach the next block tag.

However, the snapshot files themselves do not contain metadata describing the internal organization of each block, for example, whether the data are arranged as tuples or whether a particular block is present for a given particle type. Such a description is instead defined in the GADGET documentation. Therefore, we will use simple unordered maps generated using information from the GADGET documentation.

This is a multiprocess implementation using the Message Passing Interface (MPI) with a file-per-process approach, where each process reads different files to avoid file contention between processes. This is considered a common best practice in parallel I/O programming [105]. In this implementation, inter-process communication occurs only when the spawned processes reach a synchronization barrier, because all the needed data needed for processing are independently computed or retrieved from the header. It is advisable to avoid inter-process communication if it is not necessary because it can be very slow. Therefore, the only form of communication needed is a barrier, where

```
chunk = minPart[type]; //minPart[type]: minimum number of particle in file
unsigned long long n = // Number of iterations required to read all elements
    m_pHeaderType2[nFile].npart[type] / chunk;

unsigned long long Resto = // Remainder of particles to read after the n iterations
    m_pHeaderType2[nFile].npart[type] - (chunk * n);
newOffset = firstOffset[type];//Retrieving the Offset from pre-computed array
// Chunk processing loop
for (unsigned long long k = 0; k < n; k++) {
    processChunk(fileData.inFile[nFile], bufferBlock,buffers,
                 blockSize, chunk, newOffset);
    //Update offset by adding the number of bytes read in processChunk
    newOffset += blockSize * chunk * sizeof(float);
}

// Final partial chunk (if present)
if (Resto > 0) {
    processChunk(fileData.inFile[nFile], bufferBlock, buffers, blockSize,
                 Resto, newOffset);
}
```

LISTING 3: Preprocessing required to distribute data over multiple nodes and threads

each process waits for the others to finish execution. Multiprocess implementation in parallel I/O is fundamental to obtain higher bandwidth with parallel file systems, such as BeeGFS [42] or Lustre.

The implementation is also multithreaded, with threads spawned using OpenMP. Based on the previously discussed information, each thread can determine the correct read and write positions within the corresponding files. There will be no data contention because each thread reads and writes data at different positions. To prevent conflicts with the file descriptor and file pointer, each thread will independently manage its own file position. The I/O operations used will be pread() and pwrite(), which are thread-safe functions, suitable for multithreaded applications as they allow I/O operations on the same file descriptor without being affected by changes to the file offset by other threads.

Before presenting the full code executed by each thread, the Listing 3 shows the core logic that determines how particle data are divided into chunks and processed iteratively. The following excerpt illustrates how the importer computes the chunk size (`chunk`), the number of iterations (`n`), and the possible remainder (`Resto`):

This simplified code shows how the importer divides each particle block

```
1  startPos = typePosition[nBlock][type] *
2  (unsigned long long)m_pHeaderType2[nFile].npartTotal[type] +
3  fileStartPosition[type][nFile];// Calculating the starting position for the particle type
4  for (int k = 0; k < n; k++)
5  {
6      bufferBlock = new float[blockSize*chunk];
7      pread(inputFileDescr[nFile], (char *)(bufferBlock),
8            chunk*sizeof(float), offset);
9      offset += chunk*sizeof(float);
10     pWrite=((startPos*sizeof(float)) +
11              (k*chunk*sizeof(float)));
12     pwrite(outputFileDescr[type], (char *)(bufferBlock),
13            chunk*sizeof(float), pWrite);
14 }
```

LISTING 4: Code executed by each thread in the parallel importer

into manageable chunks. The value of `chunk` is defined from minPart array: it is either the total number of particles of this type in the file or a predefined upper bound, ensuring that temporary buffers remain small enough while read/write operations remain large and efficient.

The variable `n` represents the number of full chunk iterations, while `Resto` accounts for any remaining particles that form a final partial iteration.

Each chunk corresponds to a unique region in the output VBT, determined by metadata precomputed from the headers. As a result, each thread operates on a disjoint set of offsets without requiring any data exchange or synchronization beyond OpenMP's scheduling. The use of `pread` and `pwrite` ensures thread-safe, offset-based I/O without shared file-pointer contention.

We also present a simplified version of the typical code executed by each thread in Listing 4, each thread has an exclusive couple of variables nBlock and nFile. In line 1 the thread just computes the starting position in which the thread must write in the output file. "typePosition" is an array that stores for each block the number of blocks preceding it; this is done for each particle type, and it is useful because it also considers multidimensional blocks as position or velocity. "fileStartPosition" is an array that stores a number for each file the number of particles handled by previous files; this is necessary to know the position from which to start writing. In line 2 "n" represents the number of iterations needed to handle each chunk of data; it usually will be just one because GADGET snapshots distribute data over multiple files, so in each file

and for each particle type there are not many elements, but there is the requirement to handle it because there could be some edge cases. Then we just have the parallel read and parallel write, respectively, at lines 6 and 9. Each thread will execute this code on each type of particles.

4.3 Results

4.3.1 Preliminary Tests

VisIVO Importer parallel implementation has been tested on Cineca's Galileo 100 platform[1]. Each node is equipped with two Intel CascadeLake 8260 processors, allowing each node to access up to 48 cores, 384GB RAM DDR4, for the storage it uses the Lustre Parallel file system.

Tests were conducted on multiple configurations, ranging from one to eight nodes and from one to eight cores per node. Eight cores was the maximum we employed because the test snapshots had just 7 fields, and since our approach distributes the fields over multiple cores, it would not scale if the number of fields is less than the number of cores.

For each configuration, we considered one MPI process per node. For this tests, we used a 512 GB dataset in GADGET format.

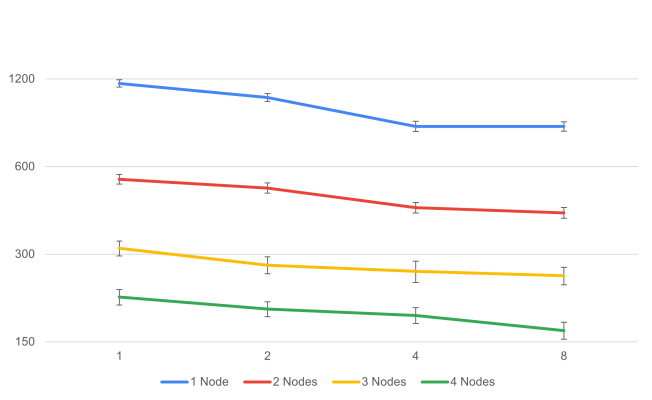

FIGURE 4.5: Execution Time on 512GB Data.

[1] https://www.hpc.cineca.it/systems/hardware/galileo100/

In figure 4.5 we show the execution times of the new GADGET parallel importer and we can see that multiple processes on multiple nodes can achieve a higher bandwidth, this behavior was expected on a parallel file system[42]. These tests were repeated 10 times each; we show the average computing time in a logarithmic scale, and the error bars represent the standard deviation.

Nodes	Execution Time (s)	Standard Deviation	Speedup	Efficiency
1	1157.94	32.66	1.00	1.00
2	542.84	20.42	1.95	0.98
4	314.29	18.22	3.56	0.89
8	213.75	13.2	5.47	0.68

TABLE 4.1: Parallel efficiency analysis for 512GB dataset

As shown in Table 4.1, using 8 nodes with 1 core each reduces execution time from 1157.14 s (1 node) to 211.65 s, corresponding to a speedup of 5.47x. The scaling is close to linear up to 4 nodes, with an efficiency of 0.89, then the efficiency decreases due to I/O contention.

Multithreading is limited by the single node bandwidth, so we can obtain a speedup by using up to 4 or 8 cores, which is a similar behavior obtained by other solutions[98]. The main reason for this slower speedup is because the importer uses multithreading over the available blocks, if the number of blocks is not enough it does not achieve significant speedup. In this case, the datasets contain only 4 blocks.

MPI-IO Comparison

As an additional preliminary test, we compared the execution times with those of another implementation of the importing module. We implemented a version of the parallel GADGET importer using MPI-IO directives combined with a producer-consumer model. This approach leverages multi-node parallel execution, where a single process acts as the producer, and the remaining processes act as consumers. The producer is responsible for distributing information about the blocks and files that need to be processed, while the consumers perform the I/O operations based on the received data.

In figure 4.6, we compare the execution times across multiple nodes for this producer-consumer approach with the previous MPI/OpenMP implementation, considering only single-threaded performance on a 512GB GADGET dataset.

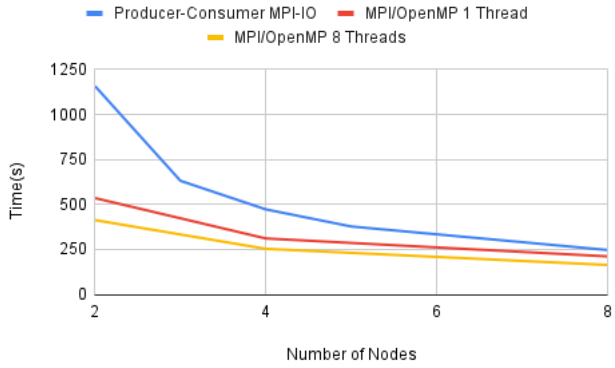

FIGURE 4.6: Comparison of Execution Time on 512GB Data.

The results indicate that the execution times for the producer-consumer approach are similar to those of the classic MPI/OpenMP approach, despite using one less process. For example, the average execution time when using 9 nodes with the producer-consumer approach is 223.475 seconds, compared to 211.654 seconds when using 8 nodes with the MPI/OpenMP approach. The slight increase in execution time for the producer-consumer model is likely due to the overhead associated with message passing and synchronization between the producer and consumers.

Additionally, while the MPI/OpenMP approach benefits from multithreading to further reduce execution time, multithreading is not feasible in the producer-consumer approach due to the limitations of MPI-IO directives, which are incompatible with multithreaded environments.

Although the MPI/OpenMP implementation can exploit multithreading to further reduce execution times, enabling thread-level parallelism is not feasible in the producer–consumer implementation. This is because MPI-IO is incompatible with multi-threaded environments, because the I/O calls are not thread safe. The approach proposed in this work replaces MPI-IO calls with POSIX system calls such as pread and pwrite, which natively support concurrent access and allow multiple threads to safely operate on shared-memory regions.

These results show that the producer-consumer approach is not the most suitable one because it uses more processes and, at the same time, it would require one more node to achieve similar results to the MPI/OpenMP approach.

4.4 Conclusion

In this chapter, we presented a parallelization approach developed for VisIVO Importer. This proposed approach combines multiprocess parallelism, using Message Passing Interface with a file-per-process strategy, and multithreading, using OpenMP, to exploit computational and I/O capabilities of modern HPC infrastructure.

This solution addresses some of the typical challenges for the parallelization of I/O-bound applications, such as file contention and limited bandwidth utilization, by ensuring that processes and threads operate independently on different portions of the dataset. We used thread-safe I/O functions, such as pread() and pwrite(), to ensure correctness in a multithreaded environment, avoiding that the changes of file offsets affect other threads.

We performed preliminary performance tests on the Galileo100 HPC system, and then demonstrated that the parallel implementation achieves significant speedup and higher bandwidth utilization compared to the sequential approach. We also compared the performance to an alternative parallel implementation based on MPI-IO and a producer-consumer model. Although this approach provided comparable performance, it introduced additional overhead due to inter-process communication and it was incompatible with multithreading.

Overall, the obtained results demonstrate that the MPI and OpenMP approach provides an efficient and scalable method to convert large-scale GADGET datasets into VBTs, enabling VisIVOImporter to better exploit HPC resources and handle large data volumes of astronomical simulation data.

Chapter 5

Workflow Management System integration

5.1 Introduction

One of the steps to allow VisIVO to exploit multiple types of infrastructure, thus improving portability and reproducibility, has been the integration with a Workflow Management System (WMS). WMSs are tools that are responsible for exposing the interactions between workflow steps to the users and orchestrating executions by managing dependencies, scheduling tasks, and ensuring that data produced in one step is correctly propagated to the next, favoring data locality when possible to avoid data transfer.

A workflow can be identified as the automation of a business process during which data or tasks are passed from one participant to another using a set of procedural rules. [121] In scientific computing, workflows formalize complex computational pipelines, abstracting both the involved data and the sequence of tool executions.

Workflows can also be defined as a directed bipartite graph, where the nodes represent either computational steps or the ports through which they communicate. The edges represent the dependency relations between steps, linking steps to ports and ports to subsequent steps.

The chosen WMS is StreamFlow, which extends a classic workflow system using a declarative description of many types of environments (such as cloud

infrastructures or HPC) and with the relations among workflow nodes and execution environments. The primary aim of StreamFlow is to enable the execution of complex workflows across heterogeneous computing platforms, such as Kubernetes clusters and traditional batch-based HPC systems, without relying on shared data spaces.

StreamFlow is based on the Common Workflow Language (CWL) standard. Defining all the details of a workflow requires creating multiple files: a YAML file to describe the workflow's inputs, and a CWL file in which all the steps of the workflow are defined, including necessary details such as input and output types. Workflow inputs can include different CWL data types, such as scalar values (e.g. integers) or complex types like Files or Directories, which reference external data or folders used during execution. Workflow outputs are typically files, representing the data produced by the workflow steps.

To describe the execution infrastructure, another YAML file is required. This file serves as the entry point for the workflow execution and performs two main tasks:

- Representing pointers to workflow and model descriptions (e.g., type, configuration, and binding) and specifying how they should relate to each other.
- Mapping each workflow step to the most suitable execution environment, thereby integrating the hybrid layer into the workflow design process.

This setup allows for precise control over which environment each task of the workflow will be executed in.

The integration of VisIVO with StreamFlow addresses the need to improve portability by decoupling multiple components from platform-specific dependencies. This separation enables the workflows to be executed across different High-Performance Computing (HPC) infrastructures without relying on a specific underlying environment. By abstracting execution environments from the application logic, the system ensures that workflows can be transferred and executed on heterogeneous systems with just minor changes.

In addition to enhancing portability, this work raises the level of abstraction by using StreamFlow to define workflows that streamline complex task orchestration. StreamFlow allows tasks to be expressed in a high-level, platform-independent manner, facilitating the seamless management of dependencies

and execution across various environments[28]. This level of abstraction is critical for ensuring the scalability and adaptability of workflows, enabling them to run efficiently on both local clusters and cloud-based platforms.

5.2 Methodology

5.2.1 Hybrid Workflows

We employed workflow abstractions to allow a portable representation of the VisIVO modular applications and their resource requirements. This approach fosters reproducibility and maintainability, allowing users to take advantage of heterogeneous HPC facilities (including also mixed HPC-Cloud resources) while minimizing data transfer overheads.

Hybrid workflows[26] allow the distribution of multiple steps of a workflow on different heterogeneous and independent computing infrastructures. This flexibility enables the orchestration of multiple kinds of workloads, where each step can be mapped to the most suitable platform, ensuring portability, reproducibility, and maintainability.

StreamFlow complements a standard workflow graph, composed of multiple steps and their interdependencies, with a declarative description of potentially complex execution environments. This hybrid workflow model allows the execution of different steps on multiple sites without the need to share data spaces, authentication protocols, or bidirectional network channels. The StreamFlow control plane takes care of all the orchestration aspects of a workflow execution, including data movements, fault tolerance, and provenance.

With the help of hybrid workflows, different VisIVO modules can run in a multi-container environment, seamlessly switching between different container runtimes such as Docker or Singularity. Moreover, independent workflow steps can be executed in a concurrent fashion on top of multi-agent ecosystems (Cloud/HPC), provided they have no interdependencies, enabling better resource utilization. Figure 5.1 shows an example of a complex deployment model for the VisIVO workflow, in this case we can see that the first workflow step of VisIVOImporter will be assigned to a cloud infrastructure, then in the second step there are two independent VisIVOFilter tasks that can be executed

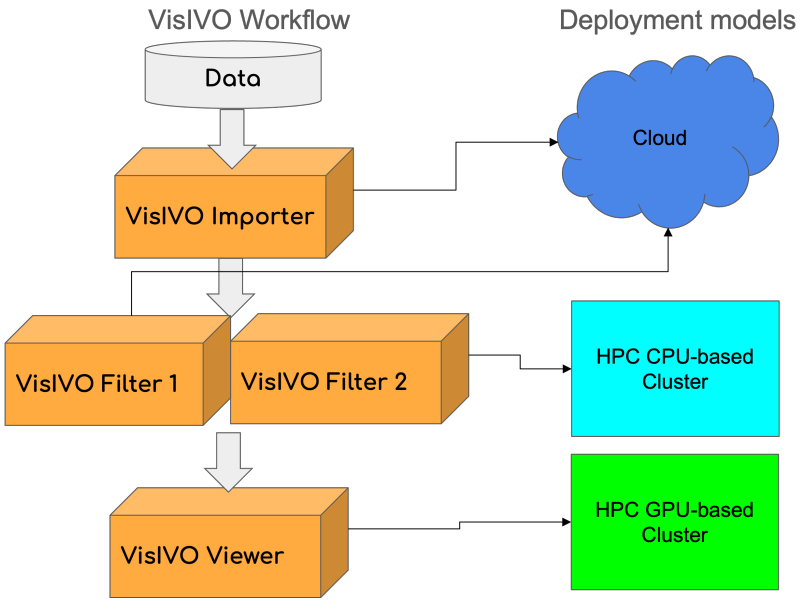

FIGURE 5.1: An example of deployment model for the VisIVO workflow.

independently on two different infrastructures, in this case Cloud and an HPC CPU-based Cluster, and the last step of the workflow can be executed on an HPC GPU-based cluster.

As a result, StreamFlow provides VisIVO workflows with portability and reproducibility. It achieves this by relying on a set of open standards widely adopted in the scientific workflow community, bringing off-the-shelf support for a broad ecosystem of workflow managers (e.g., Galaxy[1] or Airflow[2]) and avoiding technology lock-in. In particular, it relies on the CWL for workflow design, the Workflow Hub[3] platform for publishing, and the Workflow Run RO-Crate[4] format for provenance collection.

5.2.2 VisIVO workflows

We developed a set of CWL workflows composed of sequences of three applications: VisIVOImporter, VisIVOFilter, and VisIVOViewer (Figure **??**). All

[1]Galaxy, https://galaxyproject.org/
[2]Airflow, https://airflow.apache.org/
[3]Workflow Hub, https://workflowhub.eu/
[4]Workflow Run Crate, https://www.researchobject.org/workflow-run-crate

the workflows are publicly available on the VisIVOCWL GitHub repository[5] and they were assigned the Apache 2.0[6] license. We will focus on the first four Workflows.

Workflow n. 1

Workflow$_1$ performs a simple point rendering of the particles, in this case it just involves VisIVOImporter and VisIVOViewer modules. The VisIVOImporter step is common to the two workflows and it executes the following script:

```
$ VisIVOImporter --fformat ascii clusterfields4.ascii
```

The final step of Workflow$_1$ executes the command

```
$ VisIVOViewer -x X -y Y -z Z --scale
  --glyphs pixel VisIVOServerBinary.bin
```

To make this workflow work, in the directory of the workflow "docker_VisIVO_ImpView_Workflow.cwl" we needed to define some other files as inputs: paramFile_Imp.txt, clusterfields4.ascii, paramFile_View.txt, streamflow.yml.

These are the inputs of the workflow, as described in the `"docker-job_VisIVO_ImpView_Workflow.yml"` file. The importer command of the workflow, generates the `"VisIVOServerBinary.bin"` and the `"VisIVOServerBinary.bin.head"` files, that are taken as input by the VisIVOViewer command of the workflow. These two files will not exist anymore at the end of the workflow execution. The workflow generates as output four `.png` images, `"VisIVOServerImage0.png"`, `"VisIVOServerImage1.png"`, `"VisIVOServerImage2.png"`, and `"VisIVOServerImage3.png"`, that are saved in four different directories.

Workflow n. 2

Workflow$_2$ performs again point rendering of a GADGET snapshot after applying a randomizer filter, thus involving three VisIVO modules. Using a GADGET snapshot, we demonstrate correct MPI and OpenMP execution within the workflow.

[5] https://github.com/VisIVOLab/VisIVOCWL
[6] https://spdx.org/licenses/Apache-2.0

The GADGET VisIVOImporter step is common to multiple workflows and it executes the following script:

```
$ export OMP_NUM_THREADS=AA
$ mpirun --np BB VisIVOImporter --fformat gadget --out NewTable
--file snapdir/snap_091.0
```

The script above runs in parallel on a single node over `AA` OpenMP threads and `BB` MPI processes passed as parameters to the workflow itself. This command, takes as input a file of GADGET format (a GADGET snapshot) and it produces a VBT for each type of particles in the GADGET file supplied. In this specific case, it produces two files, NewTableHALO.bin and NewTableGAS.bin, and their corresponding header files, NewTableHALO.bin.head and NewTableGAS.bin.head, which contain information about the dark matter halo and the gas of the GADGET snapshot taken as input. Specifically, this command is executed as:

```
$ export OMP_NUM_THREADS=AA
$ mpirun --np BB VisIVOImporter paramFile_Imp_Par_MPI.txt
```

The `paramFile_Imp_Par_MPI.txt` parameter points to a file with the list of parameters of the VisIVOImporter command. This is also true for all the steps reported, but we always report the full list of parameters for clarity. In these test cases, the following step of the two workflows takes as input the (`NewTableGAS.bin`, `NewTableGAS.bin.head`) couple alone but the procedure is the same if we consider the (`NewTableGAS.bin`, `NewTableGAS.bin.head`) couple.

After the importer step, Workflow_2 executes a filter step that executes the following command

```
$ VisIVOFilter --op randomizer --perc 50.0
  --out NewTableHALORand.bin --file NewTableHALO.bin
```

This operation generates a new VBT containing a randomized subset that maintains just 50% of the particles from the input VBT.

Finally, the viewer step of Workflow_2 executes the command to generate four `.png` images:

```
$ VisIVOViewer --x POS_X --y POS_Y --z POS_Z
  --out VisIVOServerImage NewTableHALORand.bin
```

Workflow n. 3

Workflow$_3$ performs point rendering of a GADGET snapshot after applying two filters: PointProperty and Merge, this time it will render the density of particles.

The Importer step is the same described on **Workflow**$_2$.

Then, Workflow$_3$ executes the first filter step that executes the following command:

```
$ VisIVOFilter --op pointproperty --resolution X_RES Y_RES Z_RES
  --points POS_X POS_Y POS_Z --out density.bin --outcol density
  --file NewTableHALO.bin
```

This operation modifies the input, in this case the (NewTableHALO.bin, NewTableHALO.bin.head) couple of files. The `pointproperty` operation assigns a property to each data point in the `NewTableHALO.bin` VBT. Specifically, it creates a temporary volume using a field distribution, adopting the CIC algorithm, on a regular mesh. Then, it computes, with the same CIC algorithm, the property for each data point, considering the cells where the point is spread on the volume. The output can be stored in a new table or in a new field of the original input table. In this case, the command creates a column with the density property not present in the original VBT and it saves it in the new file `density.bin`. A correspondent `density.bin.head` is generated as output of the VisIVOFilter command. The second VisIVOFilter step executes the following command

```
$ VisIVOFilter --op merge --out NewTableHALOMerge.bin
--filelist tabselection.txt
```

This command generates a new VBT with the (`NewTableHALOMerge.bin`, `NewTableHALOMerge.bin.head`) couple of files, merging the original (`NewTableHALO.bin`, `NewTableHALO.bin.head`) VBT with the new (`density.bin`, `density.bin.head`) VBT, following the specifications reported in the `tabselection.txt` file, in which there will be defined the required columns to retrieve from each VBT to generate the final VBT.

Finally, the viewer step of Workflow$_3$ executes the command:

```
$ VisIVOViewer --x POS_X_tab_1 --y POS_Y_tab_1 --z POS_Z_tab_1
--color  --colorscalar density_tab_2 --colortable volren_glow
--logscale --out VisIVOServerImage NewTableHALOMerge.bin
```

this command generates four `.png` visualizations from the input VBT (NewTableHALOMerge.bin, NewTableHALOMerge.bin.head), using the density value as color scalar.

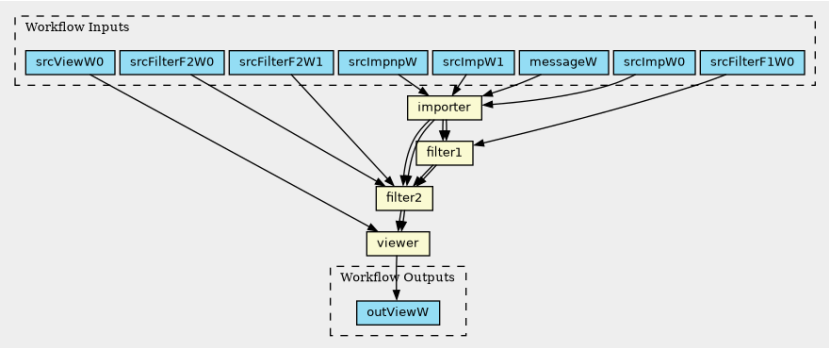

FIGURE 5.2: Visual representation of the VisIVO CWL **Workflow**$_3$.

Workflow n. 4

Workflow$_4$ performs volume rendering of a GADGET snapshot after applying the PointDistribute filter.

The Importer step is the same described on **Workflow**$_2$.

It executes then the filter step that executes the following command:

```
$ VisIVOFilter --op pointdistribute --resolution X_RES Y_RES Z_RES
--points POS_X POS_Y POS_Z --out densityvolume.bin
--file NewTableHALO.bin
```

This command takes as input the (`NewTableHALO.bin`, `NewTableHALO.bin .head`) VBT and it transforms it applying the `pointdistribute` operation, saving the new output in the `densityvolume.bin` file. Specifically, the `pointdistribute` operation creates a table which represents a volume from selected fields of the input table that are distributed using NGP, CIC (default) or TSC algorithm, and produces, by default, a density field, which is distributed and divided among the cells in the volume.

After the filter step, the viewer step executes the following command:

```
$ VisIVOViewer --volume --vrendering --vrenderingfield Constant
--color --colortable volren_glow --showlut --out img
--file densityvolume.bin
```

This command will again generate 4 `.png` images, this time performing the volume rendering using the volume VBT generated by the pointdistribute filter.

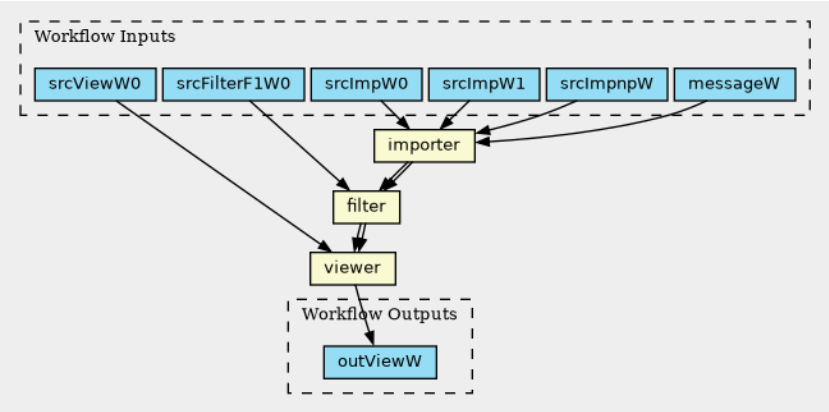

FIGURE 5.3: Visual representation of the VisIVO CWL **Workflow**$_4$.

Workflows Execution

The entire Workflows are executed with the following commands[7], where the `.cwl` files contain the workflows and the `.yml` files describe the input of the workflows. We will focus on the commands for the fourth workflow, and it will be very similar for the others:

```
$ cwl-runner --streamflow-file streamflow.yml
  docker_VisIVO_ImpFilterView_Workflow_GADGET_PD.cwl
  docker-job_VisIVO_ImpFilterView_Workflow_GADGET_PD.yml
```

The `.cwl` file import three inner `.cwl` files, corresponding to each step of the workflow. Workflow$_4$ is represented pictorially in 5.3, respectively. The `docker_VisIVO_ImpFilterView_Workflow_GADGET_PD.cwl` scripts declares

[7]The names of the two `.cwl` and `.yml` files are reported in the corresponding sections of the `README.md` files in the `VisIVOCWL` repository.

a CWL `Workflow` class whereas each `.cwl` step called within each of the two workflows contains a CWL `CommandLineTool` class. All VisIVO Docker containers are bound to their related steps by means of the CWL `DockerRequirement` feature.

Finally, the `streamflow.yml` file, the entry point of StreamFlow, is in charge of relating each workflow step with the best suitable execution environment, actually plugging the hybrid layer in the workflow execution process. As a first step, StreamFlow translates the CWL dataflow semantics into an internal workflow representation, explicitly modelling a macro dataflow graph [27]. To provide enough flexibility, StreamFlow adopts a three-layered hierarchical representation of execution environments:

- a `deployment` is the *unit of allocation*, i.e., all its components are always co-allocated when executing a step;
- a `service` constitutes the *unit of binding*, i.e., StreamFlow users can map each workflow step to a single service for execution;
- a `resource` is a single instance of a potentially replicated service and constitutes the *unit of scheduling*, i.e., each step of a workflow is offloaded to a configurable number of resources to be processed.

5.2.3 CWL Script Example

In this section the CWL defined for Workflow 1 will be presented to show how it was exploited to create a reproducible pipeline for VisIVO. The complete set of CWL files for all the presented workflow are available in Appendix B.

The listing 5 show the CWL code for Workflow 1. It defines how many inputs are required to the workflow and their types, in this case they are all files. Then, it defines the output, which in this case is an array of file, more precisely four ".png" images and the folder where they will be stored. This inputs are defined in the following ".yml" file "docker-job_VisIVO_ImpView_Workflow.yml". The listing 6 show the YML file defined for this example.

Moreover, it defines the two execution steps that compose the pipeline. The first step, importer, invokes docker_VisIVOImporter.cwl, which takes the

```yaml
cwlVersion: v1.0
class: Workflow

inputs:
  srcImpW0: File
  srcImpW1: File
  srcViewW0: File

outputs:
  outViewW:
    type: File[]
    outputSource: viewer/outView

steps:
  importer:
    run: docker_VisIVOImporter.cwl
    in:
      srcImp0: srcImpW0
      srcImp1: srcImpW1
    out: [outImp0, outImp1]

  viewer:
    run: docker_VisIVOViewer.cwl
    in:
      srcView0: srcViewW0
      srcView1: importer/outImp0
      srcView2: importer/outImp1
    out: [outView]
```

LISTING 5: CWL describing Workflow 1

two inputs (srcImpW0 and srcImpW1) and produces two intermediate outputs (outImp0 and outImp1). These outputs are not directly returned by the workflow but are instead used during the second step. The CWL file for the importer step is defined as follows:

Listing 7 defines how the importer step is executed within the workflow. The `DockerRequirement` ensures that this step runs inside the VisIVO Server container image (`visivolab/visivoserver:latest`), guaranteeing a consistent and reproducible software environment, independently of the user's system.

The two inputs are declared explicitly, with `srcImp0` bound to the first command-line position of the `VisIVOImporter` executable through the `inputBinding` directive. The outputs, `outImp0` and `outImp1`, are collected using the `glob` directive, which matches the binary VBT file and its associated header generated by the importer. These files are then passed to the second step of the workflow, completing the first step of the pipeline.

The second step, viewer, invokes docker_VisIVOViewer.cwl. It receives as input the file required for visualization (srcViewW0), which is the parameter

```
srcImpW0:
  class: File
  path: paramFile_Imp.txt
srcImpW1:
  class: File
  path: clusterfields4.ascii
srcViewW0:
  class: File
  path: paramFile_View.txt
```

LISTING 6: YML file describing the inputs

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOImporter
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcImp1)
inputs:
  srcImp0:
    type: File
    inputBinding:
      position: 1
  srcImp1:
    type: File
outputs:
  outImp0:
    type: File
    outputBinding:
      glob: VisIVOServerBinary.bin
  outImp1:
    type: File
    outputBinding:
      glob: VisIVOServerBinary.bin.head
```

LISTING 7: CWL describing Importer Step for Workflow 1

file, together with the two intermediate products generated by the importer. This step will generate the final visualization outputs, four PNG images, which are collected by the workflow's outViewW output field.

5.3 Results

5.3.1 Data description

The presented workflows have been developed to support the studies of the effects of massive neutrinos on the large-scale structure of the Universe through visualization.

Cosmological neutrinos strongly affect the evolution of the largest structures in the Universe, i.e., galaxies, galaxy clusters, cosmic voids, and filaments. In this work we use large box-sized cosmological N-body simulations, the so-called "Dark Energy and Massive Neutrino Universe" (DEMNUni) suite [22, 95]. The DEMNUni simulations have been performed using the tree particle mesh-smoothed particle hydrodynamics (TreePM-SPH) code GADGET-3 [110], specifically modified by [119] to account for the presence of massive neutrinos. This modified version of GADGET-3 follows the evolution of Cold Dark Matter (CDM) and Hot Dark Matter (HDM) neutrino particles, treating them as two separated collisionless species.

For the particular case of this work, we have considered the DEMNUni sub-set of simulations with a starting redshift $z_{in} = 99$, and characterised by a comoving volume of $(500\ h^{-1}\ \mathrm{Mpc})^3$ filled with 2048^3 dark matter particles and, when present, 2048^3 neutrino particles [54, 118]. Given the large amount of memory required by the simulations, baryon physics is not included. We consider two different simulations, with a baseline *Planck* cosmology [97], namely a flat ΛCDM model generalized to a $\nu\Lambda$CDM framework by changing only the value of the sum of the three active neutrino masses $M_\nu = \sum_i m_{\nu,i} = 0.16$ eV, and keeping fixed Ω_m and the amplitude of primordial curvature perturbations. This sub-set of simulations is characterised by a softening length $\varepsilon = 5\,h^{-1}$ Kpc, and has been run on the Marconi100 supercomputer at CINECA[8], Italy. For each simulation, we have produced 64 output logarithmically equispaced in the scale factor $a = 1/(1+z)$, in the redshift interval $z = 0 - 99$, 50 of which lay between $z = 0$ and $z = 10$. For each of the 64 output times, we have dumped on-the-fly a particle snapshot composed by both CDM and neutrino particles.

[8] http://www.cineca.it/

5.3.2 Rendering results

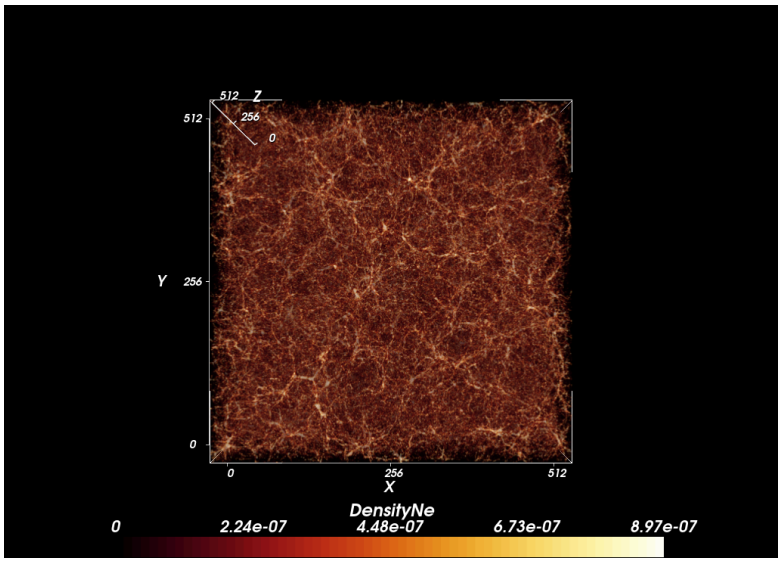

FIGURE 5.4: Volume rendering of cold dark matter particles, in the presence of neutrino particles.

The rendering was done using VisIVOViewer volume rendering implementation on the volume generated by the `pointdistribute` filter. A rendering option that was considered was to show the density of the CDM particles described in Section 5.3.1. Figure 5.4 shows the results of the volume rendering of the simulation. In addition to this, we are currently working on the rendering of an animation which shows the evolution of the simulation over time.

Then, we also compare the densities between the cases where neutrino particles are present and absent. In comparing the two densities, we calculated the difference between the densities in the volumes for the simulation with neutrino particles, ρ_1 and the case without, ρ_2, and divided this value by the absolute value of the density without neutrino particles, $\Delta\rho = (\rho_1 - \rho_2)/|\rho_2|$. Figure 5.5 shows the visualization of the particles in the case $\Delta\rho > 0$.

Another rendering result shows the time-evolution of the density of both CDM and neutrino particles as a movie of the different snapshots. The figure 5.6 shows the results of the volume rendering of the density of cold dark matter particles on the final snapshot of the simulation (corresponding to redshift

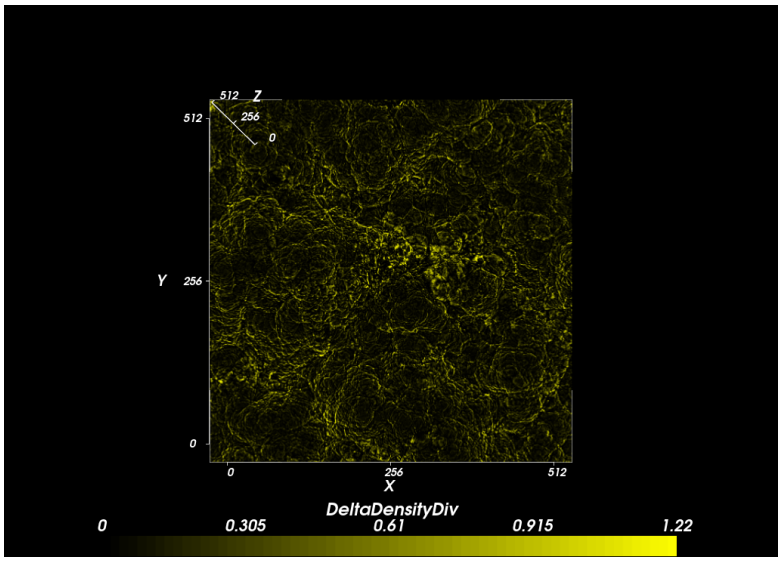

FIGURE 5.5: Volume rendering of $\Delta\rho > 0$

$z = 0$). For a visual comparison, the figure 5.7 shows a rendering of the density of the neutrino particles on the same snapshot. The two visualizations use different scales because they represent different density fields in separate domains. The first one for neutrino particles and the second one for cold dark matter particles, both occupy the same spatial region but can be represent with different scales. Different color maps were selected to better capture the characteristics of each particle type.

5.4 Conclusion

Astronomy and Cosmology (A&C) face the challenge of managing and extracting valuable insights from data sets at the petabyte scale generated by observations and simulations. The scale of these data volumes requires the development of innovative solutions to address storage, access, and analysis issues. The Visualization Interface for the Virtual Observatory (VisIVO) enables multi-dimensional data analysis and knowledge discovery within complex astrophysical and cosmological datasets.

In this chapter, we presented the implementation activities to realize high-performance visualization of outcomes coming from modern HPC applications

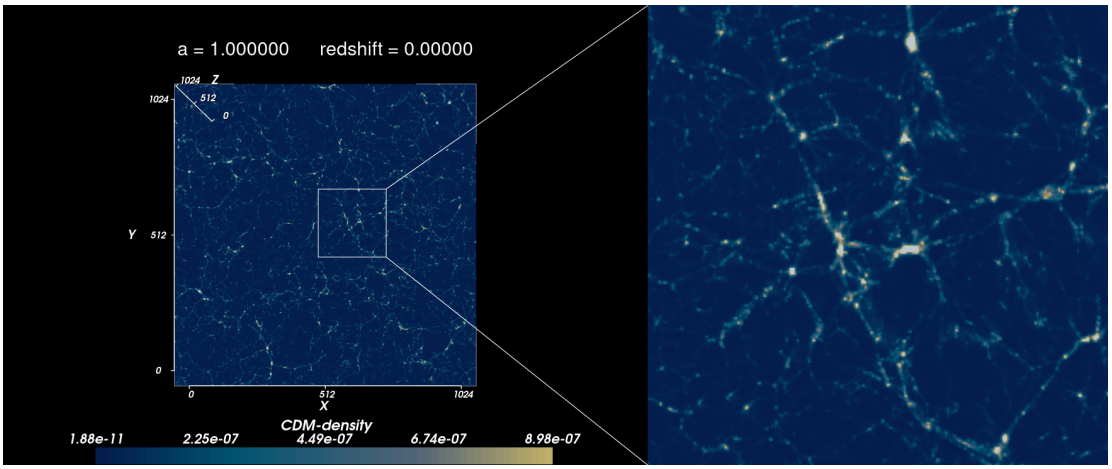

FIGURE 5.6: Volume rendering of Cold Dark Matter particles.

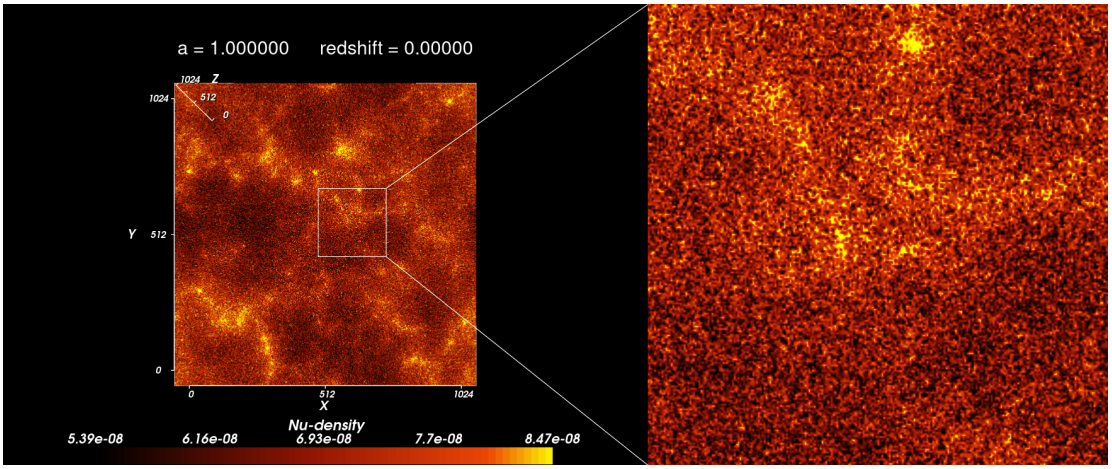

FIGURE 5.7: Volume rendering of neutrino particle density.

in A&C (such as large-scale cosmological simulations). In particular, we presented the integration of VisIVO with workflow abstractions and the StreamFlow execution models to enhance the portability of its modular applications and the related resource requirements, fostering reproducibility and maintainability, taking advantage of a flexible resource exploitation over heterogeneous HPC facilities, and including mixed HPC-Cloud resources.

We also presented a series of workflows that illustrate typical pipelines built with the VisIVO modules, and then demonstrated how they can be used to support the studies of large-scale cosmological simulations of the DEMNUni

project, highlighting the capability to produce complex visualization for analyzing the formation and evolution of cosmic structures.

Chapter 6

In-situ visualization with Hecuba

6.1 Introduction

The Astronomy and Cosmology (A&C) community is currently experiencing an unprecedented growth in the volume and complexity of data produced by both simulations and observational instruments. These datasets may reach petascale or exascale levels, thus they present significant challenges in terms of storage, analysis, and visualization. In particular, the traditional approach of storing simulation results on disk and then handled with post-processing has become a major bottleneck in high-performance computing infrastructures. The sheer size of the data means that I/O operations can dominate computational time, and storing the complete output becomes unfeasible.

One promising solution to these problems is the adoption of in-situ processing, where data analysis and visualization are performed concurrently with the simulation software using the same computational infrastructure. By eliminating the need to write results to disk, this approach not only reduces I/O overhead but also enables researchers to interact with their simulations in real-time.

In this chapter, we present our implementation of an in-situ visualization approach based on Hecuba, a distributed toolset designed to efficiently manage and access scientific data. Hecuba can retrieve data from streams using Apache Kafka or from a database (built on top of Apache Cassandra) that Hecuba uses internally to manage its own data structures, allowing to perform in-transit visualization, a particular form of in-situ visualization that exploits an I/O transport infrastructure[88]. Hecuba is intended to be integrated into the ChaNGa cosmological simulator, which generates large-scale N-body simulation data.

To prepare for this, we detail the modifications made to the VisIVO Server to support in-situ visualization using data streamed through Hecuba, bypassing the need for intermediate file storage.

6.2 Methodology

To perform in-situ processing using the multiple VisIVO Modules it is necessary to perform multiple steps.

The first step will be to understand well the type of data that we need to handle, in this case ChaNGa simulation data, and define a specific data model to adapt ChaNGa data to the Hecuba format.

The second step is to retrieve this simulation data using Hecuba within VisIVO, leveraging Hecuba's C++ API to access memory, in database or using streams.

The final step focuses on adapting the internal modules of VisIVO to operate directly on in-memory data structures. Since VisIVO traditionally relies on file-based data storage using the VisIVO Binary Table (VBT) format, this transition requires careful redesign to support in-situ workflows without intermediate file output.

6.2.1 ChaNGa using Hecuba Data Model

In chapter 2 we presented ChaNGa and Hecuba data formats, and to implement an in-situ pipeline there is the requirement to adapt ChaNGa data format using the available Hecuba data types in the most efficient way. For this reason, we worked with BSC's partner to define a possible prototype for the Hecuba's data structures that will be used to store the simulation data.

At the moment the chosen structure is based on Hecuba's `StorageDict` structure. The StorageDict is a dictionary that requires a specific Key and Value, in this case the key is defined using the following format:

```
KeyClass<double, float, float, float, float, float, float>.
```

This key format allows us to split the simulation domain into regular three-dimensional spatial blocks (referred to as data cubes). These cubes are simply

subregions of the 3D simulation volume used to partition the particle distribution. The six floats values in the key represent the lower and upper bounds of the cube along the x, y, and z axes: the first pair defines the cube's extent in x, the second in y, and the third in z. The leading double represents the simulation time in code units. The six `float` values in the key represent the spatial extensions of each data cube along the `x`, `y`, and `z` axes, precisely, the first couple of `float` parameters represents the x extension, the second couple represents y extension, and the third couple represents the z extension.

The value class at the moment is defined as:

`ValueClass<StorageNumpy>`.

This single StorageNumpy represents an array of gas particles in the data cube. At the moment we focused to make a working prototype with only a type of particles, we defined also a possible alternative value class for handling all the three type of particles available in ChaNGa datasets as follows:

```
1  class particleObject: public StorageObject {
2  public:
3      HECUBA_ATTRS(
4          StorageNumpy, gas,
5          StorageNumpy, dark,
6          StorageNumpy, star
7      );
8  };
9
10 ValueClass<StorageObject>;
11 }
```

In which we have three StorageNumpy as attributes of a StorageObject, one for each type of ChaNGa particles, inside a StorageObject.

Each `StorageNumpy` has two dimensions: the first corresponds to the number of particles of that type, while the second corresponds to the number of particle features, that is, the physical attributes stored for each particle. In the case of ChaNGa, these features typically include positions, velocities, mass, density, internal energy, smoothing length, and any additional fields defined

in the simulation. These sizes can be retrieved directly from the `StorageNumpy` metadata. This structure allows us to retrieve all the needed data at once, from the simulation side it is also possible to send to the database just one type of particle if it's the only one required.

For this solution, we also considered to adapt some header information in the form of Hecuba data model, using this simple StorageObject:

```
class headerObject:public StorageObject{
public:
HECUBA_ATTRS (
     double, time,
     int32_t, nbodies,
     int32_t, ndim,
     int32_t, nsph,
     int32_t, ndark,
     int32_t, nstar,
     int32_t, pad,
    )
};
```

6.2.2 Integrating Hecuba within VisIVO Server

To enable VisIVO Server to retrieve simulation data directly from streams or databases, rather than reading from disk, we integrated it with Hecuba. This integration allows simulation data to be streamed or queried on demand. To achieve this, we developed a new VisIVO Importer that interfaces directly with Hecuba's C++ API.

Figure 6.1 represents the UML diagram of the implemented Hecuba reader. All VisIVO reader modules are built on top of the abstract base class AbstractSource. The HecubaSource class does this too and has to implement the two virtual methods: readHeader() and readData().

The readHeader() method gathers metadata from headerObjects and stores the number of particles for each type (gas, dark matter, stars) in the class variables nsph, ndark, and nstar.

The readData() method is responsible for actually retrieving simulation data using the Hecuba API. For this first implementation we focused on working with the Gas Particles but we already defined the interface for the future methods to handle also Dark Matter and Star particles, each method corresponds to the different particle type available in ChaNGa, and they are used to delegate the handling of the data and they are name: writeGasParticles(), writeDarkParticles(), and writeStarParticles(). These methods extract the particle fields from the StorageNumpy and rearrange them to match the VisIVO Binary Table format or the in-memory structure that we will define in the following section. This step is necessary because ChaNGa stores data in an Array-Of-Structures format, whereas the VBTs use a Structure-Of-Arrays layout.

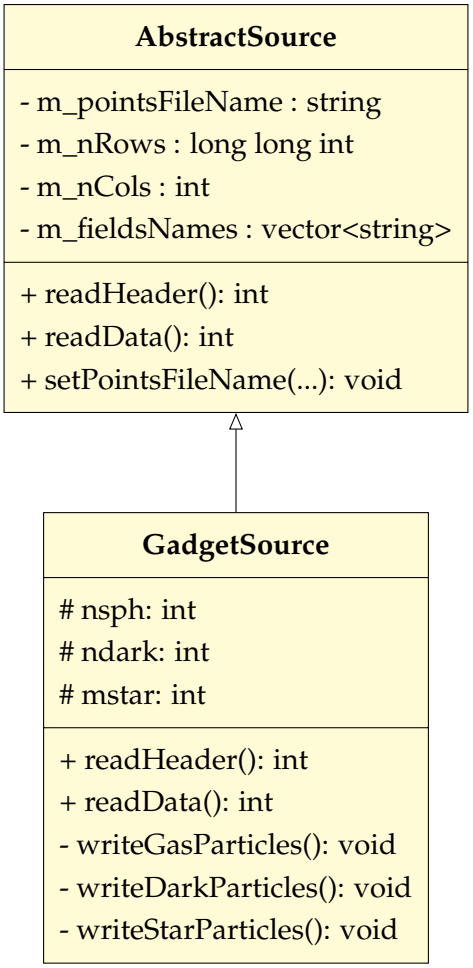

FIGURE 6.1: UML diagram of Hecuba reader class

We present the details of the code for handling the data:

```
1   Dict particleDict;
2       particleDict.getByAlias("simulationdata");
3   Key pk;
4       Value v_read;
5       StorageNumpy testNumpy;
6       for (auto it = particleDict.begin(); it != particleDict.end(); ++it) {
7               pk = *it;
8               v_read = particleDict[pk];
9               testNumpy = Value::get<0>(v_read);
10              writeGasParticles(testNumpy);
11      }
```

In the first five lines we just define all the required variables that are needed, in particular the StorageDict particleDict variable is defined and retrieved using Hecuba's API call getByAlias(). Hecuba exposes an iterator that allows us to define the right key to pass to the StorageDict to retrieve a specific Value element, this operation is handled in the lines 6-11 using the for cycle. The operation in line 9 retrieves the first StorageNumpy from the dictionary and passes it to the writeGasParticles method we defined previously.

6.2.3 Handling data in-memory

VisIVO modules rely heavily on the VisIVO Binary Table (VBT), a file-based internal data format. After the importer generates a VBT, the Filters and Viewers modules read from it directly. However, to support in-situ processing, these modules must be adapted to operate without relying on VBTs.

VisIVO Server uses a support class called VSTable, which is the core data container within VisIVO, designed to manage tables stored in the VisIVO Binary Table (VBT) format, in figure 6.2 we can see the details and its interface, it encapsulates metadata such as the number of rows and columns, column names, data types, and endianness, and provides I/O methods to read and write data from binary files. VSTable assumes a file-backed backend, meaning

that all read and write operations are performed on disk. For example, the getColumn() and putColumn() methods rely on seek operations and assume data is organized in a row-major layout across files.

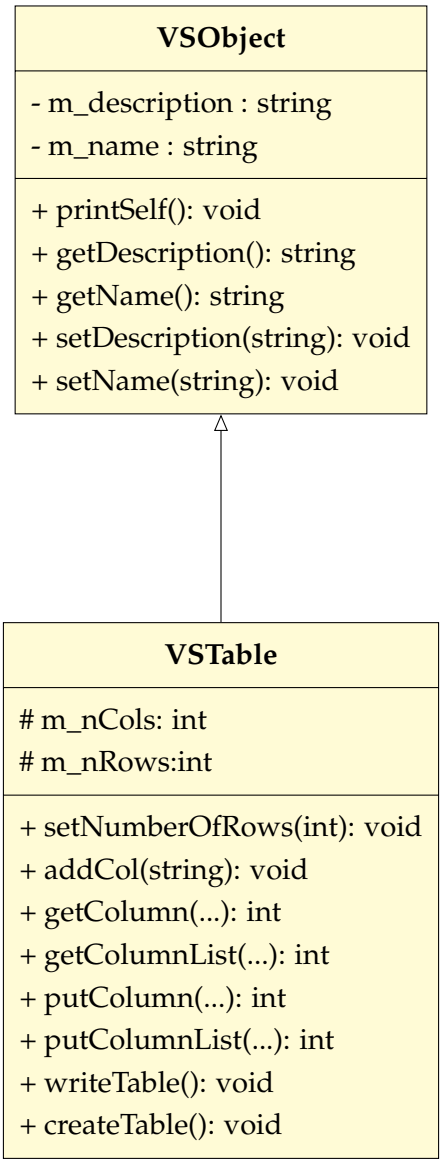

FIGURE 6.2: UML diagram of `VSTable` class inheriting from `VSObject`.

To support in-situ visualization, in which simulation data resides in memory and must be visualized without intermediate disk I/O, we used VSTable as a base because it provides all the operations that we need to handle and

developed a new subclass, named VSTableMem, that implements the same interface but stores data entirely in memory. This class replaces the use of binary files with a std::vector<std::vector<float» structure (fieldArray), allowing data columns to be accessed and modified directly. All read and write operations (getColumn, putColumn, etc.) were overridden to operate on in-memory buffers, while maintaining compatibility with VisIVO's existing module expectations. This approach allows the other modules to handle data in the preferred way by using either the parent class or the subclass VSTableMem.

Below we show a simplified example of the overridden `putColumn()` and `getColumn()` methods:

```
int VSTableMem::putColumn(const std::string &name, float *data) {
    int col = getColumnIndex(name);
    if (col < 0) return -1;

    fieldArray[col].assign(data, data + m_nRows);
    return 0;
}

int VSTableMem::getColumn(const std::string &name, float *buffer) {
    int col = getColumnIndex(name);
    if (col < 0) return -1;

    std::memcpy(buffer, fieldArray[col].data(),
                m_nRows * sizeof(float));
    return 0;
}
```

In the original `VSTable`, these methods relied on `fseek()` and `fread()` operations to access data stored in a VBT file. In the in-memory version, the constructor allocates an array of column buffers: one `std::vector<float>` per column, each sized to the number of particles. These buffers replace the file-based backend and allow all `getColumn()` and `putColumn()` operations to operate directly on memory. By fully overriding them in `VSTableMem`, we allow the rest of VisIVO Server to operate unchanged, while transparently selecting between disk-based and memory-based solutions.

This new implementation must be integrated to the various VisIVO components, here we show an example of how it can be done on the new importer based on Hecuba, implemented with the `HecubaSource` class, which can choose the desired implementation creating an instance of `VSTableMem` instead of a file-backed `VSTable`. The behavior of one of the reader support function `writeGasParticles()` is illustrated in the following code in a simplified form. After retrieving a`StorageNumpy` from Hecuba, the importer extracts each field and writes it directly into the in-memory table:

```
void writeGasParticles(StorageNumpy s) {
    nParticles = s.metas[0];
    nFeatures  = s.metas[1];
    data = (double*) s.data;

    // Select solution
    if (useMemory) {
        table = new VSTableMem();
        table->setNumberOfRows(nParticles);
        for each field in types:
            table->addCol(field);
    } else {
        open output file "GAS.bin";
    }

    // For each feature, extract column and write it
    for (t = 0; t < types.size(); t++) {

        fill buffer[] with feature t for all particles;

        if (useMemory)
            table->putColumn(t, buffer);
        else
            write buffer to file;
    }

    if (useMemory)
```

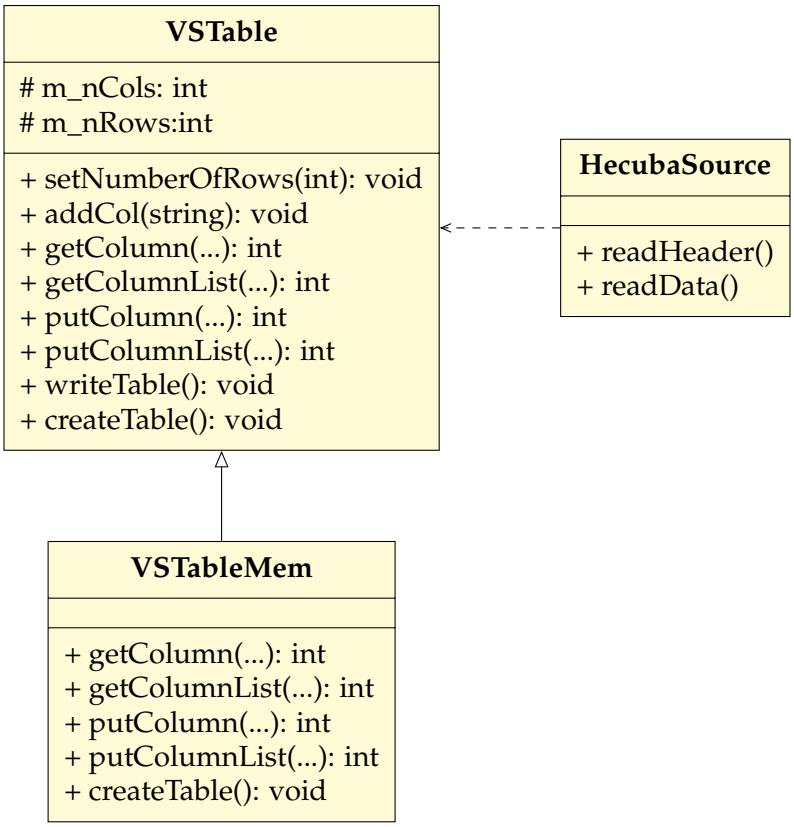

FIGURE 6.3: Class diagram for in-memory data handling. 'VSTableMem' is a subclass of 'VSTable', while 'HecubaSource' depends on the interface provided by 'VSTable'.

```
        memTables.push_back(table);
    else
        write VBT header for GAS.bin;
}
```

In figure 6.3 we show the UML diagram of the new VSTableMem subclass and the HecubaSource dependency.

We show the detailed sequence diagram on figure 6.4.

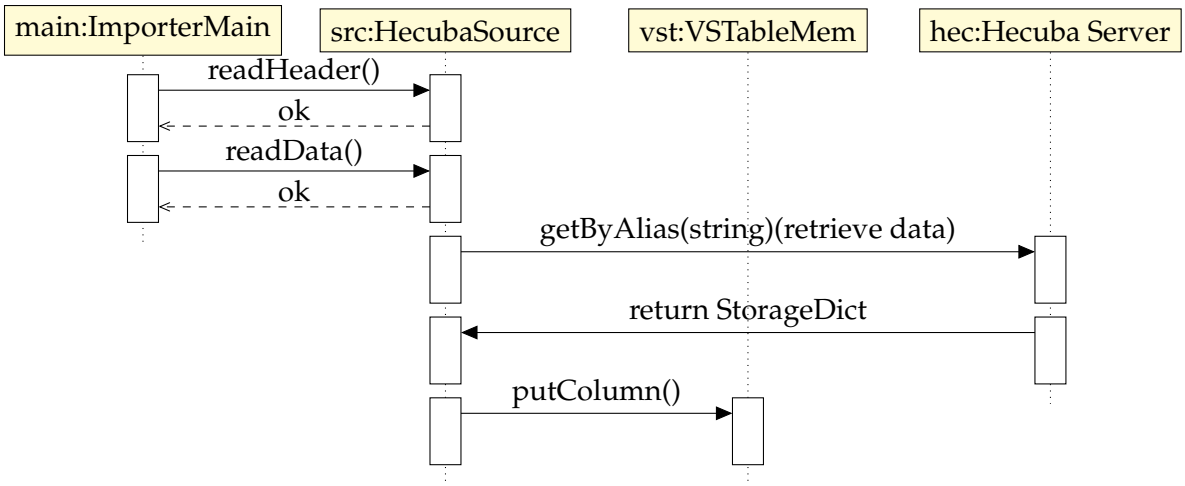

FIGURE 6.4: Sequence diagram detailing the execution of the `HecubaSource` class in VisIVO: data are fetched from Hecuba and stored in a `VSTableMem`, which is then available for future pipeline steps.

6.2.4 Extending VisIVO Library

The last work to obtain the possibility to create fully functional in-memory pipeline, that would allow in-situ visualization is to extend the already existing VisIVO Library API to support the new features presented in the previous sections.

VisIVO Library is organized using environments that are represented using a variable that contains all settings for a specific operation.

The first step is trivial, and it consists just in adding the Hecuba Importer to the supported importers. So a user will be capable to setup the Hecuba importer using the API by setting the format attribute like this:

```
VI_SetAtt(&importerEnv, VI_SET_FFORMAT, "hecuba")
```

The second step consists of adding a new environment setting to let the module use the in-memory version of the importer and the viewer classes. This was also a trivial step and the user just needs to setup the variable using this call:

```
VI_SetAtt(&importerEnv, VI_SET_INMEMORY, "")
```

Furthermore, a new library function was implemented to allow users to retrieve in-memory vectors generated by the importer with the new VSTableMem class and pass them to the Viewer. The std::vector can be easily retrieved and passed from the importer to the viewer, this operation is rather simple and just takes advantage of the new implementation.

The new function is defined as follows:

```
int VV_SetTableFromImporter(VisIVOViewer* viewer, VisIVOImporter*
                  importer, size_t tableIndex);
```

The first parameter is the `VisIVOViewer` environment, the second is the `VisIVOImporter` environment, and the third specifies the index of the desired table to retrieve. This is required because each `VisIVOImporter` can generate multiple tables, for example, if the simulation contains different types of particle.

6.3 Results

As a result we will present an example of possible complete in memory pipeline that simulates a typical usage of the VisIVO Library API while performing in-situ visualization.

6.3.1 In-situ Pipeline

Using the new added functionalities to VisIVO and its integration with Hecuba, it is possible to create a simple pipeline that uses the new Hecuba importer and then the VisIVO Viewer to visualize test data using VisIVO Library.

As for the first step, it consists on executing a producer that retrieves ChaNGa data from a reading a TIPSY sample file, and then it uses Hecuba API to send the data to the importer.

For the second step we implemented a pipeline using the VisIVO Library API. We present a snippet of code from the pipeline:

```
1 VisIVOImporter importer;
2
```

```
3  VI_SetAtt(&importer, VI_SET_FFORMAT, "hecuba");
4  VI_SetAtt(&importer, VI_SET_INMEMORY, "");
5
6  int result = VI_Import(&importer);
7  if (result != 0) {
8      std::cerr << "Import failed with code: " << result << std::endl;
9      return 1;
10 }
11
12 VisIVOViewer viewer;
13 VV_SetAtt(&viewer, VV_SET_FIELD, "POS_X POS_Y POS_Z");
14 VV_SetAtt(&viewer, VV_SET_LOGSCALE, "");
15 VV_SetAtt(&viewer, VV_SET_USEMEMORY, "");
16 VV_SetAtt(&viewer, VV_SET_AXES, "");
17 VV_SetAtt(&viewer, VV_SET_BOX, "");
18 VV_SetAtt(&viewer, VV_SET_COLOR, "");
19 VV_SetAtt(&viewer, VV_SET_COLORSCALAR, "VEL_X");
20 VV_SetAtt(&viewer, VV_SET_COLORTABLE, "volren_glow");
21 VV_SetAtt(&viewer, VV_SET_OUT, "./output.png");
22 VV_SetAtt(&viewer, VV_SET_IMAGESIZE, "medium");
23
24 if (VV_SetTableFromImporter(&viewer, &importer, 0) != 0) return 0;
25 int visResult = VV_View(&viewer);
```

The VisIVO Importer environment is defined in the first 11 lines, the required attributes are set in line 3 and 4, in this case we set the format and the "use_memory" attribute, using two library calls. Then the importer operation is executed in line 6.

Subsequently, the VisIVO Viewer environment is also defined with the required attributes in lines 12-22, in this case the particle coordinates are defined in line 13 and the "use_memory" attribute in line 15. We also define the required color settings and the desired filename for the output.

As a final step, in line 24 we use the new function that allows us to retrieve the data from the Importer, At the moment for our case having only one type of particle, it retrieves the first table that uses 0 as index, representing the gas particles.

The pipeline was tested using a Tipsy file of 2.6 million particles (which occupies 109MB). And the rendering results showed that it is working correctly, demonstrating that now VisIVO is capable of performing in-situ visualization.

Currently Hecuba's C++ API does not support streaming capabilities for the StorageDict data structure, but it is supported by the Python API. Currently, we could test the streaming capabilities on a Paraview plugin implemented in Python on the same Tipsy file. The execution time using a file based approach was 3.02s, but reduced to 0.91s when using Hecuba to stream data, that means that it achieave a speed-up of ~3.3×. This test shows what kind of of improvements we can expect when the C++ API will be updated.

Figure 6.5 represents a visualization that is possible to obtain using the presented pipeline from ChaNGa simulation output.

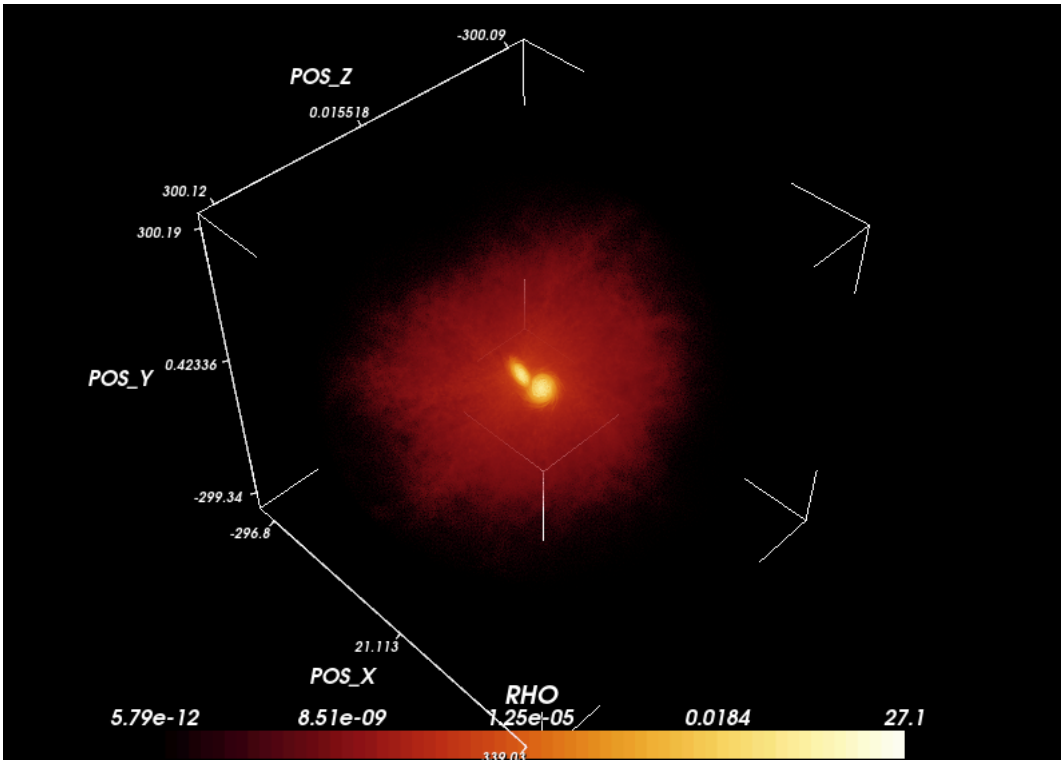

FIGURE 6.5: Visualization of ChaNGa simulation output.

6.4 Conclusion

In-situ processing is increasingly adopted to overcome the I/O bottlenecks associated with storing large-scale simulation outputs. This paradigm is especially useful in the context of Astrophysics and Cosmology, where the capability to analyze data during run-time enables more efficient workflows.

This chapter presents the development and refinement of an in-situ processing strategy leveraging the Hecuba toolset, integrated into the VisIVO visualization environment, to enable in-situ visualization during runtime.

For future work, we are working on adding the capability of in-memory data handling also to the VisIVO Filter module. This enhancement will enable more complex workflows, such as the generation of volumetric data using the data received by the Hecuba importer and then performing volume rendering without intermediate storage.

The second step would be to perform performance testing to compare the advantage of the in-situ pipeline and the scalability on simulations of various sizes. At the moment Hecuba's C++ API does not fully support the streaming capabilities, but the current integration in VisIVO will be capable to support the streaming capabilities as soon as Hecuba will be updated in the following months.

Chapter 7

Interactive High-Performance Visualization

7.1 Introduction

In recent years, Jupyter Notebooks have emerged as an interface for scientific computing, data analysis, and interactive research workflows. Jupyter Notebooks are incredibly powerful tools as it allows to interactively develop and present data science projects. Its versatility makes it a powerful tool not only for education and prototyping but increasingly as a point-of-entry to high-performance computing (HPC) infrastructures[84].

The shift towards notebook-based interfaces allow to lower the barrier of entry to new users while maintaining access to powerful computational backends. Traditional HPC environments often rely on command-line tools and batch scheduling systems, which can be efficient but intimidating for new users or researchers whose primary expertise is not computer science, as an example physicists that may need to visualize simulation data using a tool like VisIVO Server. Notebooks help improving the usability by offering an interactive, scriptable, and highly visual interface that brings HPC resources closer to the flexibility of cloud platforms.

For this reason, it would be useful to allow user to access in a easier way to VisIVO functionalities.

In this chapter, we discuss the innovative incorporation of VisIVO into the InterActive Computing (IAC[1]) service [2] provided by the Cineca HPC centre.

[1]IAC, https://jupyter.g100.cineca.it/

IAC service allows end users to obtain access to HPC compute nodes via the web browser, replacing the traditional approach to HPC resources which is limited to a command line interface via *ssh* and a queued batch system. Typical usage scenarios for IAC are interactive analyses, visualizations, and steering of simulations running on scalable compute services that abstract large-scale computing resources, which can be used for running highly parallel simulation applications, but are also suitable for data analysis tasks, involving extreme-scale data sets.

7.2 Methodology

Integrating the VisIVO kernel into the InterActive Computing service simplifies user workflow, eliminating the need for manual module loading. Previously, users logging into the Galileo 100 (G100) cluster had to rely on *ssh* and command line interface to enable VisIVO functionalities. Still, with such an approach the generated PNG files were not easily accessible within the cluster. On the other hand, in the alternative approach we tested in this work, we created a dedicated Jupyter kernel for VisIVO inside the InterActive Computing interface. The VisIVO kernel is now pre-configured, allowing users to log into the InterActive Computing service and select the VisIVO kernel, which automatically loads the VisIVO module. It enables seamless execution of VisIVO commands without additional configuration, ensuring that all generated PNG files are instantly visible within the web interface. The integration simplifies the process and enhances efficiency, allowing users to focus on their scientific visualization tasks without dealing with command-line complexities.

7.2.1 VisIVO wrappers

VisIVO is implemented in C/C++; thus, its compilation provides binary executables. Jupyter allow to integrate binary executables in its interface, but it provides out-of-the-box support for adding interaction with custom Python

environments. For this reason, we implemented a Python wrapper[2] to connect Python and the compiled VisIVO binaries and then executing VisIVO commands within a Python environment.

We mainly rely on the standard library from Python; in particular, we used the *subprocess* package to run the available VisIVO commands. Execution details are recorded using the *logging* package, which also will be useful the future developments[47].

The Python wrapper program uses multiple functions to manage the three main VisIVO commands: VisIVOImporter, VisIVOFilter, and VisIVOViewer. This is done with the idea of using decorators in the future for those functions and enhance Visivo functionalities.

This structure ensures that each function is dedicated to a specific command, each one will focus on the execution of CLI command while also managing the logging, and validating the passed arguments with the allowed option to the underlying command.

We focus on the importer function, but the work done on the other function is very similar. The options from the VisIVOImporter binary are converted as arguments of the function:

```
def importer(input_file, *flags, mpi=False, tasks=None,
             fformat=None, out=None, volume=None,
             compx=None, [...]):
```

each of the options will be processed to check the variable type usin an `options` dictionary defined as follows:

```
        options = {
        "fformat": (fformat, str),
        "out": (out, str),
        "volume": (volume, str),
        "compx": (compx, float),
        "compy": (compy, float),
        "compz": (compz, float),
        "sizex": (sizex, float),
        [...]
```

[2]VisIVO Python Wrapper: https://github.com/VisIVOLab/VisIVOPythonWrapper

This dictionary is also passed to a function that will pipe the options to the text command:

```
command = "VisIVOImporter"
_corefunction(command, input_file, *flags, mpi=mpi,
              tasks=tasks, options=options)
```

We chose this approach over more compact alternatives because we wanted to hide as much as possible the VisIVO CLI in the background to the user. For this reason, we opted to list all the possible options as function arguments, rather then embedding them in a single optional list of tuples.

We present now the main aspects of the execution of the `_corefunction`. At first it checks the type of the options values passed from the outer function:

```
worked_options = {}
if options:
    for key, (val, expected_type) in options.items():
        if val is not None:
            if not isinstance(val, expected_type):
                raise TypeError(f"'{key}' must be of
                      type {expected_type.__name__}")
            if expected_type is bool:
                if val:  # add only if True
                    worked_options[key] = val
            else:
                worked_options[key] = val
```

Then, it converts the boolean variables into options without values, and the other ones into options with values:

```
for option, value in worked_options.items():
    if isinstance(value, (list, tuple)):
        command.append(f"--{option}")
        command.extend(map(str, value))
    elif value and type(value) != bool:
        command.append(f"--{option}")
        command.append(str(value))
    else:
```

```
9           if value != False:
10              command.append(f"--{option}")
```

Finally, the VisIVO command requested to run is completed by adding the input file:

```
1     command.append(input_file)
```

Such a command run either using MPI or serially, if the desired imported module supports, and depending if the variables `mpi` and `tasks` are set or not:

```
1  if mpi:
2      mpi_command = ["mpirun", "-n", str(tasks)]
3                    + command
4      logging.debug(f"Running MPI command:
5                      {mpi_command}")
6      result = subprocess.run(mpi_command, capture_output=
7                              True, text=True)
8  else:
9      logging.debug(f"Running command: {command}")
10     result = subprocess.run(command, capture_output=
11                             True, text=True)
```

We also integrated within the VisIVO wrapper the *Pillow* module to improve image visualization efficiency, especially for PNG files. Viewing images directly from the command line can be time-consuming, often requiring external tools or manual file opening. By incorporating *Pillow*, users can display images seamlessly within the VisIVO Python environment, eliminating the need for additional steps. This enhancement significantly speeds up the workflow and provides the convenience of immediate image rendering within Jupyter notebooks, making graphical output analysis easier.

7.2.2 Deployment

In this section we present the details of the deployment. This work was performed in collaboration with CINECA colleagues, and it is useful to illustrate the steps required to have a fully functional VisIVO environment on the IAC

platform. The deployment was achieved using Ansible consistently with what it was done for ICE4HPC[3] by E4 analytics. The creation of the environment involves four steps:

1. Spack module compilation for VisIVO
2. *conda* environment creation via *Ansible* on the IAC backend nodes
3. installation of the Python wrappers from the GitHub repository
4. customization of *ipykernel*.

The first step consisted of compiling VisIVO and its dependencies using Spack[43]. While this appears straightforward, the IAC environment had some limitations that required several adjustments. In particular, the IAC service does not support a graphical display server (such as X11 or Wayland), because users access the interface exclusively through the web. As a consequence, all rendering performed by VisIVO, especially in the Viewer component, which relies heavily on VTK, must use off-screen rendering.

To satisfy this requirement, we modified the Spack package recipes of VTK, Glew, Mesa, and other graphical libraries to force the OSMesa backend. This implicitly disables any OpenGL implementation that requires a windowing system. Because VTK depends on a large number of optional rendering modules, each one had to be configured manually to ensure full compatibility with the headless pipeline. The resulting Spack module is completely self-contained: it provides a version of VisIVO compiled against a consistent stack of off-screen-capable graphics libraries. This guarantees that any rendering operation invoked from a Jupyter notebook can produce images without requiring an actual display.

The second step consisted in creating a `conda` environment on every backend node that participates in the IAC service. Because Jupyter notebooks may be executed on any of these nodes, the environment must be identical and locally available everywhere. Installing the environment on the cluster's parallel file system proved impractical: conda environments contain thousands of small files, which the parallel file system handles very inefficiently. Our tests showed

[3]ICE4HPC, https://www.e4company.com/ice4hpc/

that loading the environment from the parallel file system resulted in Jupyter start-up delays of several tens of seconds, due to metadata access bottlenecks.

To solve this problem, `Ansible` was used to deploy the conda environment directly onto the local storage of each backend node. Ansible playbooks automate environment creation, package installation, path configuration, and validation. The deployment is performed in parallel using the Ansible controller running on the login node, and the inventory explicitly lists all nodes in the IAC partition. This ensures that every node contains an identical Python environment, independently of the parallel filesystem. The environment includes: the Python standard library (used by the VisIVO wrappers); scientific packages such as NumPy, Pillow, and Matplotlib; supplementary tools required for interactive plotting and in-notebook image display.

Python provides a clear advantage over the traditional VisIVO experience by allowing the users to perform their data analysis combining both the VisIVO tools and the traditional Python tools, and also by allowing to display the generated images on the fly using *IPython* display functionalities inside the Jupyter web interface.

Once the base environment is in place, Ansible retrieves the VisIVO Python wrappers directly from the project's GitHub repository. These wrappers expose the C++ functionality of VisIVO (Importer, Filters, Viewer) as high-level Python functions, making it possible to script visualization workflows directly in the notebook. After installation, the playbooks perform a validation phase that checks:

- whether the Python modules can locate the VisIVO executables installed via Spack;
- whether the OSMesa-enabled Viewer can produce an image when invoked from Python;
- whether the wrappers correctly load, display, and manipulate a simple test dataset.

This step ensures that Python, VisIVO, and the headless rendering backend are correctly integrated before exposing the environment to users.

The final step consists in adapting Jupyter's kernel configuration so that every notebook session automatically loads the correct execution environment. In the VisIVO kernel, we added a Bash prolog that:

- loads the Spack module containing the VisIVO installation;
- exports environment variables required by OSMesa and VTK;
- activates the conda environment configured in the previous step;
- adjusts the Python path so that the VisIVO wrappers are immediately available.

This ensures that users launching a VisIVO notebook do not need to perform any manual setup. The kernel environment is guaranteed to be consistent across all nodes and across all sessions.

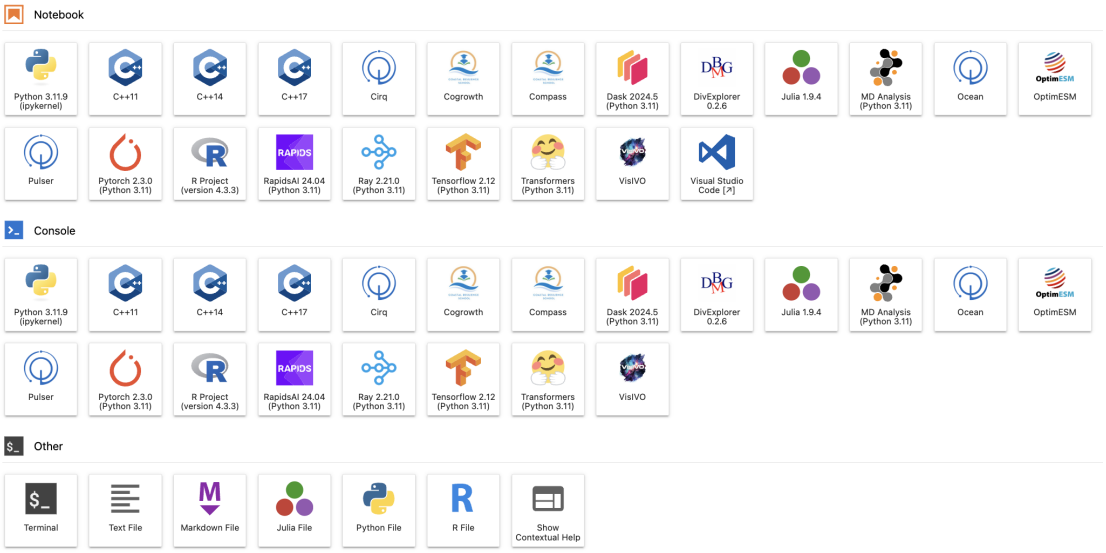

FIGURE 7.1: IAC launcher, showing the available VisIVO notebook.

In figure 7.1 we show the selection of the available environments within the IAC platform and the button for selecting VisIVO for both notebooks or console environment.

7.3 Results

In this section, we demonstrate the successful integration of VisIVO within the InterActive Computing environment through a set of interactive notebooks, each base on the CWL workflows defined in the Results section in Chapter 5. These notebooks will show the correctness of the previously presented VisIVO wrappers by showing the support of all VisIVO functionalities and the improvement on usability provided.

7.3.1 Interactive Notebooks

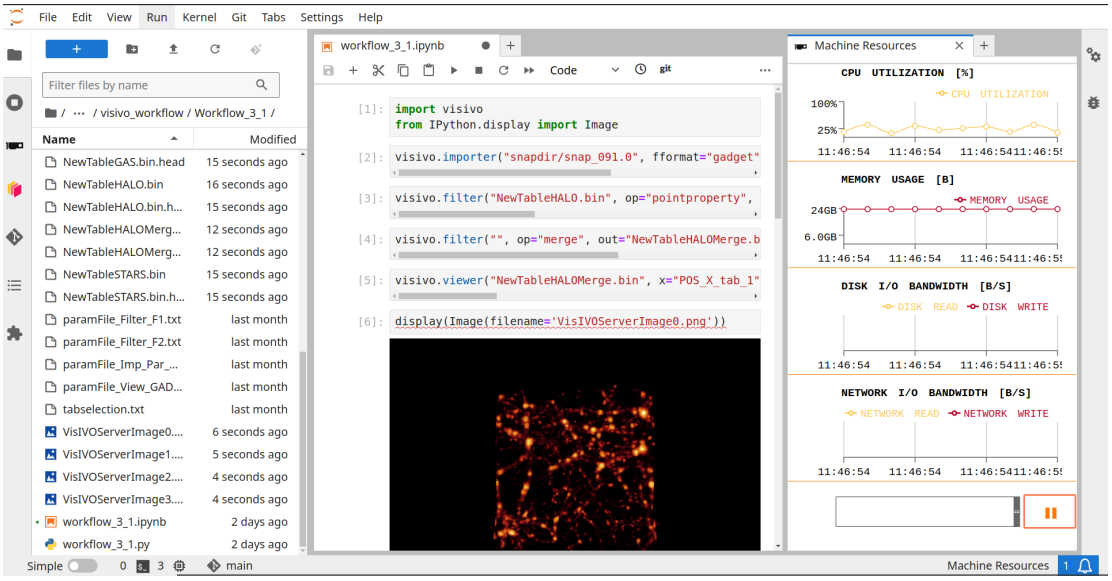

FIGURE 7.2: Jupyter notebook running VisIVO via web browser on a compute node of Galileo 100 cluster

Within the InterActive Computing infrastructure the VisIVO module is available to be launched as a notebook or console environment, in Figure 7.2 we show how the platform presents to the end user.

Notebook Workflow n. 1

This notebook reproduces the operations of **Workflow**$_1$ using the VisIVO module in an interactive environment. This workflow consisted of a simple point

rendering of particles; the same sequence is now executed step-by-step inside a Jupyter notebook.

All the interactive notebooks require the VisIVO wrappers module to be loaded, so the first step of each of the notebook will need to do this and it will also load the iPython package to display images and plots. The first cell of the notebook will have the following commands:

```
import visivo
from IPython.display import Image
```

Then, the second cell uses the importer function in the visivo module, in this case it will pass an ascii file and indicate the fformat value. The ascii importer does not support mpi, so it will also set the mpi option as false:

```
visivo.importer("clusterfields4.ascii", fformat="ascii",
                out="NewTable", mpi=False, tasks=1)
```

The third cell uses the viewer function, passing the expected VBT and all the required options:

```
visivo.viewer("NewTable.bin", x="X", y="Y", z="Z", logscale=True,
              glyphs="pixel", out="img")
```

The last cell consist on simply displaying inside the notebook the image generated by the viewer:

```
display(Image(filename='img0'))
```

Notebook Workflow n.2

This notebook reproduces the operations of **Workflow**$_2$ using the VisIVO module in an interactive environment. The original CWL workflow applied a randomization filter to a GADGET snapshot and performed point rendering on the resulting data subset. In this case we show that the wrapper supports parallel importing via MPI in an interactive mode.

The first cell loads the VisIVO wrapper and IPython display utilities as shown in the previous workflow.

The second cell imports a GADGET snapshot using MPI (4 processes):

```
visivo.importer("snapdir/snap_091.0", fformat="gadget",
                out="NewTable", mpi=True, ntasks=4)
```

This generates a VBT for each particle type in the snapshot (e.g., 'NewTableHALO.bin', 'NewTableGAS.bin'). The third cell applies a randomization filter to the HALO data:

```
visivo.filter("NewTableHALO.bin", op="randomizer",
              perc=50.0, out="NewTableHALORand.bin")
```

The last cell performs point rendering of the randomized VBT and displays the output image:

```
visivo.viewer("NewTableHALORand.bin", x="POS_X", y="POS_Y", z="POS_Z",
              out="img")
display(Image(filename='img0'))
```

Notebook Workflow n.3

This notebook reproduces the operations of **Workflow**$_3$ using the VisIVO module in an interactive environment. The original workflow applied two filters, PointProperty and Merge, to a GADGET snapshot, and then rendered the particle distribution using a scalar color map based on computed densities.

The second cell imports a GADGET snapshot using MPI:

```
visivo.importer("snapdir/snap_091.0", fformat="gadget",
                out="NewTable", mpi=True, ntasks=4)
```

This generates separate VBTs for each particle type. The third cell applies a PointProperty filter to the HALO data, computing a density field:

```
visivo.filter("NewTableHALO.bin", op="pointproperty",
              resolution="512 512 512",
              points="POS_X POS_Y POS_Z",
              out="density.bin", outcol="density")
```

The fourth cell merges the original HALO VBT and the new density VBT into a single table:

```
visivo.filter(op="merge", filelist="tabselection.txt",
              out="NewTableHALOMerge.bin")
```

The final cell visualizes the merged table, using the density field for color mapping:

```
visivo.viewer("NewTableHALOMerge.bin",
              x="POS_X_tab_1", y="POS_Y_tab_1", z="POS_Z_tab_1",
              color=True, colorscalar="density_tab_2",
              colortable="volren_glow", logscale=True, out="img")
display(Image(filename='img0'))
```

Notebook Workflow n.4

This notebook reproduces the operations of **Workflow**$_4$ using the VisIVO module in an interactive environment. The original CWL workflow applied a PointDistribute filter to a GADGET snapshot to compute a 3D density volume, which was then visualized through volume rendering.

The second cell imports a GADGET snapshot using MPI as the previous workflows:

```
visivo.importer("snapdir/snap_091.0", fformat="gadget",
                out="NewTable", mpi=True, ntasks=4)
```

The third cell applies a PointDistribute filter to the HALO data to compute a volume-based density field:

```
visivo.filter("NewTableHALO.bin", op="pointdistribute",
              resolution="256 256 256",
              points="POS_X POS_Y POS_Z",
              out="densityvolume.bin")
```

The final cell performs volume rendering of the generated density volume and displays one of the resulting images:

```
visivo.viewer("densityvolume.bin", volume=True, vrendering=True,
              vrenderingfield="Constant", color=True,
              colortable="volren_glow", showlut=True, out="img")
display(Image(filename='img0'))
```

7.4 Conclusion

This chapter presented the integration of VisIVO within the InterActive Computing (IAC) service to enable interactive, high-performance scientific visualization using a Jupyter-based environment. By developing a set of custom Python wrappers around the core VisIVO command-line tools, we created a modular and user-friendly interface that allows the user to interact with VisIVO for scientific visualization.

The deployment strategy, implemented using Ansible and Spack, ensures consistent software environments across compute nodes and supports off-screen rendering through OSMesa. Through a set of interactive Jupyter notebooks mirroring the CWL workflows introduced earlier in the thesis, we demonstrated that all core VisIVO functionalities can be executed seamlessly in a notebook interface, with support for MPI and image display. This work improves usability for HPC users.

As for future work there is the possibility to add multiple extensions to the already developed wrappers. For example, specific functions to display images with embedded plots might be easily added relying on the wide availability of Python graphical tools.

A second future development could be to exploit the function-based structure and avoid VisIVO local installation by using a REST API built using Flask Library to enable remote execution of VisIVO on a server. The main goal would be to expose the application's functionality through HTTP endpoints and provide an interactive interface that abstracts the complexity of back-end operations and it would be also easy to obtain using Flask decorator applied directly to the already defined functions without the need of refactoring the code.

Chapter 8

Conclusion

8.1 Conclusion and remarks

This thesis proposed multiple solutions to evolve VisIVO in order to exploit heterogeneous HPC environments, following the objectives set in Chapter 1.

First, we identified code mantainability and modularity as an important requirement to be able to add functionalities. We started the work by doing an analysis of the C++ codebase and then focused on refactoring some classes of the Importer, Filter, and Viewer modules. By decomposing monolithic methods, clarifying data structures, and removing dead code, the most critical hot spots saw large reductions in cyclomatic complexity, making the software easier to test, extend, and improving the readability. At the moment this refactoring has been applied on three main classes, one for each of the VisIVO Server modules.

Second, we addressed the limitations imposed by storage. One of the importers was parallelized to exploit distributed filesystems, using a hybrid MPI/OpenMP strategy capable of execution on multi-node and multi-core platforms. The design emphasized file-per-process access and large, positional I/O, which delivered consistent speedups on HPC systems, up to 5.47x on 8 nodes.

In addition, an in-situ approach was implemented by integrating Hecuba in VisIVO to limit storage usage. An in-memory data structure based on the already available VSTable interface bypassing disk I/O; coupled with a new `HecubaSource`, this enabled pipelines using importer and viewer to operate directly on in-memory data. Preliminary tests on similar pipelines with the

Python API showed speedups of 3.3× compared to file-based approaches, indicating the potential performance gains once the C++ API fully supports streaming.

Third, portability and reproducibility were addressed by adapting common VisIVO pipelines as CWL workflows and executing them with StreamFlow as hybrid workflows. Separating workflow logic from execution environments allowed the same pipelines to run on cloud and HPC backends easily by defining them with CWL standard. This approach was tested on the HPC4AI platform, and together with containerization demonstrated improved ease of use, portability and reproducibility of VisIVO's workflows.

Finally, to improve usability of VisIVO on HPC platforms, we integrated VisIVO into Cineca's InterActive Computing platform. The Python wrappers and a dedicated Jupyter kernel lowered the entry barrier for interactive, notebook-driven usage, while still supporting MPI and OpenMP execution, thus combining ease of use with scalable performance.

All of these contributions show a coherent route for the evolution of VisIVO that addresses maintainability, I/O bottlenecks, portability, and usability, making it better suited for modern and future HPC infrastructures.

Future work

Related to chapter 3 it will be needed to perform the same refactoring techniques also on other critical classes that have functions with high cyclomatic complexity such as the VolumePipe class for volume rendering, which is a fundamental function for many VisIVO workflows.

Two of the simpler future works related to chapter 6 are natural and expected. First, completing the in-memory pipeline by adding the support to the in-memory data structure to multiple filters, this would remove the file dependencies in common workflows. Second, testing the pipeline when streaming support in Hecuba's C++ API is added.

Another required future development will be to parallelize other importers similarly as the work done in chapter 4. It will also be required to explore the parallelization for filters and viewer classes, which will need a different

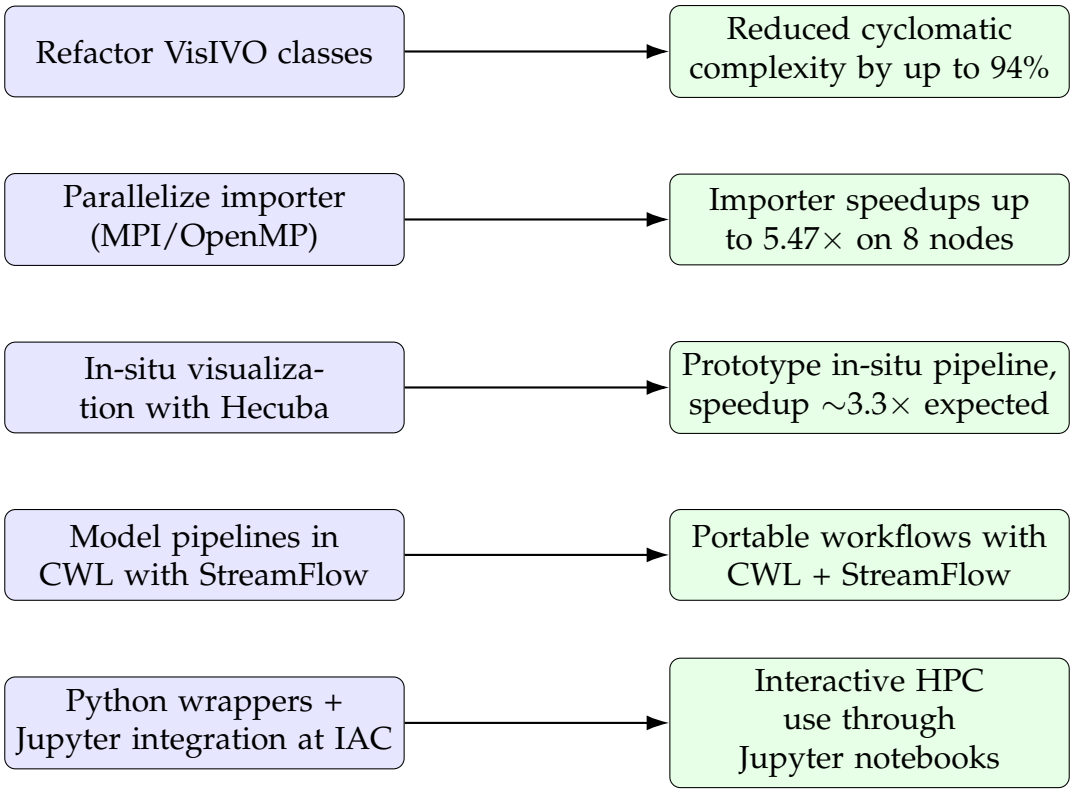

FIGURE 8.1: Thesis objectives (left) and achieved results (right).

approaches and we are currently exploring the possibility to perform the parallelization exploiting GPUs on filters using CUDA or VTK-m[16], a toolkit of scientific visualization algorithms for emerging processor architectures, supporting fine-grained concurrency for data analysis.

Appendix A

AppendixA

This appendix reports extended results obtained from the static analysis of VisIVO Server code before refactoring that we presented in part in chapter 3.

A.1 Inheritance Graphs

The following figures represent the inheritance graph for each VisIVO Server modules: Importer, Filters and Viewer. The inheritance graphs are all similar and are useful to show graphically the modularity of VisIVO's code given by a single abstract class.

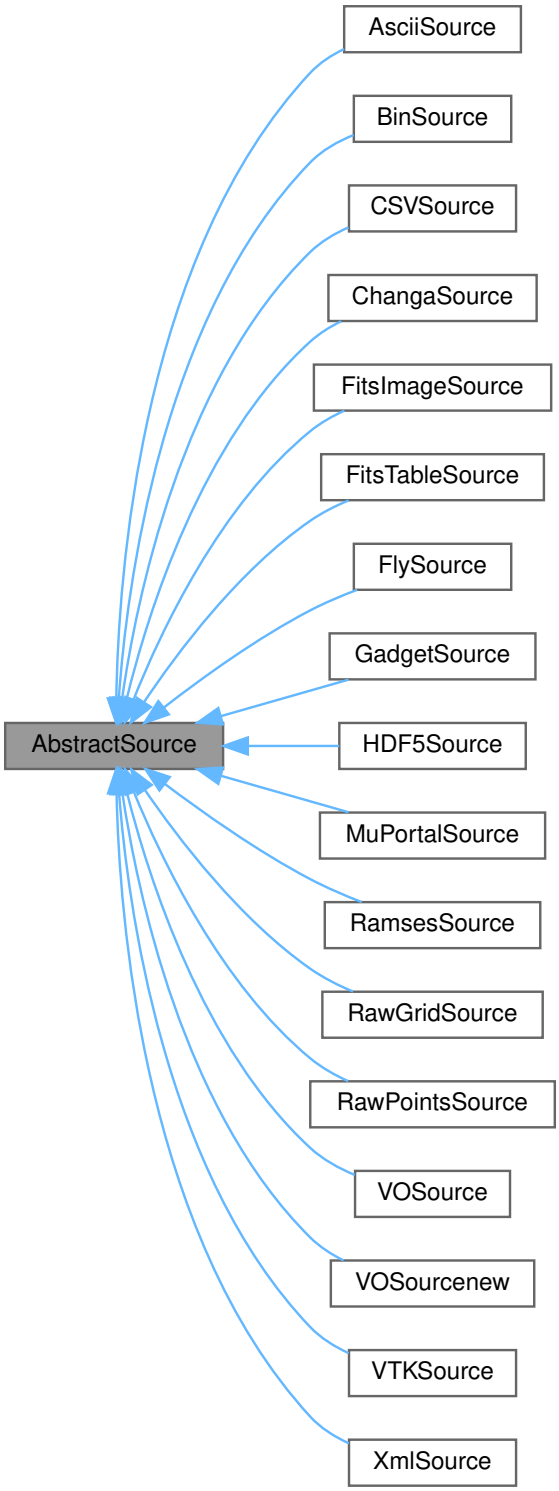

FIGURE A.1: Inheritance Graph Importer Module

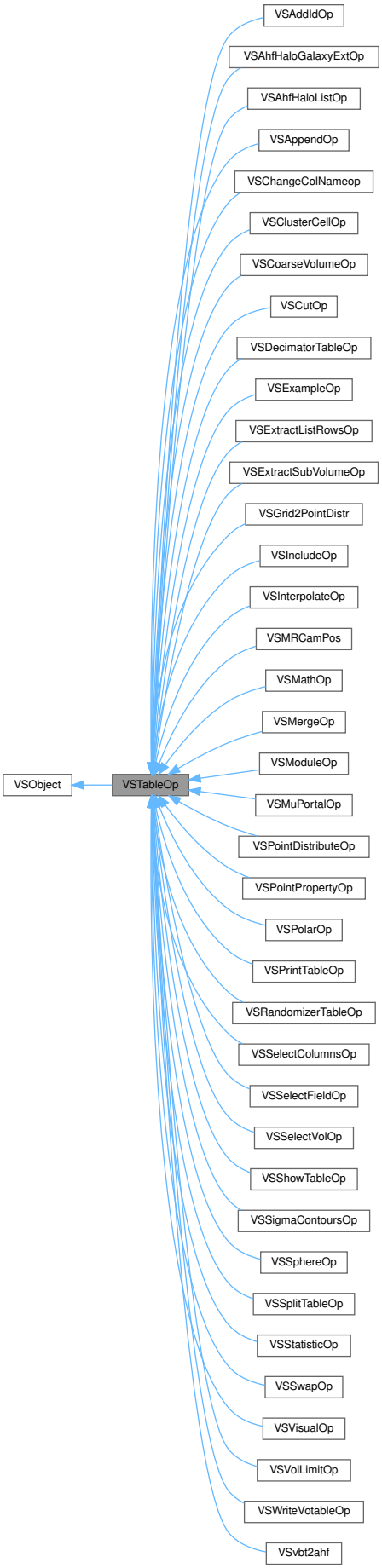

FIGURE A.2: Inheritance Graph Filters Module

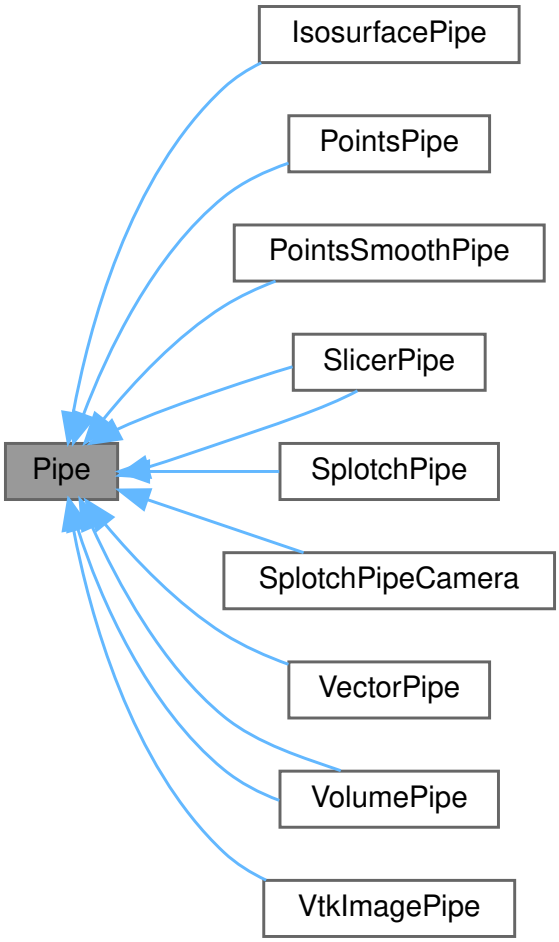

FIGURE A.3: Inheritance Graph Viewer Module

A.2 Procedural Metrics - General

This section presents procedural metrics obtained with CCCC tool of the class for the three VisIVO modules. The metrics considered for this section are LOC, MVG and COM.

A.2.1 Importer

Class Name	**LOC**	**MVG**	**COM**
AbstractSource	107	6	65
AsciiSource	242	60	71
BinSource	203	79	63
CSVSource	163	41	56
ChangaSource	172	39	70
FitsImageSource	177	32	16
FitsTableSource	553	114	77
FlySource	389	95	83
GadgetSource	557	116	88
HDF5Source	394	104	106
HecubaSource	146	15	25
MuPortalSource	213	54	69
RamsesSource	269	49	141
RawGridSource	236	55	64
RawPointsSource	359	76	82
VoSource	233	48	45
VoSourcenew	443	116	46
VTKSource	165	31	12
XmlSource	654	160	101

TABLE A.1: General Procedural Metrics — Importer Classes

A.2.2 Filter

Class Name	LOC	MVG	COM
VSTableOp	108	25	81
VSAddIdOp	145	26	23
VSAhfHaloGalaxyExtOp	261	63	27
VSAhfHaloListOp	197	40	26
VSAppendOp	193	36	69
VSChangeColNameop	69	15	32
VSClusterCellOp	475	102	55
VSCoarseVolumeOp	463	104	55
VSCutOp	437	95	45
VSDecimatorTableOp	303	61	22
VSExampleOp	100	19	68
VSExtractListRowsOp	643	149	33
VSExtractSubVolumeOp	359	88	45
VSGrid2PointDistr	681	214	71
VSIncludeOp	263	59	32
VSInterpolateOp	593	129	41
VSMRCamPos	473	86	64
VSMathOp	347	77	198
VSMergeOp	335	76	78
VSModuleOp	227	53	10
VSMuPortalOp	313	52	150
VSPointDistributeOp	901	267	126
VSPointPropertyOp	447	102	55
VSPolarOp	262	54	17
VSPrintTableOp	24	3	54
VSRandomizerTableOp	311	64	21
VSSelectColumnsOp	180	29	12
VSSelectFieldOp	469	102	47
VSSelectVolOp	438	98	46
VSShowTableOp	240	44	33
VSSigmaContoursOp	375	112	53

Class Name	LOC	MVG	COM
VSSphereOp	430	82	49
VSSplitTableOp	420	96	45
VSStatisticOp	294	121	40
VSSwapOnOp	195	48	33
VSVisualOp	381	86	52
VSVisualLimitOp	349	83	42
VSWriteVotableOp	242	43	49
VSvbt2ahf	279	53	30

TABLE A.2: General Procedural Metrics — Filter Classes

A.2.3 Viewer

Class Name	LOC	MVG	COM
Pipe	816	177	256
IsosurfacePipe	200	36	87
PointsPipe	507	140	160
PointsSmoothPipe	298	55	75
SlicerPipe	258	42	121
SplotchPipe	31	3	27
SplotchPipeCamera	398	69	88
VolumePipe	862	129	321
VolumeSlicer	668	137	85
VtkImagePipe	328	70	78

TABLE A.3: General Procedural Metrics — Viewer Classes

A.3 Procedural Metrics - Specific Classes

This section presents the same metrics considered in the previous one for some of the more critical classes from the VisIVO modules. In this case the metrics are more specific for each of the functions of each class.

A.3.1 GadgetSource Class

Function Name	**LOC**	**MVG**	**COM**
checkMultipleFiles	14	4	0
readData	361	78	71
readHeader	83	23	8
readMultipleHeaders	16	3	1
swapHeaderType1	29	4	3
swapHeaderType2	37	4	4
updateNpart2	1	0	0

TABLE A.4: Procedural Metrics — `GadgetSource` Functions

A.3.2 VSPointDistributeOp Class

Function Name	**LOC**	**MVG**	**COM**
VSPointDistributeOp	54	0	4
allocateArray(int)	192	50	6
allocateArray(int,bool)	2	0	0
computeModelBounds	94	30	20
execute	1152	386	160
executeArray	1	0	0
getOrigin	20	6	4
getSpacing	20	6	4
printHelp	52	2	4
setArray	1	0	0
setOrigin	40	14	2
setSpacing	94	34	2
~VSPointDistributeOp	24	6	4

TABLE A.5: Procedural Metrics — `VSPointDistributeOp` Functions

A.3.3 Pipe Class

Function Name	LOC	MVG	COM
colorBar	45	3	6
constructVTK	21	0	51
createPipe	15	3	6
destroyVTK	30	9	6
getCamera	14	3	6
readData	14	3	6
saveImageAsPng	181	46	26
setAxes	74	3	11
setBoundingBox	96	15	19
setCamera()	1	0	0
setCamera(...)	2	0	0
setCamera(SplotchCamera*)	287	92	119

TABLE A.6: Procedural Metrics — `Pipe` Functions

A.3.4 VolumePipe Class

Function Name	LOC	MVG	COM
VolumePipe	62	0	89
colorBar	15	1	2
createPipe	515	95	186
destroyAll	53	0	11
setLookupTable	127	33	18
~VolumePipe	57	0	11

TABLE A.7: Procedural Metrics — `VolumePipe` Functions

A.4 Object-Oriented Metrics

This section presents object-oriented metrics obtained with CCCC for the class of each of the three VisIVO modules. The specific metrics are the following ones: DIT, NOC, CBO.

A.4.1 Importers

Class Name	**DIT**	**NOC**	**CBO**
AbstractSource	0	18	18
AsciiSource	1	0	1
BinSource	1	0	2
CSVSource	1	0	1
ChangaSource	1	0	4
FitsImageSource	1	0	3
FitsTableSource	1	0	3
FlySource	1	0	3
GadgetSource	1	0	4
HDF5Source	1	0	4
HecubaSource	1	0	3
MuPortalSource	1	0	1
RamsesSource	1	0	4
RawGridSource	1	0	3
RawPointsSource	1	0	3
VoSource	1	0	2
VoSourcenew	1	0	6
VTKSource	1	0	1
XmlSource	1	0	5

TABLE A.8: Object-Oriented Metrics — Importer Classes

A.5 Filters

Class Name	DIT	NOC	CBO
VSTableOp	1	38	44
VSAddIdOp	2	0	1
VSAhfHaloGalaxyExtOp	2	0	3
VSAhfHaloListOp	2	0	3
VSAppendOp	2	0	2
VSChangeColNameop	2	0	1
VSClusterCellOp	2	0	1
VSCoarseVolumeOp	2	0	2
VSCutOp	2	0	2
VSDecimatorTableOp	2	0	2
VSExampleOp	2	0	1
VSExtractListRowsOp	2	0	4
VSExtractSubVolumeOp	2	0	2
VSGrid2PointDistr	2	0	2
VSIncludeOp	2	0	2
VSInterpolateOp	2	0	1
VSMRCamPos	2	0	1
VSMathOp	2	0	3
VSMergeOp	2	0	3
VSModuleOp	2	0	1
VSMuPortalOp	2	0	1
VSPointDistributeOp	2	0	2
VSPointPropertyOp	2	0	1
VSPolarOp	2	0	1
VSPrintTableOp	2	0	1
VSRandomizerTableOp	2	0	2
VSSelectColumnsOp	2	0	1
VSSelectFieldOp	2	0	2
VSSelectVolOp	2	0	2
VSShowTableOp	2	0	1
VSSigmaContoursOp	2	0	2

Class Name	DIT	NOC	CBO
VSSphereOp	2	0	2
VSSplitTableOp	2	0	2
VSStatisticOp	2	0	1
VSSwapOnOp	2	0	1
VSVisualOp	2	0	3
VSVisualLimitOp	2	0	2
VSWriteVotableOp	2	0	3
VSvbt2ahf	2	0	1

TABLE A.9: Object-Oriented Metrics — Filter Classes

A.5.1 Viewers

Class Name	DIT	NOC	CBO
Pipe	0	9	15
IsosurfacePipe	1	0	2
PointsPipe	1	0	12
PointsSmoothPipe	1	0	6
SlicerPipe	1	0	4
SplotchPipe	1	0	1
SplotchPipeCamera	1	0	12
VolumePipe	1	0	12
VolumeSlicer	1	0	10
VtkImagePipe	1	0	12

TABLE A.10: Object-Oriented Metrics — Viewer Classes

Appendix B

AppendixB

This appendix reports the CWL and YML files used to define the workflows we presented in part in chapter 5.

B.1 Workflow 1

B.1.1 Workflow CWL

The listing 8 show the CWL code for Workflow 1 defining the inputs are required to the workflow and their type, the outputs and the folder where they will be stored.

```
cwlVersion: v1.0
class: Workflow

inputs:
  srcImpW0: File
  srcImpW1: File
  srcViewW0: File

outputs:
  outViewW:
    type: File[]
    outputSource: viewer/outView

steps:
  importer:
    run: docker_VisIVOImporter.cwl
    in:
      srcImp0: srcImpW0
      srcImp1: srcImpW1
    out: [outImp0, outImp1]

  viewer:
    run: docker_VisIVOViewer.cwl
    in:
      srcView0: srcViewW0
      srcView1: importer/outImp0
      srcView2: importer/outImp1
    out: [outView]
```

LISTING 8: CWL describing Workflow 1

B.1.2 YML file for inputs definition

The following listing 9 show the definition of all the inputs of Workflow 1.

```
srcImpW0:
  class: File
  path: paramFile_Imp.txt
srcImpW1:
  class: File
  path: clusterfields4.ascii
srcViewW0:
  class: File
  path: paramFile_View.txt
```

LISTING 9: YML file describing the inputs

B.1.3 Workflow Steps CWL

This section shows the CWL for all the steps of Workflow 1, it consists of two steps: Import and Viewer.

B.1.4 VisIVO Importer Step

Listing 7 shows the CWL code for the importer step of Workflow 1.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOImporter
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcImp1)
inputs:
  srcImp0:
    type: File
    inputBinding:
      position: 1
  srcImp1:
    type: File
outputs:
  outImp0:
    type: File
    outputBinding:
      glob: VisIVOServerBinary.bin
  outImp1:
    type: File
    outputBinding:
      glob: VisIVOServerBinary.bin.head
```

LISTING 10: CWL describing Importer Step for Workflow 1

B.1.5 VisIVO Viewer Step

Listing 11 shows the CWL code for the viewer step of Workflow 1.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOViewer
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcView0)
      - $(inputs.srcView1)
      - $(inputs.srcView2)
inputs:
  srcView0:
    type: File
  srcView1:
    type: File
  srcView2:
    type: File
arguments:
  - position: 1
    valueFrom: $(inputs.srcView0.basename)
outputs:
  outView:
    type: File[]
    outputBinding:
      glob: VisIVOServerImage*.png
```

LISTING 11: CWL describing Viewer Step for Workflow 1

B.2 Workflow 2

B.2.1 Workflow CWL

The listing 12 show the CWL code for Workflow 2 defining the inputs are required to the workflow and their type, the outputs and the folder where they will be stored.

```
cwlVersion: v1.0
class: Workflow

inputs:
  messageW: int
  srcImpnpW: int
  srcImpW0: File
  srcImpW1: Directory
  srcFilterW0: File
  srcViewW0: File

outputs:
  outViewW:
    type: File[]
    outputSource: viewer/outView

steps:
  importer:
    run: docker_VisIVOImporter_Par_OpenMP_MPI_1.cwl
    in:
      message: messageW
      srcImpnp: srcImpnpW
      srcImp0: srcImpW0
      srcImp1: srcImpW1
    out: [outImp0, outImp1, outImp2, outImp3]
  filter:
    run: docker_VisIVOFilter.cwl
    in:
      srcFilterF0: srcFilterW0
      srcFilterF1: importer/outImp0
      srcFilterF2: importer/outImp1
    out: [outFilterF0, outFilterF1]
  viewer:
    run: docker_VisIVOViewer_GADGET_Post_F.cwl
    in:
      srcView0: srcViewW0
      srcView1: filter/outFilterF0
      srcView2: filter/outFilterF1
    out: [outView]
```

LISTING 12: CWL describing Workflow 2

B.2.2 YML file for inputs definition

The following listing 13 show the definition of all the inputs of Workflow 2.

```
messageW: 2
srcImpnpW: 4
srcImpW0:
  class: File
  path: paramFile_Imp_Par_MPI.txt
srcImpW1:
  class: Directory
  path: snapdir
srcFilterW0:
  class: File
  path: paramFile_Filter.txt
srcViewW0:
  class: File
  path: paramFile_View_GADGET_Post_F.txt
```

LISTING 13: YML file describing the inputs

B.2.3 Workflow Steps CWL

This section shows the CWL for all the steps of Workflow 2, it consists of three steps: Import, Filter and Viewer.

B.2.4 VisIVO Importer Step

Listings 14 and 15 show the CWL code for the importer step of Workflow 2.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: mpirun
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcImp1)
  EnvVarRequirement:
    envDef:
      OMP_NUM_THREADS: $(inputs.message)

arguments:
  - "--allow-run-as-root"
  - valueFrom: "VisIVOImporter"
    position: 2
```

LISTING 14: CWL describing Importer Step for Workflow 2 (Part 1)

```
inputs:
  message: int
  srcImpnp:
    type: int
    inputBinding:
      position: 1
      prefix: --np
  srcImp0:
    type: File
    inputBinding:
      position: 3
  srcImp1:
    type: Directory
outputs:
  outImp0:
    type: File
    outputBinding:
      glob: NewTableHALO.bin
  outImp1:
    type: File
    outputBinding:
      glob: NewTableHALO.bin.head
  outImp2:
    type: File
    outputBinding:
      glob: NewTableGAS.bin
  outImp3:
    type: File
    outputBinding:
      glob: NewTableGAS.bin.head
```

LISTING 15: CWL describing Importer Step for Workflow 2 (Part 2)

B.2.5 VisIVO Filter Step

Listing 16 shows the CWL code for the filter step of Workflow 2.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOFilter
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcFilterF0)
      - $(inputs.srcFilterF1)
      - $(inputs.srcFilterF2)
inputs:
  srcFilterF0:
    type: File
  srcFilterF1:
    type: File
  srcFilterF2:
    type: File
arguments:
  - position: 1
    valueFrom: $(inputs.srcFilterF0.basename)
outputs:
  outFilterF0:
    type: File
    outputBinding:
      glob: NewTableHALORand.bin
  outFilterF1:
    type: File
    outputBinding:
      glob: NewTableHALORand.bin.head
```

LISTING 16: CWL describing Filter Step for Workflow 2

B.2.6 VisIVO Viewer Step

Listing 17 shows the CWL code for the viewer step of Workflow 2.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOViewer
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcView0)
      - $(inputs.srcView1)
      - $(inputs.srcView2)
inputs:
  srcView0:
    type: File
  srcView1:
    type: File
  srcView2:
    type: File
arguments:
  - position: 1
    valueFrom: $(inputs.srcView0.basename)
outputs:
  outView:
    type: File[]
    outputBinding:
      glob: VisIVOServerImage*.png
```

LISTING 17: CWL describing Viewer Step for Workflow 2

B.3 Workflow 3

B.3.1 Workflow CWL

The listing 18 shows the CWL code for Workflow 3 defining the inputs are required to the workflow and their type, the outputs and the folder where they will be stored.

```
cwlVersion: v1.0
class: Workflow

inputs:
  messageW: int
  srcImpnpW: int
  srcImpW0: File
  srcImpW1: Directory
  srcFilterF1W0: File
  srcFilterF2W0: File
  srcFilterF2W1: File
  srcViewW0: File

outputs:
  outViewW:
    type: File[]
    outputSource: viewer/outView

steps:
  importer:
    run: docker_VisIVOImporter_Par_OpenMP_MPI_1.cwl
    in:
      message: messageW
      srcImpnp: srcImpnpW
      srcImp0: srcImpW0
      srcImp1: srcImpW1
    out: [outImp0, outImp1, outImp2, outImp3]
  filter1:
    run: docker_VisIVOFilter_F1.cwl
    in:
      srcFilterF10: srcFilterF1W0
      srcFilterF11: importer/outImp0
      srcFilterF12: importer/outImp1
    out: [outFilterF10, outFilterF11]
  filter2:
    run: docker_VisIVOFilter_F2.cwl
    in:
      srcFilterF20: srcFilterF2W0
      srcFilterF21: srcFilterF2W1
      srcFilterF22: filter1/outFilterF10
      srcFilterF23: filter1/outFilterF11
      srcFilterF24: importer/outImp0
      srcFilterF25: importer/outImp1
    out: [outFilterF20, outFilterF21]
  viewer:
    run: docker_VisIVOViewer_GADGET_Post_F2.cwl
    in:
      srcView0: srcViewW0
      srcView1: filter2/outFilterF20
      srcView2: filter2/outFilterF21
    out: [outView]
```

LISTING 18: CWL describing Workflow 3

B.3.2 YML file for inputs definition

The following listing 19 show the definition of all the inputs of Workflow 3.

```
messageW: 2
srcImpnpW: 4
srcImpW0:
  class: File
  path: paramFile_Imp_Par_MPI.txt
srcImpW1:
  class: Directory
  path: snapdir
srcFilterF1W0:
  class: File
  path: paramFile_Filter_F1.txt
srcFilterF2W0:
  class: File
  path: paramFile_Filter_F2.txt
srcFilterF2W1:
  class: File
  path: tabselection.txt
srcViewW0:
  class: File
  path: paramFile_View_GADGET_Post_F2.txt
```

LISTING 19: YML file describing the inputs

B.3.3 Workflow Steps CWL

This section shows the CWL for all the steps of Workflow 3, it consists of four steps: Import, two Filters and Viewer.

B.3.4 VisIVO Importer Step

Listings 20 and 21 show the CWL code for the importer step of Workflow 3.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: mpirun
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcImp1)
  EnvVarRequirement:
    envDef:
      OMP_NUM_THREADS: $(inputs.message)

arguments:
  - "--allow-run-as-root"
  - valueFrom: "VisIVOImporter"
    position: 2
```

LISTING 20: CWL describing Importer Step for Workflow 3 (Part 1)

```
inputs:
  message: int
  srcImpnp:
    type: int
    inputBinding:
      position: 1
      prefix: --np
  srcImp0:
    type: File
    inputBinding:
      position: 3
  srcImp1:
    type: Directory
outputs:
  outImp0:
    type: File
    outputBinding:
      glob: NewTableHALO.bin
  outImp1:
    type: File
    outputBinding:
      glob: NewTableHALO.bin.head
  outImp2:
    type: File
    outputBinding:
      glob: NewTableGAS.bin
  outImp3:
    type: File
    outputBinding:
      glob: NewTableGAS.bin.head
```

LISTING 21: CWL describing Importer Step for Workflow 3 (Part 2)

B.3.5 VisIVO Point Distribute Filter Step

Listing 22 shows the CWL code for the Point Distribute filter step of Workflow 3.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOFilter
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcFilterF10)
      - $(inputs.srcFilterF11)
      - $(inputs.srcFilterF12)
inputs:
  srcFilterF10:
    type: File
  srcFilterF11:
    type: File
  srcFilterF12:
    type: File
arguments:
  - position: 1
    valueFrom: $(inputs.srcFilterF10.basename)
outputs:
  outFilterF10:
    type: File
    outputBinding:
      glob: density.bin
  outFilterF11:
    type: File
    outputBinding:
      glob: density.bin.head
```

LISTING 22: CWL describing Point Distribute Filter Step for Workflow 3

B.3.6 VisIVO Merge Filter Step

Listing 23 shows the CWL code for the Merge filter step of Workflow 3.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOFilter
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcFilterF20)
      - $(inputs.srcFilterF21)
      - $(inputs.srcFilterF22)
      - $(inputs.srcFilterF23)
      - $(inputs.srcFilterF24)
      - $(inputs.srcFilterF25)
inputs:
  srcFilterF20:
    type: File
  srcFilterF21:
    type: File
  srcFilterF22:
    type: File
  srcFilterF23:
    type: File
  srcFilterF24:
    type: File
  srcFilterF25:
    type: File
arguments:
  - position: 1
    valueFrom: $(inputs.srcFilterF20.basename)
outputs:
  outFilterF20:
    type: File
    outputBinding:
      glob: NewTableHALOMerge.bin
  outFilterF21:
    type: File
    outputBinding:
      glob: NewTableHALOMerge.bin.head
```

LISTING 23: CWL describing Merge Filter Step for Workflow 3

B.3.7 VisIVO Viewer Step

Listing 24 shows the CWL code for the viewer step of Workflow 3.

```
cwlVersion: v1.0
class: CommandLineTool
baseCommand: VisIVOViewer
requirements:
  DockerRequirement:
    dockerPull: visivolab/visivoserver:latest
  InitialWorkDirRequirement:
    listing:
      - $(inputs.srcView0)
      - $(inputs.srcView1)
      - $(inputs.srcView2)
inputs:
  srcView0:
    type: File
  srcView1:
    type: File
  srcView2:
    type: File
arguments:
  - position: 1
    valueFrom: $(inputs.srcView0.basename)
outputs:
  outView:
    type: File[]
    outputBinding:
      glob: VisIVOServerImage*.png
```

LISTING 24: CWL describing Viewer Step for Workflow 3

Bibliography

[1] Fabio Abreu and Raul Carapuça. "The MOOD metrics suite: measuring object-oriented design quality". In: *Proceedings of OOPSLA* (1994).

[2] Sadaf Alam et al. "Fenix: Distributed e-Infrastructure Services for EBRAINS". In: *Brain-Inspired Computing*. Ed. by Katrin Amunts et al. Cham: Springer International Publishing, 2021, pp. 81–89. ISBN: 978-3-030-82427-3.

[3] Michael Albrecht et al. "Makeflow: A Portable Abstraction for Data Intensive Computing". In: *SWEEET '12: Scalable Workflow Execution Engines and Technologies*. ACM, 2012.

[4] Eman Abdullah AlOmar et al. "Do Design Metrics Capture Developers Perception of Quality? An Empirical Study on Self-Affirmed Refactoring Activities". In: *CoRR* abs/1907.04797 (2019). arXiv: 1907.04797. URL: http://arxiv.org/abs/1907.04797.

[5] V. Antonuccio-Delogu, U. Becciani, and D. Ferro. "FLY. A parallel tree N-body code for cosmological simulations". In: *Computer Physics Communications* 155.2 (Oct. 2003), 159–179. ISSN: 0010-4655. DOI: 10.1016/s0010-4655(03)00345-x. URL: http://dx.doi.org/10.1016/S0010-4655(03)00345-X.

[6] Malcolm Atkinson et al. "Scientific workflows: Past, present and future". In: *Future Generation Computer Systems* 75 (Oct. 2017), pp. 216 – 227. DOI: 10.1016/j.future.2017.05.041. URL: https://hal.science/hal-01544818.

[7] Anonymous / multiple authors. "Documenting Research Software in Engineering Science". In: *Engineering Science (ResearchGate)* (2022). Mentions using Doxygen for code documentation.

[8] Utkarsh Ayachit et al. "ParaView Catalyst: Enabling In Situ Data Analysis and Visualization". In: *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ISAV2015. Austin, TX, USA: Association for Computing Machinery, 2015, 25–29. ISBN: 9781450340038. DOI: 10.1145/2828612.2828624. URL: https://doi.org/10.1145/2828612.2828624.

[9] David Barseghian et al. "Workflows and Extensions to the Kepler Scientific Workflow System to Support Environmental Sensor Data Access and Analysis". In: *Ecological Informatics* 5.1 (2010), pp. 42–50.

[10] Michael Bauer et al. "Legion: Expressing locality and independence with logical regions". In: *SC '12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2012.

[11] U. Becciani. "Visivo–integrated tools and services for large-scale astrophysical visualization". In: *Publications of the Astronomical Society of the Pacific* (2010).

[12] U. Becciani et al. "VisIVO: A Library and Integrated Tools for Large Astrophysical Dataset Exploration". In: *Astronomical Data Analysis Software and Systems XXI*. Ed. by P. Ballester, D. Egret, and N. P. F. Lorente. Vol. 461. Astronomical Society of the Pacific Conference Series. Sept. 2012, p. 505.

[13] G. Bruce Berriman and John C. Good. "Montage: An Astronomical Image Mosaicking Toolkit". In: *Publications of the Astronomical Society of the Pacific* 118.850 (2006), pp. 1428–1437.

[14] Jean Luca Bez, Suren Byna, and Shadi Ibrahim. "I/O Access Patterns in HPC Applications: A 360-Degree Survey". In: *ACM Computing Surveys* (July 2023). DOI: 10.1145/3611007. URL: https://hal.science/hal-04173899.

[15] Daniel Blankenberg et al. "Galaxy: A Web-based Genome Analysis Tool for Experimentalists". In: *Current Protocols in Molecular Biology* (2010), pp. 19.10.1–19.10.21.

[16] Mark Bolstad et al. "VTK-m: Visualization for the Exascale Era and Beyond". In: *ACM SIGGRAPH 2023 Talks*. SIGGRAPH '23. Los Angeles, CA, USA: Association for Computing Machinery, 2023. ISBN: 9798400701436. DOI: 10.1145/3587421.3595466. URL: https://doi.org/10.1145/3587421.3595466.

[17] Lionel Briand and et al. "Empirical studies of object-oriented metrics". In: *Software Engineering Journal* (1999).

[18] Peter Bui et al. "Work Queue+Python: A Framework for Scalable Scientific Ensemble Applications". In: *Python for High Performance Computing Workshop at SC11*. 2011.

[19] Rajkumar Buyya, Toni Cortes, and Hai Jin. *Overview of the MPIIO Parallel I/O Interface*. 2002, pp. 476–487. DOI: 10.1109/9780470544839.ch32.

[20] Alexandru Calotoiu and et al. "Performance-aware software architecture analysis for HPC applications". In: *Concurrency and Computation: Practice and Experience* (2014).

[21] Neil Campbell and et al. "Cognitive complexity metrics for software maintainability". In: *Journal of Software: Evolution and Process* (2018).

[22] Carmelita Carbone, Margarita Petkova, and Klaus Dolag. "DEMNUni: ISW, Rees-Sciama, and weak-lensing in the presence of massive neutrinos". In: *Journal of Cosmology and Astroparticle Physics* 2016.7, 034 (July 2016), p. 034. DOI: 10.1088/1475-7516/2016/07/034. arXiv: 1605.02024 [astro-ph.CO].

[23] Jeffrey C. Carver et al. "Self-Perceptions about Software Engineering: A Survey of Scientists and Engineers". In: *Computing in Science Engineering* 15.1 (2013), pp. 7–11. DOI: 10.1109/MCSE.2013.12.

[24] Oscar Chaparro et al. "On the Impact of Refactoring Operations on Code Quality Metrics". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 456–460. DOI: 10.1109/ICSME.2014.73.

[25] Shyam Chidamber and Chris Kemerer. "A metrics suite for object oriented design". In: *IEEE Transactions on Software Engineering* (1994).

[26] I. Colonnelli and M. Aldinucci. "Hybrid Workflows For Large - Scale Scientific Applications". In: 2022.1 (2022), pp. 1–5. ISSN: 2214-4609. DOI: https://doi.org/10.3997/2214-4609.2022615029. URL: https://www.earthdoc.org/content/papers/10.3997/2214-4609.2022615029.

[27] Iacopo Colonnelli et al. "HPC Application Cloudification: The StreamFlow Toolkit". In: *12th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 10th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2021)*. Ed. by João Bispo, Stefano Cherubin, and José Flich. Vol. 88. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Jan. 1, 2021, 5:1–5:13. DOI: 10.4230/OASIcs.PARMA-DITAM.2021.5.

[28] Iacopo Colonnelli et al. "StreamFlow: Cross-Breeding Cloud With HPC". In: *IEEE Transactions on Emerging Topics in Computing* 9.4 (2021), pp. 1723–1737. DOI: 10.1109/TETC.2020.3019202.

[29] Iacopo Colonnelli et al. "StreamFlow: Cross-Breeding Cloud With HPC". In: *IEEE Transactions on Emerging Topics in Computing* 9.4 (Oct. 2021), 1723–1737. ISSN: 2376-4562. DOI: 10.1109/tetc.2020.3019202. URL: http://dx.doi.org/10.1109/TETC.2020.3019202.

[30] Intel Corporation. *Intel Threading Building Blocks (TBB) Documentation*. Available online. 2024. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html.

[31] Michael R. Crusoe et al. "Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language". In: *arXiv preprint arXiv:2105.07028* (2021). URL: https://arxiv.org/abs/2105.07028.

[32] CSC – IT Center for Science. *Lustre User Guide*. https://docs.csc.fi/computing/lustre/. Accessed November 2025. 2024.

[33] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[34] Ewa Deelman et al. "Pegasus: Mapping Scientific Workflows onto the Grid". In: *Across Grids Conference*. 2004.

[35] Paolo Di Tommaso et al. "Nextflow Enables Reproducible Computational Workflows". In: *Nature Biotechnology* 35 (2017), pp. 316–319.

[36] Philippe Dubois and et al. "Software engineering practices in scientific computing: survey and implications". In: *Future Generation Computer Systems* (2008).

[37] H Carter Edwards and et al. "Kokkos: Enabling performance portability across manycore architectures". In: *Proceedings of the Extreme Scaling Workshop* (2014).

[38] Thomas Fahringer, Radu Prodan, et al. "ASKALON: A Development and Grid Computing Environment for Scientific Workflows". In: *Workflows for e-Science*. Springer, 2007, pp. 450–471.

[39] Norman Fenton and Shari Pfleeger. "Software Metrics: A Rigorous and Practical Approach". In: (1998).

[40] Rosa Filgueira et al. "dispel4py: An Agile Framework for Data-Intensive eScience". In: *2015 IEEE 11th International Conference on e-Science*. 2015, pp. 454–464.

[41] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 2nd. Addison-Wesley Professional, 2018. ISBN: 978-0134757599. URL: https://martinfowler.com/books/refactoring.html.

[42] Luan Teylo Francieli Boito Guillaume Pallez. "The role of storage target allocation in applications' I/O performance with BeeGFS". In: *CLUSTER 2022 - IEEE International Conference on Cluster Computing, Heidelberg, Germany.* (2022).

[43] Todd Gamblin et al. "The Spack package manager: bringing order to HPC software chaos". In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2015, pp. 1–12.

[44] Tristan Glatard et al. "Flexible and Efficient Workflow Deployment of Data-Intensive Applications on Grids with MOTEUR". In: *International Journal of High Performance Computing Applications* 22.3 (2008), pp. 347–360.

[45] William F. Godoy and et al. "ADIOS 2: The next generation of the Adaptable IO System". In: *SoftwareX* (2020).

[46] Mark Govett et al. "Exascale Computing and Data Handling: Challenges and the Need for Co-Design". In: *Bulletin of the American Meteorological Society* 105.12 (2024), E1235–E1254. DOI: 10.1175/BAMS-D-23-0226.1. URL: https://repository.library.noaa.gov/view/noaa/69266.

[47] Miguel Grinberg. *Flask web development*. " O'Reilly Media, Inc.", 2018.

[48] {William D.} Gropp et al. "Self-consistent MPI-IO performance requirements and expectations". English (US). In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface - 15th European PVM/MPI Users' Group Meeting, Proceedings*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 15th European PVM/MPI Users' Group Meeting, EuroPVM/MPI 2008 ; Conference date: 07-09-2008 Through 10-09-2008. 2008, pp. 167–176. ISBN: 3540874747. DOI: 10.1007/978-3-540-87475-1_25.

[49] Maurice Halstead. *Elements of Software Science*. Elsevier, 1977.

[50] Robert J. Hanisch et al. "Definition of the Flexible Image Transport System (FITS)". In: *Astronomy and Astrophysics* 376 (2001), pp. 359–380. URL: https://api.semanticscholar.org/CorpusID:12441421.

[51] Jo et al. Hannay. "How do scientists develop and use scientific software?" In: *ICSE* (2009).

[52] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability - a preliminary report". In: *QUATIC 2007 - 6th International Conference on the Quality of Information and Communications Technology* (Jan. 2007). DOI: 10.1109/QUATIC.2007.7.

[53] Saskia Heldens. "The Landscape of Exascale Research: A Data-Driven Perspective". In: *Communications of the ACM* 63.11 (2020), pp. 46–54. DOI: 10.1145/3372390.

[54] Beatriz Hernández-Molinero et al. "Cosmic Background Neutrinos Deflected by Gravity: DEMNUni Simulation Analysis". In: *arXiv e-prints*, arXiv:2301.12430 (Jan. 2023), arXiv:2301.12430. DOI: 10.48550/arXiv.2301.12430. arXiv: 2301.12430 [astro-ph.CO].

[55] Michael Hilton and et al. "How modern software development teams use continuous integration". In: *Empirical Software Engineering* (2016).

[56] Lorin Hochstein and et al. "Parallel legacy code modernization". In: *IEEE Software* (2005).

[57] IEEE and The Open Group. *POSIX Threads Programming*. IEEE Std 1003.1-2017 (POSIX.1-2017). 2017. URL: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[58] Carlos Izurieta and et al. "Using historical metrics to detect architectural drift". In: *Journal of Software Maintenance* (2009).

[59] Anubhav Jain et al. "FireWorks: A Dynamic Workflow System Designed for High-throughput Applications". In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 5037–5059.

[60] B. Jansen and et al. "Software quality metrics for high-performance computing systems". In: *Concurrency and Computation: Practice and Experience* (2019).

[61] Morris A Jette and Tim Wickberg. "Architecture of the slurm workload manager". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2023, pp. 3–23.

[62] Yi Ju et al. "Understanding the Impact of Synchronous, Asynchronous, and Hybrid In-Situ Techniques in Computational Fluid Dynamics Applications". In: *2022 IEEE 18th International Conference on e-Science (e-Science)*. IEEE, Oct. 2022, 295–305. DOI: 10.1109/escience55777.2022.00043. URL: http://dx.doi.org/10.1109/escience55777.2022.00043.

[63] S. H. Kannangara and W. M. J. I. Wijayanayake. "An Empirical Evaluation of Impact of Refactoring On Internal and External Measures of Code Quality". In: *CoRR* abs/1502.03526 (2015). arXiv: 1502.03526. URL: http://arxiv.org/abs/1502.03526.

[64] Rick Kazman and et al. "SAAM: A method for analyzing the properties of software architectures". In: *ICSE* (1994).

[65] Barbara Kitchenham. "Empirical software engineering and measurement". In: *Empirical Software Engineering* (2010).

[66] Martin Kleppmann and Jay Kreps. "Kafka, Samza and the Unix Philosophy of Distributed Data". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2015).

[67] Thomas Kluyver et al. *Jupyter notebooks*. 2016.

[68] *Kafka : a Distributed Messaging System for Log Processing*. 2011. URL: https://api.semanticscholar.org/CorpusID:18534081.

[69] A. Lakshman and P. Malik. "Cassandra — A Decentralized Structured Storage System". In: *Operating Systems Review.* 44 (2010), pp. 35–40.

[70] Joseph Lazio. *The Square Kilometre Array*. 2009. arXiv: 0910 . 0632 [astro-ph.IM]. URL: https://arxiv.org/abs/0910.0632.

[71] *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*. Berlin, Heidelberg: Springer-Verlag, 1996. ISBN: 3540653066.

[72] Y. Li, F. Wang, S. Zhou, et al. "Parallel I/O Characterization and Optimization on Large-Scale Systems". In: *arXiv preprint* (2024). arXiv: 2501.00203 [cs.DC]. URL: https://arxiv.org/abs/2501.00203.

[73] Sun Lihua and Wu Yunzhi. "Visualization Toolkit and its application". In: *2010 International Conference on Mechanic Automation and Control Engineering*. 2010, pp. 5476–5478. DOI: 10.1109/MACE.2010.5535547.

[74] Jay Lofstead, Scott Klasky, et al. "Flexible I/O and Integration for Scientific Codes Through the Adaptable IO System (ADIOS)". In: *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*. 2008, pp. 15–24.

[75] Maurizio Lorenz and Jeffrey Kidd. "Object-oriented metrics in practice". In: (1994).

[76] Lustre Wiki. *Lustre Metadata Service (MDS)*. https://wiki.lustre.org/Lustre_Metadata_Service_(MDS). Accessed November 2025. 2024.

[77] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable realtime data systems*. 1st. USA: Manning Publications Co., 2015. ISBN: 1617290343.

[78] Lars Mattsson and et al. "Software engineering challenges in scientific computing". In: *Software Quality Journal* (2010).

[79] T.J. McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[80] Steve McConnell. *Code Complete, Second Edition*. USA: Microsoft Press, 2004. ISBN: 0735619670.

[81] Harshitha Menon et al. *Adaptive Techniques for Clustered N-Body Cosmological Simulations*. 2014. arXiv: 1409.1929 [astro-ph.IM]. URL: https://arxiv.org/abs/1409.1929.

[82] Tom Mens and Tom Tourwé. "A survey of software refactoring". In: *IEEE Transactions on Software Engineering* 30.2 (2004), pp. 126–139.

[83] Michael B. Milligan. "Jupyter as Common Technology Platform for Interactive HPC Services". In: *Proceedings of the Practice and Experience on Advanced Research Computing*. PEARC '18. ACM, July 2018, 1–6. DOI: 10.1145/3219104.3219162. URL: http://dx.doi.org/10.1145/3219104.3219162.

[84] Michael B. Milligan. "Jupyter as Common Technology Platform for Interactive HPC Services". In: *Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity*. PEARC '18. Pittsburgh, PA, USA: Association for Computing Machinery, 2018. ISBN: 9781450364461. DOI: 10.1145/3219104.3219162. URL: https://doi.org/10.1145/3219104.3219162.

[85] Michael B. Milligan. "Jupyter as Common Technology Platform for Interactive HPC Services". In: *CoRR* abs/1807.09929 (2018). arXiv: 1807.09929. URL: http://arxiv.org/abs/1807.09929.

[86] Yan Mo and et al. "Evaluating architectural decay in scientific software". In: *Journal of Systems and Software* (2015).

[87] Rick Mohr and Paul Peltz. "Benchmarking SSD-Based Lustre File System Configurations". In: *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. XSEDE '14. Atlanta, GA, USA: Association for Computing Machinery, 2014. ISBN: 9781450328937. DOI: 10.1145/2616498.2616544. URL: https://doi.org/10.1145/2616498.2616544.

[88] Kenneth Moreland et al. "Examples of in transit visualization". In: *Proceedings of the 2nd International Workshop on Petascal Data Analytics: Challenges and Opportunities*. PDAC '11. Seattle, Washington, USA: Association for Computing Machinery, 2011, 1–6. ISBN: 9781450311304. DOI: 10.1145/2110205.2110207. URL: https://doi.org/10.1145/2110205.2110207.

[89] Philipp Moritz and et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *OSDI*. 2018.

[90] Jingqing Mu et al. "Interfacing HDF5 with a scalable object-centric storage system on hierarchical storage". In: *Concurrency and Computation. Practice and Experience* 32.20 (Mar. 2020). ISSN: ISSN 1532-0626. DOI: 10.1002/cpe.5715. URL: https://www.osti.gov/biblio/1603709.

[91] N-Body Shop. *TIPSY: Code for Display and Analysis of N-body Simulations*. Astrophysics Source Code Library, record ascl:1111.015. Nov. 2011. ascl: 1111.015.

[92] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, 2017. URL: https://www.confluent.io/resources/ebook/kafka-the-definitive-guide/.

[93] Paul Oman and Jack Hagemeister. "Metrics for assessing a software system's maintainability". In: *Software Maintenance* (1992).

[94] Anonymous (IJERT paper). "Implementing Source-Code Metrics for Software Quality Analysis". In: *International Journal of Engineering Research Technology* 1.5 (2012). Uses CCCC tool for LOC, complexity, CK metrics, ...

[95] G. Parimbelli et al. "DEMNUni: comparing nonlinear power spectra prescriptions in the presence of massive neutrinos and dynamical dark energy". In: *Journal of Cosmology and Astroparticle Physics* 2022.11, 041 (Nov. 2022), p. 041. DOI: 10.1088/1475-7516/2022/11/041. arXiv: 2207.13677 [astro-ph.CO].

[96] Dewayne Perry and Alexander Wolf. "Foundations for the study of software architecture". In: *ACM SIGSOFT Software Engineering Notes* (1992).

[97] Planck Collaboration, P. A. R. Ade, et al. "Planck 2013 results. XVI. Cosmological parameters". In: *Astronomy Astrophysics* 571, A16 (Nov. 2014), A16. DOI: 10.1051/0004-6361/201321591. arXiv: 1303.5076 [astro-ph.CO].

[98] Edgar Gabriel Raafat Feki. "Design and Evaluation of Multi-threaded Optimizations for Individual MPI I/O Operations". In: *30th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (2022).

[99] D. Radjenovic and et al. "Software metrics for scientific software: an empirical study". In: *Empirical Software Engineering* (2013).

[100] *Refactoring: improving the design of existing code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672.

[101] Marzia Rivi et al. *In-situ Visualization: State-of-the-art and Some Use Cases*. Jan. 2012.

[102] Matthew Rocklin. "Dask: Parallel computation with blocked algorithms and task scheduling". In: *Proceedings of the Python in Science Conference* (2015).

[103] Igor Rosenberg. "Metrics for software quality and maintainability". In: *Journal of Software Maintenance* (1997).

[104] Christopher Sadowski and et al. "Continuous static analysis in large-scale software development". In: *Communications of the ACM* (2018).

[105] Sebastian Lührs Sandra Mendez. *Best Practice Guide - Parallel I/O*. Partnership for Advanced Computing in Europe, 2019.

[106] Berk Geveci Sebastien Jourdain Utkarsh Ayachit. *PARAVIEWWEB, A WEB FRAMEWORK FOR 3D VISUALIZATION AND DATA PROCESSING*. 2010.

[107] Muhammad Rabee SHAHEEN. "Survey of source code metrics for evaluating testability of object oriented systems". In: *ACM Trans. Comput. Log., vol. 20, pp. 1–18, 2014.* (2014).

[108] Rafael Ferreira da Silva et al. "A Characterization of Workflow Management Systems for Extreme-Scale Applications". In: *Future Generation Computer Systems* 75 (2017), pp. 228–238.

[109] V. Springel. "The cosmological simulation code GADGET-2". In: *Monthly Notices of the Royal Astronomical Society* (2005).

[110] Volker Springel. "The cosmological simulation code GADGET-2". In: *Monthly Notices of the Royal Astronomical Society* 364.4 (Dec. 2005), pp. 1105–1134. DOI: 10.1111/j.1365-2966.2005.09655.x. arXiv: astro-ph/0505010 [astro-ph].

[111] Hanul Sung et al. "Understanding Parallel I/O Performance Trends Under Various HPC Configurations". In: *SNTA '19: Proceedings of the ACM Workshop on Systems and Network Telemetry and Analytics* (2019).

[112] Ian Taylor et al. "The Triana Workflow Environment: Architecture and Applications". In: *Workflows for e-Science*. Springer, 2007, pp. 320–339.

[113] R. Teyssier. "Cosmological hydrodynamics with adaptive mesh refinement: A new high resolution code called RAMSES". In: *Astronomy amp; Astrophysics* 385.1 (Apr. 2002), 337–364. ISSN: 1432-0746. DOI: 10.1051/0004-6361:20011817. URL: http://dx.doi.org/10.1051/0004-6361:20011817.

[114] Rajeev Thakur, William Gropp, and Ewing L. Lusk. "Optimizing Noncontiguous Accesses in MPI-IO". In: *CoRR* cs.DC/0310029 (2003). URL: http://arxiv.org/abs/cs/0310029.

[115] University of Trieste. *Lecture: Storage for HPC Infrastructure*. https://moodle2.units.it/pluginfile.php/714202/mod_resource/content/1/lecture-03-storage-for-HPC-infrastructure.pdf. Accessed November 2025. 2024.

[116] Peter Vaillancourt et al. "Reproducible and Portable Workflows for Scientific Computing and HPC in the Cloud". In: *Practice and Experience in Advanced Research Computing (PEARC '20)*. PEARC. 2020. URL: https://doi.org/10.1145/3311790.3396659.

[117] VAST Data. *Parallel vs Distributed File Systems for HPC Storage*. https://www.vastdata.com/blog/parallel-vs-distributed-file-systems-for-hpc. Accessed November 2025. 2023.

[118] Giovanni Verza et al. "The universal multiplicity function: counting halos and voids". In: *arXiv e-prints*, arXiv:2401.14451 (Jan. 2024), arXiv:2401.14451. DOI: 10.48550/arXiv.2401.14451. arXiv: 2401.14451 [astro-ph.CO].

[119] Matteo Viel, Martin G. Haehnelt, and Volker Springel. "The effect of neutrinos on the matter distribution as probed by the intergalactic medium". In: *Journal of Cosmology and Astroparticle Physics* 2010.6, 015 (June 2010), p. 015. DOI: 10.1088/1475-7516/2010/06/015. arXiv: 1003.2422 [astro-ph.CO].

[120] Guozhang Wang et al. "Building a Replicated Logging System with Apache Kafka". In: *Proceedings of the VLDB Endowment*. Vol. 8. 12. Discusses Kafka's log, replication, partitioning architecture. 2015, [page numbers].

[121] WFMC. *Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011)*. Tech. rep. Workflow Management Coalition, Brussels, 1999.

[122] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. "Parallel in situ coupling of simulation with a fully featured visualization system". In: *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV '11. Llandudno, UK: Eurographics Association, 2011, 101–109. ISBN: 9783905674323.

[123] Wikipedia contributors. *Lustre (file system) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Lustre_(file_system). Accessed November 2025. 2025.

[124] Michael Wilde, Mariam Hategan, et al. "Swift: A Language for Distributed Parallel Scripting". In: *Parallel Computing* 37.9 (2011), pp. 633–652.

[125] Greg Wilson and et al. "Best practices for scientific computing". In: *PLoS Biology* (2014).

[126] Katherine Wolstencroft et al. "The Taverna Workflow Suite: Designing and Executing Workflows of Web Services". In: *Nucleic Acids Research* 41 (2013), W557–W561.

[127] Toshihiro Yamashita and Leon Moonen. "Exploring the impact of code smells on software maintainability". In: *Empirical Software Engineering* (2013).

[128] Jia Yu and Rajkumar Buyya. "A Taxonomy of Workflow Management Systems for Grid Computing". In: *CoRR* abs/cs/0503025 (2005). arXiv: cs/0503025. URL: http://arxiv.org/abs/cs/0503025.

[129] U. Zarić et al. "Software Metrics and Multi-lingual Code". In: *CEUR Workshop Proceedings*. Vol. 3845. Includes CppDepend among the analyzed tools. 2024, ...

[130] Ying Zhou and Henry Leung. "An empirical study of software metrics and their impact on defect-proneness". In: *Information and Software Technology* (2006).