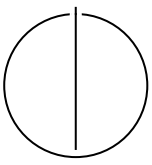
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Benchmarking of parallel I/O libraries for a Hyperbolic PDE Engine

Dilay Nurlu



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

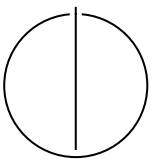
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Benchmarking of parallel I/O libraries for a Hyperbolic PDE Engine

Benchmarking von parallelen I/O-Bibliotheken für eine hyperbolische PDE-Engine

Author:	Dilay Nurlu
Examiner:	Michael Bader
Supervisor:	Marc Marot-Lassauzaie
Submission Date:	01.03.2025



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 01.03.2025 Dilay Nurlu

Acknowledgments

Thanks to my supervisor Marc, who motivated me at the right time and made great effort to support me by answering my questions. And thanks to my family, who had to hear all about HDF5.

Abstract

Peano4 is a framework for dynamically generating adaptive grids, defining the underlying mesh and its traversal. It serves as a foundation for various scientific applications. One extension of Peano4 is the ExaHyPE2 engine, which solves hyperbolic Partial Differential Equations (PDE)s for wave simulations. Currently, the output is stored in a custom file format that requires conversion for visualization. This project replaces the custom format with Hierarchical Data Format 5 (HDF5) storage and an accompanying eXtensible Data Model and Format (XDMF) file for direct visualization. The new format is evaluated based on runtime performance and file size efficiency compared to the current approach of ExaHyPE2.

Contents

1 Introduction

Natural phenomena such as tsunamis and earthquakes are a part of life on our planet. The consequences of these natural disasters can be physically and emotionally devastating for those affected. This is why it is valuable to develop advanced scientific applications that can predict such events or simulate them to understand their possible causes and outcomes.

Numerical methods are widely used for the simulation of such dynamic processes. The Peano4 framework provides dynamically generated adaptive grids as a foundation. Built on Peano4, the ExaHyPE2 engine specializes in solving PDEs for wave propagation simulations, with tsunamis being just one example.

This work aims to further improve ExaHyPE2 by introducing a new output design that utilizes the HDF5 and XDMF formats.

Peano4 and ExaHyPE2 source code is available on the GitLab repository of the Leibniz Supercomputing Centre Munich:

`https://gitlab.lrz.de/hpcsoftware/Peano`

The source code resulting from this work can be found as a fork on GitLab too:

`https://gitlab.lrz.de/00000000014AFD5A/Peano/-/tree/dilay_thesis`

2 Context and Related Work

This chapter provides an overview of the codebase used in this thesis project. It also introduces the necessary tools required for visualization and storage.

2.1 Codebase of Peano

2.1.1 Peano4

Peano, currently in its 4th version, is mainly written in C++ and has some tools in Python. This is further explained in Section 2.1.3. It is a base for multiple extensions, one being ExaHyPE2 with hyperbolic solvers. Swift 2 and MGHyPE are other extensions that are irrelevant to this work. Without an extension, Peano is not a standalone program but a code-building tool. [Pea25c] It uses space trees to derive dynamically adaptive Cartesian mesh grids and defines their traversal.[Wei19, p. 1] These processes occur hidden behind Peano, and users primarily interact with the extensions.

2.1.2 ExaHyPE2

ExaHyPE2 is a generic "Exascale Hyperbolic PDE" engine distributed as a Peano 4 extension. The engine aims to solve user-specified PDEs of first-order. Some specialized solvers also support second-order spatial operators. As the solution method, it implements "high-order ADER discontinuous Galerkin (ADER-DG) with a-posteriori subcell limiting and finite volume (FV) schemes".[Rei+20, p. 2]

From the user perspective, the engine provides a simple API to implement scenarios without dealing with the complexities of designing a high-order parallel solver, as ExaHyPE2 handles this part. The user only needs to implement the problem-specific fluxes, source terms, and quantities, ensuring their suitability for mesh refinement. The application areas range from magnetohydrodynamics to waves in elastic media, shallow water equations, and many more. [Rei+20, p.5]

2.1.3 Python API and C++

Peano 4 is written in plain C++. The architecture is organized into layers, providing interfaces for logging, plotters, observers, writers, and more. Some of these, such as

observers, must be implemented based on the chosen solver. This is where the Python API is useful.

Every application scenario needs to define an ExaHyPE2 project, which is then lowered into a Peano project and native C++ code. The project definition is done with a scenario-specific Python setup script. This is where the user chooses which solver should be used and sets the simulation parameters.

Running this top-level Python script automatically generates C++ Peano code specific to the solver settings. This is done with Python classes, which define the previously mentioned interfaces such as observers or writers. These classes are filled with C++ code templates. The values that were chosen in the top-level Python script are plugged into the templates. Plain C++ files are generated from them. With the resulting Makefile, the compilation begins, and the executable is ready. All the user needs to do is execute the top-level Python script. [Pea25a]

The Python API also provides postprocessing scripts to convert files into Visualization Toolkit (VTK) format.

2.2 Visualization Software

2.2.1 ParaView and Visualization Toolkit

ParaView is an open-source tool for scientific data analysis and visualization of large datasets. It can process and render raw data parallel to support 2D and 3D displays. Depending on the file format, the user can choose one of the provided readers to bring the data into the system. The XDMF Reader is relevant for this work. Filters can be used to extract information or customize the rendering. The visualization process can also be automated by using ParaView's batch processing offers. [Par20] The version used in this work was 5.9.0.

The Visualization Toolkit (VTK) is the foundational library upon which ParaView is built. VTK is also an open-source system developed by the same firm as ParaView, Kitware Inc. Since VTK provides a modular pipeline architecture to ParaView and is the core library, the native format is VTK-based file formats such as `.vtu`. These are the most compatible and supported file types and do not require external readers or converters to be visualized with ParaView. VTK also supports numerous other data formats, with XDMF and HDF5 being the relevant ones for this work.[VTK23a]

2.2.2 eXtensible Data Model and Format

XDMF, currently on version 3, is a standardized file format designed to exchange scientific data between High-Performance Computing codes. It serves as a metadata

layer that references datasets stored in separate binary files, making data interpretation easier for other tools. For example, it can be used to visualize HDF5 datasets on ParaView.

XDMF can be produced with native text output like C++ streams or by using the provided API's for multiple languages. Filename extensions are `.xmf` and `.xdmf`. In the later sections of this thesis, the "XMF File" refers to an XDMF file with the extension `.xmf`. [14]

XDMF differentiates between the raw data and its description. They are separated into light and heavy data, which are different in nature and stored using separate mechanisms. Light data, which describes heavy data, is smaller in size and stored using Extensible Markup Language (XML). Heavy data are the values that the data is representing. Therefore, it can be bigger, usually stored with HDF5. While HDF5 excels in the structured storage of datasets, it lacks description mechanisms for the relationships between them. However, this kind of description is required by visualization tools to interpret the domain. This is why XML is used as light data to describe the heavy HDF5 data. The separation is critical to the performance. [17]

The base structure of an XDMF file is a `Domain`. `Domains` can contain multiple `Grids`. The type can be defined by setting the `GridType` attribute. A relevant type for this work is `Collection`, which is an array of `Uniform` grids containing the same `Attributes`. When the `CollectionType` is set as `Temporal`, this can be used to describe a timestep of the simulation. `Grids` can be nested, meaning a `Temporal Grid` can contain multiple other `Grids` of `Uniform` type. This structure can correspond to a timestep displaying four subdomains, which is the base idea for the implementation in Section 3.3.2. Each `Uniform Grid` then needs to define its `Topology`, `Geometry` and `Attribute` fields. `Topology` and `Geometry` describe the data by specifying grid connectivity and the location of the nodes. The heavy HDF5 data is listed by the file name in the `Attribute` field. The values from HDF5 will be displayed within the defined grid. We can also point to a specific dataset inside the `.h5` file. [22]

2.3 Storage Technologies

2.3.1 American Standard Code for Information Interchange

American Standard Code for Information Interchange (ASCII) is a 7-bit character encoding standard published by the United States of America Standards Institute in 1968. It represents text as numerical values, assigning each character a unique 7-bit binary code. [19a] Peano's simulation output data is currently stored in ASCII-based text patch files, described in Section 3.1.1. While this format is human-readable, it is inefficient for storing large numerical datasets compared to binary formats such as

HDF5.

2.3.2 Hierarchical Data Format

Hierarchical Data Format (HDF), version 5, is a data management and storage model. Unlike plain binary or text-based formats, it extends the concepts of a hierarchical file system to a single file structure. An HDF5 file is a container that organizes multiple objects, enabling efficient retrieval of large data.

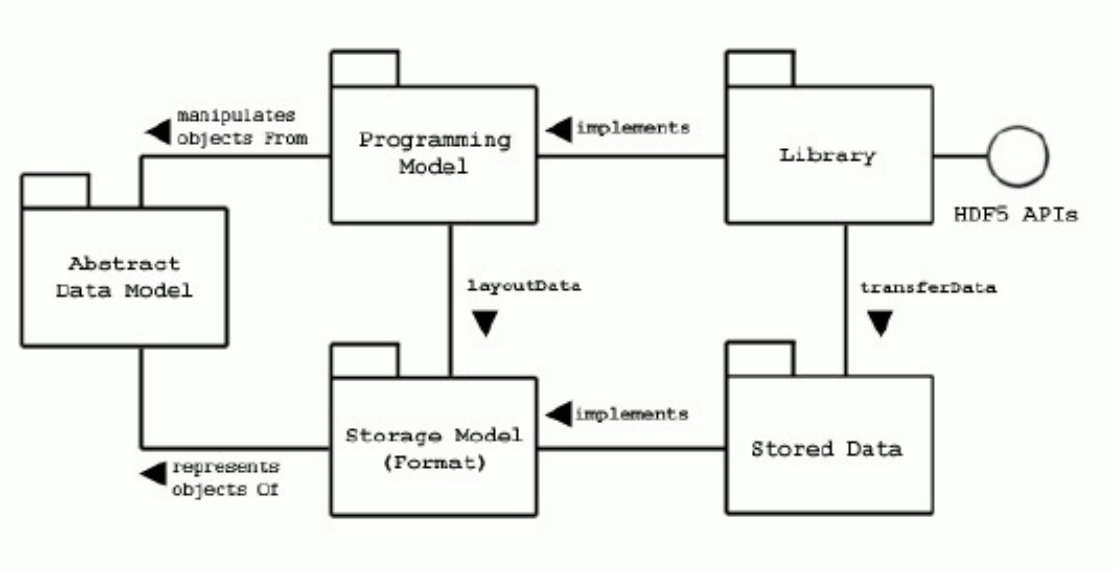


Figure 2.1: HDF5 Models and Implementations (from [The25a]).

HDF5 defines a couple of models whose interactions can be seen in Figure 2.1. The two conceptual models are the Abstract Data Model and the Storage Model. The Abstract Data Model is a conceptual model of data organization. It defines the hierarchical structures. One of the key components called `Group`, can be seen as similar to a file system directory. It can contain multiple objects whose `Group` membership is implemented by a `Link`. The root group `"/"` is special and not a member of any group. There are three classes of objects: `Dataset`, `Datatype` and `Group`. A `Dataset` is an array of data elements that can be multidimensional. The shape of a `Dataset` is defined by a `Dataspace` object. The data elements to be stored in the `Dataset` are defined by `Datatypes`. There are 11 classes of data types and multiple pre-defined types. [The25c] When a `Dataset` is created, its dataspace and datatype are set and cannot be modified afterward.

The Storage Data Model defines how data is physically structured at the byte level and stored on disk. Two key aspects, Chunked Storage and Compression, will be discussed in Section 4.3.1. Stored Data, seen in Figure 4.3, is the concrete implementation of the storage model.

The Programming Model defines operations for creating, reading, and writing of HDF5 files. The Library implements this model and provides an HDF5 API as its interface. The HDF5 core library is written in C, but it offers language bindings for other languages. This work utilizes the C++ API, specifically the `H5Cpp.h` header to implement the writer. HDF5 version is 1.10.4.

Additionally, `h5dump` is a useful command-line utility provided by HDF5. It displays the contents of an HDF5 file in human-readable format. It offers a variety of options that allow inspection in specified ways. [The25d]

3 Implementation

3.1 Previous Peano Output Design and Implementation

This section describes ExaHyPE2's custom output format called Peano Patch File. We begin by introducing the relevant sections of the codebase responsible for generating these output files. Next, the required conversion step for visualization is explained. Finally, after outlining the goals of the new design, we present the description and implementation of the new output format using HDF5 and XDMF.

3.1.1 Patch File Structure

Patch Files are text files in ASCII format that define their own structure to describe grid information and store output values. The two types of patch files are discussed in the following sections. They both start with the same header specifying the version and format.

Data Patch Files

A data patch file represents a single subdomain of the computational domain at a specific timestamp. Dividing the domain into subdomains may enable parallel file generation. How the Index Patch File brings together the individual data patch files is explained in Section 3.1.1. An example snippet from a Data Patch File can be seen in Listing 3.1.

The `dimensions` field is appended to the header in every patch file. This value remains consistent throughout all patch files. A single block of metadata follows the header. The metadata structure can vary depending on the patch file. The three fields `number-of-unknowns`, `number-of-dofs-per-axis`, and `description` are always present in the metadata. The `mapping` array can also be stored, depending on the scenario. `number-of-unknowns` specifies how many variables the simulation calculates values for, while `description` defines what those variables represent. For example, they could be water height and velocity. `number-of-dofs-per-axis` specifies the number of cells or vertices per axis in a patch (see below). The optional `mapping` array consists of $(\texttt{number-of-dofs-per-axis}^{\texttt{dimensions}}) \times \texttt{dimensions}$ entries. Groups of two consecutive

entries (in 2D) or three consecutive entries (in 3D) form a tuple or triple representing values for each axis, ordered lexicographically as (x,y,z). The first tuple (or triple) is assigned to the patch's first `number-of-dof`. The mapping array is used in the conversion step in Section 3.1.3 to compute the coordinates of the subdomain. This is done using the offset and size of the patches. Since the mapping array applies to every patch in the file, it is stored in the metadata.

After the metadata, multiple regions of blocks defining "patches" follow. Each patch data file represents a subdomain. The patch blocks inside the data file can be seen as a further granulation that makes up the subdomain. Each patch block includes offset and size values for each axis. As mentioned above, these values are used to calculate coordinate points by applying the mapping, if applicable. A `begin patch` block can store two different types of values for the unknowns: values for grid cells in `begin-cell-values` and values for vertices of the grid in `begin-vertex-values`. The size of this values array is $(\texttt{number-of-dofs-per-axis}^{\texttt{dimensions}}) \times \texttt{number-of-unknowns}$. The data is stored in a one-dimensional layout. All unknowns are stored consecutively for the first cell or vertex. Then, we move to the second cell or vertex and store all its unknowns.

```
#
# Peano patch file
# Version 0.2
#
format ASCII
dimensions 2

begin vertex-metadata "SWERadialDamBreakQ_old"
  number-of-unknowns 4
  number-of-dofs-per-axis 4
  description
  mapping  6.943184e-02 6.943184e-02 3.300095e-01 6.943184e-02 6.699905e-01 ...
end vertex-metadata

begin patch
  offset 4.629630e-01 -4.629630e-01
  size 3.703704e-02 3.703704e-02
  begin vertex-values "SWERadialDamBreakQ_old"
     2.00000e+00 0 0 0 2.00000e+00 0 0 0 2.00000e+00 0 0 0 ...
  end vertex-values
end patch
```

Listing 3.1: Example Peano Patch File

Index Patch File

The Index Patch File is a metafile that links to actual data by organizing multiple datasets. Each dataset corresponds to a single timestamp and includes multiple data patch files as subdomains that collectively describe the whole computational domain at that moment.

As shown in Listing 3.2, for each timestep, four `.peano-patch-file` files from the `solutions` folder are referenced in the index file. Since four data files are included, the computational domain consists of four subdomains. As the simulation progresses to the next timestep, a new set of data patch files will be included to reflect the updated simulation values. These data patch files are uniquely identified by numbers preceding the `.peano-patch-file` extension, ensuring proper association within the dataset. The Index Patch File shown in Listing 3.2 is named `solution-SWERadialDamBreak.peano-patch-file`, also located in the `solutions` folder. Unlike the individual data patch files, the index file does not contain a number in its name, distinguishing it from the data patch files.

```
#
# Peano patch file
# Version 0.2
#
format ASCII

begin dataset
  timestamp  0.00000000000000000000000e+00

  include "solutions/solution-SWERadialDamBreak-0.peano-patch-file"
  include "solutions/solution-SWERadialDamBreak-1.peano-patch-file"
  include "solutions/solution-SWERadialDamBreak-2.peano-patch-file"
  include "solutions/solution-SWERadialDamBreak-3.peano-patch-file"
end dataset

begin dataset
  timestamp  5.01911059143469936993 9310e-02

  include "solutions/solution-SWERadialDamBreak-4.peano-patch-file"
  include "solutions/solution-SWERadialDamBreak-5.peano-patch-file"
  include "solutions/solution-SWERadialDamBreak-6.peano-patch-file"
  include "solutions/solution-SWERadialDamBreak-7.peano-patch-file"
end dataset
```

Listing 3.2: Example Snippet from a Peano Index File

3.1.2 Generation With Writers

This section describes the parts of the codebase responsible for generating the Patch Files. Two writers collaborate to produce the output files: the Text Patch File Writer and the Cell (or Vertex) Data Writer. The Plotter, generated by the Python API, manages the creation and execution of these writers.

Text Patch File Writer

The `PeanoTextPatchFileWriter` is derived from `PatchWriter` and provides specialized implementations for handling both cell-based and vertex-based data. It is located under the namespace `tarch::plotter::griddata::blockstructured`. A separate Text Patch File Writer is instantiated for each subdomain per timestep, generating its corresponding Patch Data File. Each writer maintains an internal string stream to buffer the data incrementally. The Text Patch File Writer performs patch plotting with its `plotPatch()` function by writing the offset and size to the buffer. Once all patches and their values have been buffered in the string stream, the data is flushed to the Patch Data File. The plotting of cell or vertex values within each patch is handled by the `CellDataWriter` or `VertexDataWriter` classes, explained in the next chapter.

Additionally, all `PeanoTextPatchFileWriter` instances modify the shared Index Patch File to include the Patch Data File they generate. There are three modes to manage the index file:

- **NoIndexFile** ignores the index file management. Every writer just dumps data to its Patch Data File.
- **CreateNew** creates an empty index file with every writer instance and adds the first patch data file into the dataset. It throws away any old index file that existed with the given file name.
- **AppendNewData** creates a new index file if no file with the given name exists. If a file with that name exists, it checks the last timestamp that was written. If the last timestamp is greater than the current timestamp being written, it creates a backup of the current index file and generates a new one. If the last timestamp is smaller than the current timestamp, it begins a new dataset and includes the patch data file. If the last and current timestamps are equal, it only adds the patch data file to the existing dataset.

Cell and Vertex Data Writer

The other writers defined in `PeanoTextPatchFileWriter`, as mentioned in the previous section, are the `CellDataWriter` and `VertexDataWriter` (which will be summarized as

Data Writer for the rest of the chapter). Both Data Writers provide similar functionality and are derived from the inner abstract classes inside `PatchWriter`.

One Data Writer instance is associated to a Text Patch File Writer and can access its internal string stream to buffer data values for each patch in the subdomain. Information about metadata, the number of unknowns, and the number of Degrees of Freedom (DOF)s per axis (3.1.1) is passed to the Data Writer during its creation.

The Data Writer's `plotCell()` function is called for every single DOF (cell or vertex, explained in 3.1.1) inside the patch. Essentially, the values for every unknown for the current DOF will be buffered in the Data Writer's internal string stream. For each patch, the `plotCell()` function will buffer $(\texttt{number-of-dofs-per-axis}^{\texttt{dimensions}}) \times \texttt{number-of-unknowns}$ amount of values. Once all DOFs in the patch have been processed, the Data Writer's internal stream is flushed into the Text Patch File Writer's stream, and the Data Writer has a clean stream to process the next patch's DOFs.

Python API for Text Patch File Writer

The Python class that defines a plotter using the Text Patch File Writer is implemented in `PlotPatchesInPeanoBlockFormat.py`. It contains function templates that accept variables such as the file name and generate the plotter as a C++ file using these values. In the generated file, one Text Patch File Writer and one Data Writer are instantiated per subdomain. To manage access to the shared index file, lock structures are used.

3.1.3 Visualization

The Patch File output format described in Section 3.1.1 is not compliant to any standard machine-readable format. Therefore, the generated data files need to be converted to a standardized file format which can be recognized by visualization tools. Peano provides a Python script that converts patch files to VTK and ParaView Data (PVD) files to visualize on ParaView.

ParaView Data Format

In practice, a PVD file is very similar to Peano's Index Patch File. It is an XML-based file that lists multiple datasets corresponding to different timestep values, pointing to VTK files that store the actual data. The single PVD file is loaded into ParaView, allowing the simulation to be visualized by stepping through the timesteps and retrieving data from the corresponding VTK files. An example can be seen in Listing 3.3.

VTK Data Format

The Visualization Toolkit (VTK), described in Section 2.2.1, provides standardized file formats for storing and writing scientific data. The three different styles of file formats are Legacy, XML and VTKHDF. Peano converts patch data files to the XML based VTK format with the extension `.vtu`, which stands for serial unstructured data [VTK23b]. Not every patch file is directly converted into a single VTU file. While each patch file defines a subdomain, a VTU file describes the entire computational domain.

VTU files are structured in an XML hierarchy containing three main components. `<Points>` store the spatial coordinates of grid points. `<Cells>` define the grid topology, including connectivity and cell types. `<CellData>` contains simulation variables mapped to the computational cells. All numerical data, including coordinates, connectivity, and simulation variables, is stored in <DataArray> elements using inline binary encoding for efficiency.

Compared to Peano patch files, VTU files are less human-readable but offer better performance and compatibility with visualization tools such as ParaView. An example VTU file snippet is shown in Listing 3.4.

```
<?xml version="1.0"?>
<VTKFile type="Collection" version="0.1"
         byte_order="LittleEndian"
         compressor="vtkZLibDataCompressor">
  <Collection>

    <DataSet timestep="0.0" group="" part="0"
        file="solution-Euler.EulerQ-0.vtu" />

    <DataSet timestep="0.013843647798131657" group="" part="0"
        file="solution-Euler.EulerQ-1.vtu" />

    <DataSet timestep="0.023064728510227304" group="" part="0"
        file="solution-Euler.EulerQ-2.vtu" />

  </Collection>
</VTKFile>
```

Listing 3.3: Example PVD File Snippet

```
<?xml version="1.0"?>
<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">
  <UnstructuredGrid>
    <Piece NumberOfPoints="23409" NumberOfCells="20736">
      <CellData Scalars="Unknowns">
        <DataArray type="Float64" Name="Unknowns" NumberOfComponents="5" format
    ="binary">
          GgAAAACAAAAAKAAAYyYAAGI5...
          <!-- Binary data following -->
        </DataArray>
      </CellData>
      <Points>
        <DataArray type="Float32" Name="Points" NumberOfComponents="3" format="
    binary">
          CQAAAACAAABMSQAA+hwAAMYc...
          <!-- Binary data following -->
        </DataArray>
      </Points>
      <Cells>
        <DataArray type="Int64" Name="connectivity" format="binary">
          FQAAAACAAAAAIAAAoxAAAPwPA...
          <!-- Binary data following -->
        </DataArray>
        <DataArray type="UInt8" Name="types" format="binary">
          AQAAAACAAAAAUQAALAAAAA==
        </DataArray>
      </Cells>
    </Piece>
  </UnstructuredGrid>
</VTKFile>
```

Listing 3.4: Example VTU File Snippet

Conversion

The conversion of Peano Patch Files into VTU files begins by invoking the `render.py` script provided by Peano. This script is executed with the Peano Index File specified in the command, where the file contains a list of timestamps and their associated patch data file names (as mentioned in Section 3.1.1). The script counts the number of snapshots by iterating over all the timestamps in the index file and then instantiates a visualizer object. This object is instantiated from the `VTUUnstructuredGrid.py` module, which extends the generic `Visualiser.py` class.

The visualizer first calls the superclass's `display()` method to read the index file one dataset at a time. For each dataset, a snapshot instance is created that encapsulates the patch data files for that timestep. This design results in one VTU file per timestep, rather than one per patch file. The VTU file represents the entire computational domain at a given timestep, whereas each patch file corresponds to a single subdomain.

The visualizer processes the patch files in the snapshots by creating a `PatchFileParser` instance for each patch. This parser object extracts metadata from the patch file, such as the mapping array and the number of unknowns (as discussed in Section 3.1.1). If no specific mapping is provided, the parser calculates a default mapping to ensure a uniformly distributed patch grid. After parsing the metadata, the actual patch regions containing the unknown values are processed and stored in the helper class `UnknownAttributes`. Any filters specified via `render.py`'s command-line options are applied at this stage. For example, the `-norm-calculator` filter computes the norm at each point, and the `-filter-average` filter averages over cell data.

The parsed and aggregated data is then passed to the `VTUUnstructuredGrid` visualizer object, which converts it into a VTK unstructured grid. VTK points, cells, and data arrays corresponding to the patch geometry and unknowns are created. Finally, a VTU file is generated using VTK's XML writer. At the end of the process, a PVD file referencing the VTU files across snapshots is written.

3.2 Requirements For New Design

Peano's patch file output format lacks standardization and is not directly compatible with other software. Maintaining this custom format requires an ongoing effort from developers. Additionally, users unfamiliar with Peano may find it difficult to understand, as it does not follow a common standard.

Without a conversion step, Peano's output format cannot be visualized. The conversion process adds overhead by invoking a separate Python script outside the simulation executable. Additionally, users must learn how to use this script. Any issues that arise at the conversion stage would not originate from the simulation itself, making debugging more complex. Instead of only inspecting the solver code, users would also need to verify the conversion script to not produce errors.

To mitigate these challenges, the new data output format is designed with the following goals in mind:

- The conversion step should be eliminated. The simulation should output the data files and the index file in a standardized format that can be directly recognized by visualization software.

- The new output design should ensure that the results are displayed correctly when visualized.
- The speed and file size of the new format should be equal to or better than the current format.
- The data and index files should be generated in parallel to support efficient execution.
- The new implementation must use Peano's existing interfaces and introduce minimal external dependencies.

The patch data files will be replaced by HDF5 files. Peano already supports HDF5, requiring the `-lhdf5` and `-lhdf5_cpp` libraries. The `<H5Cpp.h>` header, which is used for generating HDF5 files, will then be available. To eliminate the conversion step, the index file will be replaced with an incrementally written XDMF file, which is used to load HDF5 files into ParaView for visualization. No additional processing is needed. Since this work outputs the XDMF file using plain C++ streams, no library linking is required.

3.3 New Output Format Design

3.3.1 HDF5 Files Structure

Peano's original approach of saving data for one subdomain in a single patch data file has been adopted for the HDF5 format. An example HDF5 file, which saves data for one subdomain, can be seen in Listing 3.5. It contains one group called `Metadata` and multiple individual groups for each patch in the subdomain, indexed as `Patch_i`. To keep the HDF5 files as minimal as possible, only the most relevant information for the visualization is stored. The only data types used are the HDF5 C++ API's `PredType::NATIVE_DOUBLE` and `PredType::NATIVE_INT`. Since the simulation code directly provides data as C++ `doubles`—such as the mapping array, unknown values, offset, and size— `NATIVE_DOUBLE` is used. On most modern platforms, this type maps to the 64-bit floating point `H5T_IEEE_F64LE` format. The reason behind not choosing `NATIVE_FLOAT` is discussed in Section 5.1. For other values such as the number of DOFs per axis and the number of unknowns, the `NATIVE_INT` type is chosen, which maps to `H5T_STD_I32LE`.

The `Metadata` group stores any information that applies to each patch in the file. These values are `dofs_per_axis` and `number_of_unknowns`, which are stored as scalars in a dataspace for one element. If a mapping array is provided, it is also stored within `Metadata` as a one-dimensional dataspace.

Each patch is stored in a separate group named `Patch_i`. The `offset` and `size` of each patch are stored in a one-dimensional dataspace of length 2 or 3, depending on whether it is a 2D or 3D scenario. Unlike the original patch data files, all unknowns are not stored in a single array. In the HDF5 file structure, each unknown is stored separately in a one-dimensional dataspace, indexed as `unknown_i`. Storing them separately is the easiest and most direct way to reference the unknowns from the XMF file, which is explained in Section 3.3.2. Each `unknown_i` has a length of `number-of-dofs-per-axis`$^{\texttt{dimensions}}$.

3.3.2 XMF File Structure

The XMF file's purpose is to reference the correct HDF5 files at the appropriate time values while describing the domain. On the highest level, the file consists of a `Temporal Collection` grid for each timestep. The `Collection` grid type represents an array of `Uniform` grids, all sharing the same `Attributes`. The timestep is specified with the `<Time Value = "0.0">` tag.

Under each timestep grid, the four subdomains are separately defined as a `Collection` grid and contain a series of `Uniform` grids defining the patches. A `Uniform` grid is a homogeneous single grid. Essentially, we reference all patch data from one HDF5 file under the subdomain grid.

Each `Uniform` patch grid must contain a `Topology` and `Geometry` element to define the mesh structure. `Topology` element describes the general organization of the data and how data points are connected to form the grid. `Geometry` element defines the spatial positions of points and specifies the coordinate system. Depending on whether the scenario includes a mapping array, there are two versions of XMF files, which mainly differ in `Topology` and `Geometry` elements.

For scenarios without a mapping array, the subdomain is assumed to be equidistant. In this case, the `TopologyType` is set to `2DCoRectMesh` since this type is used when the axes are perpendicular and spacing is constant. The `2DCoRectMesh` type is compatible with the `ORIGIN_DXDY GeometryType`. The points do not need to be explicitly listed, only the offset and spacing needs to be provided. The patch offsets for each axis are specified in a `DataItem` field of the `Geometry`. Note that XDMF expects values in the `DataItem` fields in Y-X order, meaning the Y-axis offset comes first, followed by the X-axis offset. Spacing is computed using the patch size and the number of DOFs per axis and is provided in a `DataItem`, similar to the offset. An example of this file type can be seen in Listing 3.7.

For scenarios with a mapping array, the subdomain is no longer assumed to be equidistant. This requires the `TopologyType 2DSMesh`, where the point values need to be provided to XDMF explicitly. Since these point values are only required for the XMF file, saving them in the HDF5 files and referencing them would be redundant. Instead,

they are written inline to the `Geometry` field, which works with `GeometryType="X_Y"`. The point coordinates are computed by applying the mapping array to the patch offset, scaled with the patch size. An example of this file type can be seen in Listing 3.6.

In both scenario cases, the `Dimensions` of the `Topology` and `Geometry` fields are determined based on the number of DOFs per axis. For 3D scenarios, the `TopologyType`, `GeometryType` and their `Dimensions` are correctly adjusted, for example by using `3DSMesh` and `"X_Y_Z"`.

Lastly, the `Attribute` elements of a patch grid define values associated with the mesh. These are the values that will be displayed for the visualization. There is one `Attribute` element per unknown. Inside the `Attribute` elements, there is a `DataItem` field which sets the `Dimensions` based on the number of DOFs per axis and the dimensions of the scenario (2D or 3D). The actual values are referenced from the HDF5 datasets using the corresponding patch and unknown index. Since referencing each unknown separately is the most direct way to extract the values on XDMF, they are stored in separate HDF5 datasets. Note that the XMF file needs to be in the same location as the HDF5 files as relative paths are used. Depending on whether a mapping array is provided, the `Center` type of the `Attribute` element is defined accordingly. When mapping occurs, the simulation provides vertex-based values, which requires `Node` type. Otherwise, the type is `Cell`.

Throughout the XMF file, `NumberType="Float"` is used with `Precision="8"` to reference `double` values. A precision value of 8 corresponds to 8 bytes (64 bits), which is the correct precision for the stored `double` values.

```
HDF5 "solution-SWERadialDamBreak-0.h5" {
GROUP "/" {
   GROUP "Metadata" {
      DATASET "dofs_per_axis" {
         DATATYPE  H5T_STD_I32LE
         DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
      }
      DATASET "mapping" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 32 ) / ( 32 ) }
      }
      DATASET "number_of_unknowns" {
         DATATYPE  H5T_STD_I32LE
         DATASPACE  SIMPLE { ( 1 ) / ( 1 ) }
      }
   }
   GROUP "Patch_0" {
      DATASET "offset" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 2 ) / ( 2 ) }
      }
      DATASET "size" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 2 ) / ( 2 ) }
      }
      DATASET "unknown_0" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 16 ) / ( 16 ) }
      }
      DATASET "unknown_1" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 16 ) / ( 16 ) }
      }
      DATASET "unknown_2" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 16 ) / ( 16 ) }
      }
      DATASET "unknown_3" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 16 ) / ( 16 ) }
      }
   }
  }
 }
```

Listing 3.5: Example h5dump Output Scenario With Mapping

```
<?xml version="1.0"?>
<Xdmf Version="3.0">
  <Domain>
    <Grid Name="TimeStep_0" GridType="Collection" CollectionType="Temporal">
      <Time Value="0.000000000000000000000000e+00"/>
      <Grid Name="subdomain" GridType="Collection">
        <Grid Name="Patch_0" GridType="Uniform">
          <Topology TopologyType="2DSMesh" Dimensions="4 4"/>
          <Geometry GeometryType="X_Y">
            <DataItem Dimensions="4 4" NumberType="Float" Precision="8" Format="XML">
 4.655345127483e-01 4.751855362299e-01 4.877774267331e-01 4.974284502147e-01 (...)
            </DataItem>
            <DataItem Dimensions="4 4" NumberType="Float" Precision="8" Format="XML">
 -4.603914131777e-01 -4.603914131777e-01 -4.603914131777e-01 -4.603914131777e-01 (...)
            </DataItem>
          </Geometry>
          <Attribute Name="unknown_0" AttributeType="Scalar" Center="Node">
            <DataItem Dimensions="4 4" NumberType="Float" Precision="8" Format="HDF">
              solution-SWERadialDamBreak-0.h5:/Patch_0/unknown_0
            </DataItem>
          </Attribute>
          <Attribute Name="unknown_1" AttributeType="Scalar" Center="Node">
           ...
          </Attribute>
           ...
        </Grid>
        <Grid Name="Patch_1" GridType="Uniform">
        ...
        </Grid>
        ...
      </Grid>
      <Grid Name="subdomain" GridType="Collection">
      ...
      </Grid>
      ...
    </Grid>
    <Grid Name="TimeStep_0.0501911" GridType="Collection" CollectionType="Temporal">
    ...
    </Grid>
  </Domain>
</Xdmf>
```

Listing 3.6: Example XMF File for Scenario With Mapping

```
<?xml version="1.0"?>
<Xdmf Version="3.0">
  <Domain>
    <Grid Name="TimeStep_0" GridType="Collection" CollectionType="Temporal">
      <Time Value="0.000000000000000000000000e+00"/>
      <Grid Name="subdomain" GridType="Collection">
        <Grid Name="Patch_0" GridType="Uniform">
          <Topology TopologyType="2DCoRectMesh" Dimensions="17 17"/>
          <Geometry GeometryType="ORIGIN_DXDY">
            <DataItem Dimensions="2" NumberType="Float" Precision="8" Format="XML">0 0</
    DataItem>
            <DataItem Dimensions="2" NumberType="Float" Precision="8" Format="XML">
    0.0069444 0.0069444</DataItem>
          </Geometry>
          <Attribute Name="unknown_0" AttributeType="Scalar" Center="Cell">
            <DataItem Dimensions="16 16" NumberType="Float" Precision="8" Format="HDF">
              solution-Euler-0.h5:/Patch_0/unknown_0
            </DataItem>
          </Attribute>
          <Attribute Name="unknown_1" AttributeType="Scalar" Center="Cell">
          ...
          </Attribute>
        </Grid>
        <Grid Name="Patch_1" GridType="Uniform">
        ...
        </Grid>
      <Grid Name="subdomain" GridType="Collection">
      ...
      </Grid>
      <Grid Name="subdomain" GridType="Collection">
      ...
      </Grid>
      <Grid Name="subdomain" GridType="Collection">
      ...
      </Grid>
    </Grid>
    <Grid Name="TimeStep_0.0138436" GridType="Collection" CollectionType="Temporal">
    ...
    </Grid>
  </Domain>
</Xdmf>
```

Listing 3.7: Example XMF File for Scenario Without Mapping

3.4 New Output Design Implementation

3.4.1 HDF5 and XMF File Writer

The `PeanoHDF5FileWriter` is responsible for generating both the HDF5 files and the XMF file. It uses the same API as `PeanoTextPatchFileWriter` and implements its own version of the inner cell data writer class, `PeanoHDF5FileWriter_CellDataWriter`, which is described in Section 3.4.2.

Each HDF5 File Writer instance is paired with a single Cell Data Writer object. Together, they generate one HDF5 file representing a single subdomain. All writers share access to a common XMF file, where they append the corresponding subdomain entry for the HDF5 file they generate, as explained in Section 3.3.2.

The HDF5 File Writer is primarily responsible for incrementally updating the shared XMF file. Like the Peano Patch File Writer, it maintains an internal string stream that buffers content before writing it to the XMF file. After the HDF5 file has been fully written, the `writeToFile()` function is called to append the subdomain to XMF. Calculations such as applying the mapping array are performed at this step.

Similar to the Peano Patch File Writer, the HDF5 File Writer constructor supports three XMF creation modes: `CreateNew`, `NoIndexFile`, and `AppendNewData`. Depending on the mode, it creates the XMF file if it does not already exist. The same logic as the Peano Patch File Writer is implemented to decide when a new `Temporal` grid should be created to start a new timestep.

To incrementally generate a correctly structured XMF file, the HDF5 File Writer implements the `removeClosingTags()` function. A critical aspect of XMF generation is correctly handling the placement of closing tags. Since the XMF file contains multiple nested grids, it must include closing tags at precise positions to remain well-formed. The `</Domain>` and `</Xdmf>` tags are always located at the end of the file, enclosing all grid elements. Each grid's data must also be written between its respective opening and closing tags. This structure means that new subdomain data cannot simply be appended to the end of the XMF file, as it needs to be placed before the closing tags. Additionally, it is not feasible to defer writing the closing tags until the last writer completes, as the system does not track which writer will finish last.

To ensure proper XMF structure at all times, the `writeToFile()` function appends all necessary closing tags immediately after each subdomain is written—regardless of whether the simulation is finished. Then, whenever a new writer is instantiated to add another subdomain, the constructor traverses to the end of the existing XMF file, evaluates the provided timestep, and compares it with the last recorded timestep. Based on this evaluation, it removes any previously appended closing tags to allow the new subdomain to be inserted at the correct position. After the final writer completes,

the XMF file remains well-formed.

The HDF5 File Writer's `plotPatch()` function is responsible for creating a patch group and writing the offset and size values into the HDF5 file. The writing of all other datasets in the HDF5 file is handled by the Cell Data Writer.

During the Hotspot Analysis in Section 4.3.2, the HDF5 writer has been extended with a pointer to the HDF5 file to minimize close/open calls on the file.

3.4.2 Cell Data Writer

The Cell Data Writer is instantiated by the HDF5 File Writer and has an internal association to it. Almost all of the HDF5 writing is handled by the Cell Data Writer, except for the patch offset and size. Each patch group in the file is created by the `plotPatch()` function of `PeanoHDF5FileWriter`.

During initialization, the Cell Data Writer writes the metadata group with its datasets.

The `plotCell()` function is responsible for creating and writing the `unknown_i` datasets. The values array from the simulation contains all unknowns in a single array. The `plotCell()` function traverses this array to separate values for each unknown and writes them into individual datasets. Unlike the `PeanoTextPatchFileWriter` implementation, the `plotCell()` only needs to be called once per patch.

Notably, the Cell Data Writer does not modify the XMF file.

3.4.3 Python API

Similar to `PlotPatchesInPeanoBlockFormat.py` (see Section 3.1.2), the Python API has been extended with `PlotPatchesInHDF5Format.py` to define a plotter using the HDF5 writers. The plotters defined by the Python API for the two writer types have minimal differences. Unlike the `PeanoTextPatchFileWriter`, the HDF5 writer's `plotCell()` function does not need to be called for every cell inside the patch. Instead, it is called only once per patch in `PlotPatchesInHDF5Format.py`, eliminating the need for the for-loop present in `PlotPatchesInPeanoBlockFormat.py`. The HDF5 writer also requires shared access to the XMF file, as each writer outputs its subdomain data into this file using the `writeToFile()` function in parallel. Since the XMF file must be carefully structured in the correct order, synchronization between concurrent writers is necessary. The semaphore originally used in `PlotPatchesInPeanoBlockFormat.py` during writer creation is now employed as a lock to coordinate `writeToFile()` function calls in `PlotPatchesInHDF5Format.py`.

Additionally, the Python API has been further extended to allow users to choose between `PeanoTextPatchFileWriter` and `PeanoHDF5Writer`. The Python files that define abstract base classes for the solvers have been updated with a function called

`set_plotter`, which accepts either "peano" or "hdf5" as an argument and assigns the corresponding value to `self._plotter_option`. Depending on this variable, the base solver script triggers the creation of a plotter using either the HDF5 writer or the Peano Text Patch File Writer. The modified base solver classes include `FV.py`, `ADERDG.py`, `RungeKuttaDG.py`, and `CellCenteredFiniteDifferences.py`. Users interact with the `set_plotter` function through the scenario project setup Python script. The choice of writer can be specified either via command-line arguments or by directly modifying the `set_plotter` call before executing the script.

4 Performance Analysis

4.1 Method and Tools

We will discuss the choice behind the utilized scenarios and comparison metrics to analyze the new implementations' performance.

4.1.1 Runtime Measurements

ExaHyPE2's output on the terminal was used to measure the runtime. Once the scenario terminates successfully, information about the time spent in each program section is displayed. The time output named `plotting` is used for the performance analysis since this value isolates the time spent with writers. Additionally, if Peano is compiled with `-DTrackStatistics`, the function `writeToCSV()` uses Peano's statistics backend to output a Comma Separated Value (CSV) file. This file contains detailed runtime statistics, such as how many tasks are in flight at any time or how long MPI routines had to wait for incoming data. [Pea25b]

The scenarios were run five times to get an accurate first measurement, and the average of the `plotting` values was calculated. The solution folder was cleared between each run to ensure precise measurement. Leaving existing solution files from a previous run triggers different modes in the writer, leading to unequal results between runs.

4.1.2 File Sizes

File sizes are a second indicator of performance. The size of the total solution folder was saved with the `du -sb` command in bytes.[19b] Except for the additional case, the solution folder also includes the XMF file size. Ultimately, the average file size was manually calculated similarly to the runtime. File sizes are displayed with International System of Units (SI).

4.1.3 Scenarios

We will run the chosen scenarios with both writers for a consistent comparison. The performance analysis will be divided into 2D and 3D scenarios. The primary comparison will be done with the 2D scenarios since they are smaller and faster to execute.

3D scenarios are much more prominent in size and will be used to observe how the writers scale.

Whether the scenario consists of a mapping affects file sizes and the runtime. Therefore, we will differentiate between two scenario types. The 2D non-mapping case will use an Euler equation scenario from the ExaHyPE2 tutorials. The patch size is 16, and the number of unknowns is 4. "The scenario describes the movement of a fluid if there is neither loss of energy due to frictional forces nor heat conduction in the system." [Pea25d] For the 2D mapping case, Radial Dam Break (RDB) will be used, implemented with Shallow Water Equations (SWE) located in the tests folder. "SWE describe the flow of fluid given the assumption that the horizontal length scale is much higher than the vertical length scale."[Pea25e] A RDB scenario is a common test case for SWE. It simulates the sudden collapse of a circular dam, which leads to an outward wave propagation. Initially, a circular high-water region is surrounded by a dry water region simulating a dam. When the scenario is run, the dam is removed, and water from the high-level region spreads outward in a radial propagation. This scenario's patch size and number of unknowns are both 4. The solver was compiled with the polynomial type Gauss-Legendre, which is responsible for the grid spacing seen in Figure 4.1 (right).

For both scenarios, an additional parameter will be included. Both writers will be compiled with the `NoIndexFileMode`, which excludes the generation of the XMF file or the index file. So, the solution folder will only contain storage files such as `.h5` or `.peano-patch-file`. This additional case is interesting because the index file generated by the `PeanoTextPatchFileWriter` cannot be directly used for visualization. Hence, it is much smaller than the XMF file, which directly serves as a visualization tool, leading to a much bigger file size. The generation of this XMF file can also increase the runtime of `PeanoHDF5FileWriter`. With this additional case, we can make a more transparent comparison of both writers without the extra overhead of an additional XMF file.

To observe how the writers scale with a bigger 3D scenario, Euler was compiled with 3 dimensions, patch size 16 and volume size 0.1.

4.2 First Results

The column charts representing runtime and file sizes for the 2D scenarios can be seen in Figure 4.2 and Figure 4.3, respectively. The additional case mentioned in Section 4.1.3, where both writers only produce individual subdomain files, has also been included in these figures. 3D scenario results are shown in Figure 4.4.

Since the amount of data being written for both scenarios differs, the runtime will be compared relative to the number of cells written. The total number of

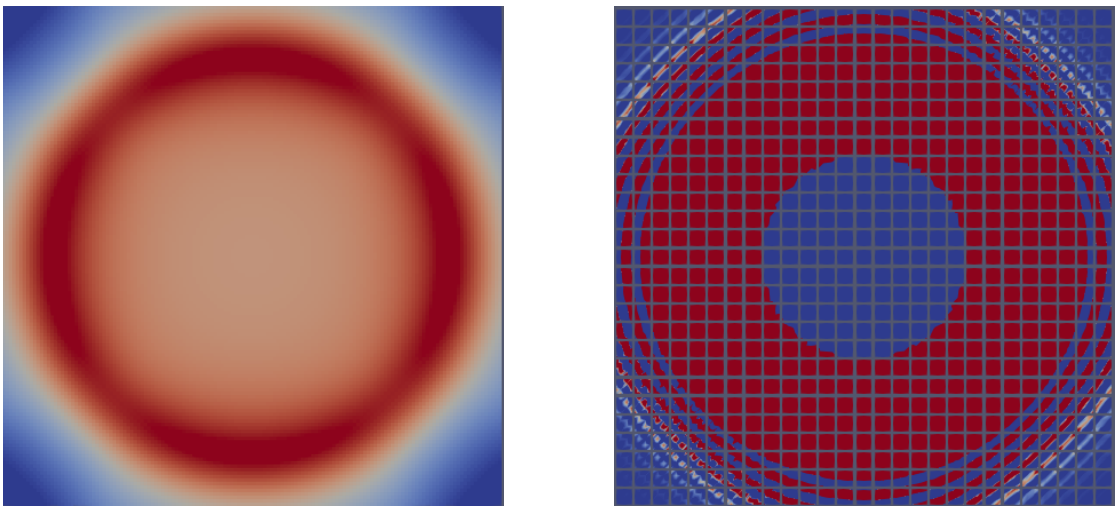


Figure 4.1: 2D Euler (left) and 2D RDB Scenario (right) Visualized With Paraview

cells per scenario is calculated using the formula: $Total\ cells = Number\ of\ files \times Number\ of\ patches\ per\ file \times Number\ of\ cells\ per\ patch\ in\ the\ file$. The runtimes can be relatively compared between both scenarios with this value: $Runtime\ per\ cell \approx Runtime\ from\ plotting \times Total\ cells$. The same approach can be used to compute file size per cell.

The Euler scenario has 404 individual `.h5` or `.peano-patch-file` files (not including the XMF or index file). Out of every 4 patch files, 3 of them contained 20 patches, and 1 of them had 21 patches. Therefore, the average 20.25 will be used as the patches per file value in the calculation for total cells. This also applies to the RDB scenario and the 3D scenario. The ASCII writer takes approximately 0.000006 seconds per cell, while the HDF5 writer takes 0.000042. This means the HDF5 writer is about 6.8 times slower per cell than ASCII. When the XMF file is omitted, the HDF5 writer improves very slightly (0.000041) but remains 5.3 times slower than ASCII (0.000008).

The RDB scenario has 44 individual files. 182.25 was the average patch number in one file, calculated similarly to the Euler scenario. ASCII writer took around 0.000023 seconds per cell, while HDF5 needed 0.000179 seconds. This means the HDF5 writer is 7.6 times slower than ASCII. For the additional case with no XMF generation, the HDF5 writer improves to 0.000125 seconds but is still 5.3 times slower than ASCII (0.000023). Without XMF, the HDF5 writer achieves a 43% speedup in the RDB scenario and a 28% speedup in Euler. This confirms that writing the XMF for a scenario with mapping introduces additional algorithmic steps increasing runtime.

For the Euler scenario, the HDF5 writer generates 0.000054 MB per cell, which is 1.2

times more than the ASCII writer (0.000046). With the additional case, HDF5 file sizes are 1.05 times more than ASCII, meaning both writers scale similarly. For the RDB scenario, the ASCII writer generates 0.000058 MB per cell and HDF5 0.000436 MB. This is 7.5 times more than ASCII. When the XMF is omitted, the ratio falls to 5.1.

The same total cells calculation was used to observe the change from 2D to 3D Euler scenario. There were 352 files. Of 4 files, 3 had 6 patches, and 1 had 9 patches, resulting in 6.75 as the average patches per file. ASCII writer took 0.000002 seconds per cell and HDF5 needed 0.000003. This means that in the 3D scenario, the HDF5 was 1.6 times slower. Despite still being slower than ASCII, the HDF5 writer scaled better in 3D than in 2D, where it was 6.8 times slower. In terms of file size, HDF5 produced 0.000042 MB per cell, while ASCII produced 0.000055 MB. In 2D, HDF5 files were 1.2 times larger than ASCII, but in 3D, they were only 0.75 times as large. As the simulation scales to 3D, HDF5 format becomes more storage-efficient.

As a result, we can say that the `PeanoHDF5FileWriter` currently operates slower than the `PeanoPatchFileWriter`. This is expected, as HDF5 involves more complex data structure management than ASCII files. The runtime bottlenecks will be investigated in Section 4.3.2 with using the Intel VTune Profiler. HDF5 writer also produces bigger files, especially when the scenario contains mapping. Chunking and compression were used to try and minimize the file size difference in Section 4.3.1.

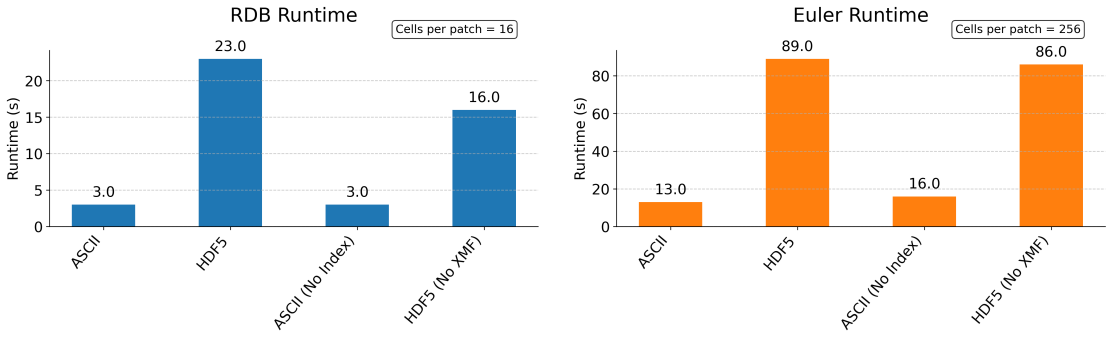


Figure 4.2: 2D Scenario Comparison of RDB and Euler Runtimes

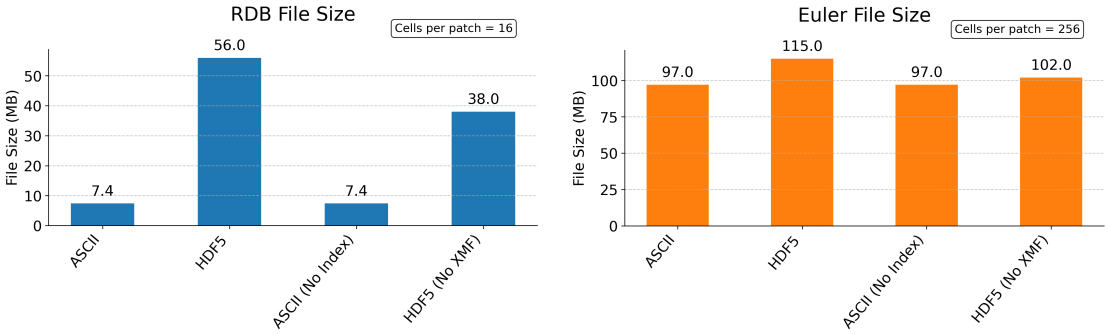


Figure 4.3: 2D Scenario Comparison of RDB and Euler File Sizes

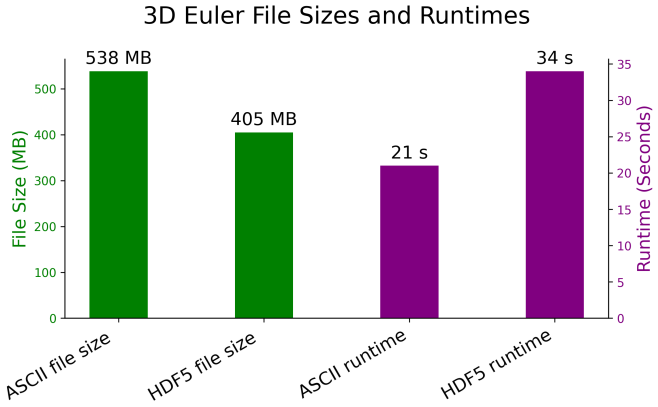


Figure 4.4: 3D Euler Scenario Runtime and File Size Comparison

4.3 Optimization

In this section, we aim to improve the results of the first performance analysis run, which were explained in Section 4.2. The focus is on optimizing both the runtime and file size.

4.3.1 File Size

To decrease the size of the HDF5 files generated by `PeanoHDF5FileWriter`, the `gzip` compression filter is used. `gzip` is the standard and supported by default in HDF5. Other available filters include `n-bit`, `scale-offset` and `szip`.

Chunking

When using compression, it is required that the HDF5 files are chunked. Chunking is mandatory for extendable unlimited datasets regardless of compression. This work uses fixed-size datasets with a contiguous storage strategy where chunking is only necessary when compression is enabled. Therefore, the results mentioned in Section 4.2 were not chunked. It is important to note that choosing an incorrect chunking arrangement can degrade performance. Therefore, we first determine an optimal chunk size. [PK17]

The mapping and the values for unknowns are stored as one-dimensional datasets in HDF5. Therefore, these are the candidates for chunking. These datasets have fixed sizes of $dimsMapping = cellsperpatch \times dimensions$ and $dimsValues = cellsperpatch$. Both of these size values will be referred to as `dims` from now on. There are two constraints for chunk shape and size: The dimensions of a chunk must be less than or equal to that of the dataset, and the chunk size cannot be more than the size of the dataset [The25b]. Because our datasets are one-dimensional, the chunk dimension can only be set to 1. For chunk size, values between 1 and `dims` come into question. Since `dims` can vary according to the scenario's number of unknowns and cells per patch, we can not set explicit numbers for the chunk size. It needs to be set in relation to `dims`.

Multiple chunk sizes were tested: `dims`, `dims/2`, `dims/4`, `dims/8`, `dims/16`, and `dims/32`. Without chunking, the Euler scenario had a runtime of 89 seconds and a file size of 115 MB. With all tested chunk sizes, the file size increased to 175 MB and remained constant. File sizes are expected to increase after enabling chunking. Each chunk introduces metadata overhead, as chunks are not stored in a predefined order.

Since file size optimization is primarily achieved through compression, selecting the chunk size with the lowest runtime seems most appropriate. However, runtime differences among chunk sizes were minimal and likely within each other's rounding

error margins. The measured `plotting` runtimes (in seconds) for different chunk sizes were:

$$\frac{dims}{1} = 88s, \frac{dims}{2} = 83s, \frac{dims}{4} = 87s, \frac{dims}{8} = 82s, \frac{dims}{16} = 83s, \frac{dims}{32} = 94s$$

As previously mentioned, all tested chunk sizes resulted in a file size of 175 MB. Therefore, we ultimately chose a chunk size of `dims` to maximize compression efficiency. Storing each dataset as a single chunk of size `dims` allows the compression filter to process the entire dataset at once, leading to higher compression ratios, especially for uniform data. Multiple smaller chunks are beneficial for partial I/O operations. However, since our writers always read and write entire datasets, smaller chunks provide no advantage. Thus, using a single chunk per dataset is optimal for our I/O patterns.

The next section explores compression techniques to mitigate the increased file size caused by chunking.

Compression

The HDF5 library provides a built-in deflate compression filter based on `gzip`. Compression levels, ranging from 0 to 9, can be set using the `H5Pset_deflate()` function. Higher compression levels result in better compression ratios but slower processing speeds, with level 9 offering the highest compression at the cost of performance. Setting the level to 0 does not disable the `gzip` filter but applies no compression during data processing. Further details on this function can be found in the HDF5 documentation.

The Euler scenario from the previous chapter, where the mandatory chunking was applied, serves as the baseline for comparison with 175MB file size and 88s runtime.

Applying compression at level 1 reduced the file size to 160 MB, while the runtime increased to 128 seconds. Higher compression levels, including level 9, did not reduce the file size further, but runtime continued to increase, with level 9 taking 145 seconds. These results suggest that compression beyond level 1 provided no additional file size reduction beyond 160 MB.

This raised concerns about whether the compression was functioning correctly, given that the original unchunked file size was 115 MB. The `h5dump` tool with the `-pH` option was used to inspect each dataset's compression levels (Listing 4.1). A compression ratio of 89.043:1 indicates that the uncompressed data was 89.043 times larger than the compressed data. The effectiveness of compression depends on data redundancy and entropy: datasets with many repeated values, especially zeros, compress more efficiently. For instance, `unknown_1` stored only zeros, whereas `unknown_0` contained only ones. As Listing 4.1 confirms, the compression filter was successfully applied with high ratios.

However, the benefits of compression were outweighed by the overhead introduced by chunking. Before compression and chunking, the file size was 115 MB; afterward, it increased to 160 MB. As a result, compression and chunking were ultimately disabled.

The primary reason for this overhead is likely the storage structure of the unknowns, as they are stored in separate datasets. For instance, if there are 1000 patches, each with 4 unknowns, this results in 4000 separate chunked datasets, each adding management overhead and resulting in increased file size. To examine whether this was the cause, the next section explores an alternative approach where all unknowns of a patch are stored together in a single dataset. This reduces the total number of datasets from 4000 to 1000, potentially mitigating the overhead.

```
DATASET "unknown_0" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 256 ) / ( 256 ) }
         STORAGE_LAYOUT {
            CHUNKED ( 256 )
            SIZE 29 (70.621:1 COMPRESSION)
         }
         FILTERS {
            COMPRESSION DEFLATE { LEVEL 9 }
         }
         (...)
      }
      DATASET "unknown_1" {
         DATATYPE  H5T_IEEE_F64LE
         DATASPACE  SIMPLE { ( 256 ) / ( 256 ) }
         STORAGE_LAYOUT {
            CHUNKED ( 256 )
            SIZE 23 (89.043:1 COMPRESSION)
         }
         FILTERS {
            COMPRESSION DEFLATE { LEVEL 9 }
         }
         (...)
      }
```

Listing 4.1: Example h5dump -pH Output of Compressed and Chunked Euler Scenario

Optimization Approach - Single Unknowns Dataset

To reduce the number of datasets and potentially decrease file sizes, all unknowns of a patch were stored together in a 3-dimensional dataset. For 2D scenarios, the dataset

dimensions are defined as: `number-of-dofs-per-axis` × `number-of-dofs-per-axis` × `number-of-unknowns` For 3D scenarios, this would extend to a 4-dimensional dataset, incorporating an additional axis.

To correctly extract each unknown from this multidimensional dataset, the HDF5 referencing in the `Attribute` field of the XMF file (Section 3.7) must be adjusted. Instead of defining multiple `Attribute` elements for each unknown, a single `Attribute` element with `AttributeType="Matrix"` is used to reference the entire dataset. The `Dimensions` field directly corresponds to the dataset's shape. An example is shown in Listing 4.2, where 16 DOFs per axis and 4 unknowns exist. Apart from this modification, the XMF file structure remains identical to the example in Listing 3.7.

```
<Attribute Name="AllUnknowns" AttributeType="Matrix" Center="Cell">
    <DataItem Format="HDF" NumberType="Float" Precision="8" Dimensions="16␣16␣4">
        solution-Euler-0.h5:/Patch_0/unknowns
    </DataItem>
</Attribute>
```

Listing 4.2: Optimization Approach: Updated XMF Attribute Element

Executing the 2D Euler scenario introduced in Section 4.2 with this new approach confirmed that the dataset structure had been a significant factor outweighing the benefits of compression. In the previous approach, the Euler scenario had a runtime of 89s and an uncompressed file size of 115 MB. The optimized approach yielded the following results:

- 1. Without chunking and compression: 108 MB and 55s runtime.
- 2. With mandatory chunking, where chunk dimensions and size match the dataset dimensions exactly (see above): 119 MB and 71s. As expected, chunking brought an initial file size increase.
- 3. With compression level 3: 95 MB and 49s. (including XMF file: 5.8 MB)

This design change successfully decreased the file size from 108 MB to 95 MB. The HDF5 writer now produces smaller files than the ASCII writer, which yielded 97 MB for the same scenario. It is important to note that the 95 MB from the HDF5 writer includes an XMF file of 5.8 MB, whereas the ASCII writer does not generate an XMF file. This means the actual HDF5 file size is around 89 MB, which is even smaller than the ASCII writer's 97 MB.

With this optimization approach, the visualization in ParaView was accurate for the Euler scenario, which generates cell-based values and does not use a mapping array. However, the RDB scenario, which uses a mapping array and provides vertex-based values, introduced in Section 4.2, could not be accurately displayed in ParaView.

The primary issue arises because the simulation provides vertex-based unknown values when a mapping array is present. In XDMF, vertex-based values require the attribute `Center="Node"`, which works correctly in the current approach that stores unknowns separately, as explained in Section 3.3.2. However, when unknowns are stored in a multidimensional dataset, ParaView fails to read attributes with `Center="Node"`. Instead, it only reads when set to `Center="Cell"`, which is incorrect for vertex-based values and results in inaccurate visualizations (see Figure 4.5).

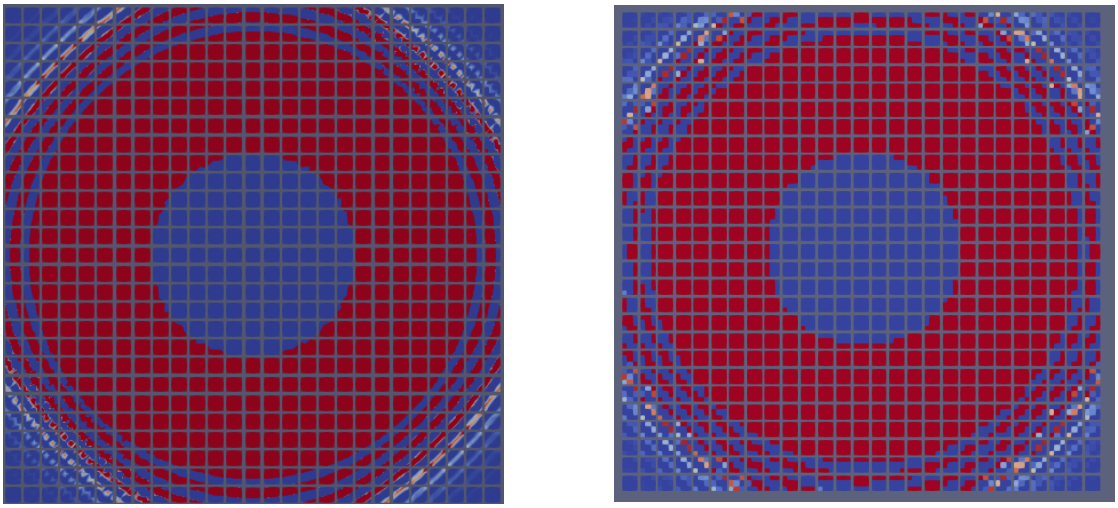


Figure 4.5: 2D RDB Scenario With Optimization Approach (right) and Main Approach (left)

After re-evaluating this issue, multiple attempts were made to adjust the multidimensional XDMF referencing in Listing 4.2 to correctly extract the values while maintaining `Center="Node"`. Various configurations of `AttributeType="Matrix"` were tested, along with storing the unknowns in 1D or 2D datasets instead of 3D. HyperSlab, an alternative method in XDMF for referencing subsets of multidimensional datasets, was also tested. However, ParaView failed to read the data in every case when using `Center="Node"`. Since the same dataset worked when using `Center="Cell"`, the issue appears to be related to ParaView's handling of multidimensional values specifically with `Center="Node"`, rather than an issue with the HDF5 storage structure itself.

As a result, while this alternative optimization approach successfully reduced file size and improved performance, it could not be fully adopted due to ParaView's limitations in handling vertex-based values stored in a multidimensional datasets.

4.3.2 Runtime

To reduce the runtime of `PeanoHDF5FileWriter`, Intel VTune Profiler is used to perform a hotspot analysis. A hotspot refers to a function or loop in the code that consumes significantly more execution time compared to others. Once potential hotspots are identified and optimized through code modifications, compiler optimization flags will be applied to further improve performance.

Hotspot Analysis

The VTune Profiler from Intel was used to perform a hotspot analysis of the executable. This tool measures program runtime and ranks function calls or loops based on the time spent in the CPU.

The initial hotspot analysis was conducted on the 2D Euler scenario from Section 4.2, which had an average `plotting` runtime of 89 seconds. This value does not represent the total CPU execution time but instead refers to the time spent specifically on plotting, as printed by the Peano API in the terminal.

The analysis revealed that the `removeClosingTags()` function of the HDF5 File Writer (see Section 3.4.1) was the most time-consuming operation. Out of a total CPU time of 221 seconds, 11.6% (34 seconds) was spent executing this function. The original implementation processed the entire XMF file in a while loop, scanning from the top to remove the closing tags at the end of the file. This operation has to be repeated in each writer constructor call, adding significant overhead.

To optimize this process, a new algorithm was implemented. The closing tags are always located at the end of the file and have a constant word length. The new approach leverages this knowledge by directly positioning the read cursor near the end of the file, skipping unnecessary processing of unrelated content. Instead of scanning from the beginning, the algorithm now seeks the closing tags from the end and adjusts the file position accordingly. The file is then resized at the adjusted position, effectively removing the closing tags without iterating over the entire file. This optimization drastically reduced the CPU time of `removeClosingTags()` from 34 seconds to 0.043 seconds, significantly improving performance. As a result, the `plotting` runtime decreased from 89 seconds to 71 seconds.

The analysis also revealed that the `plotCell()` and `plotPatch()` functions each had a CPU time of 7.5 seconds (out of 221 seconds), making them the third most time-consuming operations. Upon examining their implementation, it was observed that the HDF5 file was being opened and closed for each read/write operation, leading to unnecessary overhead.

To optimize this, a pointer to the HDF5 file was introduced in `PeanoHDF5FileWriter`

and initialized within its constructor instead of in the Cell Data Writer (see Section 3.4.2). By using this pointer, the HDF5 file remains open throughout the entire plotting process, significantly reducing the number of open/close calls. The file is now closed only once all datasets have been written, which occurs in the `writeToFile()` function. With this optimization, `plotPatch()` had a CPU time of 1.242s and `plotCell()` had the slightly bigger CPU time 2.333s, since it contains additional loops. The overall plotting has now decreased to 50s.

Compiler

To further optimize the runtime, compiler flags and directives can be used. For this section, the project was compiled with the optimization level `-O3`.

Compiler directives are chosen dependent on the used compiler. If the code has been compiled with OpenMP, the `#pragma omp simd` loop directive can be used. This OpenMP SIMD directive is generally portable between `gcc` and `clang`. Therefore, the OpenMP directive will be applied. This directive tells the compiler that the iterations are independent and it should vectorize the loop. It requires the for loop to have canonical loop form and iterations to not have dependencies [Ope18].

To identify potential parts in the code which can be optimized by using compiler directives, the VTune Profiler was used. There were three for loops which had a significantly long execution time. The longest loop, from `getLatestTimeStepInXMFFile()` in `PeanoHDF5FileWriter` had a CPU time of 9 seconds. However, this loop requires the `while` loop form and therefore exluced the application of the `#pragma omp simd`. Regardsless, the `-O3` option descreased the CPU time to 3.3 seconds. Two loops in the `writeToFile()` function are also excluded from the directive application since the for loop iterations contain data independencies. The `-O3` flag decreased the longest loop in the function from 0.475s CPU time to 0.256s.

The `plotCell()` function from the Cell Data Writer had a CPU time of 2 seconds. For this loop, the compiler directive is appropriate. By setting the `-O3` flag and using the `#pragma omp simd` directive, the CPU time was descreaed to 1.9s only.

As a result of all optimizations in this chapter, the overall plotting time of the HDF5 writer for the 2D Euler scenario has now decreased from the initial 89 seconds to 11 seconds.

4.4 Discussion

The HDF5 writer produces larger file sizes compared to the Peano Text Patch File Writer. Due to the structure of HDF5 datasets, compression does not provide significant file size reductions.

Through runtime optimizations, the HDF5 writer achieved an 87% performance improvement and is now approximately 15% faster than the Peano Text Patch File Writer. Additionally, unlike the Peano Text Patch File Writer, the HDF5 writer eliminates the need for a separate conversion step by directly generating an incrementally written XMF file. This improvement is fully accounted for in the measured runtime.

5 Future Work

A successful HDF5 and XMF writer has been implemented and analyzed for performance within the scope of this work. However, there is still room for improvement and future work, which is discussed in this chapter.

5.1 Using Float Storage in HDF5

The currently implemented writer stores all values in the HDF5 files as `double`. These values can be categorized into unknowns and geometry-related data, such as mapping and offsets.

To reduce file size, an alternative approach was tested by storing values as `float` instead of `double`. However, when loading the XMF file in ParaView, the grid did not appear correctly. To investigate further, a mixed approach was attempted: storing geometry data as `double` while saving unknowns as `float`. This partially reduced file size, and the grid shape was displayed correctly. However, the simulation results in ParaView were incorrect, indicating that storing unknowns as `float` led to inaccuracies.

The reason behind this issue was investigated but could not be identified. As a result, the decision was made to use only `double`. If a method can be found to store unknowns as `float` while preserving accurate visualization, it could enhance performance. The XMF file `Precision` field must be adjusted accordingly.

5.2 Unknown Names

Currently, the unknowns are displayed and stored under the generic name `unknown_i`. A potential improvement would be to retrieve the actual names of the unknowns directly from ExaHyPE2's interfaces and update the dataset names accordingly. This enhancement would improve clarity, making the stored data more interpretable and aligned with the underlying simulation variables.

5.3 ParaView Loading Improvement

In some scenarios, such as the Elastic, loading the XMF file into ParaView and running the simulation takes significantly longer than using the PVD file. The underlying reason for this could be further investigated. A likely explanation is that the XMF file explicitly contains all geometry data for each timestep within a single file, making parsing more time-consuming. In contrast, ParaView loads only the data for the current timestep when using a PVD file.

One possible alternative is to use the `h5tovtk` command, which converts the generated HDF5 files into VTK format [19c]. These converted files could then be displayed using a PVD file, potentially improving loading speed. However, this approach introduces additional conversion overhead, which contradicts the original goal of eliminating extra processing steps in this work.

5.4 Vertex Data Writer

The Cell and Vertex Data Writer share similar functionality. Therefore, only the Cell Data Writer has been fully implemented so far. The Vertex Data Writer is declared within the `PeanoHDF5FileWriter.h` to allow compilation but has not been implemented since it was not strictly necessary.

5.5 XDMF3 C++ API

In this work, the XDMF generation was done using only the C++ output file streams and the formatted output insertion operator. Another approach is to utilize the XDMF API, which provides functions for this purpose. When using pure C++, we must manually ensure that the output adheres to the correct XMF formatting. Using the XDMF3 C++ API can make this process more convenient by handling the structure and syntax. However, this API needs linking against a library and can bring a performance overhead compared to using pure C++. [14]

Abbreviations

ASCII American Standard Code for Information Interchange

CSV Comma Separated Value

DOF Degrees of Freedom

HDF Hierarchical Data Format

HDF5 Hierarchical Data Format 5

PDE Partial Differential Equations

PVD ParaView Data

RDB Radial Dam Break

SI International System of Units

SWE Shallow Water Equations

VTK Visualization Toolkit

XDMF eXtensible Data Model and Format

XML Extensible Markup Language

List of Figures

Listings

Bibliography

[14] *XDMF- Xdmf3 C++ API*. 2014. URL: https://www.xdmf.org/index.php/Xdmf3_C%2B%2B_API (visited on 02/14/2025).

[17] *XDMF - Main Page*. 2017. URL: https://www.xdmf.org/index.php/Main_Page (visited on 02/14/2025).

[19a] *Ubuntu Manpage: ascii - ASCII character set encoded in octal, decimal, and hexadecimal*. 2019. URL: https://manpages.ubuntu.com/manpages/focal/en/man7/ascii.7.html (visited on 02/19/2025).

[19b] *Ubuntu Manpage: du - estimate file space usage*. 2019. URL: https://manpages.ubuntu.com/manpages/trusty/man1/du.1.html (visited on 02/14/2025).

[19c] *Ubuntu Manpage: h5tovtk - convert datasets in HDF5 files to VTK format*. 2019. URL: https://manpages.ubuntu.com/manpages/lunar/man1/h5tovtk.1.html (visited on 02/14/2025).

[22] *XDMF - Model and Format*. 2022. URL: https://www.xdmf.org/index.php/XDMF_Model_and_Format (visited on 02/14/2025).

[Ope18] OpenMP Architecture Review Board. *OpenMP specification - SIMD Directives*. 2018. URL: https://www.openmp.org/spec-html/5.0/openmpsu42.html (visited on 02/28/2025).

[Par20] ParaView Developers. *ParaView Documentation - 1.1 Introduction*. 2020. URL: https://docs.paraview.org/en/latest/UsersGuide/introduction.html (visited on 02/17/2025).

[Pea25a] Peano Development Team. *Peano4 Documentation - Architecture*. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/db/d3f/page_architecture_home.html (visited on 02/17/2025).

[Pea25b] Peano Development Team. *Peano4 Documentation - Logging, tracing, statistics, profiling and assertions*. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/d1/d9e/tarch_logging.html (visited on 02/17/2025).

[Pea25c] Peano Development Team. *Peano4 Documentation - Peano 4*. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/dd/da2/page_peano4_home.html (visited on 02/17/2025).

[Pea25d] Peano Development Team. *Peano4 Documentation - The Euler Equation*. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/d3/df5/page_exahype2_tutorials_toyproblems_euler.html (visited on 02/20/2025).

[Pea25e] Peano Development Team. *Peano4 Documentation - The Shallow Water Equations*. 2025. URL: https://hpcsoftware.pages.gitlab.lrz.de/Peano/db/d10/page_exahype2_tutorials_toyproblems_swe.html (visited on 02/20/2025).

[PK17] E. Pourmal and L. Knox. *HDF5 Data Compression Demystified – Part 2: Performance Tuning*. May 2017. URL: https://www.hdfgroup.org/2017/05/24/hdf5-data-compression-demystified-2-performance-tuning/ (visited on 02/17/2025).

[Rei+20] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems." In: *Computer Physics Communications* 254 (2020), p. 107251. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2020.107251.

[The25a] The HDF Group. *HDF5 Data Model and File Structure*. 2025. URL: https://support.hdfgroup.org/documentation/hdf5/latest/_h5_d_m__u_g.html#sec_data_model (visited on 02/19/2025).

[The25b] The HDF Group. *HDF5 Datasets - Storage Strategies*. 2025. URL: https://support.hdfgroup.org/documentation/hdf5/latest/_h5_d__u_g.html#sec_dataset (visited on 02/14/2025).

[The25c] The HDF Group. *HDF5 Predefined Datatypes*. 2025. URL: https://support.hdfgroup.org/documentation/hdf5/latest/predefined_datatypes_tables.html (visited on 02/19/2025).

[The25d] The HDF Group. *The HDF5 h5dump Tool*. 2025. URL: https://support.hdfgroup.org/documentation/hdf5/latest/_h5_t_o_o_l__d_p__u_g.html (visited on 02/19/2025).

[VTK23a] VTK Developers. *VTK Documentation - Supported Data Formats*. 2023. URL: https://docs.vtk.org/en/latest/supported_data_formats.html (visited on 02/17/2025).

[VTK23b] VTK Developers. *VTK Documentation - VTK File Formats*. 2023. URL: https://docs.vtk.org/en/latest/design_documents/VTKFileFormats.html (visited on 02/17/2025).

[Wei19] T. Weinzierl. "The Peano Software-Parallel, Automaton-Based, Dynamically Adaptive Grid Traversals." In: *ACM Transactions on Mathematical Software* 45.2 (June 2019). ISSN: 0098-3500. DOI: 10.1145/3319797.