Implementation and Evaluation of Parallel I/O Interfaces in ExaStencils

Richard Angersbach

Master Thesis
Implementation and Evaluation of Parallel I/O Interfaces in ExaStencils

Richard Angersbach
Master Thesis

Aufgabensteller: Prof. Dr.-Ing. Harald Köstler
Betreuer: Dr.-Ing. Sebastian Kuckuk
Bearbeitungszeitraum: 12.08.2020 – 06.04.2021
Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Master Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 6. April 2021
## Contents

1 Introduction 7

2 Related Work 9

3 Code Generation 11
   3.1 ExaStencils 11
      3.1.1 Numerical Methods 11
      3.1.2 Domain-specific Languages 12
      3.1.3 Workflow 12
      3.1.4 Data Partitioning 15
      3.1.5 Computational Grids 16
      3.1.6 Fields 18
      3.1.7 Former State of I/O Interfaces 18
   3.2 GHODDESS 20
      3.2.1 Overview 20
      3.2.2 Grid Types 21
      3.2.3 Workflow 21

4 Parallel Input/Output 23
   4.1 Background 23
      4.1.1 Data Representation and Layout 23
      4.1.2 Parallel I/O Infrastructure 24
      4.1.3 Parallel File Systems 25
      4.1.4 Parallel I/O Patterns 27
   4.2 Parallel I/O Interfaces 28
      4.2.1 Outline 28
      4.2.2 MPI-I/O 29
      4.2.3 HDF5 31
      4.2.4 Parallel NetCDF 34
      4.2.5 SIONlib 36

5 Visualization Formats 39
   5.1 VTK 39
   5.2 XDMF 39
   5.3 NetCDF 42
      5.3.1 Meta-data conventions 42
      5.3.2 ExodusII 42

6 Implementation 44
   6.1 Design 44
      6.1.1 Goals 44
      6.1.2 Overview 44
   6.2 I/O Interfaces 45
      6.2.1 Structure 45
      6.2.2 C++ Standard Library Implementation 48
      6.2.3 MPI-I/O Implementation 49
      6.2.4 HDF5 Implementation 53
      6.2.5 PnetCDF Implementation 55
      6.2.6 SIONlib Implementation 56
   6.3 Visualization Interfaces 58
      6.3.1 Organization 59
      6.3.2 XDMF Implementation 60
      6.3.3 ExodusII Implementation 62
   6.4 Field I/O interfaces 63
      6.4.1 ReadField and WriteField 63
      6.4.2 PrintField 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5 Optimizations</td>
<td>66</td>
</tr>
<tr>
<td>6.5.1 Hints</td>
<td>66</td>
</tr>
<tr>
<td>6.5.2 Data Reduction</td>
<td>69</td>
</tr>
<tr>
<td>7 Evaluation</td>
<td>71</td>
</tr>
<tr>
<td>7.1 Usability</td>
<td>71</td>
</tr>
<tr>
<td>7.2 Portability</td>
<td>72</td>
</tr>
<tr>
<td>7.3 Performance</td>
<td>72</td>
</tr>
<tr>
<td>7.4 File size</td>
<td>76</td>
</tr>
<tr>
<td>7.5 Visualization Capabilities</td>
<td>77</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>79</td>
</tr>
<tr>
<td>8.1 Summary</td>
<td>79</td>
</tr>
<tr>
<td>8.2 Future Work</td>
<td>80</td>
</tr>
</tbody>
</table>
Abstract

In the scope of High Performance Computing (HPC), simulations for larger problems are nowadays commonly run on highly parallel supercomputers producing immense amounts of data. In accordance to the parallelization of an application’s computations, the same notion of parallelism is necessary for its Input/Output (I/O) operations. Since I/O is an essential component in HPC applications but also a well-known bottleneck, providing solutions for an efficient storage is crucial.

In this thesis, parallel I/O techniques are evaluated in the context of the ExaStencils code generation framework which will be introduced in the beginning. Despite the fact that ExaStencils provides means to perform I/O in a parallel environment, these are rather rudimentary and do not fully utilize the I/O hardware. Therefore, the main aim of this thesis is a coupling of parallel I/O interfaces and ExaStencils. For this purpose, general I/O strategies and libraries are analysed at first. For more structure, these were encapsulated in a dedicated I/O package implemented within the ExaStencils compiler. Additionally, visualization interfaces which utilize the new I/O layer are implemented. To further minimize the time spent in I/O optimization techniques are proposed. All these functionalities were also exposed to the GHODDESS framework which utilizes ExaStencils’ source-to-source compiler underneath. An evaluation of the new I/O interfaces in comparison to the prior state is made in scope of applications from the GHODDESS project, e.g. the simulation of tidal flows around the Bahamas islands. Accordingly, scaling experiments of the I/O performance are made on the Emmy cluster.
1 Introduction

Partial Differential Equations (PDEs) are used in various fields of Computational Science and Engineering (CSE) as they serve as mathematical models of many physical phenomena. These application domains range from areas in natural science such as physics, biology and chemistry to fields of engineering, e.g. electrical engineering. One particularly interesting model in the scope of this thesis is the so-called Shallow Water Equation (SWE). This is a commonly used model for problems of the field of oceanography, meteorology and coastal engineering [28].

In the context of computing a numerical solution for the often large systems of equations that originate from the discretization of PDEs, scalable and efficient solvers are required. The multigrid method is known as one of the most efficient class of solvers. For larger problems, these solvers are typically run on HPC clusters which nowadays consist of various hardware components such as different Central Processing Units (CPUs), memory architectures and accelerators, e.g. in form of Graphics Processing Units (GPUs) [77]. Most notably, the implementations of such solvers are commonly tuned towards these highly parallel and heterogeneous systems for optimal utilization and therefore require profound knowledge in programming and hardware in addition to the mathematical models of the corresponding application domain. Additionally, in case that the code is to be ported and executed on another target system, a revision of the applied optimizations needs to take place which makes common goals such as performance portability hard to achieve. For these reasons, the software development process proves to be a challenging task. One increasingly popular practice, especially in the scope of HPC applications, is to use Domain-specific Languages (DSLs) in combination with code generation techniques. In this thesis, the code generation framework ExaStencils which specializes in the application domain of geometric multigrid solvers on (semi-)structured grids will be thoroughly examined. ExaStencils employs its own multi-layered DSL named the ExaStencils language, in short ExaSlang, where each layer is customized towards a certain group of domain experts. From the program specification in ExaSlang, source code in the target language C++ with support of parallelization back-ends such as MPI, OpenMP and CUDA can be output by ExaStencils’ source-to-source compiler.

A popular field of research in the scope of CSE is the simulation of ocean currents, tides and coastal ocean circulation [28]. For these, the SWE can be used as a model. There exist many approaches to discretize and solve this kind of equation efficiently. One promising solution is the application of a quadrature-free Discontinuous Galerkin (DG) scheme. The Generation of Higher-Order Discretizations Deployed as ExaSlang Specifications (GHODDESS) project extends the ExaStencils toolchain with a Python interface that allows users to easily specify their discrete problem in a symbolic algebra description, that is, by using the symbolic algebra package SymPy. GHODDESS then maps the symbolic formulation to an ExaSlang specification and thus makes fully use of ExaStencils code generation capabilities.

The utilization of Input/Output (I/O) in scientific simulations is ubiquitous since these, regardless if sequential or parallel, are often required to store and retrieve data from files. In the scope of HPC, there are many uses cases for file accesses, such as reading input parameter files at the setup phase of the simulation, checkpointing for fault tolerance and restarting capabilities, emitting log files and writing results for further data analysis and visualization during post-processing. With the advances in parallel processing hardware, scientists are increasingly using parallel computers to solve problems that require both large amounts of data and computing power. Moreover, due to the immense amounts of effort spent into a more efficient usage of computing resources, HPC applications follow the trend towards simulating at larger scales and finer granularities. As a result, the aggregate amounts of data increase rapidly. However, while there has been great progress in CPU and communication performance, similar advances have not been made in the speed of I/O devices, but rather in their capacities and densities [88]. Hence, I/O speed is a commonly known bottleneck in HPC codes and plays an important role in the overall time to completion. Despite the fact that the performance gap cannot be filled without breakthroughs in I/O technology, it is important to use storage devices as efficiently as possible to mitigate the impact of I/O operations.

Visualization plays an important role in many fields of CSE as it turns large amounts of data into comprehensible images that often reveal important aspects of the data and thus grant scientists more insight to the data. In principle, there are two different approaches to visualize the data. The first is to couple the visualization software with the simulation which allows to process the data in memory. While this interactive approach is definitely beneficial as it removes the bottleneck
of disk I/O and gives steering capabilities for simulation, it is sometimes not feasible to use this approach since the data is transient. For instance, scientific simulations can run for several days on said HPC systems which makes a permanent interactive visualization not possible. This is mitigated via automated batch processing, where users can specify instructions on how data should be visualized and the results stored for later analysis. However, this solution also has its limitations since these instructions must be known beforehand, restricting the exploration potential. In the second approach the results are rendered independently after the simulation has been completed. In contrary to the first approach, all data is stored on the file system and is always available for visualization but at the cost of disk I/O [65]. In the scope of this thesis, the second approach will be further investigated.

The main goal of this thesis is to narrow the gap created by the I/O performance by extending the ExaStencils framework with new I/O capabilities. With ExaStencils being able to generate highly parallel programs, parallel I/O, defined as concurrent I/O requests from processes in a parallel program, will be the focal point in this thesis. In order to achieve this goal, multiple parallel I/O techniques are examined and integrated to ExaStencil’s code generator. These new interfaces can then easily be put into practice via simple statements in the DSL code. Furthermore, a bridge between the parallel I/O interfaces and new visualization interfaces will be build. Additionally, these newly added I/O capabilities will be further put into test for simulations from the scope of the SWE. For this purpose, the Python module of GHODDESS is extended.

This thesis is structured as shown in the following. Research in relation to parallel I/O techniques is presented in Section 2. Section 3 introduces the two code generation projects of this thesis, namely ExaStencils and GHODDESS, and describes their fields of application, workflow and how I/O is currently handled. As a next step, Section 4 gives a thorough explanation for the most relevant topics in I/O, including the possible approaches and interfaces when performing parallel I/O. In accordance to the previous chapter, visualization capabilities in combination with each approach and possible visualization formats are presented in Section 5. After the theoretical part of this thesis, Section 6 explains the implementation of the parallel I/O interfaces and their collaboration with their corresponding visualization interfaces in detail. Additionally, optimization features are demonstrated. Afterwards, the various parallel I/O approaches are evaluated based on criteria described in Section 7. One of the most important criteria is performance. For this purpose, applications from the scope of the GHODDESS project are used as benchmark. In combination with the test system, a brief description of the overall test scenarios is given. Finally, the results made in the present work is summarized and further ideas that can be additionally developed in the future are discussed in Section 8.
2 Related Work

Due to its great importance in the field of scientific computing, parallel I/O has been utilized in a wide-spread range of application fields and has been thoroughly researched. Moreover, various interfaces providing different levels of abstraction were developed for the sake of performing I/O in a parallel environment. In many cases, interfaces originally intended for serial applications, e.g. Portable Operating System Interface (POSIX) I/O, are used in a parallel environment, for example by creating a separate file for each process or synchronizing the accesses to a shared file between them. However, since these interfaces are meant for serial programs using them for the often complex access patterns employed in many parallel programs is often not sufficient.

For this purpose, parallel I/O libraries like MPI-I/O have been developed to allow users to convey more information about the (global) access pattern of all processes [88]. Furthermore, high-level I/O libraries like netCDF or HDF5 have also been enriched with parallel I/O capabilities by making use of the MPI-I/O middleware layer, namely in the PHDF5 and PnetCDF libraries [18]. Both deliver sophisticated data models which allow users to specify their data structures concisely. In HDF5 files are structured hierarchically and resemble a UNIX file system. It also enables users to define and store even the most complex data structures. On the contrary, the data model employed in PnetCDF is intended to be more simple [59]. Here, the structure of the files is flat and is defined for storage and retrieval of array-oriented objects. Moreover, both libraries supply portable and self-describing file formats which contain meta-data about the stored data structures including data types, dimensions, the purpose of the data, etc. This way, the files are suitable for long-term storage. While higher-level libraries focus on the preservation of data, lower-level interfaces, which mainly focus on high performance instead of said features, are often used for data that is only needed temporarily on the same system [33]. Note that these three parallel I/O libraries only need to write to a shared file while maintaining an independent, process-local view [34, 33]. Like MPI-I/O, it does not come with an advanced data model but rather treats data like a stream of bytes. Another high-level library is ADIOS [62] which basically is a componentization tool for various standard I/O interfaces such as POSIX I/O and MPI-I/O. Here, the I/O requirements including the transport method are configured externally in a XML file. From the XML description, code is generated for the I/O commands. This way, users have only a single interface for multiple I/O back-ends while also reducing the complexity in the source code. More details on general I/O techniques and parallel I/O libraries will be described in Section 4.

Applications

All in all, software frameworks from different application fields make use of the named libraries for parallel I/O. As will be shown, in many cases not only a single I/O technique is implemented in the frameworks but also a selection where users can choose from is provided.

The widely applicable Lattice Boltzmann from Erlangen, in short waLBerla [7], software framework which is specialized for Computational Fluid Dynamics (CFD) simulations with the Lattice Boltzmann Method (LBM) provides an MPI-I/O interface to write or read to disk. One of the use cases of this I/O layer is the storage of files in the Visualization Toolkit (VTK) format which allows for multiple processes to write to a shared file while maintaining an independent, process-local view [34, 33]. Like MPI-I/O, it does not come with an advanced data model but rather treats data like a stream of bytes. Another high-level library is ADIOS [62] which basically is a componentization tool for various standard I/O interfaces such as POSIX I/O and MPI-I/O. Here, the I/O requirements including the transport method are configured externally in a XML file. From the XML description, code is generated for the I/O commands. This way, users have only a single interface for multiple I/O back-ends while also reducing the complexity in the source code. More details on general I/O techniques and parallel I/O libraries will be described in Section 4.

Applications

All in all, software frameworks from different application fields make use of the named libraries for parallel I/O. As will be shown, in many cases not only a single I/O technique is implemented in the frameworks but also a selection where users can choose from is provided.

The widely applicable Lattice Boltzmann from Erlangen, in short waLBerla [7], software framework which is specialized for Computational Fluid Dynamics (CFD) simulations with the Lattice Boltzmann Method (LBM) provides an MPI-I/O interface to write or read to disk. One of the use cases of this I/O layer is the storage of files in the Visualization Toolkit (VTK) format which allows for a visualization during post-processing.1

Another code from the domain of CFD is Code_Saturne which solves Navier-Stokes (NS) equations using the Finite Volumes (FV) method on any mesh type. In accordance to (in-)compressible flows, turbulences and heat transfers can also be simulated. As I/O layer, Code_Saturne has also employed MPI-I/O. For systems which do not support it, it provides a fall-back to serial I/O performed only by the root process. The I/O interface is also used to store visualization data in the EnSight gold format.

Furthermore, parallel I/O capabilities have been evaluated for the Computational Mechanics (CM) simulation code Alya [95] which can be used to solve PDEs on unstructured meshes using the Finite Elements (FE) methods. The work in [22] describes the addition of parallel I/O support using the parallel implementation of the Hierarchical Data Format version 5 (HDF5) and compares it to its previous I/O model where the data is communicated towards a root process which is then

1https://i10git.cs.fau.de/walberla/walberla
directly writing to disk via POSIX I/O. The HDF5 implementation has also been coupled with the eXtensible Data Model and Format (XDMF) in order to introduce an open-standard format and to enable visualization of the data. In fact, the FEniCS [53] project, a well-known computing platform for solving PDEs, also makes use of the HDF5 and XDMF combination.

However, there are also other visualization techniques used in conjunction with HDF5. Storage, analysis and visualization of VPIC\(^1\), a particle-in-cell plasma physics code, datasets with trillion of particles using HDF5 have been proposed and evaluated in [13]. Here, the H5Part library, an extension of HDF5 tailored for I/O in particle simulations, has been evaluated in comparison to an approach where each process writes its own binary file. Accordingly, a plug-in for VisIt [94] has been developed which makes use of the FastQuery [19] indexing and query system. This allows particles of interest within the massive H5Part files to be selected for visualization since rendering all is often not feasible.

In scope of NASA’s Goddard Earth Observing System model version 5, short GEOS-5, which is used for weather and climate forecasting, simulation of air temperature, etc. parallel I/O capabilities via PnetCDF and ADIOS have also been demonstrated in [61]. Here, both approaches have been compared to the prior state in which I/O was solely performed by root processes via the serial netCDF interface.

Another tool for computational science is the FLASH\(^2\) code which can be used to solve problems of astro and plasma physics, CFD, etc. Parallel I/O is implemented utilizing either the PnetCDF and HDF5 interface. Both can be used for checkpointing or to create plot files for later visualization and analysis. These I/O strategies have been evaluated in a case study [55] for both applications and different file layouts.

SIONlib has also been integrated to multiple HPC software projects. This includes modules from the DUNE [9] project, abbreviated for Distributed and Unified Numerics Environment, which provides a set of tools to solve PDEs with the focus lying on grid-based methods. One of them is the DUNE-ALUGrid [4] module which is a parallel, adaptive manager for unstructured grids and also provides means for dynamic load balancing. Here, SIONlib can be used by so-called backup and restore methods utilized for post-processing or checkpointing. Alternatively, users can use standard file streams to store the data in a binary file created for each process.

Another project is The Neural Simulation Tool, in short NEST [83], used in applications of computational neuroscience such as simulating the human brain. To collect simulated quantities such as spikes from neurons and synapses, virtual recording devices are introduced which determine if data is stored based on the activity. For this purpose, multiple record back-ends like SIONlib, HDF5 and individual text files per process are provided.\(^3\)

\(^{1}\)https://github.com/lanl/vpic
\(^{2}\)http://flash.uchicago.edu/site/flashcode/
\(^{3}\)https://nest-simulator.readthedocs.io/en/latest/guides/recording_from_simulations.html
3 Code Generation

This chapter begins with an introduction to the code generation framework ExaStencils and addresses key concepts from the realm of code generation. This introduction covers a brief explanation of the framework’s custom language and its facets, data partitioning schemes, grid types and also gives an overview of the current I/O implementation. Subsequently, a description of the most important aspects in the GHODDESS project which utilizes ExaStencils’ source-to-source compiler is given.

3.1 ExaStencils

ExaStencils [58] is a code generation framework that is capable to generate highly efficient stencil codes for geometric multigrid solvers. The framework comes with a source-to-source compiler written in Scala in addition to its own language geared towards the application of multigrid. Said solvers can be formulated in an abstract fashion which is then transformed to the target language C++. Since these formulations are more compact and concise than their generated counterpart, productivity is increased. Moreover, the emitted code is highly parallel since back-ends such as MPI, OpenMP and CUDA are supported, enabling it to run efficiently on large-scale compute clusters.

3.1.1 Numerical Methods

PDEs are commonly solved using numerical methods. In order to solve them numerically, the physical domain $\Omega$ must first be transitioned to its discretized counterpart $\hat{\Omega}$, also known as computational domain. These are often organized as grids and consist of multiple discretization points, on which functions can be approximated on. These approximations can, for example, be obtained by computing the value of the function on these points. Alternatively, basis functions which will be explained in Section 3.2 can be used. Note that the discretization points can reside in different locations within the grid. ExaStencils’ supported grid locations will be presented in 3.1.5. Systems of linear equations, formulated as a matrix equation $Ax = b$ with $x$ being the unknown, that arise from the discretization, can either be solved directly by inverting the matrix, i.e. $x = A^{-1}b$, or with iterative solvers. Despite being able to deliver an exact solution in absence of rounding errors, using direct solvers is often not practicable due to its high arithmetic complexity. Iterative solvers, on the other hand, update an initial guess iteratively until the termination criterion is fulfilled. This criterion checks if the residual term of the approximation is sufficiently small. Additionally, they allow for parallelization potential, e.g. by dividing the unknowns into sets of colours where in each set an unknown can be approximated independently from other unknowns. A well known iterative solver that makes use of this colouring is the red-black Gauss-Seidel method [49].

The multigrid method is an attractive choice to compute an approximate solution for (non-)linear systems of equations due to its complexity of $O(N)$, i.e. linear with the number of unknowns [44]. It is important to note that classical iterative solvers, like Jacobi or Gauss-Seidel, tend to have a good convergence on high frequency error components which cannot be said about their low frequency counterpart. Thus, the convergence rate decreases after a few iterations. Multigrid, on the other hand, achieves higher convergence rates despite actually using said solvers. In short, multigrid makes use of two principles. The first one is the smoothing property of named iterative solvers, often referred as smoothers, that allows the high frequency part of the error to be damped. The other principle is that a smooth function can be represented satisfactorily on a coarser grid which allows to solve it with less discretization points and thus less processing power, either directly or with so-called coarse-grid approximation schemes. The approximated error term from the coarser grid is interpolated to the original and finer grid and is then used to correct the approximate solution because the exact solution is given by the subtraction of the approximative with its error. As the name implies, multigrid is typically implemented with hierarchy of grids of different resolution. These are traversed recursively from finer to successively coarser grids in a so-called multigrid cycle. All in all, the multigrid method has a lot of algorithmic components and more importantly, the overall performance of the solver strongly depends on the constellation of those. Hence, it is necessary to facilitate the specification of said components and allow them to be interchangeable [58, 49].
3.1.2 Domain-specific Languages

As mentioned, the implementation of numerical solvers for the present compute clusters can be a challenging task since not only expertise in the application domain and the algorithmic components of the solver but also deep knowledge of the cluster’s hardware elements is required. In many cases the implementations of very specific problems including optimizations for a certain target platform are crafted by hand. These implementations are commonly written in traditional HPC host languages, e.g. C/C++ or Fortran, and can be perfectly optimized due to the low-level nature of these languages under the circumstances that this process requires a lot of expertise and time. Another major disadvantage of hand-written codes is that they require a re-optimization when ported to other target systems which can, again, be a time-consuming task. On the other hand, users can make use of scientific libraries and frameworks that provide support for many application fields and therefore grant flexibility. However, their performance is often sub-optimal since only general optimizations can be applied due to their flexibility. With the application of code generation technology in combination with DSLs, these disadvantages can be mitigated [49].

Basically, a DSL is a custom language, potentially with its own syntax and semantics, which allows for a separation of algorithm and implementation. This way, domain experts can specify their problems in a more concise way that does not involve implementation details as in hand-optimized codes which, in the end, allows for an increased productivity. The generator, on the other hand, takes the more abstract description in its corresponding DSL as input and produces source codes geared towards a specific target system. In contrast to scientific libraries which mostly employ more general optimization techniques, code generation frameworks are often tailored for a specific application domain which awakens more optimization potential. Here, not only general but also domain-specific optimizations can be applied. Additionally, for new hardware constellations, the hardware and optimization knowledge is put into the generator such that every program profits from the extension. From this point, the algorithmic description can be input to a compiler and highly optimized code is output for the new target system [77, 49].

In general, there are two kinds of DSLs, namely internal or external [77]. Internal DSLs are embedded in an existing host language, often General-purpose Languages (GPLs) such as C++ and Java, and extend it with domain-specific features including data types, e.g. fields, functions and macros while using the host language’s syntax. On the contrary, external DSLs define a new language with its own syntax and semantics making them more flexible and expressive. As a result, external DSLs have the drawback of a higher design cost, since the new language with its features and the corresponding compiler is built from scratch. ExaStencils employs its own external DSL called the ExaStencils language, in short ExaSlang, which will be discussed in the following section.

3.1.3 Workflow

Due to the abstract nature of external DSLs they often enable important concepts such as separation of concerns which, in other words, separates the description of what to compute from how it is actually done [74]. With this kind of abstraction, domain experts can formulate algorithmic problems in a fashion without having to focus on implementation particulars. ExaSlang employs a multi-layered DSL design [58, 77, 49] as depicted in Figure 1. The goal of this design is to enable users to choose a layer that is best suited for them. For this purpose, ExaSlang provides four layers with each geared towards a certain community of users. Furthermore, each layer employs its own language adjusted for the corresponding user group which allows them to specify their problems in a syntax they are familiar with. Each layer in correspondence to their expected group of users is described in the following.

Layers

Layer 1, also called ExaSlang 1, is the most abstract in the hierarchy and is proposed to be used by natural scientists and engineers with only little programming expertise. Here, problems such as PDEs to be solved are specified as a continuous equation in addition to their domain and boundary conditions. The associated syntax is similar to LaTeX. A specification with Unicode symbols is also allowed which creates the possibility of copy-pasting problem descriptions from research publications.
Layer 2 is on a lower level than the previous one and therefore a bit less abstract. At this stage, the problem’s discretized version is specified. In the scope of ExaStencils, mostly the Finite Differences (FD) or Finite Volumes (FV) methods are used. Both approximate functions with a single value per discretization point. In short, the FD method approximates differential operators, e.g. the Laplace operator $\Delta$, with linear combinations of function values at the grid nodes, typically formulated in a compact stencil notation. The FV method, without going into details, approximates integrals of the PDE over the volumes of the computational domain tessellated into so-called control volumes. In contrast to FD, the function values are typically stored in the cell-center and are constant for the cell’s volume. This layer is mostly used by mathematicians but also by advanced natural scientists and engineers.

In layer 3, multigrid has its first appearance. Here, as a supplement for layer 2, a solver algorithm for the discretized problem is specified in a syntax that resembles Matlab. This solver can be modified such that different smoothers, inter-grid transfers and multigrid cycles can be specified. Since this requires profound knowledge in iterative solvers and their algorithmic components, this layer is mostly used by mathematicians and computer scientists.

Layer 4 is the most concrete layer and holds the complete program specification. This layer follows the paradigm of procedural programming and provides common language elements like loops, functions, data types, etc. Despite that it resembles the syntax of some GPLs, especially Scala, in some parts it is still more abstract as it provides multigrid-specific language elements. At this layer, parallelization-related concepts such as domain partitioning and communication are made accessible for users. Since the general program flow including special statements for I/O, visualization, communication and performance evaluation is specified here, this layer is mostly used by computer scientists.

The platform description is orthogonal to the other layers and specifies details about the execution platform. Here, a description of the utilized hardware components and also the software, e.g. target code compilers, is given. Currently, these details are given by a configuration file, but concepts for a designated language, namely the Target Platform Description Language (TPDL), have been developed as well [78].

The only layer that was not mentioned yet is the Intermediate Representation (IR). With it being the link between the complete program description in layer 4 and the final generated code, most implementation work in the scope of this thesis is done here. Most notably, this representation can be interpreted as an additional layer in the code generation process which is not visible for users. Its role in the code generation workflow is shown in the following.

The implementation of the problem can be done by providing different components of the program on multiple levels or work on one layer exclusively and by generating more concrete specifications from it. Note that it is possible to intervene the generation process at any layer in case
that modifications to parts of the generated code are wished [48]. Most notably, a fully automatic
derivation of lower layers is hard to achieve since more domain knowledge is necessary. For this
purpose, so-called hints were introduced. These provide the compiler with knowledge such that
the discretization, the solver and the application can be automatically derived during the gener-
ation. For example, with discretization hints specified on layer 1, layer 2 can be generated. For
an automatic generation of a layer 4 specification, however, it is important to note that it requires
information from both layer 2 and 3 as they essentially are counterparts to each other. This way,
users have more flexibility as they can make use of the semi-automatic or traditional approach,
or combinations of both. Moreover, one can directly start implementing at layer 4 as it holds the
specification for the whole program and thus does not require the input from the more abstract
layers [49, 58].

**Code Generation Pipeline**

In order to explain ExaStencils’ code generation pipeline, a brief introduction to compilers is given.
A compiler can be defined as an instance that translates a source language to an equivalent program
in a target language. A general compilation pipeline begins with the lexical analysis. Here, a
so-called lexer reads the input program, discards blanks and comments, and recognizes language-
specific keywords such as identifiers, constants, etc. The program is then split into a stream of tokens
and fed to the parser where the syntax analysis takes place. Here, said input is matched according
to the rules of a grammar and mapped towards a parse tree. Simply put, the grammatical structure
of the token input is represented in tree-like form, where the nodes correspond to a construct in
the source program. Conceptually, the parser is also responsible for reporting syntax errors. The
parse tree is often transformed to a simplified counterpart, denoted as Abstract Syntax Tree (AST),
where not all details from the real syntax, like parenthesis or semicolons, are represented. After
semantic consistency is checked in the semantic analysis which, amongst others, makes sure that
in an operation the types of the operands match, compilers often generate an explicit Intermediate
Representation (IR) to represent a source program for an abstract machine. Note that during the
translation process of a program the compiler can construct sequences of IRs where one distinguishes
between high-level, i.e. close to source language, and low-level, i.e. close to target language. For
instance, syntax trees are a form of IR and belong to the former class. From here on, machine-
independent optimizations, such as dead code elimination, can be employed to improve the quality
of the target code. After this, machine-dependent optimizations can be applied to fully make use
of the hardware’s capabilities. At the back-end of a compiler, code which could be formulated
in another programming language or machine code is generated from the IR. In ExaStencils, the
former approach, that is source-to-source compilation, is employed. The emitted C++ code can
then be compiled by general-purpose compilers such as the GCC, CLANG, etc. This approach
has been chosen as it facilitates the verification and extension of the generated code compared to
machine code [2, 49].

The compiler written in Scala is the core of the ExaStencils framework. Due to the differences in
each language, each layer has its designated lexer and parser tailored to them. The same applies to
the transformations applied to the AST produced by the parser. To be more precise, transformations
can be used to add, delete or replace a node and thus are an key concept in each compiler. In
ExaStencils transformations make fully use of Scala’s pattern matching capabilities. Here, the AST
is traversed and for successful matches the specified transformation can be applied. In order to
execute transformations, strategies are required. Strategies can be interpreted as a collection of
transformations which share a common goal. Another important instance is the StateManager
which is the central entity to execute transformations and provides means for bookkeeping of the
strategies [77, 79].

In accordance to its design, ExaStencils employs a layer-related hierarchy of node types [49].
Those from a certain layer L with \( X \in \{1, 2, 3, 4\} \) inherit from the \( \text{L}X\_\text{Node} \) trait which is a subclass
from the general \( \text{Node} \) trait. While this implies that some nodes may exist on each layer, this does
not hold true for all since with more concrete layers new language elements and thus node types
are introduced. As mentioned, in the course of the generation pipeline nodes will be elevated to a
more concrete layer \( X + 1 \). Since transformations are locked to a specific layer, they cannot be used
for this purpose. Instead, the glue logic for this purpose is implemented in a \text{progress} function
originating from the \( \text{L}X\_\text{Progressable} \) trait. This function is called recursively from the root node
in order to progress a whole AST to the next layer.

The code generation process in ExaStencils can be started at any given layer \( L_X \). The AST received from the corresponding parser is then processed by the application of multiple transformations. These transformations are bundled in a so-called layer handler. Naturally, these are also layer-specific. At the end of each layer handler the nodes are progressed towards a more concrete layer. Once the layer 4 specification has either been generated or provided by the user, transformations are applied by the layer handler and its nodes are progressed towards the IR. The IR layer is the most complex as it forms the base of the generation of C++ code and also since most domain-specific transformations, including low-level optimizations like vectorization, are applied here. At the end of the pipeline the target code in C++ is output by pretty-printing the nodes from the IR AST.

In addition to the ExaSlang specifications, configuration files can be handed over to the compiler in order to influence the outcome of the generation process. These include options to specify the generator’s input and output paths in a settings file. As already mentioned, hints about the hardware and software components allowing for more optimization potential can be provided in a platform file. Lastly, the knowledge files provide parameters for the generation pipeline. This includes features to be enabled, configurations for the parallelization and other traits.

### 3.1.4 Data Partitioning

The current state-of-the-art in nowadays compute clusters is to exhibit parallelism on different hardware layers and thus to distribute both work and data on them. In the scope of this thesis, we focus on shared-memory and distributed-memory computers and their most common programming models, OpenMP and MPI. In shared-memory systems, multiple processors share a physical address space. Here, in the meaning of memory accesses, one distinguishes between Uniform Memory Access (UMA) and cache-coherent Nonuniform Memory Access (ccNUMA). In the former, all processes can access the memory with equal performance, i.e. bandwidth and latency. In the latter, the memory is physically distributed but is logically shared. For example in a multi-socket system, memory is locally connected for the processor cores in a socket. The memory from other sockets can be accessed through internal network connection. Thus, access performance for local and remote memory accesses differ. In distributed-memory architectures, processors have a private local memory which cannot be directly accessed by others. Instead, these transfer data explicitly via messages over a communication network. In today’s HPC systems, the hybrid approach is the most common in which shared-memory compute nodes, e.g. multi-socket systems, are connected in the communication network [39].

Note that this hierarchical principle can also be applied to other hardware layers and their programming paradigms. In today’s heterogeneous compute clusters, accelerator hardware such as GPUs and FPGAs is commonly present. Therefore, a rule on how data is distributed and mapped towards hardware resources must be conceptualized first. Additionally, the data between the partitions must be synchronized, typically with only neighbouring partitions in the domain. Since for performance reasons not the whole data of a partition is communicated with the others but only a certain fraction, these particular parts must be distinguishable from others, i.e. split into logical groups. In ExaStencils, there exist two partitioning concepts that enable the generation of parallel target codes by addressing both aspects [50, 49].

The so-called physical data partition describes how data is distributed across the given hardware resources. Since nowadays HPC systems are typically organized hierarchically and exploit multiple levels of parallelism, it is important for the partitioning to be flexible. However, in ExaStencils, the implemented hierarchy is rather flat and is restricted to a depth of three levels. This, on the other hand, makes the partitioning less complex and suffices for most applications. ExaStencils’ hierarchy is depicted in Figure 2. The blocks are the top of the hierarchy and can be used to represent entities in the scope of distributed memory parallelism, e.g. MPI processes. These are further subdivided into fragments, where each fragment belongs to exactly one block. These can, for example, be mapped to sockets or accelerators [58]. Finally, each fragment consists of a portion of the computational grid in form of grid nodes or cells and the data related to them, denoted as leaf elements. Technically speaking, these fragments are composed of a constant number of unit fragments, which are cubic and contain a fixed number of grid points in each dimension. Despite the fact that multiple constellations of hardware components in combination with parallelization
approaches can be represented with given implementation, in the scope of this thesis it is assumed that blocks correspond to entities from an MPI parallelization, i.e. MPI processes, and fragments are mapped to distinct OpenMP threads.

Figure 2: Hierarchical domain partitioning in ExaStencils (from [50])

For synchronization purposes, data is communicated periodically between fragments and, in most cases, only where they interface. In order to access data from adjacent partitions, the field layout is extended such that it additionally holds read-only copies of a neighbouring fragment’s leaf elements. These copies are only updated via communication and no computations take place on them. For this particular reason, it is important to tell them apart from the regular leaf elements. Therefore, ExaStencils employs a logical data partition and presents four types of elements. These are shown in Figure 3 which depicts the grid nodes of two adjacent fragments, their corresponding logical groups and their regions within the computational grid. The first group corresponds to a fragment’s regular leaf elements, denoted as inner layers. Secondly, the previously mentioned copies are also often referred as ghost layers. Another group where the elements also exist in multiple fragments are the so-called duplicate layers. They have multiple occurrences in areas where two neighbouring fragments interface. These can be communicated but, unlike ghost layers, are included in the computations and can, due to the order of operands, differ from their counterparts in other fragments. ExaStencils provides a specialized protocol to handle the variance in duplicate layers [50]. Lastly, padding layers can be enabled to prepare aligned vector operations [49]. Most importantly, these layers can be configured per dimension and are specified in field layouts.

Figure 3: Global view of grid nodes distributed over two fragments and their assignment to the logical groups: Padding, Ghost, Duplicate and Inner (from [50]).

3.1.5 Computational Grids

As previously mentioned, the simulation domain of a PDE solver is often represented by a grid, or mesh. Here, the domain is discretized into a set of cells with connectivity information. In general, meshes can be categorized to be either structured or unstructured [5]. Structured meshes can be recognized by all interior nodes and cells (or elements) in the mesh having an identical connectivity that is implicitly known. They are often associated with quadrilateral (quad) or hexahedral (hex) meshes due to advantages described in [5]. Structured grids have the advantage to be computationally efficient but are inflexible and difficult to generate for complex domains [5, 98]. Figure 4 illustrates different types of structured grids.
Unstructured meshes, on the other hand, are not bound to a pre-defined structure as shown in Figure 5 and thus need the connectivity to be explicitly defined. Although almost being used synonymously with triangle (tri) or tetrahedral (tet) meshes, these can also be composed of quad or hex cells. Due to their flexibility and thus simplicity in their construction, they are commonly used to solve problems on geometrically complex domains. However, the performance degenerates compared to structured meshes due to indirect memory accesses and high cache-miss rate [98, 29].

A hybrid concept of both are the so-called block-structured meshes, also known as multi-block structured meshes, where the domain is divided into topological blocks. Each block itself is structured but joined together in an unstructured manner. This approach provides a compromise between both purely used methods. By using the hybrid approach, the performance is better than fully unstructured grids and also gives more flexibility than fully structured grids [98].

In summary, ExaStencils supports the structured grids depicted in Figure 4. Additionally, these can be patched together resulting in a block-structured grid as shown in Figure 6. Besides the already mentioned models, there are also meshfree approaches, e.g. particle simulations, in which data is scattered and no connectivity information is required. However, these are not supported in ExaStencils [49].

Localizations

Another important aspect in the scope of grids is the localization of the discretization points as it denotes how values are attached to the grid. These are traditionally at grid nodes (Node) or cell centers (Cell). Most notably, the way data is visualized also depends on the localization. For cell-centered quantities, the value of the whole cell is represented by its value at the center. Thus, the whole area or volume of the cell is shaded in a constant color, i.e. no interpolation takes place. Node-centered variables, on the other hand, are assigned to the grid nodes and the grid cells are typically shaded by a color gradient arising from the interpolation of node-centered quantities across a cell’s area or volume [97].

Additionally, ExaStencils also supports staggered grids which introduces face-centered variables. Face-centered variables have a stagger dimension, e.g. Face_x, in which the location can be handled as node-centered, whereas for the other dimensions it can be treated as cell-centered. All supported localization types for a 2D domain are illustrated in Figure 7. Staggered grids are used to locate...
quantities, e.g. velocity and pressure in a CFD application, on different positions mitigating stability issues which would arise if those quantities were on the same locations [49, 51].

3.1.6 Fields

Fields are a common concept in numerical simulations as they represent quantities on domains. Note that in ExaStencils these can be viewed differently on each layer. For example, on layer 1 it is a continuous function and on others its discretized version. However, the focal point here will lie on an ExaSlang 4 specification. Here, a field can be interpreted as an array linked to a computational domain. In layer 4, a field specification consists of two parts: the declaration of a field and its field layout. This way, the layouts can be re-used resulting to fewer declarations. Field layouts contain information about its data type, localization and the previously described logical data partitioning. For scalar fields, simple data types such as floating point values (Real) can be used, whereas for others, aggregate data types such as Vector or Matrix are employed. Note that in layer 4, definitions like the field layout are considered for a single fragment as this approach allows a representation independent on the number of fragments in use. In addition to its layout, a field’s domain and boundary conditions must be set. Listing 1 shows an exemplary field specification for each multigrid level. Here, the layout describes a node-centered vector quantity with communication on ghost and duplicate layers enabled. The field is declared on a 3D unit cube with boundary conditions set to zero [49, 80, 77].

Listing 1: Field and field layout declaration of a vector field in 3D

```
Domain global< [0.0, 0.0, 0.0] to [1.0, 1.0, 1.0] >

Layout NodeVecWithComm< Vector < Real, 3 >, Node >@all {
    duplicateLayers = [1, 1, 1] with communication
    ghostLayers = [1, 1, 1] with communication
}

Field Solution< global, NodeVecWithComm, 0.0 >@(all)
```

Another concept that stemmed from the integration of non-uniform grids are so-called virtual fields. These are language elements which can be used to retrieve the position of neighbouring grid points and cell centers, the cell widths and more. Ideally, when a user specifies a program in the DSL, code can interchangeably be generated for different grid constellations. However, different amounts of data must be stored depending on the specified grid. For example, the node position of uniform meshes can easily be determined with a single expression due to the constant cell width, whereas for non-uniform and axis-aligned meshes, the node positions on each axis must be specified, i.e. an array for each dimension. Therefore, ExaStencils minimizes the memory to be allocated and only stores the information when necessary. Hence, not actual but virtual fields are introduced. Note that these are not to be specified by the user. For example, `vf_nodePosition_x@current@(1, 0, 0)` can be used to retrieve the x-position of the right neighbour in x-dimension. The compiler resolves accesses to these fields and replaces them appropriately.

3.1.7 Former State of I/O Interfaces

In this section, statements that are responsible for performing I/O will be presented. Note that this only includes interfaces where the enhancement with parallel I/O capabilities would be the most beneficial. That is, when data is accessed frequently and whenever it is possible to access large aggregates of data in few requests. One possible example where implementing parallel I/O would have only little or even no gain is when reading parameter files. Since these are commonly only read once during the setup phase of a simulation and are typically in the range of several kilobytes. For simplicity, it is often implemented that only one MPI process reads the file and broadcasts the parameter values to the other involved processes. Figure 8 shows the functions which are expected to benefit from an integration of parallel I/O interfaces.
As can be seen, ExaStencils does not have a designated I/O package and the selection of I/O classes has different origins. This is due to the fact that these are utilizing the stream-based I/O library from the C++ standard. This I/O library provides a simple and yet flexible way to implement I/O routines. In the generator specifically, the stream operators have designated nodes in the IR, namely \texttt{IR\_Print} and \texttt{IR\_Read}, which are currently used to perform file I/O and write to console by simply feeding them with a sequence of expressions.

Most of the presented I/O statements implement the \texttt{IR\_Expandable} trait. It has been introduced to reduce the sheer number of transformations that would have been necessary to transform an AST node to another type of node(s), thus called expansion. Since the whole AST needs to be traversed to perform this action when applying transformations for each of those, it is be more efficient to just identify nodes to be expanded and perform the expansion in one go. Nodes that make use of this interface must implement an \texttt{expand} function which returns either one or multiple \texttt{IR\_Node} instances or \texttt{None} in case that the node is simply removed [49].

As already mentioned, ExaStencils employs hierarchical data partitioning by splitting the global domain into fragments. Thus, information such as the adjacency between the fragments must first be made available in the simulation. ExaStencils employs two ways, configurable per knowledge parameters, to generate code that provides this kind of topological information. The first is only available for rectangular domains. Here, a fragment’s position and connectivity to other fragments in the global domain can be automatically determined due to the geometric properties. For other domain types, however, this information must be made available explicitly. This marks the second approach and is done by reading data from disk. Hence, a function in the target code is generated from an \texttt{IR\_InitDomainFromFile} node [49].

Other from that, the remaining functions with their corresponding signature shown in Listing 2 can be called in ExaSlang 4. On the left side are statements to perform I/O for fields, whereas on the right side a visualization for a specific application can be output.

The I/O functions from the field package have different use cases. The \texttt{IR\_ReadField} function is used to initialize a field with the data stored in a given file. For MPI parallel simulations, each MPI process opens its own unshared file and reads the field values from it. \texttt{IR\_PrintField}, on the other hand, can either be used to store plain field values or to enable rudimentary visualization capabilities. For the latter, a Comma-separated Values (CSV) file is printed where in addition to the field values, e.g. vectors, their corresponding location within the grid is given. Tools like ParaView

![Class hierarchy of candidates to be extended with parallel I/O.](image-url)
are fully capable of reading and visualizing such files. However, note that no geometric indicators of the grid are specified and in many cases more positional data is provided than necessary. For example for uniform meshes, only the origin and spacing per dimension must be specified to derive the position of all grid nodes. Another difference to the `IR_ReadField` statement is how I/O is performed. Instead of creating exclusive files for each process, they write into a single shared file with the application of a locking mechanism which basically allows a process exclusive access the file. Note that multiple function names exist for the `printField` and `readField` method as highlighted with the brackets. For example, when calling `printFieldWithGhost` a field's ghost layers will be included in the output.

The VTK interfaces are responsible for supplying a fully fledged visualization of a certain kind of application by providing the data in the VTK format. The supported applications are solvers for the Navier-Stokes (NS) equations and the SWEs. Here, a pre-defined set of fields is selected and visualized on unstructured grids. Depending on the application, different geometries are used: triangular (2D) for SWE and quad or hex meshes for NS. Moreover, these interfaces also make use of the locking mechanism to handle I/O in a parallel environment.

```plaintext
// field.ir
def printField[Binary][Values][WithGhost](
    field: IR_Field,
    filename: IR_Expression = field.name + ".txt",
    condition: IR_Expression = true
)

// field.ir
def readField[WithGhost](
    field: IR_Field,
    filename: IR_Expression = field.name + ".txt",
    condition: IR_Expression = true
)
Listing 2: ExaSlang 4: Primary I/O statements that frequently access large amounts of data

3.2 GHODDESS

The Generation of Higher-Order Discretizations Deployed as ExaSlang Specifications (GHODDESS) project aims to further extend the ExaStencils toolchain with higher-order discretization schemes such as the quadrature-free Discontinuous Galerkin (DG) methods. For this purpose, a Python interface to specify discrete DG formulations and build the bridge with ExaStencils in addition to a grid generator that is capable of creating grids for more complex domain geometries has been implemented.

3.2.1 Overview

For realistic simulations in the application fields of oceanography, meteorology and coastal engineering efficient and scalable solver implementations are a necessity. For these kinds of applications, it is often the case that the vertical scales of motion in a fluid is relatively small in comparison to the horizontal ones. Thus, the SWE can be used as a model. While there are many ways to discretize and solve them, one may see that there is a vast difference between its discretized formulation and final, optimized implementation. Hence, this again requires profound knowledge in numerical methods, hardware details and programming. As shown in the previous section, one way to bridge this gap is to use code generation technology. In fact, the generation of SWE solvers using a Finite Volumes (FV) discretization has also been evaluated in the scope of ExaStencils [48]. Additionally, adding support for DG discretizations would be beneficial due to their discontinuous nature which is depicted in Figure 9. Here, each triangular element has independent vertices. A grid node, however, bundles vertices at the same position from adjacent elements. The number of vertices per node depends on the number of elements attached to the same node. Note that for this discretization scheme, the function values are not solely approximated by evaluating a function at a certain discretization point, but by a linear combination of so-called basis functions with a collection of coefficients. Here, the number of basis functions is dependent on the degree of a polynomial. These basis functions do only have influence within a specific region, e.g. a triangular cell.
Thus, state variables within an element are approximated with a contiguous function but there is no inter-element continuity required. The discontinuities between the elements enable many possibilities. For instance, complex geometries can be handled since elements do not need to match up at edges. Additionally, this enables a high degree of adaptivity such as different polynomials for the elements and locally refined meshes. Furthermore, its implementation is highly parallelizable [23, 52, 21].

However, ExaStencils does not support higher-order discretization schemes like DG. Instead of directly incorporating the formulation for such methods into the compiler, the GHODDESS Python module which translates the DG scheme to an ExaStencils specification first was developed. This scheme is formulated in a set of symbolic SymPy expressions. SymPy [64] is a well-known Python library for symbolic mathematics. It provides many features such as simplification of expressions, integration and differentiation of functions while allowing the terms to be expressed in a concise and mathematically close fashion, thus, increasing productivity. In order to enable the translation to ExaSlang, the symbolic expressions were extended with concepts from ExaStencils, such as fields [28, 29].

3.2.2 Grid Types

In realistic ocean models, geometrically complex domains are commonly found. For accurate representations of these domains, using fully structured grids is often not feasible. As previously mentioned, block-structured grids grant more flexibility while being computationally more efficient than unstructured meshes. In the scope of GHODDESS, block-structured grids can be automatically produced by a grid generator. Here, an unstructured triangle grid (2D) is simplified and a block structure is derived. Each block is then uniformly refined such that each block is structured. Additionally, the boundary and element density is adapted under certain constraints. In the end, a block-structured mesh consisting of either quadrilateral or triangular elements is output in a mesh file. Note that in ExaStencils triangular meshes are not directly supported. However, it is handled such that quadrilateral elements are logically divided into two triangles with distinct orientation, namely lower and upper triangle.

3.2.3 Workflow

The complete code generation process in the scope of the GHODDESS project is depicted in Figure 10. The start of the toolchain begins with a quadrature-free DG formulation, manually derived from the continuous problem, which is implemented in the Python module where classes for concepts such as triangles, basis functions and fields are already provided. The integrals and derivatives in the formulation are then solved analytically by SymPy. From here on, the symbolic expressions are mapped to a valid ExaSlang specification which can then make fully use of ExaStencils’ optimization and parallelization potential.

The resulting simulation then potentially reads a mesh file, i.e. via IR_InitDomainFromFile, created by the grid generator in order to initialize geometrical and topological information for each
fragment. Additionally, values of auxiliary fields are read from file, i.e. by the layer 4 `readField` function. Over the course of the simulation results and visualization files are output to disk for further post-processing. For this purpose, GHODDESS makes use of the aforementioned print statements.

Figure 10: Code generation toolchain employed in GHODDESS (modified from [28])
4 Parallel Input/Output

This section is divided into two parts. The first is more about the theoretical background and gives insight to general aspects from the scope of parallel I/O. The second part is rather application-oriented and introduces I/O interfaces which are commonly used in many HPC frameworks.

4.1 Background

As previously mentioned, simulations access data from disks for various reasons. However, the characteristics of I/O vary from application to application. One of the more straightforward properties is the size of data that is transferred between main memory and disk. Commonly, a simulation only reads small data inputs at the start-up phase and generates large amounts of data from it, but how much data is transferred depends on the I/O needs. For instance, when checkpointing is used, a snapshot of the simulation state is stored on disk to ensure fault tolerance. The data extents of this snapshot are in the same order of magnitude as the data held in memory. In other cases, only simulation results without its byproducts and thus only a small fraction of the data in memory is stored. Another important factor is the frequency of the I/O operations. Checkpointing and storage of results occur periodically, whereas reading input files typically occurs once. In order to minimize the time spent in I/O, the integrity of data is often compromised by reducing the data extents or frequency of the accesses.

In this section, a more thorough introduction to I/O characteristics is given. This includes a description on how the accessed data can be organized, an illustration of typical I/O architectures, parallel file systems and a presentation of techniques to perform I/O operations in a parallel environment.

4.1.1 Data Representation and Layout

Depending on the use case of the files in- or output in a general application, the way in which values are represented in the file plays an important role. Here, we essentially distinguish between binary and ASCII files. The American Standard Code for Information Interchange (ASCII) standard enables encoding data values to a human-readable and machine-independent representation in text form, whereas in the binary case data from memory is stored to file as-is. Note that for an ASCII representation data needs to be converted from binary first. Additionally, ASCII files generally require far more disk space than their binary counterpart since each digit of a human-readable numeral is represented by a character. Thus, ASCII files cause a performance penalty. Files in binary form, on the other hand, are more efficient but machine-dependent. Here, porting and using these on another system can prove to be cumbersome since the byte-order, or *endianness*, differs depending on the architecture [33].

Subsequently, meta-data plays an important role for portability. By providing meta-information of the data types, the structure of the data, e.g. dimensionality of an array, the format of the file, e.g. binary, the endianness and many more, these files can be used for multiple purposes. In contrast, for files without meta-data applications need to know this information beforehand and often explicitly define it in manners of direct coding. However, these files are mostly meant for a single application and are thus not reusable. Meta-data can either be directly incorporated into the same file or specified in an additional file, e.g. in XML format. High-level file formats like HDF5 or NetCDF make use of the former approach. These employ so-called self-describing formats which, in addition to the aforementioned meta-information, contain application-related meta-data such as the meaning and structure of data. Moreover, these can be further annotated with descriptive attributes. Thus, these files offer portability and longevity [33].

As shown in Section 3.1.4, data can be distributed among multiple compute resources. With this kind of partitioning, the question arises in which order the data should be stored. In serial programs, data is typically written in contiguous chunks, whereas in parallel programs the processes access the (shared) file simultaneously but have dedicated regions of the file assigned to them. In the scope of this thesis, we distinguish between two kinds of file layouts which are depicted in Figure 11. The first one will be called *fragment-wise order*, in which each MPI process is given a contiguous slice of the file space which is then partitioned contiguously among its fragments. In the figure, each process has a single fragment. This, in return, means that a fragment’s data is not only laid
out linearly in memory but also on disk. The global start position of a fragment’s data is dependent on the index of the process and its index among the process-local fragments. The second one will be denoted as canonical order and is meant for rectangular domains. Here, the values are stored as if the domain was never partitioned. As shown in the figure, the mapping between the memory and file space is not as trivial as in the fragment-wise order. While a fragment’s data is laid out contiguously in memory, this does not hold true any more when mapped to the file space of the partitioned 3D domain. Thus, for this layout data needs to be rearranged such that it is stored in the geometrically correct order.

Both approaches have their advantages and disadvantages. For instance, the fragment-wise storage usually leads to large and contiguous file accesses and thus is expected to have good performance. However, post-processing with such files can be cumbersome since the data needs to be potentially reordered, e.g. when restarting the simulation using a different number of fragments. For this purpose, the number of fragments and their dimensionality need to be known. In the canonical order, on the other hand, data is stored independently from the partitioning into fragments and can easily be redistributed again. However, storing the data in this layout can prove to be an additional cost since each process accesses multiple small and non-contiguous portions of the file. Without employing the correct techniques for this matter, the performance is significantly reduced. For this purpose, parallel I/O libraries provide special optimizations that promise a similar performance as a fragment-wise order [33, 60].

4.1.2 Parallel I/O Infrastructure

As previously mentioned, nowadays compute clusters exhibit parallelism by distributing work on multiple hardware resources. Here, we distinguish between shared and distributed memory systems. In most modern HPC systems, a hybrid approach is utilized where shared-memory compute nodes are connected in a communication network, i.e. memory is shared on the compute nodes but not between processors located on different nodes. Often, compute nodes do not have local hard disk drives available to applications, as is the case for the Emmy cluster. Moreover, in case that local disks are employed, these are still not accessible for other compute nodes [88]. Thus, in most cases the compute nodes store data on a global, distributed file system accessible with use of network protocols [69].

Basically, a file system can either be attached internally or externally to a parallel system [33, 69]. In the former case, file system nodes interface a supercomputer’s interconnect network directly.

---

Figure 11: Global storage and file layouts of a 3D domain decomposed into eight fragments

---

https://hpc.fau.de/systems-services/systems-documentation-instructions/clusters/emmy-cluster
Moreover, there are also file systems accessible from multiple supercomputers, where each may use a different networking technology. These center-wide file systems are often connected via network switches or bridge nodes to the supercomputers [33]. Externally attached file systems are connected with so-called I/O nodes which are linked to the compute nodes’ communication network. In essence, the function of the I/O nodes is to act as servers for the parallel file system [88].

For architectures employing I/O nodes, compute nodes often do not have direct access to the file system but I/O requests are forwarded through the I/O nodes where the file system calls are then invoked. In other words, the I/O nodes operate on behalf of the compute nodes and forward their requests towards the file system. This allows compute nodes to employ reduced versions of the OS kernels, whereas the I/O nodes run fully functional ones. Typically, the processing elements of I/O nodes are similar to the ones from the compute nodes. Moreover, a selection of compute nodes is associated with I/O nodes, e.g. an I/O node to compute node ratio of 1:8 to 1:64 for IBM Blue Gene/P. This delegation of I/O requests to dedicated I/O nodes is known as I/O forwarding and can reduce network traffic as the I/O nodes aggregate, reschedule and cache I/O requests. Furthermore, from an application’s perspective this concept is applied transparently [3, 56].

Figure 12 depicts an abstract I/O infrastructure of HPC systems with external file systems. In [33], this infrastructure is explained using the notion of an abstract I/O node. In accordance to the input and output streams each abstract I/O node has, it also comprises an internal buffer for the data input or output which is asynchronous to the actual disk I/O. Likewise, file systems follow the same principle. As will be explained, parallel file systems often consist of several server nodes which are connected to the file system clients, i.e. compute or I/O nodes. The server nodes are attached to individual or an array of disks and, as they transfer data between disk(s) and clients, their memory is equivalent to an abstract I/O node’s internal buffer. Overall, parallelism in parallel I/O architectures is established with multiple I/O nodes, file system servers and storage devices [88].

**4.1.3 Parallel File Systems**

Parallel File Systems (PFSs) are another pillarstone from the scope of parallel I/O, as these provide global access to data from all compute nodes of an HPC system. The file system by itself consist of multiple server components which are interfaced by the clients, e.g. compute nodes. Supercomputers often make use of the Network File System (NFS) which allows a file system to be accessible from different computers of an HPC site. This enables the users’ home directories to be accessed directly from other machines. Despite all conveniences, NFS is slow and not meant for parallel access from multiple clients, but its home directory is typically used to store configuration files and source code files. However, using high-performance file systems, such as GPFS [81] and Lustre [6], during the actual job execution is often mandatory for good I/O performance [42, 88, 33].

Most of the parallel file systems are compliant to the POSIX standard and thus provide clients a familiar interface. The I/O interface of POSIX provides many well-known functions, such as `open()` and `write()`, and command-line tools that can be used to obtain services from the operating system. POSIX was developed when a single machine had its own file system. Moreover, it employs
Figure 13: Lustre file system architecture (from [86])

semantics to ensure the atomicity for write accesses and read-after-write consistency which can impact the overall performance in parallel applications. There are file systems which either sacrifice this consistency or implement more relaxed versions of it, e.g. NFS which guarantees consistency after closing a file [33]. However, developers need to be aware of this behaviour. Nonetheless, for convenience most parallel file systems are POSIX-compliant. In order to enable parallel access, these make use of the so-called striping technique. Here, a file is split into chunks and is distributed across several servers for storage which allows clients to access the chunks concurrently [69, 85].

Since the Emmy cluster utilizes a Lustre variant as parallel file system, Lustre will be introduced in the following.

Lustre

Lustre [6, 33, 86] is a high-performance file system for Linux supercomputers and is implemented entirely in the Linux kernel. In basic terms, Lustre separates the storage of meta-data from the storage of block data. For this purpose, it implements a distributed object-based storage. These objects can be categorized as data objects, which represent byte arrays stored in files, or index objects, which are in form of key-value pairs and are used for the storage of file system inodes. As shown in Figure 13, data is stored on so-called storage targets which can either be represented by Meta Data Target (MDT) units used for namespace operations in the file system or Object Storage Target (OST) units meant for storage of file data.

One or more OSTs are exported per Object Storage Server (OSS) node which interfaces with the HPC system’s network and is responsible for handling data retrieval and storage. For data storage each OST maintains its own local filesystem and is attached to a number of disks, e.g. in a RAID system. In addition, each OST operates independently from the others. In Lustre, parallelism is granted by striping a file over the OSTs. Striping is performed such that a file is distributed into chunks that are assigned over a selection of OSTs in a round-robin manner. Lustre provides a set of command-line tools to inspect or set a layout. In order to set the layout for a file, the client can call the `lfs setstripe` command and specify the number of OST stripes, the stripe size and the OST storage pool. However, Lustre clients can not only define different file layouts for individual files but also for parent directories. The total storage capacity of a Lustre file system is determined by the aggregation of the OSTs’ capacities.

Likewise, one or more MDTs are exported by Meta Data Servers (MDSs). The MDTs store the filesystem namespace, e.g. directories or filenames, as well as attributes for the file access, e.g. permissions or ownership, for the file layout, etc. However, unlike OSSs, which employ an active-active failover mechanism, Lustre only utilizes an active-passive failover for MDSs where only one is actually active and the others are used for redundancy [6]. The corresponding MDS controls the selection of OST objects that store a file. Unless the client specified this selection directly, the MDS assigns OSTs to files for load balancing [6]. This, in return, means that an MDS needs to track the
storage locations of a file in association with the correct set of OSTs and their OSSs.

The Management Server (MGS) is responsible for managing configurations for client and server nodes. It also logs the configuration information and notifies clients on server restarts and configuration changes.

Technically speaking, a Lustre client combines the objects exported by the data and meta-data targets to build a single coherent file system from the cluster of Lustre servers. When creating or accessing a file, the aforementioned storage locations, which are requested from the MDS, are then used to directly access the OSSs and OSTs for I/O operations. Once a file is opened, the MDS is not involved in I/O until the file is closed, which requires meta-data updates for the file on the MDS.

In order to ensure consistency for concurrent file and meta-data operations, Lustre employs a distributed lock manager which serializes conflicting file system operations and ensures coherent caches for the clients. There are different modes to protect against concurrent accesses such as protected read locks to allow read-only accesses to a shared file, exclusive locks for a parent directory’s inode requested by the MDS when creating a new file, byte-range locking to support shared file I/O and many more [96]. Each storage target makes use of this concept such that the locks are located with the objects to be protected improving the scalability of concurrent locking [6].

4.1.4 Parallel I/O Patterns

In general, parallel I/O can be defined as simultaneous write or read accesses of shared data which are performed by multiple processes in a parallel application [88]. However, there are different approaches when performing parallel I/O [18, 59, 33, 69]. Figure 14 depicts three common I/O patterns employed in parallel computing.

![Figure 14: General I/O patterns in parallel applications](image)

The first and conceptually most straightforward one is the Single Writer I/O pattern shown in Figure 14.a. In principle, there is one designated process, typically the root process, in charge of performing I/O operations to a single file via a serial I/O API. In other words, this process carries out the I/O requests on behalf of the others and is therefore also required to aggregate and distribute data when accessing the file. However, one problem of this approach is the writer’s limited memory capacity which might lead to multiple gather/scatter operations to aggregate/distribute the data for file accesses [34]. Due to its limited I/O capabilities, the single writer becomes a critical resource and prevents the other processes from making progress [54]. While this approach is simple, it is only reasonable for small datasets and will certainly result in a bottleneck in the realm of large-scale application runs, where thousand-way parallelism is routine [54, 33, 69]. For these reasons, this pattern will not be investigated any further.

Alternatively, it is also possible that all processes perform File-per-process I/O. With this approach, each process accesses a separate file. Since no files are shared, no synchronization of the I/O operations is required in the application [33, 88]. Moreover, in contrast to the previous approach,
no data shipping between processes as in the previous pattern is required. Hence, the processes can perform their I/O independently with each using a serial API, making this method easy to implement. Despite all these conveniences, this method also has severe drawbacks. One disadvantage is the management of the datasets spread across a large number of files [18, 34]. For example, when the number of processes that read the file is different from the number of processes when the files were originally created, additional effort to redistribute the data is required [60]. Note that this problem is similar to the storage of a single-shared file in fragment-wise order. Additionally, copying files to a tape archive can be slowed down due to the large number of files. However, merging these into a single file is time consuming and also challenging since the storage space is temporarily duplicated during the merge process. Besides the difficult management of such files, large-scale file operations can lead to side effects such as temporary disruption of services for other users on the overall system [34]. Therefore, the number of files that users can possess are often limited. This approach can achieve promising I/O bandwidths, however, operations such as parallel file creation can be very costly because the simultaneous creation of thousands of files in the same directory may lead to a serialization arising from meta-data contention. To ensure consistency, file systems need to serialize the access, e.g. via locking, to a directory's inode which stores pointers to the files' inodes and thus is a shared resource, especially in Lustre where all inodes are managed by one MDS [34, 33]. All in all, this approach can achieve high bandwidths but has scalability and file management issues.

The last pattern is the so-called Single Shared File I/O. In this pattern, all processes access the same shared file. In order to prevent conflicts when accessing the same file regions concurrently, coordination is required. For this purpose, either a temporal or spatial distribution can be applied [33]. The former describes a procedure where the access to the file is serialized among the processes and equals the locking mechanism described in Section 3.1.7. The latter denotes a mapping of non-overlapping portions of the file to each process. Moreover, in case of simultaneous accesses to the same file regions, they are serialized by the file system by a lock manager to guarantee file consistency [69, 33]. With only one file, both problems of the previous access pattern are overcome, i.e. data management will be facilitated and meta-data contention is mitigated. Furthermore, most parallel I/O libraries make use of this approach and also provide semantics for parallel access [18]. Moreover, they supply collective buffering methods which basically move the coordination work from the file system up to the application-layer [69, 13]. Here, a portion of the processes is responsible for aggregating the data from other processes and cluster them into large contiguous chunks which are then written to the file system. With less processes writing to the file, contention for the underlying I/O infrastructure is reduced. Overall, this approach can achieve high performance if requirements such as sufficient I/O hardware, a parallel file system and an efficient implementation of the I/O interface are fulfilled [60].

4.2 Parallel I/O Interfaces

In this section, multiple parallel I/O interfaces are introduced including their interaction with other software layers. The main goal in here is to illustrate their core characteristics and application domains. A further comparison of the selected interfaces in the scope of ExaStencils and GHODDESS can be found in Section 7.

4.2.1 Outline

In accordance to the different access patterns, it is important to get an overview of the interaction between an application and the (parallel) file system. For each access pattern, there exist various software layers with each providing different levels of abstraction and technologies. As a result, the concept of an I/O stack has arisen. An exemplary I/O stack that illustrates the covered I/O interfaces can be found in Figure 15.

At the bottom of the stack is the (parallel) file system which in most cases is POSIX-compliant. Next to POSIX I/O, there are also similar I/O interfaces which are typically realized on top of POSIX I/O [33]. For instance, the C language provides interfaces from the ANSI C standard, e.g. the `fwrite()` function, whereas in C++ streams are provided. Both treat file data as streams and buffer data to reduce the number of system calls. In a parallel application, however, these APIs lack constructs that convey the I/O intention of all involved processes. Instead, these perform their I/O
operations in an uncoordinated fashion. As already shown, there often exist data layouts in which processes write in an interleaved and non-contiguous manner. With low-level abstractions, each process would need to seek and access each of its portion of the file individually, resulting in many small and non-contiguous file accesses. Thus, these interfaces are better suited for non-interleaved accesses, e.g. file-per-process.

Middleware layers such as MPI-I/O provide more suitable APIs for parallel HPC applications. Here, users can convey all information about the access patterns of the processes to the interface. For this purpose, the middleware layer provides abstractions that allow the global access pattern in a shared file to be formulated conveniently. Additionally, with this kind of information optimization techniques are made possible.

On the top of the stack, there are high-level libraries such as HDF5 and PnetCDF. In contrast to lower-level interfaces, these supply data models and APIs with abstraction levels well above mere byte sequences. Thus, users are able to describe their data objects in a more natural way than with lower-level interfaces. Additionally, these employ self-describing file formats which allow for a long-term storage and portability. Most of them also make use of the middleware layer to provide parallel I/O capabilities.

4.2.2 MPI-I/O

MPI-I/O [60, 88] is a portable interface for parallel I/O and was defined as part of the MPI-2 standard. Furthermore, it can be used in languages such as C, C++ and Fortran. As the middleware of parallel I/O, it can either directly be used by application programmers or is used under the hood of high-level libraries. While MPI-I/O could theoretically also be used in a File-per-process manner, it is originally intended to be used by MPI parallel programs which read or write parts of a single common file [60]. The same can also be said about the high-level libraries which sit on top of MPI-I/O. There are multiple implementations of the MPI-I/O standard which can be portable or vendor-specific [88]. The most popular implementation is ROMIO [76, 35] as it is part of various MPI distributions such as MPICH [66], OpenMPI [72], etc. Thus, all optimization techniques and other features presented in this chapter are explained in the context of the ROMIO implementation. Moreover, the implementation of MPI-I/O is based on an interface for abstract devices called ADIO, which itself has separate implementations optimized towards different file systems and grants portability for MPI-I/O implementations such as ROMIO [89].

Features

One important aspect is to specify the location in the file at which the I/O operations take place [60]. MPI-I/O offers three different options. The first one is to use individual file pointers. Here, each process stores and maintains its own pointer independently. Just as in regular POSIX I/O, the
pointer can be set explicitly and is updated automatically when performing read or write operations. Another possibility is to specify the position in the file directly via explicit offsets. Here, the file offset is passed as an argument when accessing the file. In this case, the individual file pointer is not updated. The last method is to use a shared file pointer which is basically a common file pointer shared among the involved processes. Similar to the individual file pointer, the file pointer can be moved and the pointer is updated when reading or writing the file, but in a synchronized way [57]. One typical use case for this method is to write log files [60]. However, maintaining such a pointer can result in some synchronization overhead and is therefore for performance reasons not advised.

In order to convey more information about a user’s I/O intentions to MPI-I/O, file views have been introduced [88, 60, 35]. These essentially specify the region of a file which is visible to a process. This way, each process can partition the file logically. In order to describe such partitioning, MPI datatypes can be used. These essentially embody access patterns of the memory or a file and can express regular or irregular data layouts. In general, MPI datatypes come in two flavours: basic and derived. The basic datatypes are analogous to primitive datatypes of the used programming language, e.g. integers. The derived datatypes, on the other hand, consist of multiple basic datatypes which are either located contiguously or non-contiguously in file or memory. In addition, derived datatypes can also be constructed recurrently from others. Therefore, any possible data layout can be expressed. Initially, when a file is opened, the entire file is visible to each process. The MPI datatypes can then be used set their view on the file. Once the file view is set, additional MPI datatypes can be used to describe the access pattern in memory when reading or writing a file, e.g. regions without ghost or padding layers. As shown in Section 4.1.1, a process often writes to non-contiguous portions in the file. With the view set accordingly, only the specified regions in the file are accessed and any gaps are skipped. However, these non-contiguous regions are then logically combined into a contiguous region which is finally written or read as if the view was a contiguous stream of bytes.

MPI-I/O supports multiple data storage representations [89] of binary data which can be provided as a string literal when calling the appropriate function to read or write a file. The first one is the "native" representation and basically means that data is stored in the file as it is in the memory, i.e. no conversion happens. The "internal" representation is implementation-defined, i.e. a degree of file portability is ensured when the same MPI-I/O implementation is provided. On the other hand, files written with "external32" are guaranteed to be portable with any MPI-I/O implementation. However, this approach has its limitations. Within MPI-I/O, converting datatypes from or to this representation can cause byte swapping, truncation or padding. Consequently, precision loss can also occur. Additionally, it is possible to specify user-defined representations by providing data-conversion functions. However, this kind of portability is only ensured when using the MPI-I/O layer and is only focused on the binary representation of data types. That is, the structure of the file itself, such as the kinds of data structures written including their dimensions, datatypes, etc. must either be implicitly known, i.e. directly specified in the program code, or must be provided in form of meta-data. Meta-data can either be incorporated in the file itself, e.g. high-level formats like netCDF or HDF5, or in an external file, e.g. XML. Thus, files without meta-data are mostly dedicated to a single application. On the contrary, the addition of meta-data makes a file truly portable and allows re-usage in other applications, e.g. visualization. Therefore, it is important to further analyse additional means to ensure portability.

In MPI-I/O and therefore also in high-level interfaces, data is moved between processes and files via read and write function calls. Most notably, these data storage operations can either be performed collectively or independently [60]. In case that the independent option is chosen, a process can call these functions independently from others. As a result, the function calls of a process do not need to be matched with calls on other processes and there is no limit on the number of calls a process can make. For the collective variant, on the other hand, all processes that collectively opened the shared file need to participate in a function call. The participants can be identified by the MPI communicator that was passed as an argument when opening the file. Since all processes are involved in these functions, collective function names end with _all. While the independent functions are less restrictive and require no function call matching, i.e. synchronization, the collective function calls provide a bigger picture of the user intent to the MPI-I/O library. Since each process is involved in the I/O operation, access information can be exchanged and I/O requests can be reorganized among the processes. Hence, this kind of collaboration enables I/O optimization techniques which will be presented in the following.
I/O Optimization Techniques

MPI-I/O presents two major optimization techniques in the scope of accessing distributed arrays from files [90, 88, 60, 35]. Both aim to improve the performance by reducing the number of small, non-contiguous accesses to the file. I/O typically has a high latency which, amongst other things, arises from the overhead of system calls.

The first optimization is the so-called data sieving [88]. This optimization allows an application to request few, contiguous chunks of data from the file system despite the fact that the user wants to access multiple, small and non-contiguous parts of the file. For such access patterns, instead of making a request for each non-contiguous piece of data separately, a single I/O request over a large file segment, i.e. the size of the sieve, can be made even if it contains data which was not requested. This segment can then be stored into a temporary buffer and the data of interest is copied to the user’s buffer. Despite the fact that more data than originally requested is transferred, the performance improves due to the reduced number of I/O requests [60]. Figure 16.b illustrates an example of data sieving applied when reading data from a file. As can be seen, instead of accessing the file seven times, four data requests in the size of the sieve are made and the unwanted data is omitted. When writing to a file, data sieving applies a read-modify-write mechanism to make sure that data between the contiguous data segments is not destroyed. Additionally, the accessed region of the file must be locked to block concurrent updates and ensure file consistency [60].

The second technique is the Two-phase I/O [60, 90, 35], also known as collective buffering. Two-phase I/O essentially exchanges data between processes and rearranges them such that they can be processed by the file system with the best performance. As the name implies, this approach is conceptually divided into two phases [60]. The first is the request aggregation phase in which a subset of processes is selected as I/O aggregators. Their task is to serve as I/O proxies for the remaining processes. Firstly, the whole file region aggregated from the collection of all I/O requests is split up amongst the set of aggregator processes into contiguous chunks, also called file domains. Then, the other processes transfer their data from or to the aggregators where the data is also rearranged. Finally, in the file access phase, each I/O proxy will interact with the file system by making contiguous I/O requests. In addition, the aggregator processes are a subset of the communication group which further prevents a congestion of the I/O subsystem. In other words, a file domain is a buffer on an aggregator process which also holds data on behalf of other processes. For instance, when reading a file, the aggregators firstly read their respective file domain from file and then redistribute it to the corresponding locations. Moreover, when writing a file, data from the other processes is first collected and redistributed before it is written to file. Figure 16.a depicts the process when writing to a file via two-phase I/O. Here, processes P0 and P2 were selected as aggregators and the globally accessed file region is split amongst them. As can be seen, the data from the other processes is shuffled and communicated to the respective file domains which can then be written as contiguous chunks. The advantage of this method is that the price of the additional communication to rearrange the data is typically low compared to the I/O time that is then saved. Therefore, the performance is close to what could be achieved with large I/O requests made in parallel [88]. As can be seen, configuration capabilities are essential for both optimization techniques. For instance, the choice of the sieve size and the number of aggregator processes can have a critical impact to the performance. For this purpose, MPI-I/O introduced a hinting mechanism which can be used by users to pass this kind of information, in form of key-value pairs, to the library. These will be covered more thoroughly in Section 6.5.1.

4.2.3 HDF5

The Hierarchical Data Format version 5 (HDF5) [46, 38, 18] developed by the National Center for Supercomputing Applications is a high-performance suite that embodies a data model, library and file format for the storage and management of data. The data model provides abstract concepts that allow users to specify their domain-specific objects. The HDF5 library itself is implemented in C but also provides wrappers and APIs for languages such as C++, Fortran, Python and Java which can be used to access and manage HDF5 files. However, instead of using the C++ wrappers, the C library is directly used in the scope of this thesis. HDF5 files are stored in a portable, self-describing, extensible and backward-compatible format. Additionally, the HDF5 suite has multiple command-line tools for compilation, library management, library performance measurement, gen-
eral file operations such as conversion and comparison, etc. Besides serial I/O, HDF5 also provides a parallel counterpart PHDF5 which is taking advantage of the MPI-I/O middleware.

### Data Model

In an abstract way, an HDF5 file is a container with a collection of objects. These are organized in a fashion similar to a UNIX file system. Here, the two primary objects that describe this hierarchical structure are *groups* and *datasets* which resemble directories and files respectively. Groups consist of zero or more objects and in turn each object must be a member of a group, except for the root group "/". Furthermore, group membership is implemented via *links* which not only enable a hierarchical structure but directed graphs. Also, additional *attributes* in form of key-value pairs can be specified for an object to further annotate the file.

**Datasets** While groups and links more or less only describe the structure of the file, datasets contain an application’s substantial array data. Conceptually, datasets are a multidimensional array of data elements. Data elements are represented by a *datatype* that must be the same for each entry in the array. These range from atomic data types such as numerals or characters to compound types which resemble C structs. Furthermore, they almost provide unlimited flexibility since users can even define custom atomic datatypes. How these elements are actually laid out is described in a *dataspace*. Physically, a dataset is stored in two parts, that is, a header and a data array. The header provides meta-data that describes the dataset as well as information required to interpret the array. To be more concrete, this information mainly includes the dataset’s name, dataspace, datatype and storage layout.

**Dataspaces** In principle, dataspaces describe the dimensionality of the multidimensional data, that is, with the number of dimensions (rank) in addition to their current and maximum extents. Dataspaces can be categorized into three types: *null*, *scalar* or *simple*. Null spaces consist of no data elements. Scalar spaces consist of a single element and thus have a zero-rank. Nonetheless, the datatype of a scalar element may be very complex, e.g. a composition of multiple and potentially nested datatypes. Simple dataspaces are the most used in the scope of this thesis as they represent regular, multidimensional arrays of data elements.

Next to the creation of datasets, dataspaces also mark elements that are involved in I/O operations. Either all or a selection (partial I/O) of the dataset is transferred. One reoccurring use case for the latter is the storage of distributed data in parallel applications. In HDF5, partial I/O
is specified with hyperslabs or lists of individual points. Figure 17 illustrates basic examples in which a hyperslab is selected from a source dataspace and transferred to a destination dataspace with potentially different dimensionality. Basically, hyperslabs are either rectangular sub-arrays as shown in Figure 17.a or a regular pattern of points or blocks within the dataspace as in Figure 17.b. More complex regions can be specified by the union of multiple hyperslabs. The point selection is mostly used when a sequence of points without a regular pattern is accessed. For both selection approaches, the specified rank of the selection must be identical to the rank of the original dataset but may still select regions such as a 2D plane in a 3D dataset. In case of collective I/O, parallel HDF5 internally maps regular hyperslab selections to MPI derived datatypes but uses independent I/O for non-regular selections. Similar to the selection in a file’s dataspace, regions within a data array can be marked for transfer by selecting a portion in its memory dataspace. One common scenario in the scope of this thesis is the exclusion of ghost layers.

Property Lists Essentially, property lists provide optional information that is relevant to alter the default behaviour of the library. These lists consist of key-value pairs and are passed to functions for file creation and access, dataset creation and transfer, etc. For instance, file access properties can be used to select a file driver to open a file. In parallel applications, MPI-I/O can be selected, whereas for serial applications the POSIX driver can be chosen. Since this only involves changing the argument passed to the create or open function, the differences between serial and parallel I/O implementations are further minimized. Moreover, there exist various properties to optimize the performance for a specialized environment, i.e. these properties are used as a hint mechanism. Users can also use them to specify whether collective or independent I/O is chosen when transferring a dataset. Another important use case is the customization of the storage layout of a dataset.

Storage Layouts Logically, a dataset is laid out as a rectangular array. However, it can be physically stored in different ways with each having an effect on how much space in the file is allocated. In total, there are three strategies: contiguous, chunked and compact. By default, a dataset is laid out contiguously. Here, the dataset is stored as a meta-data header and the data array is flattened in row-major order. With this layout, neither unlimited dimensions nor filters, e.g. compression, can be used. With chunked storage, the array data is stored as equally-sized subunits. These can have various size and shape combinations, e.g. 2D planes in a 3D dataspace. Most notably, in this layout I/O operations are performed per chunk now [38]. This layout offers great storage flexibility as it makes unlimited dimensions as well as utilization of filters possible. Additionally, this decomposition can allow performance optimizations for specific access patterns. However, this approach must be used carefully since it also involves complicated management of the indices to locate the chunks of a dataset in the file [31]. Thus, the choice of appropriate chunk sizes is crucial. Compact storage allows the storage of a small and continuous data array in the header block. This way, dataset header and data are not stored separately. A separate storage has negative effects for small datasets since at least two I/O operations are required to access the data, that is, one access for the meta-data and the rest for the data [38].
Storage Model

With this kind of flexible and hierarchical structure of an HDF5 file, the question arises how this structure can be mapped to a linear address space, i.e. for storage on disk. For this purpose, HDF5 established a format specification separated into three levels that describes how elements from the data model are represented in the file [38]. In basic terms, the level zero specification introduces a superblock that includes the format signature, version numbers, e.g. of the superblock, pointers to entries within the file and parameters such as drivers to describe the file layout. The file infrastructure including the group hierarchy is described in level one. In order to describe such a flexible and complex layout, data structures like binary-trees, heaps, free-space lists, symbol tables, etc. are defined and stored. The data objects that represent scientific data defined by the user can be found on level two and any other data in the file can be interpreted as framework to access these. As mentioned, datasets are typically stored as header and data information and are not necessarily laid out contiguously on disk. Thus, the layout of an HDF5 file on disk can be highly irregular. A more detailed description of an HDF5 file’s format can be found in [37].

4.2.4 Parallel NetCDF

Unidata’s netCDF [68] is also a high-level I/O library that provides a simple API to store multidimensional typed array variables in a portable, self-describing data format. NetCDF is widely utilized in many application fields such as ocean, earth and atmospheric modelling. The netCDF library itself is written in C but there are bindings for other languages such as C++, FORTRAN, Python and Java. All in all, netCDF provides multiple file formats, two different data models and different means to perform parallel I/O.

Format Versions

During the lifetime of netCDF multiple variants of the file format were developed. In general, these can be categorized into two types, classic and netcdf-4.

Classic formats follow the same (classic) data model which will be presented in this section. Moreover, they can be further categorized into three chronologically ordered formats. The first is the default format CDF-1, the second is the 64-bit offset format CDF-2, i.e. files greater than 2 GiB, and the third is the 64-bit data format CDF-5, i.e. array variables with more than 4 billion elements. Despite the fact that there is no netCDF-3 format, the term is commonly used to abbreviate the collection of classic formats [67, 75]. Note that older versions of the library might not support CDF-2 or CDF-5 files. Still, as long as the format is supported, files of that format are backwards-compatible for older versions of the library, e.g. CDF-1 has maximum portability [68, 75].

The netCDF-4 library [75], on the other hand, has been developed in collaboration with the HDF and netCDF groups to enrich the netCDF data model with features from HDF5. As a result, netCDF-4 employs an additional data model, often called enhanced data model, that enables support for multiple unlimited dimensions and user-defined datatypes such as compound datatypes, group hierarchies, etc. Furthermore, the netCDF-4 library enables compatibility with existing netCDF software and files. The netCDF-4 format refers to an HDF5 file that was created by the netCDF-4 library. However, while each netCDF-4 file is an HDF5 file, the same does not apply in the other direction since netCDF-4 only supports a selected subset of HDF5’s features in order to keep the data model and API simpler. For instance, features including non-hierarchical structures such as group containment cycles, custom atomic datatypes, etc. are not provided. Moreover, these features cannot be applied for classic files due to the differences in the data models. For users that want to use the HDF5 storage layer but do not want to use complex features such as a group hierarchy or compound datatypes, the netCDF-4-classic format can be used, but in turn, these files cannot be extended by said features. NetCDF-4-classic files can still benefit from the HDF5 storage. This, for instance, facilitates the extension of meta-data in an existing netCDF-4 classic file which, as will be shown later, can be cumbersome with netCDF-classic files. This format has been added for convenience since existing netCDF-3 applications can simply be re-linked and upgraded to the netCDF-4 library.
Classic Data and Storage Model

The classic data model of netCDF is rather simple in comparison to HDF5. Instead of employing a hierarchical organization of data, the classic data model focuses on a flat design in which data is stored in an array-oriented dataset. Datasets consist of variables, attributes and dimensions. A variable holds a multidimensional array of data with identical datatype. In contrast to HDF5, only a small set of primitive datatypes, in summary six types representing numerals and characters, is provided. The shape of a variable is given with a list of dimensions. Dimensions can be shared by multiple variables and consist of a name and a length. In classic formats, only one dimension can be unlimited. Furthermore, attributes can be used to annotate the dataset itself in form of global attributes or the variables individually. Similar to HDF5, these are key-value pairs that hold meta-data.

As shown in Figure 18, a classic dataset file can be divided into three regions. The file’s header contains meta-information about the file contents such as dimensions, variables and attributes. The following two regions in the file represent the stored values of the variables. Here, we distinguish between fixed-sized and variable-sized variables. Fixed-size variables are stored contiguously after the header and the variable-sized ones, often referred to as record variables, are immediately stored after the last fixed-size variable. More specifically, the most significant dimension of record variables is the shared unlimited dimension in which these variables grow along with. Record variables are stored as a sequence of records. A record contains the data of all record variables for a certain record dimension index, e.g. the i’th time step, with its entries stored interleaved in the order the variables were defined. Note that each entry in a record is of fixed-size.

In accordance to the separation of the file into header and data portions, netCDF introduces a bimodal programming paradigm comprising of define and data mode. A dataset is initially set to define mode when it is first created. Here, the definition of dimensions, variables including their shape and type and attributes takes place. By defining the data extents beforehand, space allocation for the file is facilitated. After the definition, the application switches to data mode in which array data is stored to file. The file then stays in data mode unless instructed otherwise. One downside to this bimodal approach is the cost of extensions. In case that the define mode is re-entered and new variables are added or the header is extended with more meta-data, stored data must potentially be reordered and moved. However, in many cases it is known beforehand which data will be written.

While HDF5 introduced the concept of hyperslabs to select a portion of a multidimensional array, netCDF provides designated data access functions for this purpose. The calls to these functions pass a pointer to contiguous regions in the memory and depending on the function type different regions in the file can be described. These regions can be single elements, whole arrays, (strided) subarrays and multiple non-contiguous regions.

Parallel I/O

Initially, the netCDF library was designed for serial applications and did not support parallel I/O. Over the years, parallel I/O capabilities for netCDF files have been granted by two different libraries. The first is the aforementioned netCDF-4 library that uses HDF5 underneath and thus can make use of its parallel I/O capabilities when accessing netCDF-4 files.

The second interface is the so-called Parallel netCDF (PnetCDF) library, a joint project of the Northwestern University and Argonne National Laboratory, which is built on top of MPI-I/O. On the contrary to netCDF-4, PnetCDF retains the classic netCDF data model and formats.
The PnetCDF library has APIs for C, Fortran 77 and Fortran 90. In short, PnetCDF aims to maintain backwards compatibility by making use of the classic formats and providing a so-called high-level API that is similar to the serial one allowing users to easily adopt to the parallel interface [54, 56].

One major drawback of the data access functions of the serial API and also its similar high-level API is that it assumes a pointer to a contiguous region in the memory is passed. Hence, these functions cannot describe more complex memory layouts and data must often be copied to a temporary buffer accordingly before the function call is made. For this reason, PnetCDF introduces an additional flexible API which allows non-contiguous memory regions to be specified with MPI datatypes.

With the addition of MPI-I/O, PnetCDF also distinguishes between collective and independent I/O operations. For this purpose, PnetCDF has further split the data mode into (non-)collective data mode in which only functions corresponding to their mode are allowed be called, e.g. collective data access functions with suffix _all as in MPI-I/O can only be used in collective data mode.

Furthermore, PnetCDF minimizes the overhead of meta-data I/O by keeping a local copy of the header on each process. Hence, many meta-data accesses can be performed with in-memory operations instead of I/O. Moreover, in case of I/O operations to the header, they are only performed by the root process. When opening a file, the root process gets and broadcasts the header to the other processes. In case of changes to the header, the root process is also responsible for writing the changes at the end of define mode. Moreover, accesses to the meta-data are performed collectively. Note that this does not necessarily mean that communication between the processes is always required, e.g. for inquiry functions. However, it is required when the header is modified and consistency between the copies needs to be ensured. In this case, the arguments passed by all processes need to match.

Another reason for the choice of the classic format is its linear file layout. While HDF5 uses a highly flexible and hierarchical file structure where data is laid out in an irregular fashion, the layout of classic netCDF formats is highly predictable and regular. This allows user-defined data regions to be easily mapped to the MPI-I/O layer by passing meta-information such as MPI datatypes and views as well as optimization hints to it. Hence, only little overhead and a performance similar to a pure MPI-I/O implementation is expected for raw data I/O operations [59].

In conclusion, netCDF-4 and PnetCDF libraries employ different formats and data models. The NetCDF-4 interface makes use of HDF5 and introduces the enhanced data model which enables more features and flexibility. However, it intentionally contains only a portion of HDF5’s feature set. In contrast to netCDF-4, PnetCDF embodies the classical data model and promises minimal overhead compared to an MPI-I/O implementation. Therefore, instead of using the (hybrid) netCDF-4 approach we compare the native HDF5 library with its complete feature set with the PnetCDF approach in order to get more insight about the differences of both pure variants. Hence, netCDF-4 will not be covered any more in the scope of this thesis.

4.2.5 SIONlib

The SIONlib [17, 33, 34] library was developed in collaboration with the Jülich Supercomputing Centre (JSC) and the German Research School for Simulation Sciences. SIONlib embodies the file-per-process model on a logical level while using a shared file as a container. Moreover, for this paradigm it introduces a more general notion, namely task-local parallel I/O. Here, the logical task-local files are mapped to a shared file to mitigate the disadvantages that originate from creating a physical file per task, i.e. the meta-data overhead and management of the large number of files. With this concept, SIONlib aims to provide scalable I/O capabilities to typical task-local use cases such as log files, scratch files and especially checkpoints. SIONlib itself is implemented in C but also supplies a mapping for Fortran. The library also comes with command-line tools to manage, convert and analyse SIONlib files. The SIONlib API is conceptually close to the ANSI C interface and can be seen as an extension for it. Furthermore, read and write calls from the ANSI C or POSIX interfaces can be used on SIONlib files demanding only little adoptions, i.e. changing the open and close calls, for existing task-local ANSI C implementations to upgrade to SIONlib. Unlike NetCDF or HDF5, SIONlib does not provide a high-level data model but handles data as a stream of bytes. Additionally, despite the fact that SIONlib also employs a single-shared file approach it is fundamentally different to the previously introduced interfaces. These store their data in a global view which, from a programmer’s perspective, requires knowledge about the distribution and
structure of the data. Instead, since SIONlib preserves the view of (logical) task-local files, each task can write its data individually without this global knowledge.

Most notably, SIONlib uses a file format in which *chunks* are dedicated to a single task and stored in a linear fashion to disk. In contrast to regular task-local files which can grow almost indefinitely, this chunk-based format imposes a limit to the chunk size which cannot be exceeded by each task, since otherwise data from subsequent tasks may be overwritten. Furthermore, the chunk sizes between the tasks may differ. By specifying the chunk size before the first task writes to file, the library can calculate the start offsets of each chunk. In order to enable reading such files, the offsets and sizes of the chunks need to be known. For this reason, this meta-information must also be stored. Hence, the open and close operations in SIONlib are performed collectively. When opening the file, all tasks communicate their chunk size to a master task which writes a meta-data block and returns the respective start offsets to the tasks. When closing the file, the number of written bytes per task is communicated and stored in the meta-data block at the beginning of the file. Since the requirement to specify the total amount of data written by each task can be too confining, this limitation can be lifted by only providing the maximum size required for a single write call by each task. With this concept, the file is now structured in *blocks* in which one chunk per task exists. In case that a task requests more space than available in the chunk, it receives a new chunk of the same size. Under the hood, a new block with its whole set of chunks, one for each task, is allocated. Since the size of the block can easily be calculated with the specified chunk size, no synchronization is required to determine the file offsets of the new block. However, this kind of repetitive allocation of space will lead to severe fragmentation if chunks are only requested for a small portion of all tasks. In most cases, the fragmentation occurs only logically since no file system blocks are physically allocated for the gaps but when post-processing these files, it is important to de-fragment the files first to prohibit an actual allocation [33]. For this purpose, more meta-data consisting of the number of chunks per task and free space is required. This information is stored in a second meta-data block at the end of the file as soon as the collective close function is called. Note that between the open and close functions no further collective operation is required. Figure 19 illustrates the layout of a SIONlib file that is organized with blocks [34].

In accordance to the presented file layout, SIONlib introduces optimization techniques for simultaneous accesses to a single-shared file from a large number of tasks. One scalability issue is the locking mechanism of the (parallel) file system which must be implemented due to the consistency semantics POSIX imposes. Here, regions of the file can be locked by a task before performing I/O. For write operations, exclusive locks can be used to serialize access to the region, whereas for read operations shared locks can be used to read the file in parallel. Despite the fact that a chunk in a SIONlib file is not directly accessed by multiple tasks concurrently, it can occur that a file system block is shared between two tasks. Since in many parallel file systems exclusive locks are given on basis of file system blocks, a false sharing situation can occur and accesses to the blocks are serialized. Thus, SIONlib allows the alignment of chunks with the boundaries of file system blocks to overcome this problem. However, since file system blocks on nowadays HPC systems are commonly in the magnitude of one megabyte this could lead to severe (physical) fragmentation for a large count of tasks that write small amounts of data. Instead of dropping the block alignment, SIONlib solves this problem with so-called *coalescing I/O* in which, identical to MPI-I/O’s two-phase I/O, data is aggregated by a set of tasks before it is written in larger contiguous chunks. With the reduced amount of tasks that are involved in I/O operations, not only fragmentation but also overcrowding of the underlying I/O system is mitigated. For further optimization potential, SIONlib introduces the concept of *multi-files*. Instead of writing the data from all tasks to a single file, SIONlib allows the task-local data to be distributed and stored to a requested number of phys-
ical files. With this concept, software and hardware parallelization may be better utilized [34]. For instance, the overhead of the meta-data management of a large task count accessing a single-shared file can be reduced [33]. Here, file meta-data such as references to file system blocks is stored in the inode structure. However, in very simplified terms, there exists only one inode for a single file and thus updates need to be synchronized [34]. Hence, increasing the number of physical files can mitigate the meta-data handling.

Note that the previous approaches define parallel I/O only on the MPI layer. Moreover, shared-memory parallelization models like OpenMP do not support parallel I/O and require users to synchronize accesses to the file themselves [71]. However, data written to file is not thread-private and can be written outside parallel regions by the master thread. Despite the fact that I/O is serial in the view of a shared-memory entity, parallel I/O is still possible on the MPI layer and is typically done this way in practice [33]. However, there are applications that require data to be read or written per thread in a parallel region. Thus, SIONlib also provides APIs for MPI, OpenMP and OpenMP+MPI parallel applications. In addition to the parallel interfaces, SIONlib also provides a serial API which can be used to access multi-files from a serial application. For instance, it is possible to (sequentially) read the data of a specific task or the whole global data from a file that was written in parallel [33, 34].
5 Visualization Formats

At this point, the different means to store data on disk have been discussed. However, visualization programs need a description of the underlying data in order to comprehend and visualize it. This section briefly demonstrates the visualization formats which were used in combination with the previously mentioned parallel I/O patterns. The presented formats can be directly visualized with tools such as ParaView [73] and VisIt [94] and therefore facilitate the workflow of a user since these tools are operable via a graphical user interface (GUI) and provide several filters and analysis functions.

5.1 VTK

The Visualization Toolkit (VTK) [92, 82] is a software system for scientific visualization, information visualization, image processing, volume rendering, etc. The VTK library is implemented in C++, but also provides bindings to Python, Java and Tcl. The VTK software system introduces software layers such as rendering techniques to display scientific data and a data model that comprises a set of data objects, such as (un-)structured data, image or volume data and linear primitives, e.g. polygons. Furthermore, it employs a visualization pipeline in which processing steps, e.g. filters, on the data objects are described. In fact, the aforementioned visualization tools are based on VTK.

Most notably, VTK employs its own file format which, naturally, is also supported by these visualization tools and can thus be directly visualized [25, 24]. Here, it is important to note it is not mandatory to use the VTK API in order to create such files. Instead, it is possible to create them with standard I/O interfaces, e.g. C++ I/O streams, by following the guidelines of the format. In fact, this is how the VTK output in ExaStencils is implemented. More specifically, ExaStencils currently uses the serial legacy format. In short, the legacy format is straightforward to use but is inflexible. For instance, when data is represented in binary form, the byte ordering must be big-endian. Thus, data potentially needs to be converted to big-endian first in order to be understood by VTK readers. Furthermore, the binary representation of the legacy format is not fully portable since differences in the number of bytes per data types that are machine-dependent cannot be handled. Thus, the VTK output in ExaStencils is currently restricted to an ASCII representation.

However, VTK has also introduced another set of formats based on eXtensible Markup Language (XML). On the contrary to the legacy format, the endianness and byte size of the used data type can be specified to avoid aforementioned problems. Another benefit of the XML format is the support for file-per-process parallel I/O. For this purpose, XML VTK provides two types of files formats, namely serial and parallel. The parallel formats are not actually used to store data but reference the pieces that originate from the domain partitioning. Here, each process writes a serial XML file for its assigned piece of the global domain. The parallel file then specifies a list of pieces where each entry provides minor structural information and the path to the source file in the serial format. However, using the file-per-process approach is optional. Another possibility is that all processes write to a single-shared file (serial format) in a coordinated fashion, e.g. via the temporal locking currently used in ExaStencils or by using MPI-I/O as it can be done in waLBerla. In addition, binary data can be encoded to base64 and compressed.

5.2 XDMF

The eXtensible Data Model and Format (XDMF) [20, 41, 91] was developed to exchange scientific data between HPC codes and tools used for visualization, analysis, etc. XDMF has an C++ API which also enables reading and writing XDMF data in other languages such as Python, Perl, Tcl and Java by using wrappers. However, using this interface is not mandatory as it is also possible to write XDMF files using native text output and is in fact also done by several HPC codes. Furthermore, it also allows binary data to be compressed [91]. One key principle of XDMF is the categorization of data based on two attributes.

The first is the size. Data can be either categorized as light data or heavy data, where the latter belongs to extents of several megabytes or even greater. Besides the logical distinction of heavy data and light data, these can also be physically separated, i.e. in different files. Light data is stored using XML and is mainly used to describe the data model. The data model basically describes what the heavy data represents. XML was chosen since it specifies how data is stored
and structured in a standardized way. Additionally, XML provides means to reference data from other files. For instance, with the XInclude mechanism one can include files that are not necessarily well formed XML, e.g., plain text files. In order to reference particular elements in a XML file, the XPath mechanism can be applied. Heavy data is typically embodied by enormous data structures and can either be stored as regular binary or HDF5 files. This separation grants more functionality and flexibility since HPC applications can now write their datasets in a layout which is natural to their internal data storage in order to perform I/O operations optimally. The properties of the stored datasets are then described in the light data fraction which is comparatively small and also portable. Without this kind of separation, HPC codes might be required to rearrange their data in order to conform to a certain format which can lead to performance penalties [20].

The second attribute is the function of data. Data can either refer to format or model. The former provides information such as the dimensions and the rank of a data array, whereas the latter instruments the usage of the data, e.g., whether an array of floating point values expresses the coordinates of a mesh or a computed vector field. This distinction removes possible constraints for HPC programs as these only need to focus on producing a file with a proper format without being bothered by the data model of the tools.

<?xml version="1.0" encoding="utf-8"?>
<Xdmf Version="3.0">
  <Domain>
    <Grid Name="Grid" GridType="Uniform">
      <Topology NumberOfElements="2" Type="Triangle">
        <DataItem DataType="Int" Dimensions="2 3" Format="XML">
          0 1 2
          3 2 1
        </DataItem>
      </Topology>
      <Geometry Type="XY">
        <DataItem DataType="Float" Dimensions="4 2" Format="XML">
          0.0 0.0
          1.0 0.0
          0.0 1.0
          1.0 1.0
        </DataItem>
      </Geometry>
      <Attribute Name="Field" AttributeType="Scalar" Center="Node">
        <DataItem DataType="Float" Dimensions="4" Format="XML">
          0.0
          1.0
          2.0
          3.0
        </DataItem>
      </Attribute>
    </Grid>
  </Domain>
</Xdmf>

Listing 3: XDMF example of a simple triangular mesh with two triangles in a 2D Domain

Data Model

This section illustrates the data model of XDMF by using the preceding example in Listing 3 which represents a unit square split diagonally into two triangles. The square consists of four vertices on which a scalar quantity named field is stored. As shown here, the organization structure in XDMF starts off with so-called Xdmf elements. These elements consist of a hierarchy of one or more Domain elements. These Domains may consist of at least one Grid element. These indicate the beginning portion of the data model in XDMF. A Grid can be viewed as a group of elements with a homogeneous topology and their corresponding values. To be more concrete, it contains information related to 2D and 3D points, connectivity for (un-)structured meshes and its values.
Grid elements can be classified by their GridType attribute. In the example, only a single, homogeneous Grid (Uniform) is used. On the other hand, in case that different topologies are present, these belong to separate Grid elements. In such cases, more complex GridTypes can be used, e.g. a Tree, a Collection or a SubSet of Grids. In the scope of XDMF, a Grid has a Topology, a Geometry and optional Attribute elements. The Topology basically describes a Grid’s connectivity and also its organization of data. For structured grids, the connectivity is already implicitly given. For unstructured grids, on the other hand, the connectivity must generally be provided. In order to specify the Topology of a Grid, the Type attribute must be set. In XDMF, structured grids are categorized as curvilinear (2DSMesh or 3DSMesh), rectilinear (2DRectMesh or 3DRectMesh) and rectilinear with constant spacing (2DCoRectMesh or 3DCoRectMesh). For unstructured grids, a name for the cell type must be provided. In the example the type is Triangle. The lower triangle consists of the vertices 0, 1 and 2 with (x,y) positions (0,0), (1,0) and (0,1). XDMF supports linear, quadratic and mixed element types. For mixed topologies, a number representing the cell type must be specified in addition to a node list, that consists of indices to the specified node positions in the geometry section for a certain element. The linear cell types in addition to their cell type number are depicted in Figure 20. For cell types where the number of nodes are not implicitly known, the NodesPerElement attribute must be set, e.g. 8 for octagons [91].

![Figure 20: Linear cell types for unstructured meshes (from [91])]()
The Dimensions of a dataset are defined in the KJI order in which the slowest varying dimension is on the leftmost side of the specified list. In order to access a raw binary file at a certain position, the Seek attribute can be applied. Despite the fact that HDF5 is a self-describing data format with metadata such as the dimensions and number types of all the arrays that exist in the file, this metadata is also redundantly stored in the XDMF file. This enables applications to determine the storage requirements without having to access the heavy data [41]. Similarly to a Grid’s GridType, the ItemType attribute of a DataItem can be declared as Uniform which represents a single array of values and is used by default. Furthermore, there are also more complex ItemTypes such as HyperSlab to select a subset of another DataItem, Function to perform operations, e.g. joining scalar components into a vector, on its provided children DataItems and others described in more detail in [41].

5.3 NetCDF

Unfortunately, both previously presented approaches cannot be used to visualize data from classic netCDF files without having to convert them first. Thus, two additional approaches are introduced for this purpose.

5.3.1 Meta-data conventions

Since netCDF is often used in ocean, earth and atmosphere sciences, the Climate and Forecast (CF) conventions [26] have been introduced as a meta-data standard for netCDF files in order to facilitate sharing and processing them. In short, the conventions define meta-data used to give a description about each data variable in conjunction with temporal and spatial information. Most notably, this enables visualization and other post-processing operations with minimal effort for users. For instance, visualization tools like ParaView and VisIt have dedicated readers for netCDF data following the CF conventions. However, the CF conventions are originally modelled for fields of climate and forecast, atmosphere and other sciences. Thus, it provides features related to geographic coordinate systems such as projections other than the typical latitude and longitude coordinates on earth. Since this is not the application domain of ExaStencils and GHODDESS, only a subset of the convention’s feature set is used. Instead, these conditions are only followed loosely in a sense that the positions of the discretization points are specified such that the corresponding file reader in ParaView or VisIt can produce a suitable visualization for the data on the discretization points.

One downside of the CF-conventions and the aforementioned readers for visualization is that only structured meshes are supported [93]. Unstructured data, however, can only be defined as scattered data without any topological information. As an alternative, the so-called UGRID conventions [93] can be used to also provide this kind of information in a netCDF file. Additionally, there is an external plug-in for ParaView that allows visualization of following the new convention [10]. Unfortunately, this plug-in is restricted to 2D triangular, quadrilateral and a mix of both meshes. Alternatively, users can develop their own readers and visualization tools for UGRID-conform files. For instance, the Python module gridded [36] can be used to simplify reading data following the CF or UGRID conventions. The read data can then be visualized with packages like matplotlib [63]. Despite the fact that these packages facilitate the process, this poses a great burden for users and, in contrast to general-purpose tools, the visualization capabilities are rather rudimentary since filters would need to be developed. On the contrary to the meta-data approach, scientific libraries can be used instead.

5.3.2 ExodusII

In order to overcome the mentioned problems for unstructured meshes, the ExodusII [84, 70] library can be used. ExodusII is a library for the storage of data from Finite Elements (FE) applications. ExodusII provides APIs for applications written in C/C++, Fortran and Python. These APIs are geared towards FE codes and facilitate the specification of the employed data objects. Furthermore, ExodusII makes use of netCDF as storage layer in order to achieve portability. This way, also the netCDF API can be used to access the data from an ExodusII file.

In the ExodusII data model, three different data groups are distinguished. The initialization data describes the dimensions of datasets, e.g. number of nodes or elements, and optional meta-data.
The *model data* is static and consists of node positions, connectivity of the elements, attributes and others. ExodusII provides a rich set of element types similar to XDMF and also supports mixed topologies. Besides elements, ExodusII supports the entity types: nodes, edges and faces. For an efficient storage, ExodusII groups elements with identical type into blocks, e.g. a block containing only triangular elements. There are also other entity groups such as sets or sidesets but these are not covered in this thesis. Since the model data is assumed to be static, meshes with time-varying geometry must be stored in separate ExodusII files. The *result data* allow storing variables for the aforementioned entity types that were grouped together, e.g. in a block.

Similar to netCDF, ExodusII library was originally developed for serial applications. For this purpose, the NemesisI [70] project was developed to enhance ExodusII for parallel environments. The NemesisI routines were integrated to the ExodusII library. However, parallel I/O is done in a file-per-process manner and external tools can be used to split the global domain into pieces or join them together. In order to enable visualization capabilities in conjunction with the PnetCDF library, we just use the ExodusII library to create an infrastructure for visualization. In other words, components such as nodal coordinates, connectivity of the elements, variables, etc. are only defined via the ExodusII interface but the large data arrays are not stored with it. Instead, once the data objects have been defined, the ExodusII file is closed and re-opened via PnetCDF. From here on, PnetCDF can be used as storage layer.
6 Implementation

In this section, the focus lies on how the previously discussed concepts can be put together. Basically, that is the generation of parallel I/O interfaces in addition to new visualization techniques. Here, mainly the changes to the ExaStencils DSL and its source-to-source compiler are discussed since, in the end, GHODDESS is build on top of ExaStencils. Thus, GHODDESS only needs to be adapted to support the new DSL elements that will be introduced. At first, an introduction to the general aims for the design is given. Afterwards newly added software packages and their components are shown and their most important aspects of their implementations discussed. In the end, the employed optimization techniques are presented.

6.1 Design

This section shortly introduces the overall design goals and gives an overview of the new software modules in ExaStencils and their interaction.

6.1.1 Goals

Since the requirements for I/O operations change from application to application, there should be multiple options available for users. These options include different I/O patterns, libraries, data formats and layouts. By integrating multiple interfaces to the code generation framework, users can choose the one most suitable for their I/O needs. Furthermore, code for each approach can be generated in different configurations, e.g. domain partitioning with different number of processes, facilitating the evaluation process of the approaches based on quantitative metrics, e.g. I/O bandwidth. Since multiple I/O interfaces are added to the ExaStencils compiler, one important goal is to provide an easily extensible class hierarchy. Moreover, these interfaces should be designed such that they are attachable for different use cases, e.g. visualization or checkpointing.

Despite the fact that the focus mainly lies on parallel I/O capabilities, serial applications should also be able to benefit from the newly introduced concepts, e.g. the portability of high-level libraries. With this in mind, in case that a library also has a serial API, it is also supported in ExaStencils. In fact, with the exception of MPI-I/O each presented library can also be used in serial applications.

As shown in Section 3.1.7, the implementations for the printField and readField statements use different I/O patterns. In accordance to the terminology from Section 4.1.4, the printField statement makes use of the single-shared file I/O approach by explicitly serializing the accesses to the file at a given time (locking). On the other hand, the readField function is following the file-per-process I/O pattern, where a non-shared file is assigned to each process. Correspondingly, while not being directly relevant in practice, the exemplary sequence of a printField followed by readField statement with identical arguments would result in unexpected behaviour in the target code. Thus, another design goal of this thesis is to implement an I/O interface where the user can explicitly specify the required I/O pattern and library to allow such sequences to be authentic.

In accordance to the previous goal, existing DSL code should still be valid and maintain the same functionality. Thus, ensuring backwards compatibility is another important aspect.

6.1.2 Overview

In the scope of this thesis, the overall changes made to the compiler can be classified into three groups of interfaces which are mainly present at the IR layer.

The first group consists of the I/O interfaces. In the prior state of ExaStencils, there was no dedicated package for I/O as everything could be done with C++ I/O streams. There, a sequence of expressions were fed to instances of the IR_Read or IR_Print classes which then are pretty-printed in form of stream operators. However in contrast to a stream-based I/O method, the presented I/O interfaces are more sophisticated as they convey information about the global access pattern of all processes and often supply advanced data models. Thus, the newly added I/O interfaces have been encapsulated in a designated I/O package which is used by the remaining two groups.

Since visualization of data is ubiquitous in HPC applications, we further evaluate visualization capabilities in combination with the newly added storage layers. As mentioned, the legacy VTK interface from the prior state has its limitations. Furthermore, it cannot be combined with the
formats from libraries such as HDF5 and NetCDF. Thus, new visualization formats have been introduced to ExaStencils, which marks the second group.

The third group consists of the aforementioned file access functions for fields. These were restricted to file-per-process for reading a field and the locking mechanism for printing fields. In order to clarify which I/O interface is actually used, a new set of I/O statements for ExaSlang 4 is introduced. Furthermore, we distinguish between visualization interfaces tailored for a specific application type, e.g. SWE, and more general interfaces providing visualization capabilities for a single field for any kind of application. The latter conceptually belongs to the third group.

The interaction of the three groups is rather straightforward. The I/O interfaces are the foundation of this thesis and are directly used by the other groups. Put simply, the other packages create an instance of an I/O interface, denoted as I/O handler here, and pass it information about the file name, the data to be accessed, and other properties about the file access. By design, this instance is responsible for all I/O operations and, naturally, is also involved in the code generation process when used by the other interfaces. For instance, a visualization interface must notify the handler which field and mesh data needs to be stored for a proper visualization. When target code for the visualization is created, statements from the I/O handler instance are also generated and appended to it. This way, the I/O interfaces are encapsulated and can be re-used for other occasions, e.g. checkpointing.

6.2 I/O Interfaces

The core of this thesis, that is the implementation of the new I/O interfaces, is discussed in this section. First of all, an overview of the I/O package’s general class structure is given. Moreover, since only theoretical concepts of the proposed file storage approaches were specified in Section 4, insights to the programming paradigm of each approach in conjunction to the integration into the code generator are given.

6.2.1 Structure

The overall organization of the I/O package is illustrated in Figure 21. As will be shown in the following, its main components can be categorized into three class types. Note that this figure is by no means complete and only demonstrates the key components of the I/O interface and their relation.

I/O Interface Hierarchy

The typical procedure when accessing a file is identical, independent of the I/O technique that was used. First, the file is created or opened, followed by either read or write operations, and is closed lastly. Moreover, since the differences in the implementations of read and write accesses are generally small, it would make most sense to supply an interface that is responsible for both access types to reduce the code base. This course of action is also defined in the abstract IR_FileAccess class which is to be implemented by each of the presented I/O interfaces. Similar to the I/O statements from the prior state, the abstract class also inherits from the IR.Expandable trait and defines an expand function which calls the open, read, close, etc. functions from a derived class, e.g. IR_FileAccess_HDF5, during the expansion pass in the generation pipeline and concatenates the statement collections together before they are pretty-printed to C++ code. It is also responsible for handling the specified dependencies, e.g. addition of headers and libraries, on behalf of each employed I/O approach.

As can be seen in the figure, an instance of a IR_FileAccess subclass consists of multiple IR_DataBuffer objects which represent the basis for each I/O request. Since most presented I/O libraries require a specification of the databuffer’s extents, often in form of an array with the number of entries for each dimension, the abstract class provides declaration statements for these to prevent code duplication in the subclasses. As will be shown, two different memory layouts are supported for file access. Here, we distinguish between two modes: block-wise or fragment-wise in-memory layout of data buffers. In the fragment-wise case, the data of each fragment is transferred between disk and memory individually in a loop over all fragments contained by a block. For block-wise storage looping over the fragments is not necessary as all data in the buffer can be transferred
in a single call. Thus, the derived classes need to implement handling for I/O requests occurring fragment- or block-wise.

Furthermore, the subclasses also employ members for typical properties of file accesses, including the file name, a flag to determine a read or write access, a file mode flag, e.g. trunc or append. The values for these members are set by the constructor of the subclasses. For more flexibility, each derived class can implement a startup function occurring between file opening and file access and a cleanup function which is applied before closing the file. For instance, these are useful for I/O libraries such as HDF5 and PnetCDF where datasets need to be created or fetched from file first in order to perform I/O requests for these.

In Figure 21, only the HDF5 interface is depicted to provide a better overview but there are also additional interfaces. In summary, interfaces for following I/O methods were implemented over the course of this thesis:

- **File-per-process**: via standard C++ I/O library, serial and parallel
- **Single-shared file**:
  - Locking: via C++ I/O library, serial and parallel
  - MPI-I/O: parallel only
  - HDF5: serial and parallel
  - PnetCDF: serial and parallel
  - SIONlib: serial and parallel

![UML diagram of the new I/O package and its class hierarchy.](image)

**Figure 21**: UML diagram of the new I/O package and its class hierarchy.

### Memory Buffer Classes

**Fields** Since in I/O operations data is transferred between memory buffers and files, one key aspect is the definition of the memory buffers. In ExaStencils, the most commonly used representation are the fields which can be seen as arrays linked to the computational domain. As mentioned, fields
are elements of the DSL and specified by its user. In the generated target code, these are declared as global variables [80]. Corresponding to the definition of fields and field layouts per fragment, in the target code fields are also stored in a fragment-wise manner. For instance, a declaration of a field is often in the form: `double* fieldData_test[numFragsPerBlock];`

**Temporary Buffers** Data often needs to be stored or copied in temporary buffers. For example, when visualizing unstructured meshes, the connectivity data is often stored in a temporary buffer before it is written to file in one piece. Therefore, the `IR_IV_TemporaryBuffer` class, facilitating memory (de-)allocation and initialization, has been added. Note that these were designed to hold the data for all fragments within a block. Since temporary buffers typically only contain data of interest, that is no ghost or padding layers are copied into them, this results in a single, large and contiguous I/O request per block, which generally perform well in contrast to multiple smaller accesses [88]. This way, the overhead from allocating, initializing and freeing the memory of a buffer can be mitigated.

**General Data Buffers** In order to provide an abstraction from both concepts, `IR_DataBuffer` wrapper class has been developed and is an important building block for the I/O interfaces. To be more concrete, `IR_FileAccess` expects a collection of databuffers and defines the `read` and `write` functions to be per entry of the collection. When the expand function is executed, these functions are called for each supplied databuffer and are appended to the statement block of the whole file access.

Its companion object provides `apply` methods that accept either instances of `IR_Field` or `IR_IV_TemporaryBuffer` as parameters and extract properties such as data types, grid localization, the number of dimensions and their data extents in memory, etc. to construct an `IR_DataBuffer` object. One concern is the memory layout of fields with higher-dimensional datatypes, e.g. vectors or matrices. By default, these are represented as Struct of Arrays (SoA) which means that the dimensions for the matrix and vector components are the outermost [47]. Stream-based I/O interfaces employ a buffering mechanism internally [33] and facilitate components of such fields to be written in an interleaved fashion, i.e. by performing a field access for all components at a given discretization point. As will be shown, the implementation with MPI-I/O based approaches, however, is not meant for such fine-grained accesses and would require data to be explicitly rearranged in a temporary buffer before storage on disk. Thus, in this scenario it would be more efficient to store the field as-is instead of rearranging it. Moreover, visualization formats and tools provide means to interleave the components during post-processing again, e.g. XDMF’s `Function` mechanism.

In accordance to this, there are also occurrences where only a single component is required to be written. For instance, in case of non-uniform grids the (virtual) field for the node positions is stored as a vector. Since some formats, e.g. ExodusII, require the node positions of each dimension to be stored in separate variables, the companion also provides a constructor which allows a component to be treated as an individual databuffer. Another feature of the companion is the bookkeeping of dimensions that are declared, i.e. prevent a duplication of dimension declarations that might occur for databuffers with identical memory layout. This way, the output target code is simplified. For this purpose, a cache for the declarations has been introduced and is looked up whenever a buffer’s data extents must be specified.

Furthermore, in single-shared file approaches the file is shared among all processes in a parallel program and since, in our case, fragments must not overlap [49], each process has its own dedicated region within the file. Here, we distinguish between the (global) file space and the (local) memory space, and must also be able to define mappings between both. For this purpose, information such as the start indices, the stride, the total dimensions, etc. for both spaces are required and therefore encapsulated in the `IR_DataBuffer` class. Note that the global information is calculated in dependence of the aforementioned file layouts: canonical or fragment-wise order. In addition to this, this class also employs a flag to distinguish between the different (local) in-memory layouts of temporary buffers stored block-wise and fields stored fragment-wise.

```scala
val pattern = IR_AccessPattern(
  callback = (idx : IR_Index) => IR_FieldAccess(field, slot, idx.toExpressionIndex)
)
```

Listing 4: Callback registration for a databuffer’s access pattern instantiation
Figure 22: Left: Logical partitioning of quad cells into triangles with (L)ower and (U)pper orientation including the vertex indices for each. Right: access pattern denoted as sequence of $x,y$ offset pairs for node-centered quantities to be output for each triangle’s vertices.

**Access Patterns**

Most notably, in order to access the underlying instance of a databuffer, e.g. a field, the IR_Databuffer class owns an instance of IR_AccessPattern. This class employs a callback mechanism, where a callable function is registered when creating an object of the access pattern class. Besides a manual construction of said objects, the IR_DataBuffer’s companion object is also capable of creating an instance automatically. For example, the (anonymous) function from Listing 4 can be registered to access a databuffer’s underlying field for a given index. Another trait of the IR_AccessPattern class is that it allows the specification of a list of offsets which are considered for each access of a discretization point, e.g. a cell-center. This was mainly designed for the access pattern for applications from the domain of GHODDESS. As mentioned, in DG discretizations each element, in our case each triangle, has individual vertices. These can be grouped into nodes in which the number of vertices per node depends on the number of elements joining at that node. In ExaStencils, triangle meshes are handled as quadrangular cells divided logically into lower and upper triangle as shown on the left side of Figure 22. The corresponding vertex indices $i \in [0,2]$ for a lower or upper triangle are given in counter-clockwise order with the zero being on the opposite site of the diagonal that splits the quad. Furthermore, this vertex-based storage is also reflected in a type of field, which will be denoted as discontinuous fields in this thesis. Since there are two triangles in a quad cell and each has three individual vertices, these fields are stored as six separate and cell-centered arrays: fieldDisc_lower0, fieldDisc_lower1, fieldDisc_lower2 for the lower and fieldDisc_upper0, fieldDisc_upper1, fieldDisc_upper2 for the upper triangles of the quads. As will be shown, the discontinuous fields can mostly be written as they are without any sophisticated handling. However, for other field types which need to be adapted to this access pattern, the offset list can be made use of. For instance, for visualization the node positions, which are stored per grid node in memory, must be written for all triangle vertices instead. Since there is usually more vertices than nodes, e.g. in this example six vertices in comparison to four grid nodes, this, in return, means that a node’s positions is printed more than once. Therefore, for each quad cell the access pattern is described in form of $(x,y)$ offset pairs as shown on the right side of Figure 22. In summary, besides the callback mechanism, IR_AccessPattern is also responsible for handling such access patterns. By default, however, it is assumed that a databuffer is accessed regularly and specifying a list of offset pairs is optional.

### 6.2.2 C++ Standard Library Implementation

As previously described, the approaches for FPP and single-shared file I/O via locking were already employed for certain scenarios in the generator. Both approaches benefit from the flexibility that the stream-based I/O that the C++ standard supplies.

However, the I/O library from the standard can be further divided: the object-oriented I/O library and a set of functions similar to I/O in C [14, 15]. The former is used in the prior state of the generator and makes use of the streaming operators, i.e. `<</>>`. However, these are commonly used for formatting of data and are inappropriate for binary I/O [30]. Therefore, the interfaces have been adapted such that functions from the C-like I/O interface are used for binary I/O, whereas
when formatting to ASCII the existing implementation with the stream operators is utilized.

The interfaces for both I/O patterns have been developed in the IR_FileAccess_Locking and the IR_FileAccess_FPP classes derive the aforementioned abstract class. Moreover, since both make use of the same I/O library, methods tailored to transfer binary or ASCII data between databuffers and files are enclosed in the IR_Iostream trait. These also allow users to specify conditions which determine if a file access for an entry in the databuffer should occur. Additionally, users can also overwrite the default value for the separator, which is the space symbol. Another important similarity is that for serial simulations both I/O classes produce equivalent codes.

In essence, the main difference between both interfaces is the way they perform I/O in a parallel environment as shown in Listing 5. The listing demonstrated a simple example where each process writes the values of a scalar 1D field `double arr[NX]`. For higher dimensional datatypes, e.g. vectors, or when a access pattern was specified these would be written in an interleaved fashion via fine-grained accesses, i.e. by iterating over all involved discretization points and accessing all components for the current iteration individually. In this example, not only the the difference between the I/O patterns but also the main features are demonstrated.

On the left side, locking I/O is used in combination with stream operators for text output. In addition, the user set the condition to only print every second value and specified them to be comma-separated. As the name implies, locking I/O grants a process exclusive access to a file. This is achieved with synchronization barriers and a pre-defined order in which access is given. Here, a process simply queries if it is granted access. If that is not the case, it waits until the process in turn reaches the barrier. Note that when reading the file via locking, once the process currently in charge of accessing the file finished its I/O requests, it determines its pointer within the file before closing it. This pointer is then communicated towards the next process in turn so it can continue from there on.

On the right side, the C-like API from the standard library is used to write the array out in binary form. In contrast, the user specified no condition and separators are omitted. Since no condition is specified and the whole array is requested, e.g. no ghost or pad layers to be excluded, it can be written as a whole with a single function call. Instead of coordinating the access to a single file, I/O is handled such that each process is assigned an unshared file.

```c
// Locking I/O: ASCII output
if (0==mpiRank) {
  ofstream str("test.txt", std::ios::trunc);
  // potentially write header into file ...
  str.close();
}
for (int rank = 0; rank < rank; ++rank) {
  MPI_Barrier(mpiCommunicator);
  if (mpiRank==rank) {
    ofstream str("test.txt", std::ios::app);
    for (int i = 0; i < NX; ++i)
      if (i % 2 == 0)
        str << arr[i] << "," << "\n";
    str.close();
  }
}
```

```c
// File-per-process I/O: binary output
stringstream buildString;
buildString << "test_" << mpiRank << ".bin";
string filename = buildString.str();
ofstream str(filename, ios::trunc | ios::binary);
char* address = (char*)(&(arr[0]));
int byteSize = NX*sizeof(double);
str.write(address, byteSize);
str.close();
```

Listing 5: Exemplary C++ target codes writing a 1D array via the standard I/O library.

### 6.2.3 MPI-I/O Implementation

The implementation of I/O routines using MPI-I/O is rather similar to message passing. In fact, the starting point of the parallel I/O concepts originated from the rough idea to view write and read accesses to the file system as send and receive operations of messages [88].

The main features of the MPI-I/O implementation are demonstrated in the basic example implemented in Listing 6. Here, a rectangular 2D domain is partitioned into two blocks by halving the domain in x-direction. Each block is further divided into two fragments in y-direction. In total, each process has two fragments, each with a number of 3x3 inner points in addition to ghost layers at both start and end of each dimension, resulting to a total of 5x5 points. However, in this example
the ghost layers were chosen to be omitted for file transfer. The global distribution of the fragments’ inner points, the layout in memory and in the linearized file space are shown in Figure 23.

Figure 23: Global distribution of inner points in the 2D domain partitioned into a total of 4 fragments. The file is laid out in a canonical order. FX denotes the fragment id in the domain and PX the process id.

Listing 6: MPI-I/O example for writing a field in a 2D domain partitioned into two blocks with two fragments each to a binary file.

```c
// preparations
int globalDims[2] = { 6, 6 }, globalStart[2] = { 0, 0 }; 
MPI_File fh; 
MPI_Info info = MPI_INFO_NULL; // use default behavior
MPI_Status status;
for (int fragmentIdx = 0; fragmentIdx<2; ++fragmentIdx) {
    globalStart[0] = 3*fragmentIndex_1[fragmentIdx]; 
    globalStart[1] = 3*fragmentIndex_0[fragmentIdx]; 
    MPI_Type_create_subarray(2, globalDims, count, globalStart, 
        MPI_ORDER_C, MPI_DOUBLE, &(globalSubarray[fragmentIdx]));
    MPI_Type_commit(&(globalSubarray[fragmentIdx]));
}
MPI_Type_create_subarray(2, localDims, count, localStart, 
    MPI_ORDER_C, MPI_DOUBLE, &localSubarray);
MPI_Type_commit(&localSubarray);

// access file
MPI_File_open(mpiCommunicator, "data/src.bin", 
    MPI_MODE_WRONLY | MPI_MODE_CREATE, info, &fh);
for (int fragmentIdx = 0; fragmentIdx<2; ++fragmentIdx) {
    MPI_File_set_view(fh, 0, MPI_DOUBLE, globalSubarray[fragmentIdx], "native", info);
    MPI_File_write_all(fh, &(field[fragmentIdx][0]), 1, localSubarray, &status);
}
MPI_File_close(&fh);
```

Similar to the POSIX `open()` function MPI-I/O provides a `MPI_File_open()` function to create or open a file. This function accepts an MPI communicator, a file name, a file mode, e.g. read-only, hints for performance improvements and a file handle, which is modified and returned by the function. Note that the open function is collective, which means that each process in the specified communicator must participate in this call.

In order to store the data from the field to disk, preparations are made first. To recapitulate, for an efficient storage of this file layout (canonical), derived data types are utilized and the global data is stored with collective I/O. The first occasion of MPI derived datatypes is to describe a process's portion in the global domain as in the beginning the whole file is "visible" to all processes. These are created with the `MPI_Type_create_subarray` function and essentially describe a mapping of a process-local sub-array to the global array. Here, we pass information such as the global dimensions, the count of elements of the local array and the global dimensions. The global information can be computed with Internal Variables (IVs) and knowledge parameters provided in ExaStencils. For instance, global start can be determined with the fragment index in x- and y-direction, `fragmentIndex_0` and `fragmentIndex_1` respectively, and the number of rows and columns each process writes
which is $3 \times 3$. Note that in 2D the zero'th entry of the dimensions is meant for the rows and the first entry is for the columns. This datatype is then used to restrict a process’s view to its own portion of the file. A file view consists of a displacement, an elementary type and a file type. The displacement denotes an offset in bytes from the start of the file, the elementary type denotes the smallest entity for positioning and file access, and, in this example, the file type is the sub-array that was created. Since `field` is of type double the elementary type is chosen accordingly. In this example, the displacement is set to zero since only one field is written and the global offsets for it are already handled by the file type [60].

Once the view for a fragment is set, the field data can be written to file via `MPI_File_write`. The write function accepts the pointer to a buffer, an MPI derived datatype and the number of datatypes to be stored. Since the ghost layers still need to be excluded for I/O, a derived datatype to describe the whole in-memory layout is created. In this example, this is done analogously to file space with the only difference that only process-local dimensions were specified. The file is then written collectively (_all suffix) which allows the MPI-I/O layer to employ the two-phase I/O optimization technique with the information conveyed by all processes. Once the data is written, the file can be closed with `MPI_File_close`. Likewise the procedure of reading a field from file is almost identical. At first, a derived data types for a fragment’s memory space and the mapping between memory and file space are created. Then the `MPI_File_read` function is called and the file can be closed.

The handling for a fragment-wise layout is different. Here, each process writes all data of its first fragment ($3 \times 3$) followed by its next fragment’s data, etc. In this example, the storage order of the fragments would be: $F_0, F_2, F_1, F_3$. This way, each process has a contiguous slice of the file space. In other words, the starting half of the file space would belong to process $P_0$, whereas the other half corresponds to $P_1$. On the contrary, with the storage order dependent on the fragment-id, i.e. $F_0, F_1, F_2, F_3$, in this example this kind of continuity would not be given anymore. Thus, the former is employed in the scope of this thesis. In order to describe the global partitioning for this file layout, a fragment-dimension is introduced. In this example, the global dimensions would be specified as `int globalDims[3] = {4, 3, 3}` with the number of fragments being the slowest varying one. Since this adds an additional dimension to the ones of the domain, special handling is required and is implemented in the `IR_DataBuffer` class. Moreover, in contrast to the canonical layout it is not required to set a view for each fragment within a block. Here, a databuffer’s view is only set once per block such each process knows its starting position within the file. From here on, each process can access the data in a fragment-wise manner where each process’s individual file pointer is automatically incremented such that afterwards the next fragment can be written. Since setting a view is a collective operation, this can reduce synchronization costs.

### Implementation in the Generator

The code generated by the compiler is rather similar to the previous example. At first, preparations for the file access are made. Mostly, this involves the declaration of MPI-related variables and arrays used for the creation of derived datatypes. The file is then opened followed by read or write accesses for each databuffer made in the order these were specified. Depending on the memory layout of a databuffer, i.e. fragment- or block-wise, the statements are potentially wrapped around an `IR_LoopOverFragments`. Other from the example where the file displacement was set to zero, for multiple databuffers the corresponding displacement needs to be set to prevent overwriting a previous databuffer’s data. In this work, this is handled by accumulating the global sizes of a databuffer’s predecessors.

As can be seen, the main difficulty when using MPI-I/O is the creation of derived datatypes to describe the layout of data in file and memory. Note that in ExaStencils there exists a `MPI_Datatype` class already, but it is tailored for communication and does not supply datatypes such as the aforementioned sub-arrays. Thus, the implementation for the creation of derived datatypes designed for I/O is encapsulated in an additional `MPI_View` class. Since in the creation of a datatype mostly pointers for the dimensions are passed, `MPI_View` follows a similar route: it mainly expects an `IR_VariableAccess` for the arrays with the total dimensions, the start dimensions and the count respectively, which allows both local and global datatypes to be created easily. Similar to `IR_DataBuffer`, it’s companion object introduces a cache for already created datatypes. This way, already created datatypes can be reused which also minimizes the generated code base. In
case that a new derived datatype is created, it is declared as a global variable and code for the (de-)initialization is appended to the corresponding functions in the generated program. This way, each employed datatype is created and committed once and can be utilized during the whole simulation.

Besides sub-arrays, datatypes to describe the memory layout originating from the scope of GHODDESS have also been implemented. Note that this is not meant for discontinuous fields but for node-centered fields, e.g. the aforementioned node positions, which are output for each triangle’s vertices. For this purpose, a list of (x,y) offset pairs can be set in the IR_AccessPattern class. For this access pattern, another class of MPI datatypes is used, namely the indexed datatypes. In short, an indexed datatype is created via MPI_Type_create_indexed_block in which a list of displacements, constructed with the offsets pair, for a single quad cell is passed. This datatype is then used recurrently for all cells in the domain.

Since in collective I/O functions the calls to this function are matched among all processes [60]. This involves synchronization [18] it is important to ensure that each process participates in the call to prevent possible deadlocks in the simulation. This can cause problems when fragments are not valid for a field’s domain. In ExaStencils, it is possible that fragments for a sub-domain are defined but not valid in case that more than one domain is specified [49]. To check if a fragment is valid for a domain, an IV named IR_IV_IsValidForDomain can be used which corresponds to a boolean array with entries for all defined fragments. As a result, a fragment may be valid for one process but can be invalid for another process. Thus, handling for collective I/O in combination to a variable number of fragments on each process must be implemented. For this purpose, so-called I/O No Operations (NOPs) were introduced. In short, in case that a fragment is invalid, it participates in the call but passes arguments such that no data is actually read or written by it. As an alternative, each process collect the data of all valid fragments in a temporary buffer but this reduces the available main memory significantly. Another idea would be to put processes with identical distributions of valid fragments in communication groups, which access the file collectively. However, this limits the gain from collective I/O operations to begin with. Also, the atomicity of operations between these groups can become another problem. Since HDF5 and PnetCDF are build on top of MPI-I/O and also provide collective I/O operations, invalid fragments are also handled with I/O NOPs. This, in turn, also means that each process writes different amounts of data. When the file is laid out fragment-wise, this requires the knowledge on how many valid fragments each process contains to calculate the displacement in file correctly and avoid unnecessary gaps in the file. Therefore, code to synchronize the valid fragments is generated and executed once during the simulation. All this information is contained in IVs and managed in IR_IV_FragmentInfo. Since the following two interfaces handle these scenarios identically, this handling will not be recapitulated.

In contrast to the implementation with the C++ standard library, the data in this interface is only output in binary format. While it would be possible to output it in ASCII as well, this approach can be quite costly. At first, the data would need to be copied to a temporary buffer where it is then encoded. The increased amount of data originating from this encoding is an additional cost. These steps can degenerate the performance significantly. Despite the fact that ASCII files are more portable, writing the data in binary is more efficient and is the employed representation in this thesis. Moreover, MPI-I/O also provides the aforementioned data representations which can provide a degree of portability. However, this portability is only tailored for MPI-I/O applications and can cause conversion errors. Another problem is that MPI-I/O implementation do not necessarily need to implement the representations besides "native" [89]. While these representations are still supported in this work and error checking is performed in case these are not supported, achieving portability by providing an additional XML-based file would be more elegant. The error checking can be enabled via the parIO_generateDebugStatements flag. One of these XML-based formats is XDMF, as it supplies a standardized method to exchange scientific data between HPC applications with its underlying XML structure [41, 27]. As shown, meta-data such as the endianness, the dimensionality and the (primitive) datatype is provided in an XDMF file. In fact, for visualization data is written in the "native" representation and the corresponding meta-data is supplied in the XDMF file. This way, data is written from memory as-is and converted only on-demand.

Another constraint in comparison to stream-based I/O interfaces is that conditions are not supported for the MPI-I/O based interfaces. While this would also be possible, data is always required to be copied towards a temporary buffer first. Depending on the condition, each process might hold different amounts of data. In order to write to the correct (global) position within the file space, the sizes of each process’s temporary buffer must be communicated to the others such
that the file offset of each process can be computed. Moreover, this complicates the process of creating derived datatypes since for each time data is written, these must be created anew. Hence, for this type of file access it is encouraged to use stream-based approaches instead.

Moreover, using collective I/O for the fragment-wise layout may not be beneficial and independent I/O can be used instead. In order to determine whether independent or collective I/O is used, the \texttt{parIO\_useCollectiveIO} knowledge flag can be set.

6.2.4 HDF5 Implementation

This section covers the programming model of HDF5. Since HDF5 provides a rather extensive API, not all but the most relevant features for the implementation in ExaStencils are shown here. An exemplary target code for the previous domain distribution in Figure 23 is shown in Listing 7.

```c
// initialization
hsize_t stride[3] = { 1, 1, 1 }, count[3] = { 1, 3, 3 };  
hsizet localDims[3] = { 1, 5, 5 }, localStart[3] = { 0, 1, 1 };  
hsizet globalDims[3] = { 4, 3, 3 }, globalStart[3] = { 0, 0, 0 };  
hid_t fh, group;  
hid_t fapl = H5Pcreate(H5P_FILE_ACCESS);  
hid_t fcpl = H5Pcreate(H5P_FILE_CREATE);  
hid_t dcpl = H5Pcreate(H5P_DATASET_CREATE);  

// open file with MPI and handle groups
H5Pset_fapl_mpio(fapl, mpiCommunicator, MPI_INFO_NULL);  
fh = H5Fcreate("data/field.h5", H5F_ACC_TRUNC, fcpl, fapl);  
group = H5Gcreate2(fh, "/test", H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);  

// setup property list to enable collective I/O
hid_t transferList = H5Pcreate(H5P_DATASET_XFER);  
H5Pset_dxpl_mpio(transferList, H5FD_MPIO_COLLECTIVE);  

// describe memory space and select only the inner layers
hid_t memspaceField = H5Screate_simple(3, localDims, NULL);  
H5Sselect_hyperslab(memspaceField, H5S_SELECT_SET, localStart, stride, count, NULL);  

// describe file space, set chunks to the size of each fragment and create dataset
hid_t dataspaceField = H5Screate_simple(3, globalDims, NULL);  
H5Pset_chunk(dcpl, 3, count);  
hid_t datasetField = H5Dcreate2(fh, "/test/field", H5T_NATIVE_DOUBLE,  
dataspaceField, H5P_DEFAULT, dcpl, H5P_DEFAULT);  

// select each fragment's portion of the global dataset and write
for (int fragmentIdx = 0; fragmentIdx<2; ++fragmentIdx) {  
  globalStart[0] = 2*mpiRank+fragmentIdx;  
  globalStart[1] = 0;  
  globalStart[2] = 0;  
  H5Sselect_hyperslab(dataspaceField, H5S_SELECT_SET, globalStart, stride, count, NULL);  

  H5Dwrite(datasetField, H5T_NATIVE_DOUBLE, memspaceField, dataspaceField,  
   transferList, &field[fragmentIdx][0]);}  

H5Gclose(group);  

// close property lists, datasets, data spaces, ...
H5Fclose(fh);  
```

Listing 7: HDF5 example code for writing a field in a 2D decomposed domain using the fragment-wise storage layout.

On the contrary to the MPI-I/O code snippet, a fragment-wise file layout is used. As a result, a third dimension representing the index of a fragment is set as the slowest-varying dimension. Similar to MPI-I/O, the slowest varying dimension is stored at the zero'th index of a dimension.
array and the fastest varying dimension is stored at its last index. This is also referred as KJI order. The HDF5 API is distributed into sub-interfaces where each is meant for a specific HDF5 object type. For instance, functions starting with prefix H5F are used for controlling file accesses, e.g. opening or closing the file. In the generator, the most relevant functions have been encapsulated in the IR_HDF5_API trait to reduce the code base of the actual implementation for the I/O interface. Moreover, the conversion from primitive C/C++ datatypes to HDF5’s own set of datatypes, e.g. H5T_NATIVE_DOUBLE, is implemented here. As shown in the listing, HDF5 objects are opened, used and closed in the end.

One crucial aspect for the integration to the code generator is the miniscule difference between a serial and parallel application. In order to tell HDF5 that a file is opened in parallel, a property list for the file access, in short fapl, is set-up and the MPI communicator in addition to hints for the MPI-I/O layer are passed. For serial applications, on the other hand, this property must not be appended to the fapl. Furthermore, a file creation property list fcpl can be created to set the sizes of the underlying file structure for further optimization, e.g. the size of the underlying B-trees in the file. In this example, a default fcpl is used. Once the file is opened, further HDF5 objects can be created and added to the file. In the example, the user requested the field data to be stored in a group "/test". In the generator, the dataset paths of all databuffers are collected and distinct groups are extracted. Normally, code for error checking, e.g. if a group was already created, is generated but was left out in the example for the sake of simplicity. Additionally, a property list to specify whether to use independent or collective I/O is set-up.

As indicated in Section 4.2.3, datasets and dataspaces are the main objects when it comes to storage and retrieval of heavy data. In simple terms, datasets represent the application’s data structures and contain information such as the its name, datatype, storage layout and dataspace. In this example, the dataset consists a plain array of double values. The dataset’s dimensionality is described with a dataspace object and is set to the global dimensions of the field, i.e. containing the values of all fragments. When creating the dataset, all this information is passed in addition to a set of property lists. The property list for the dataset creation dcpl enables using storage layouts other from the contiguous layout. In the scope of this thesis, the chunked layout can also be used, mainly since it enables compression filters and potential for optimizations. GZIP can then be enabled for compression. However, this feature might not be usable for writing compressed datasets in parallel applications, especially when using older versions of HDF5\(^1\). The following knowledge flags to enable these features can be found in Section 6.5.1. Note that dataspaces are not restricted to regions in the file but can also be used to describe the dimensions of a buffer in memory.

Analogous to MPI derived datatypes, regions of interest within the memory and file space are selected using the concept of hyperslabs. In the example, a rectangular portion of both spaces are marked for transfer. More specifically, a fragment’s whole memory region is first described with an array defining the total dimensions of a fragment, i.e. including unwanted regions such ghost and padding layers. In order to only transfer a fragment’s inner layer, arrays containing the start index and the size of the corresponding rectangular sub-array are also passed as arguments for the hyperslab selection. The same procedure can be applied for the global file space. Here, each fragment is stored individually in a loop over all fragments within a block and each selects its corresponding portion of the file. The required dimensions to describe the size of both file and memory space in addition to the dimensions for the selection are encapsulated in the IR_DataBuffer class. Finally, when writing the field to the storage system, both memory and file space are passed as arguments. Again, in this example there is only one databuffer to be written. On the contrary to MPI-I/O, the handling for writing or reading multiple databuffers is different as HDF5 is a high-level library and abstracts from low-level details such as file pointers. Instead, for each individual databuffer there exists a dataset in the HDF5 file. Similar to the MPI-I/O datatypes, a cache for the dataspaces has been employed to allow reusing the same dataspace for databuffers with identical data extents.

The procedure of reading a dataset is rather similar. However, since the file space of the global dataset is already stored in the file, it can be queried from the file. Analogously, the memory space of a fragment is created and each fragment’s portion of the global dataset is selected via a hyperslab before it is read into memory via H5Dread. Most notably, the debugging mode can be enabled with the previously named knowledge flag. In this mode, error messages are printed in

\(^1\)https://www.hdfgroup.org/2018/04/why-should-i-care-about-the-hdf5-1-10-2-release/
case that a library call returned an invalid value. In addition, when reading a dataset from file the size of the dataspace is compared to the requested size of the corresponding read access in order to prevent reading datasets with mismatching dimensions.

Internally, for regular hyperslab selections HDF5 builds derived data types for the underlying MPI-I/O driver [45]. Moreover, both collective and independent I/O can be used for these selections. For non-regular selections, e.g. the aforementioned access pattern for applications from GHODDESS, independent I/O is used internally. Initially, this access pattern was implemented via the H5Sselect_elements function to specify a list of points to be accesses. However, this has led to a vast amount of small and individual accesses to the file system and degraded the performance significantly. Thus, for these access patterns data is first copied to a buffer and then written to file. This insight has been captured with the Darshan I/O profiling tool which provides compile time wrappers and dynamic library preloading to instrument applications. For these applications, a log file is produced which can then be converted to a PDF file with help of provided tools. This file depicts characteristics about the overall accesses to the file system, e.g. the number and size of individual accesses, the time spent in I/O and many more.

Finally, all instantiated HDF5 objects including the file handle are closed.

6.2.5 PnetCDF Implementation

In this section, the programming paradigm for the PnetCDF library is demonstrated. Like HDF5, PnetCDF is a high-level I/O library and provides abstractions from the view of data as linear stream of bytes as it is the case in low-level libraries, e.g. MPI-I/O or POSIX I/O. More specifically, it employs a data model based on multidimensional arrays of typed data. Listing 8 shows the main aspects of implementing an I/O kernel with PnetCDF. Again, the same domain partitioning from Figure 23 is used as an example and stored in fragment-wise order, however, with the twist of also storing the ghost layers and using independent I/O. For brevity, error checks are omitted.

```c
// initialization
MPI_Offset stride[3] = { 1, 1, 1 }, count[3] = { 1, 5, 5 };
MPI_Offset localDims[3] = { 1, 5, 5 }, localStart[3] = { 0, 0, 0 };
MPI_Offset globalDims[3] = { 4, 5, 5 }, globalStart[3] = { 0, 0, 0 };
int ncFile;
int varId;
int dimIds[3];
```

```c
// create file and enter "define mode"
ncmpi_create(mpiCommunicator, "data/field.nc",
    NC_64BIT_DATA | NC_CLOBBER, MPI_INFO_NULL, &ncFile);
ncmpi_def_dim(ncFile, "frag", globalDims[0], &dimIds[0]);
ncmpi_def_dim(ncFile, "Y", globalDims[1], &dimIds[1]);
ncmpi_def_dim(ncFile, "X", globalDims[2], &dimIds[2]);
ncmpi_def_var(ncFile, "field", NC_DOUBLE, 3, dimIds, &varId);
```

```c
// leave "define mode" and enter "data mode"
ncmpi_enddef(ncFile);
ncmpi_begin_indep_data(ncFile);
for (int fragmentIdx = 0; fragmentIdx<2; ++fragmentIdx) {
globalStartNode_00[0] = 2*mpiRank+fragmentIdx;
globalStartNode_00[1] = 0;
globalStartNode_00[2] = 0;
ncmpi_put_vara_double(ncFile, varId, globalStart, count, field[fragmentIdx][0]);
}
```

```c
ncmpi_close(ncFile);
```

1https://www.mcs.anl.gov/research/projects/darshan/
Listing 8: PnetCDF example code writing a 2D field including ghost layers in fragment-wise order via independent I/O.

As already mentioned, the serial and parallel version of netCDF employ a bimodal programming model. In the example, a CDF-5 file is created via the `ncmpi_create` function where an MPI communicator and a hint object are passed as arguments. For serial applications, these arguments are omitted. Initially, when a file is created, it is put into define mode. Here, the dimensions describing the shape of the global field are first defined. These in addition to a primitive datatype are used to create a variable representing the field. Despite the fact that this is not shown in the example, because the field is stored as a fixed-size array, an unlimited dimension can be defined and used as the most significant dimension of a record variable. In the scope of this thesis, record variables are also supported and find mainly use for visualization. Note that this introduces an additional dimension and requires appropriate handling. In the example, a record variable for the field would have four dimensions in total. In the generator, an optional truth table to determine if a databuffer is a record variable was introduced. The dimensions of the record databuffers are then adjusted automatically. Moreover, IVs for the time index and time value were also introduced.

Once the dimensions and variables have been defined, the header is written to file by the root process and the application switches over to data mode [59]. While it is possible to re-enter the define mode, this can cause a costly reordering and rewriting of the file and is therefore not supported in the generator. In data mode, the actual storage and retrieval of variable data is done. At this point, the application is put into collective data mode by default [1]. Hence, the `ncmpi_begin_indep_data` function is called to be able to perform independent I/O operations, i.e. call functions without ".all" suffix. PnetCDF provides multiple APIs for data access. In short, there are functions to store single elements (`var1`), whole (`vara`) and strided arrays (`vars`), and multiple non-contiguous regions (`varm`) in a variable. In the example, the `vara` functions can be utilized since the ghost layers are not omitted and the array can be written as a whole. The starting position for each file access is set in the fragment loop from which count contiguous field values are stored at once. For more complex access patterns, PnetCDF's flexible API can be used as it allows passing MPI derived datatypes as argument and describe the layout in memory with them. An exemplary function call for the flexible API is also shown in Listing 8. In the generator, the flexible API is used when non-contiguous regions in memory are transferred, e.g. when ghost layers are excluded or for the access pattern employed in GHODDESS applications. This way, copying to an additional buffer can be avoided. In ExaStencils, the existing MPI_View class has been reused for this purpose. Serial applications, however, cannot utilize the flexible API and use the `varm` API instead. Note that the differences between parallel and serial applications are rather small. The function names in serial applications begin with "nc_", whereas in parallel applications they begin with "ncmpi_". Similar to the implementation of the HDF5 interface, the functions for both serial and parallel APIs are encapsulated in a trait, namely `IR_PnetCDF_API`, which also provides debugging capabilities.

As in HDF5, meta-data including information about the dimensions and the datatype of a variable is stored in the file can be queried when a dataset is read. When a file is opened for read accesses, it is automatically put into data mode [1]. Then, the same sequence of steps can be applied to read a fragment’s portion of a stored variable.

6.2.6 SIONLib Implementation

SIONlib is fundamentally different to the MPI-I/O based interfaces as the task-local view is preserved despite writing to a single-shared file. In essence, SIONlib can be seen as an extension of the ANSI C interface and its stream-based I/O. In a sense, this implementation resembles the ones using the C++ standard library. This section introduces the main aspects of SIONlib’s API for MPI parallel applications. Since all previous libraries parallelize I/O on MPI level, SIONlib’s OpenMP-based APIs will not be part of this evaluation. Serial I/O, on the other hand, is supported. Since the well-known ANSI C and POSIX I/O functions can be directly used for SIONlib file containers, the main focus will lie on the process of creating the structure to allow task-local I/O on a shared file. An exemplary target code for writing the inner layers of a 2D field which is scalar and node-centered to file is shown in Listing 9.

Similar to the other shared-file approaches, the file is opened collectively via the `sion_paropen_mpi` function such that each task’s starting offset can be computed accordingly [34,
each process specifies its individual chunksize, that is the maximum number of bytes written in a single write call, from which the aforementioned block structure in a SIONlib file is built. The chunksize is determined by accumulation of all databuffers’ local dimensions, say the inner layers of an underlying field, times the number of bytes of its datatype. Naturally, when a non-default access pattern or higher dimensional datatypes are employed, their size is also taken into consideration. Identical to the implementation using C++ standard functions, for both cases the values are stored in an interleaved fashion. Most notably, not only the maximum size per write call but the total size written by a process is specified. Hence, the file’s chunks are organized in a single block. As mentioned, SIONlib allows the (logical) task-local files to be mapped to multiple physical files by passing a local communicator \(\text{lComm}\) for the tasks sharing an underlying physical file. Alternatively, \(nFiles\) can be set which splits the global communicator into groups. The resulting communicator and the filenames are returned in \(\text{lComm}\) and \(\text{newPhysFileenames} \) respectively. By default, the number of physical files is set to one but can be changed via the \text{sion_phys_files} knowledge flag. Another important parameter is the filesystem block size as it is used for the alignment of chunks. Here, we pass minus one in order to let it be automatically determined.

```
int nFiles = 1;
sion_int64 chunkSize = 2*(33*33*sizeof(double)); // 2 fragments, each with 33x33 grid nodes
sion_int32 fsBlockSize = -1;
int globalRank = mpiRank;
FILE* filePtr = NULL;
MPI_Comm lComm = mpiCommunicator;
char* newPhysFileenames = NULL;
int fileId = sion_paropen_mpi("test.sion", "bw", &nFiles, mpiCommunicator, &lComm,
    &chunkSize, &fsBlockSize, &globalRank, &filePtr, &newPhysFileenames);
```

```
for (int fragmentIdx = 0; fragmentIdx<2; ++fragmentIdx) {
    for (int i1 = 0; i1<33; i1 += 1) {
        for (int i0 = 0; i0<33; i0 += 1) {
            fwrite(&(field[fragmentIdx][35*i1+i0+36]), 8, 1, filePtr);
            // or alternatively:
            // sion_fwrite(&(field[fragmentIdx][35*i1+i0+36]), 8, 1, fileId);
        }
    }
}
sion_parclose_mpi(fileId);
```

Listing 9: Exemplary target code for writing a 2D field into a SIONlib container.

Once the function returns, a file handle for SIONlib (fileId) or ANSI C (filePtr) can be used for the functions of both APIs. Furthermore, the file pointer is set to a task’s corresponding position. From here on, all write operations can be performed individually. The SIONlib API provides wrapper functions such as \text{sion_fwrite} that tracks the remaining space in a chunk, allocates a new one on demand and sets the pointer accordingly. In this example, the limit of a chunk cannot be exceeded and thus the regular ANSI C \text{fwrite} is used. Besides, the ANSI C file handle can be converted to a file descriptor for POSIX I/O functions. All in all, these three APIs can be used in a mixed manner. Once the data is written, the file is closed collectively and meta-data collected from the tasks is stored on file [33].

Since most parameters of the aforementioned open function are passed via the call-by-reference semantic, this function can also be used to open a SIONlib file for read accesses. Here, the parameters are filled with information extracted from the file’s meta-data block. While the open and close functions are collective, the reading operations occur independently. These can be either performed in the limits of one chunk or without this limit, i.e. reading more than one chunk at a time. The former is performed via the ANSI C or POSIX functions, i.e. \text{fread} or \text{read}, for which the bytes left in the chunk can be queried via \text{sion_bytes_avail_in_chunk}. For the latter the \text{sion_fread} can be used to read more data than stored in one chunk and removes the need for users to loop over the chunks in order to do so.

In this example, the values on the ghost layers are excluded and the inner values are output via multiple \text{fwrite} calls in a 2D loop since these functions always expect pointers to contiguous memory regions. Note that despite the fact that the ANSI C functions use buffers internally and do not perform a system call for each I/O operation, the overhead of the library calls can
still be a dominating factor. In this example, one idea to reduce the calls is to write a whole row of inner layers at once, i.e. reducing the 2D loop to a 1D loop. However, this can only be done when no condition is specified and can still result in too many library calls. Therefore, a manual buffering technique has been implemented to minimize these calls and can be enabled via `parIO Streams use Intermediate Buffer` flag. For write accesses data is copied to a temporary buffer before finally writing it with a single POSIX `write` call. Likewise for read operations where the outer layers are excluded and no condition is specified, i.e. it is known how much data is read, the buffering can be employed as well. That is the data is read with one `read` call to a buffer where it is then copied to the specified destination. Note that the manual buffering can also be used when accessing binary data with the C++ standard library.

Another feature that SIONlib provides is that a file’s endianness is encoded in the metadata. This way, one can determine whether the endianness is different when a file is read on another architecture. SIONlib provides the `sion swap` function to perform byte-order swapping and `sion endianness swap needed` to check if swapping is necessary. In order to ensure that the byte order is swapped on-demand when reading a SIONlib file, the `sion ensure byteswap read` knowledge flag can be set.

Additionally, the aforementioned knowledge parameter to enable debugging capabilities can be set to check if each I/O request occurred successfully.

6.3 Visualization Interfaces

This work provides various interfaces for the visualization of field data on the different mesh types supported in ExaStencils. Note that in ExaStencils there were layer 4 functions for a specific application type, in which a pre-defined set of fields on an unstructured grid is printed to file, and functions for the visualization of one field usable in any kind of application. In this section, the focus lies on the former type of functions. This includes the implementation of different formats and a demonstration on how these are coupled with the presented I/O methods. Despite the VTK format’s popularity in many HPC application fields, the interface is not further extended in the scope of this thesis since it only supports binary and ASCII formats [92]. Datasets from HDF5 or netCDF files would need to be converted during post-processing in order to conform to the format. Instead, XDMF has been chosen as the core format as it enables visualization data stored in HDF5 files and, as shown, also provides a manifold of features. Unfortunately, it cannot be used for classic netCDF files and hence, additional techniques must be introduced. As mentioned, tools like ParaView and Visit supply readers for netCDF files following meta-data conventions and enable visualization at the cost of additional user effort. However, as shown these are not the most user-friendly when it comes to unstructured grid. Hence, for netCDF we introduce two interfaces: For (block-)structured grids we make use of the CF meta-data conventions but for unstructured ones, the ExodusII library and its format is used. VTK’s XML format would be very suitable for FPP I/O as it allows referencing domain pieces stored in separate files. However, XDMF also provides an identical mechanism and thus, to ease code maintenance, this I/O approach also makes use of the XDMF interface.

SIONlib is tailored for task-local I/O operations such as checkpointing, log files and scratch files [34]. Thus far, SIONlib files cannot be directly visualized. On the one hand, there are no readers for SIONlib files on the visualization tools such as ParaView and VisIt [24, 25]. On the other hand, there exist no formats such as XDMF which allow to reference the binary data within SIONlib files to enable direct visualization. Thus, the data stored in SIONlib files must be converted to another format in order to visualize it. Naturally, this step is done during post-processing. Here it is important to note that the alignment to file system blocks must be undone first. For this purpose, the `sion defrag` utility which comes with the installation of SIONlib can be used to defragment the data before copying it to another file of the requested visualization format. However, note that this step can be quite expensive and is only beneficial in case that the performance of SIONlib is far better than the other approaches. So far, the responsibility of converting the data for this purpose is given to the user and will not be covered in this thesis.
6.3.1 Organization

In the prior state of the generator, the VTK interfaces for unstructured meshes were contained in the `visualization.ir` package. As it was the only provided format, the whole infrastructure for the visualization of unstructured meshes was contained in the abstract `IR_PrintVtk` class. With the addition of new formats and interfaces, parts of infrastructure, which all involved interfaces have in common, were moved to the `IR_PrintVisualization` trait. All in all, this trait marks the top of the visualization class hierarchy. The overall structure is shown in Figure 24. Note that this figure is not complete for the sake of providing a better overview. Moreover, the class hierarchy has been separated into two parts which in the actual implementation are not separated at all.

In Figure 24.a, a hierarchy of traits is shown. These, again, were introduced to reduce the overall code base and are extended by the corresponding visualization interfaces. For instance, for all formats the connectivity of an unstructured triangular cell is defined identically and is therefore moved to `IR_PrintVisualizationTriangle` trait. Information such as the selection of fields to be visualized are found in the application-related traits. The fields and their layout can be used to easily determine the number of dimensions, nodes and elements.

The other part of the structure, shown in Figure 24.b, is the class hierarchy of the actual printer classes. Like in the former state of the generator, the separation of abstract printer classes in the `visualization.ir` package and their actual implementation in `application.ir` is kept. Naturally, these classes also make use of the aforementioned traits, e.g. the `IR_PrintXdmfSWE` extends both `IR_PrintXdmf` and `IR_PrintVisualizationSWE`.

Each concrete visualization class in the `application.ir` package has a counterpart in the DSL, i.e layer 4 functions callable by the user. These functions and their signatures are enlisted in Listing 10. Calls to these functions are then resolved in the compiler and transformed into instances of the respective class, e.g. the `IR_PrintXdmfSWE` extends both `IR_PrintXdmf` and `IR_PrintVisualizationSWE`.

In the compilation process, their `expand` function is called and in the end the resulting statement block is pretty-printed to C++ code. Since multiple I/O approaches can be utilized in combination with XDMF each combination has its own layer 4 function. All functions accept an expression of the filename and a multigrid level. By design, not only string literals can be passed as filename, but also expressions such as an access to a string variable which contains a newly build filename for the current step in a print series. Note that the filename should contain the corresponding extension of
the visualization format, e.g. ".xmf" as proposed in the listing, such that visualization tools can automatically choose the correct reader for the files.

Additionally, another contribution of this thesis is the adaption of the SWE printers to accept a list of fields to be visualized instead of a predefined selection of them. This is implemented such that users can flexibly pass accesses to fields as arguments. Here, we distinguish between discontinuous and node-centered fields. As mentioned, a discontinuous field is represented by six separate cell-centered (quad) fields. Hence, a convention for the field access list has been evoked: when a cell-based field was detected while iterating through the list, the next following five fields must have the same grid localization as well.

```c
// VTK printers (ASCII). Extension: ".vtk"
printVtkSWE(filename: Expression, level: Int, fields : FieldAccess*)
printVtkNS (filename: Expression, level: Int)
printVtkNNF(filename: Expression, level: Int)
// XDMF printers: file-per-process (binary or ASCII). Extension: ".xmf"
printXdmfSWE_fpp(filename: Expression, level: Int, useBinary: Boolean, fields : FieldAccess*)
printXdmfNS_fpp (filename: Expression, level: Int, useBinary: Boolean)
printXdmfNNF_fpp(filename: Expression, level: Int, useBinary: Boolean)
// XDMF printers: single-shared file with either: xyz = mpiio or hdf5
printXdmfSWE_xyz(filename: Expression, level: Int, fields : FieldAccess*)
printXdmfNS_xyz (filename: Expression, level: Int)
printXdmfNNF_xyz(filename: Expression, level: Int)
// ExodusII printers. Extension: ".e"
printExodusSWE(filename: Expression, level: Int, fields : FieldAccess*)
printExodusNS (filename: Expression, level: Int)
printExodusNNF(filename: Expression, level: Int)
```

Listing 10: Callable layer 4 functions for the visualization of data on unstructured grids.

### 6.3.2 XDMF Implementation

The XDMF format has not only been chosen due to its sophisticated feature set for visualization but also since it supports heavy data to be stored in multiple file formats: XML, HDF and Binary. This, in return, also allows for a coupling with multiple I/O interfaces, especially HDF5 and its file format. For the implementation of this interface, the association between formats and I/O interfaces plays an important role. While the association for HDF5 is obvious, for FPP I/O and its implementation with the C++ standard library data can be both represented in ASCII (XML) or binary. As mentioned, only a binary representation of data is supported in this work when using MPI-I/O.

Most notably, in order to create XDMF file we make use of native text output capabilities given in IR_Print instead of using the XDMF library. This eases the burden of the user, which would otherwise need to install another dependency.

In most cases, data is supplied in a separate file and the values are referenced in XDMF’s DataItems. However, due to the disadvantages of FPP I/O, keeping the number of files to a minimum is essential. In this implementation, the number of files is \(N + 1\) with \(N\) being the number of MPI processes. Similar to the XML-based VTK format the additional XMDF file, denoted as main file here, references the pieces from the global domain. For FPP storage in ASCII encoding, i.e. in XML, the main file references the XDMF files for each piece by simply specifying a path, whereas in binary mode special handling to reference each process’s raw binary data is required. While in most cases only one XDMF file is written and paths to other files with the heavy data are specified, in the XML case the data is directly incorporated in a piece’s XDMF file. Thus, functions responsible for writing XDMF elements distinguish between these cases.

Moreover, since single-shared file approaches store data in a global file in contrast to the process-local storage from FPP I/O, their differences in the data’s dimensions, file offsets, etc. must be handled as well. Accordingly, the printer of XDMF elements distinguish between global or local storage.
Structure

To briefly recapitulate XDMF’s data model, a file is structured hierarchically. On top of the hierarchy are the Domain elements which contain one or more Grid elements. Each Grid consists of a Geometry, Topology and Attributes. Actual values for them are specified via DataItems, whose dimensions, e.g. of the node positions, are supplied in the traits for the corresponding application. Naturally, this structure is also reflected in the implementation within the generator and is mainly represented by following methods:

**writeXdmfGeometry**(global: Boolean) This function is mainly responsible for providing the coordinates of the grid nodes. Since there are differences on how these are specified, e.g. the aforementioned access pattern from GHODDESS, the derived classes for a certain application type, e.g. SWE, implement this function. In ExaStencils the node positions can be retrieved by accessing the aforementioned virtual fields which, in case that they are actually stored and not replaced with simple expressions, are stored in a non-interleaved fashion. Thus, for a more efficient storage of these coordinates, instead of rearranging them to an interleaved order they are stored as in memory and specified in separate DataItems, e.g. for geometries with Type="X_Y".

**writeXdmfTopology**(global: Boolean) Since the focus of this section lies on the visualization of unstructured grids, the element connectivity and the mesh type, e.g. hex, are mainly specified here. As the element type and the connectivity differ for each application kind, this is specified in the derived class for the corresponding application.

**writeXdmfAttributes**(global: Boolean) Here, the subclasses specify which fields from the application are visualized. In summary, these values can be mapped towards a grid’s cells or nodes and represent a scalar or vector quantity. As face-centered variables do not map well for visualization tools [41], they are mapped to cell-centered variables via interpolation. As mentioned, for SWE applications the user can supply accesses to these fields as arguments for this purpose, whereas in NS simulations these are still pre-defined. For the former application, this implementation makes use of XDMF’s Function mechanism in order to join the values of discontinuous fields together so these are interleaved when reading the dataset. This way, each component can be written separately without having to interleave them in a temporary buffer during the simulation.

**writeXdmfGrid**(global: Boolean) This function prints an Uniform Grid element and is implemented in the abstract class. Here, the previous functions from the subclasses are called in order to do so.

**writeXdmf()** The skeleton for writing a whole XDMF file can be found in the abstract class. Here, in case of FPP a filename for each domain piece is created by appending the rank of a process, whereas otherwise the user-defined filename is directly used. For single-shared file approaches, the XDMF file is only written on the root rank. With these steps in mind this function opens the file, calls the **writeXdmfGrid** function and incorporates the resulting grid in a Domain element.

**writeData()** This function is responsible for writing the heavy data to a separate file. When data is stored as XML, it is already incorporated in each XDMF file. In this case this function will not be called. This function builds the bridge between this visualization interface and its supported I/O methods. From the user-defined filename for the XDMF file, a filename for the heavy data is built. This function is found in the abstract class and simply creates an instance from an I/O interface, denoted as I/O handler, by passing it the constructed filename and the databuffers for the topology, geometry and attributes specified by the subclass for the respective application.

**expand()** Naturally, all these concepts are utilized during the expansion. Furthermore, the handling for the main file in FPP is provided here. For this purpose, XDMF’s feature to specify a Collection of the Uniform grids is utilized. In the XML format the main file simply consists of relative paths to each piece, whereas in binary format the main file must be complete, i.e. contain
a Grid element with a Geometry, Topology and Attributes. Besides the handling of FPP main files, this function simply calls the `writeXdmf()` and `writeData()` function.

### 6.3.3 ExodusII Implementation

The supplied ExodusII interface enables a coupling with the PnetCDF interface. On the contrary to XDMF, for this approach a library is used and requires the user to install it first. This section focuses on giving insights to the library’s programming model. As mentioned, in this work the library is mainly used to supply a basic structure to enable the visualization of data. Once this infrastructure is provided, heavy data is stored via the PnetCDF library. For serial applications, however, the ExodusII library is directly used as it also makes use of the (serial) netCDF library as storage layer.

This section describes which steps in the target code are required to build this structure. A file is created via the `ex_create` function which takes a filename, a mode, I/O and computation word sizes for possible conversions as argument and returns a file handle used by all following functions [84]. Analogously it can be opened via `ex_open`. As mode, the EX_CLOBBER and EX_LARGE_MODE1 flags are set to erase files with an identical name and to create datasets greater than 2 GiB, i.e. CDF-2 format. The aforementioned CDF-5 format is not supported yet. It is also possible to use netCDF-4, but this feature is still in beta and not covered in this thesis [84].

Once the file is created, initialization data is defined. For this purpose, the `ex_put_init` function is called to set the number of dimensions, grid nodes, elements and blocks which is defined in the corresponding trait for the application type. Since the type of all elements is identical, e.g. triangular, all elements are bundled in one block. It is also possible to use node- or side-sets to reference a group of those together and reference those with a single ID, but this feature is not being used in this thesis.

Subsequently, the model description is given. Here, we specify the element block with the `ex_put_block` function, including its number of elements, the element type and the number of grid nodes per element. Afterwards, names for the coordinates are specified with the `ex_put_coord_names` function. Serial applications also make use of the library’s `ex_put_coord` function to provide the nodal coordinates for each dimension and the element block’s connectivity via `ex_put_conn`. These functions accept pointers to memory locations where the coordinate and connectivity data lies. Similar to XDMF, the node positions for each dimension are stored as separate variables.

 Afterwards, the last data group, that is the results data in form of nodal or element variables, is specified. For this purpose, the number of fields and their entity type, e.g. EX_NODAL or EX_ELEM_BLOCK, based on their localization are supplied with the `ex_put_variable_param` function. Additionally, serial applications define a time value `ex_put_time` and provide field values `ex_put_var` for a certain time step. In order to write field values the time step, entity type, index of both variable and block, and the number of values must be provided in accordance to the pointer of the memory location. While PnetCDF provides means to describe in-memory layouts via MPI derived datatypes, this concept cannot be used in serial applications. While there exist functions to describe multiple non-contiguous regions, these are not suitable for the 1 dimensional variables created by ExodusII. Thus, in case that ghost or padding layers are omitted, data must be copied to a temporary buffer before storing it. However, there is also a case in parallel applications where data must be reorganized before writing it, that is when writing discontinuous fields in a SWE application. For a correct visualization, the six components of such a field must be stored in an interleaved fashion. In XDMF, on the other hand, this problem could be handled with Function DataItems.

Finally, the ExodusII file is closed via `ex_close`. Note that for parallel applications, no data has been stored yet but a dataset where dimensions, meta-data, fixed-size arrays, e.g. for the coordinates and connectivity, and record variables for the field values were defined. This dataset is opened via PnetCDF and can be extended over the course of the simulation. As will be shown, the described steps to set-up an ExodusII file must only be done once. Afterwards data is stored via PnetCDF for each time step.

In the generator, the implementation of these steps is reflected in the abstract `IR_PrintExodus` class. Here, the differences between serial and parallel applications are also handled. Similar to the

---

XDMF interface, its subclasses for the different application types must specify a list of databuffers to be written. Besides, they also provide the entity type of the variables, the element name, e.g. "hex", and the number of nodes per element.

The coupling with the PnetCDF interface is more complex than in XDMF, especially since serial applications solely use the ExodusII library. In addition to the user-defined filename, the databuffers and a truth table indicating which databuffer is a record variable are passed to the I/O handler. Since the PnetCDF interface defined data as multidimensional arrays and ExodusII defines all variables as linearized 1D arrays, e.g. with the number of grid nodes as dimension, special handling for the storage of data is implemented in the abstract class. The parts besides storing the variables such as opening or closing the file, writing time values, etc. of the I/O handler are directly re-used in this interface.

Besides, this implementation also offers a debug mode which checks if errors occurred during library calls to ExodusII and prints an error message. This, again, can be enabled via the knowledge parameter parIO_generateDebugStatements.

6.4 Field I/O interfaces

This section covers the extension of the functions responsible for storage and retrieval of a single field’s data. Originally, these made use of two distinct parallel I/O techniques which was not explicitly made known to the user of the DSL. This interface was extended such that additional I/O methods are supported and the statements in the DSL reflect which one is applied. Essentially, these can be categorized as functions for storage and retrieval of a field to or from disk and functions to visualize a field. The counterparts of the statements in the IR, e.g. IR_ReadField, inherit from the abstract IR_FieldIO class which provides a generateFileAccess function to facilitate creating an I/O handler for each of the field I/O interfaces.

6.4.1 ReadField and WriteField

In the prior state of the generator, the printField function was not only responsible for printing a CSV file for a field, but also to store the plain values without the coordinate. For this purpose, the printFieldValues function could be used. While storing plain values this way is still supported due to backwards-compatibility, it is encouraged to use the new DSL statements given in Listing 11 to store a field’s data on disk. Note that the function signature for data retrieval, e.g. readField_hdf5, is identical to a data storage function, e.g. writeField_hdf5.

```c
// Locking: Binary or ASCII
writeField_lock(filename: Expression, field: FieldAccess,
                includeGhost: Boolean = false, useBinary: Boolean = false,
                condition: Expression = true, separator: Expression = " ")
// File-per-process: Binary or ASCII
writeField_fpp(filename: Expression, field: FieldAccess,
                includeGhost: Boolean = false, useBinary: Boolean = false,
                condition: Expression = true, separator: Expression = " ")
// MPI-I/O: only for MPI parallel applications. Binary
writeField_mpiio(filename: Expression, field: FieldAccess,
                  includeGhost: Boolean = false, canonicalOrder: Boolean = false,
                  repr: String = "native")
// (P)HDF5
writeField_hdf5(filename: Expression, datasetPath: Expression, field: FieldAccess,
                 includeGhost: Boolean = false, canonicalOrder: Boolean = false)
// (P)netCDF
writeField_nc(filename: Expression, varName: Expression, field: FieldAccess,
              includeGhost: Boolean = false, canonicalOrder: Boolean = false)
writeField_sion(filename: Expression, field: FieldAccess,
                 includeGhost: Boolean = false, condition: Expression = true)
```

Listing 11: New layer 4 functions to store field data on disk.

As can be seen, the function signature of each approach is different based on the features each I/O interface provides. For instance, stream-based I/O techniques like SIONlib or the C++
implementations support conditional file accesses. Moreover, MPI-I/O-based libraries provide an additional file layout for rectangular domains where data is organized in a canonical order. For HDF5 users can specify a path for the group on which the field’s dataset is stored in. In case that the groups are not already existent in the file, they are created automatically. Likewise, users of the PnetCDF interface can specify the variable name used in the file. However, due to its flat file hierarchy no paths are allowed. All in all, these functions are meant for the storage of plain values and can be directly translated into instances of the corresponding I/O interfaces.

Another trait of these statements is that for the same I/O interface read and write operations are symmetrical as long as the arguments passed to the respective functions are identical. In other words, this means that when a field is written to a file "foo.h5", modified and then read from this file, the field values are identical to the point of time when they were stored on disk.

6.4.2 PrintField

In contrast to the previous visualization interfaces, where multiple fields were output on unstructured grids for certain types of applications, e.g. SWE, the interfaces covered in this section are meant to provide visualization capabilities for a single field in any kind of application. In the prior state of the generator, this was done by providing the field values for each discretization point in a CSV file. While this approach works for each provided mesh type in ExaStencils, often more (positional) data than necessary is output. For instance, even when uniform structured grid are used, data is output in form of a mesh-free representation, i.e. the positions of each grid point is output. Furthermore, additional work during post-processing is required to display the data as structured grid again. Despite the fact that tools like ParaView provide filters\(^1\) for this purpose, it is an additional burden for the user. Therefore, means to visualize fields on (block-)structured meshes are also provided in this work.

For these functions, the association of I/O interfaces and visualization formats is rather similar. For backwards-compatibility, the CSV format is kept when using the locking mechanism. The supported I/O methods when using XDMF are the same. The main difference is that instead of using the ExodusII library we utilize the CF-conventions in combination with the PnetCDF I/O interface.

The corresponding DSL functions are enlisted in Listing 12. Similar to the other visualization statements, these expect the filename to contain the extension of the corresponding visualization format. Note that for most visualization formats ghost layers are not involved in the plot such that only relevant data is depicted. Also, the user is not obligated to remove them during post processing in order to only see the values at the inner layers. Moreover, interfaces such as MPI-I/O, HDF5 and PnetCDF provide the two aforementioned file layouts and handle their differences in the visualization interfaces accordingly. Specifically for XDMF, the visual results are identical independent from the actual storage order.

\(^1\)https://www.paraview.org/Wiki/ParaView/Data_formats

Listing 12: Layer 4 functions to print visualization files for fields on (block-)structured grids.

XDMF interfaces

Similar to the interfaces for unstructured grids, the classes illustrated in Listing 12 are organized hierarchically for the different mesh types. On the top of the hierarchy is the aforementioned \texttt{IR_PrintXdmf} abstract class which provides the functions responsible for print the different components of an XDMF file. This overall structure is mostly reused in the depicted subclasses.
**IR_PrintXdmfMeshless**  The mesh-less variant is an XDMF equivalent of the CSV format, i.e. the field is visualized in form of a scattered data plot, and this kind of representation can be enforced via the `parIO_vis_forceMeshlessVisualization` knowledge parameter, independent on the actual underlying grid. This can be used as fall-back in the future in case that ExaStencils is further extended with other mesh types. In contrast to CSV, the constant data reduction scheme which will be introduced in shortly can be applied.

**IR_PrintXdmfStructured**  For the visualization of fields on (block-)structured grids the hierarchy was extended by the `IR_PrintXdmfStructured` abstract class which encapsulates the steps necessary to print an XDMF file for such grids. Most notably, the resulting XDMF file is now organized such that a Grid element is defined per fragment, i.e. the `writeXdmfGrid(global: Boolean)` function was overridden such that a Collection of grids is printed. This has the advantage that no complex handling for adjacent fragments and the duplicate layers located at the fragment interfaces is required. In addition, this allows fragments not only to be organized in form of a rectilinear domain but also allows them to be organized as general block-structured grids, e.g. for a plus-shaped domain. Moreover, since on most I/O methods the XDMF file is written on the root process and information of the fragments from other processes is required, e.g. the start position of a fragment in each dimension, it is gathered once on the root process at the beginning of the simulation. Furthermore, since according to [41] face-centered variables are not encouraged to be used, these are interpolated towards the cell-centers here.

Another step that is handled in the abstract class is providing Attribute data, i.e. the field values, for a fragment. Since the field data is not only stored in a process-local or global file but also in a fragment-wise or canonical order this step must consider all these scenarios. When a file is laid out fragment-wise, extracting data for a fragment can simply be done by setting a DataItem’s pointer to corresponding byte offset via `Seek` option. When data is stored in an HDF5 file or stored canonically, the portion of the global dataset must be extracted via DataItems of type `Hyperslab` which consists of two further DataItems. The first specifies the slab in form of start, stride and count dimensions to be selected from the second DataItem. Assuming higher dimensional datatypes, e.g. vectors, are not stored in an interleaved fashion, their separately stored components are interlaced via `Function` DataItems.

The derived classes are responsible to specify their Topology and Geometry. Two dimensional uniform grids, for example, can specify a `2DCoRectMesh` topology and only need to provide a fragment’s origin and spacing, i.e. in total four values, in the domain as geometry data. For the other grid types, on the other hand, more geometry data is required which, in turn, means that an additional databuffer for the node positions must be passed to the I/O handler. However, when visualizing uniform grids via VisIt, it can occur that a reverse order for the origin and spacing entries is expected, resulting to an erroneous plot of the data. In this case, the `parIO_vis_generateVisItFiles` flag should be enabled.

**NetCDF interface**

For the visualization with PnetCDF, the CF meta-data conventions can be loosely followed in order to get a visual representation for the resulting `.nc` files. Particularly, the standard readers of ParaView and VisIt allow uniform and axis-aligned meshes to be visualized by specifying the coordinate values for each axis in the domain. In addition, VisIt provides a reader for 2D curvilinear meshes. For other mesh types, data could be visualized in form of scattered data. For instance, VisIt’s "Scatter" plot operator can be used for this purpose. Despite the fact that ParaView and VisIt provide readers for other conventions, these are tailored for specific application codes, e.g. FVCOM or CCSM, and mostly belong to the application domain of climate and forecast simulations and atmospheric modelling. In other words, data is often expected to be stored on spherical longitude and latitude coordinates. Moreover, the conventions for the different application types often derive from the CF conventions. One particular problem of the CF conventions is that cell-centered quantities are required to specify cell boundaries as data is normally expected to be node-centered [26]. However, specifying these for a 2D non-uniform grid would result in four additional values per cell center and is rather impractical. For this purpose, the SGRID\(^1\) conventions for

\(^1\)https://github.com/sgrid/sgrid
staggered grids can be used, however there are currently no readers in ParaView or VisIt. Therefore, these are currently output as node-centered quantities positioned on the cell-centers. Likewise, face-centered variables are positioned on the corresponding face-center. Another constraint with these conventions are higher-dimensional quantities. These will be not detected by the readers and must be output as multiple scalar components. Tools like ParaView’s 'Calculator’ can be used to create a higher dimensional datatype from multiple scalars.

6.5 Optimizations

In this section optimization techniques are presented. At first, parameters for tuning an I/O library’s performance are introduced. Afterwards, schemes to reduce the amounts of data output on disk are shown.

6.5.1 Hints

As shown in Section 4, the software stack consists of multiple layers including the file system and I/O interfaces ranging from the low-level POSIX I/O to higher-level libraries such as HDF5 and PnetCDF. In this section, the focus lies on demonstrating the employed hinting mechanisms that allow tuning the performance for the elements of each layer individually. In essence, these hints provide the implementations of I/O libraries information to further control the employed optimization techniques but also to impose restrictions to available resources. In summary, hints awaken the potential for an increased performance but do not change the semantics of the I/O interfaces [32, 89].

MPI-I/O Hints

```c
MPI_Info info;
MPI_Info_create(&info);
MPI_Info_set(info, "cb_nodes", "4");
MPI_Info_set(info, "romio_ds_write", "automatic");
MPI_Info_set(info, "romio_cb_write", "enable");
MPI_File_open(MPI_COMM_WORLD, "test.txt", MPI_MODE_WRONLY | MPI_MODE_CREATE, info, &fh);
...
```

Listing 13: Example of specifying hints for MPI-I/O via info objects.

As previously described, MPI-I/O provides hints to further tune its two major optimization techniques, namely data sieving and two-phase I/O. In section Section 6 the default behaviour was
always assumed, i.e. MPI_INFO_NULL. However, the default values are often not suitable for an
application. In MPI-I/O hints are generally specified per file, that is for actions such as creating
or opening a file and setting a view. As shown in Listing 13, an info object is filled with key-value
pairs and is passed to the open function. Note that hints are not only meant for the configuration
of the two optimization techniques but can also be used to describe file system parameters such as
a file’s stripe size and the stripe count in Lustre. The MPI standard defines a set of reserved keys
for the info objects which have a prescribed functionality. In case that an implementation of the
standard supports the interpretation of these keys, it must follow the provided functionality, e.g.
that the implementation uses four nodes for collective buffering as in the example. Otherwise, the
key is ignored [32].

In ExaStencils, these hints were integrated in form of knowledge flags. A comprehensive list of
the parameters and the possible values is shown in Listing 14. Since all MPI-I/O based libraries
also make use of the hinting mechanism, it has been encapsulated in the abstract IR_IO_Hint
class which is basically an IV of type MPI_Info. I/O interfaces that want to make use of this technique
simply need to create an instance of a subclass, e.g. MPI_Info(), and call the setHints() function
to generate code responsible for setting up the key-value pairs for the corresponding IV.

```java
// Striping parameters
// 0 = off or externally defined, else = user-defined
var stripe_count : Int = 0 // number for OSTs
var stripe_size : Int = 0 // number of bytes written to one OST

// Data sieving
// "automatic" = default, otherwise: "enable", "disable"
var romio_ds_read : String = "automatic"
var romio_ds_write : String = "automatic"

// Collective buffering
// 0 = default, -1 = auto = number of OSTs, else = user-defined
var cb_nodes : Int = 0
// 0 = default (4194304 = 4 MB), else = user-defined
var cb_buffer_size : Int = 0
// "automatic" = default, otherwise: "enable", "disable"
var romio_cb_read : String = "automatic"
var romio_cb_write : String = "automatic"
```

Listing 14: Knowledge flags for MPI-I/O performance tuning.

HDF5 Property Lists

HDF5 follows a similar approach for the hinting mechanism with its concept of property lists.
Moreover, since PHDF5 utilizes MPI-I/O as file driver, the aforementioned hints can also be passed
when setting up the property list for parallel file access via H5Pset_fapl_mpio. In addition to this,
other properties for the optimization of HDF5 on Lustre file systems are employed as proposed
in [40].

One possible optimization technique for HDF5 is to use the chunked storage layout in which a
dataset is split into equally sized chunks. Each chunk is a contiguous block of data in the file and is
written or read as a single I/O operation [38]. This layout not only grants optimization potential,
but enables features such as multiple unlimited dimensions and filters, e.g. compression. However,
the chunk size must be chosen carefully as this can prove to be a performance pitfall [31]. The
work in [40] proposes choosing the chunk size such that each process maps to one chunk in the file.
However, since in ExaStencils the number of fragments for each process may differ, the chunk size
was set to the requested extents of a fragment, e.g. the inner layers.

Similar to SIONlib, HDF5 allows data objects to be aligned with a advantageous block boundary.
While in SIONlib this is more or less done automatically, i.e. by passing -1 as argument for the
file system block size, in HDF5 the files are typically meant to be more compact. While alignment
increases the size of the file, it can improve the performance significantly as lock contention can be
reduced. The H5Pset_alignment(hid_t fapl, hsize_t thres, hsize_t align) function can
be used to set the alignment for the file access property list fapl. The threshold thres specifies
at which byte size an HDF5 is actually aligned on an address beginning at multiple of \texttt{align}. Accordingly, the chunks of each process can be aligned with block boundaries, e.g. the Lustre stripe size. One concern for chunking can be meta-data operations. Chunked datasets employ a B-tree for the mapping between requested elements at given array coordinates to offsets in the file for each individual chunk. While the B-tree is regularly meant to be compact in both depth and breadth, its width can be configured towards the total number of chunks via the \texttt{H5Pset_istore_k(hid_t fcpl, unsigned ik)} function where \texttt{fcpl} is the file creation property list and \texttt{ik} the half of the tree’s rank to store chunked data. In HDF5, the number of entries before splitting a node in the B-tree is \texttt{2*ik}. Thus, \texttt{ik} is set to the half of the total number of chunks. This allows minimizing the number of small I/O operations to bring the B-tree into memory and thus allows mapping the requested elements at a given array coordinate to the file offset more efficiently. This is done automatically when chunking is enabled [40].

```c
// use chunked storage layout
// size is automatically set to the extents of a fragment/block
var hdf5_use_chunking : Boolean = false

// use zlib compression when writing a dataset
// compression rate in range [0,9]. 0 = no compression
var hdf5_write_zlib_compression_level : Int = 0

// automatic meta-data evictions.
// careful: if the application crashes before meta-data is flushed to file, the file is unusable
var hdf5_auto_metadata_flush : Boolean = true

// threshold and alignment of HDF5 objects in bytes
// default = 1, -1 = auto = stripe_size, else = user-defined
var hdf5_object_alignment_threshold : Int = 1
var hdf5_object_alignment_size : Int = 1
```

Listing 15: Tunable parameters for HDF5.

Similar to PnetCDF, caching of meta-data is applied in HDF5 to enable faster access times for recently used objects. While in serial applications dirty meta-data items are just evicted individually to file, in parallel applications flushing the meta-data cache is done by batching the evictions into so-called eviction epochs to reduce the communication overhead. An eviction epoch ends when the amounts of dirty meta-data exceed a specified threshold. Here, synchronization takes place and the oldest dirty meta-data is flushed to file. The frequency of these epochs can be configured in HDF5 via the file access property list. Ultimately, automatic flushes can be disabled completely and may improve the performance since each flush causes small writes to the file system which can degrade the overall performance. In this case, the application is responsible for flushing the meta-data to file. This can be done via \texttt{H5Fflush} or when the file is closed. However, note that in case of crashes occurring before meta-data flushes can cause the file to be unusable [40]. In ExaStencils, the automatic flushing of meta-data can be disabled completely via the corresponding knowledge parameter enlisted in Listing 15.

**PnetCDF Hints**

```c
// 1 = default, -1 = auto = stripe_count, else = user-defined
// alignment of the header.
var nc_header_align_size : Int = 1

// alignment of fixed-size arrays.
var nc_var_align_size : Int = 1

// alignment of record variable section.
var nc_record_align_size : Int = 1
```

Listing 16: Knowledge flags for PnetCDF hints.

Likewise, PnetCDF can also utilize the MPI-I/O specific hints. Unlike HDF5, however, it does not employ an additional concept to provide hints meant for the abstraction levels provided by netCDF,
e.g. for variables or records. Instead, netCDF hints are also passed via info objects and interpreted by the PnetCDF implementation. These hints are then used internally or converted to MPI-I/O hints [59]. The PnetCDF hints employed in ExaStencils are shown in Listing 16. The hints meant for PnetCDF have been encapsulated in the subclass \texttt{PnetCDF\_Info()}. 

6.5.2 Data Reduction

Since the time spent with I/O strongly depends on how much data is accessed, the first optimization approach focuses on reducing the data extents. In this section, we introduce two data reduction schemes tailored for the visualization interfaces.

The first scheme is the so-called \textit{nodal data reduction} which was mainly developed for applications from the domain of GHODDESS. The general idea of this method is to omit the discontinuity information between adjacent elements from the triangulation by printing the field values per grid node instead of for each element’s individual vertices. For instance, in the triangle mesh depicted in Figure 26, the total number of vertices corresponds to the number of triangles, which is two times the number of quads \((2 \times 4^2 = 32)\), times three \((32 \times 3 = 96)\). In comparison, the number of grid nodes is \(5 \times 5 = 25\). Asymptotically, for finely resolved meshes the amounts of data can be reduced to one sixth as \(\lim_{n \to \infty} \frac{(n+1)^2}{n^2 \times 6} = \frac{1}{6}\) with \(n\) being the number of quad cells in each dimension. Therefore, storing the values per grid node can reduce the data extents significantly and can be viewed as a trade-off for the missing discontinuities. In order to enable this reduction, a mapping of data stored per vertex, in our case in discontinuous fields, towards a node-based storage needs to be defined. This is done by calculating the average over the field values on the vertices at which the triangular elements join at a common node. The results are then stored to a node-centered temporary buffer which is then written to file. In case of C++ streams, the computed values can be written directly without having to employ a temporary buffer. Figure 26 illustrates the most important aspects of the reduction. In general, four different cases, classified by the number of vertices per node, need to be distinguished. Depending on the case, a different number of discontinuous field entries, denoted by the orientation of the triangle and a vertex index \(i \in [0, 2]\), are considered for the average computation. For instance, in the lower left corner of the aforementioned figure the lower triangle’s zero’th entry of the discontinuous field can be mapped directly without having to compute an average. For the inner points, on the other hand, six values per grid node are considered for averaging. The reduction can be used by enabling the knowledge parameter \texttt{swe\_nodalReductionPrint} and only has an effect on the \texttt{printVtkSWE}, \texttt{printXdmfSWE} and \texttt{printExodusSWE} statements.

![Nodal data reduction scheme](image)

Figure 26: Nodal data reduction scheme. The shaded areas depict the triangles involved in the average calculation. The corresponding values of the discontinuous field are averaged over the vertices that share the highlighted grid node.

The second method, called the \textit{constant data reduction}, is more general and can be applied for any application with a mesh that does not vary over the course of the simulation. The general idea is to write constant data, in our case mostly geometrical and topological information such as node positions and element connectivity, only once in the beginning and re-use it afterwards. In the prior state of the generator, this approach was not possible since it was not supported by legacy

---

\(^1\)http://cucis.ece.northwestern.edu/projects/PnetCDF/doc/pnetcdf-c/PNETCDF\_HINTS.html
VTK and the CSV format. Thus, this method can only be used with the new visualization formats. As mentioned, XDMF allows including and referencing parts of other files and hence enables reusing data such as the geometry and topology information. Despite the fact that it would be possible to store the XDMF description for all print calls in a single XDMF file, all XDMF printers assume to write a series of files when the constant data reduction is used. The file, or parts of it, containing the constant data is then referenced by the other files of the series. A file name for each print call can be build with the `buildString` function from ExaSlang 4. This decision was made since parsing a single large XDMF file in ASCII format can be very expensive and often causes visualization tools to crash, especially when also the data values are directly incorporated in the file. The classic format of netCDF, on the other hand, also provides means to reduce constant data duplication indirectly. Other from a visualization with XDMF and its referencing mechanism, netCDF requires the mesh data to present in each file. However, the mesh data is written once in form of fixed-size arrays, whereas the varying data is stored as record variables. Therefore, when this reduction is applied, the file name is expected to be equal for each print call in order to work properly. In order to memorize that the constant data was already written, an IV with the name of the file containing the constant data has been introduced. Algorithm 1 sketches the general procedure of the constant data reduction in pseudo-code. When printing for the first time, the data buffers declared as constant are written in addition to the varying ones. Once the data was written, the file name containing the constants is stored in the IV. In the following print call, the IV is not empty any more and thus the else-branch, where only varying data is output, is executed. This reduction scheme is enabled by default and can be disabled with the knowledge flag `parIO_vis_constantDataReduction`. In case that it is disabled, the algorithm is adapted such that the if-condition is always true and hence, the constants are always written as well.

```plaintext
init
  constantsWritten ← empty string;
  filename ← user-defined expression for file name;
  buffersConst ← data buffers holding constant data;
  buffersVarying ← data buffers holding time varying data;
if constantsWritten is empty then
  for constBuf in concat(buffersConst, buffersVarying) do
    write(filename, buf);
  end
  constantsWritten = filename;
else
  reference constant data from file constantsWritten;
  for constBuf in dataBuffers do
    write(filename, buf);
  end
end
```

**Algorithm 1**: Procedure when constant data reduction is enabled.
7 Evaluation

The focus of this section lies on the comparison of the different I/O access patterns and the libraries which were presented over the course of this thesis. The different approaches and their characteristics are compared to each other based on a set of criteria which are crucial for both users and developers of ExaStencils and GHODESS. Naturally, this also includes the performance of the individual approaches. For this purpose, applications from the domain of the GHODESS project were chosen as test scenarios and the I/O performance was measured on the Emmy cluster. In the end, the visualization capabilities for these applications are demonstrated.

7.1 Usability

One key aspect for each of the presented interfaces is their usability, i.e. in terms of user-experience. While this is generally a very broad term, this criterion is further divided into four sub-categories.

File management The manageability of the files produced by the individual I/O techniques plays an important role for the user-experience. In general, we distinguish between FPP and single-shared file approaches here. As already mentioned, FPP approaches have several issues for this criterion as additional effort during post-processing is necessary to rearrange or merge the data from the individual files together. Moreover, on compute clusters the number of files a user can create is often limited. Single-shared file approaches do not have these management issues since only one file is created and shared amongst the processes. Additionally, there are also several parallel I/O libraries which employ semantics and optimization techniques which allow storing distributed data in a canonical order, i.e. as if the data was never partitioned. Despite the fact that this does not hold true for SIONlib with its chunk-based file layout, meta-information, such as the endianness and the number of tasks that wrote the file, in addition to the command-line tools that are provided, facilitate the procedure of remapping data when the file is used with a different number of tasks, e.g. when restarting a simulation with a different amount of compute nodes.

Dependencies Dependencies for external software components also play an important role in terms of user-experience. Here, the most convenient ones are the C++ standard library interfaces, i.e. for locking and FPP, and MPI-I/O. As already mentioned, MPI-I/O is implemented in many MPI implementations and therefore does not impose users to install additional software to perform parallel I/O. While the other libraries are normally required to be installed before utilization, these can further be categorized as light- or heavy-weight. SIONlib does not rely on external software except optional bindings for MPI or OpenMP, hence it is rather light-weight. In contrast, PnetCDF and HDF5 depend on other software libraries. Thus, their installation is a bit more time-consuming.

Debugging Potential Another factor is the debugging potential which is very pronounced in high-level libraries such as HDF5 and PnetCDF. Not only do they facilitate finding problems in faulty programs with comprehensive error messages, there exists multiple visual browsers such as HDFView and NcView to quickly inspect the binary datasets. While SIONlib also provides tools for managing files in its own format, these are not as refined as for the previously named formats. As MPI-I/O can store data in binary and ASCII, regular text or hex editors can be used for to inspect the stored data, however, for binary data this step is rather laborious.

Feature Set Since MPI-I/O and SIONlib are rather low-level interfaces and mainly provide means to store or retrieve data from or to files, the differences of the feature sets between HDF5 and PnetCDF are discussed here. In short, PnetCDF employs a relatively simplistic data model which is based on multidimensional array data with at most one unlimited dimension, whereas HDF5 supplies a more flexible and sophisticated data model. It allows for a description and storage of even the most complex data structures, e.g. structs, and several unlimited dimensions. However, for the current requirements in ExaStencils, PnetCDF’s data model has proven to be sufficient. HDF5 also allows applying filters to datasets such as (de-)compression and techniques such as combining multiple files via software mounting [59]. As will be later discussed, PnetCDF provides means for non-blocking I/O. All in all, HDF5 is the more feature-rich library, whereas PnetCDF sacrifices these features for an improved performance [59].
7.2 Portability

An aspect which has been extensively discussed in this thesis is the portability of the approaches. MPI-I/O provides a degree of portability with different representations for the binary data. However, converting the binary data to these non-native representations can cause conversion errors and degrade the performance. Moreover, this portability is only given when using the MPI-I/O layer. Additionally, an MPI implementation is not obligated to support these representations. While storing data in ASCII format is another possibility, the additional conversion cost and increased amounts of data can further reduce the performance. Providing an external description of the binary data, e.g. in form of meta-information encoded in an XML file, is a more efficient solution. High-level binary formats such as netCDF and HDF5 incorporate meta-data in the file and are fully portable across different machine architectures and (mostly) library versions. The possible conversions, e.g. endianness, when switching between different systems occur fully transparently. Furthermore, these formats also enable users to specify attributes within the file to further annotate the stored datasets. SIONlib also provides a degree of portability, e.g. by storing meta-data for the endianness of the data, but cannot be compared to the portability of netCDF or HDF5.

7.3 Performance

The most dominant factor for the comparison of the different I/O interfaces is the I/O performance. In order to get more insight about the scalability and overall throughput for the individual I/O access patterns and libraries, measurements for applications in different configurations are made. At first, the characteristics of the target system are briefly explained. Afterwards, the measured test scenarios are introduced and their I/O performance is discussed.

Setup

As previously mentioned, the performance of the individual I/O interfaces was measured on RRZE’s Emmy cluster. It consists of 560 compute nodes where each is equipped with two Xeon 2660v2 Ivy Bridge chips. Each chip has ten physical cores running at a frequency of 2.2 GHz and employ a 25 MB shared cache. Moreover, each compute node has 64 GB of RAM and is interconnected to the others over InfiniBand with a 40 GBit/s bandwidth. Most notably, it provides a Lustre variant called LXFS as file system with 400 TB capacity and an aggregated parallel I/O bandwidth of 7000 MB/s.

In terms of I/O libraries, HDF5 version 1.12.0, PnetCDF 1.12.1 and SIONlib 1.7.6 were employed. Additionally, the OpenMPI 3.1.1 implementation was used and an MPI process was always pinned to a socket. The I/O bandwidth was calculated with the measured file size in MB and the time spent in I/O in seconds, which was averaged over all processes with help of ExaStencils timing module.

Test Scenarios

For the measurements, following test scenarios and configurations have been chosen:

- **Analytical_L8**: This application is an artificially manufactured analytical solution on an uniform mesh and is on the resolution level 8, i.e. with \(2^8 \times 2^8\) quad cells per fragment. The scalability of the approaches is tested by increasing the number blocks, starting with 2 blocks and doubling this amount until 128 blocks are reached.

- **Analytical_L5**: This application itself is identical to the previous one. However, its configuration is different. The resolution has been reduced to level 5. Moreover, instead of increasing the number of blocks and thus fragments, the distribution is modified such that a total of 80 fragments is distributed amongst 4, 10, 20, 40 and 80 MPI processes. For these measurements, a notation of \(^*bX_fY^*\) is used where \(X\) corresponds to the number of blocks and thus MPI processes, whereas \(Y\) symbolizes the number of fragments.

- **Bahamas_L5**: This application simulates a tidal flow at the Bahamas islands and therefore is a more realistic example. Here, two different distributions of blocks and fragments are measured: 82 fragments distributed amongst 4 or 82 MPI processes.
In the following examples, SIONlib is solely used for performance comparison. That is, data is output as in the other interfaces, i.e. the same databuffers in the identical order, but no visualization files are output. Since the size of external visualization files, e.g. XDMF, is typically in the range of tens of kilobytes, this should not falsify the measurements.

Additionally, not each of the presented reduction schemes were employed for the measurements. While the nodal data reduction can theoretically reduce the amounts of data to one sixth of the original, the same does not apply to the time spent in I/O. Thus, only the constant data reduction was enabled for all following measurements.

Measurements

The performance results for the Analytical\_L8 examples are split into two graphs for a better visibility. Figure 27 depicts the I/O bandwidth of the stream-based interfaces, i.e. SIONlib, FPP and locking, in relation to the number of MPI processes. Note that the y-axis in all following plots is in logarithmic scale. As can be seen, the locking mechanism has a constant bandwidth of roughly 20 MB/s as there is only one process capable of writing to file at a given point of time. For file-per-process approaches, the bandwidth increases with the number of processes and gets close to the peak bandwidth at roughly 5600 MB/s when storing binary files. As can also be seen, the meta-data bottleneck for FPP has not been reached for these measurements due to the small number of processes. Overall, SIONlib has also shown a good scaling.

Figure 27 depicts the performance for MPI-I/O, PnetCDF and HDF5 using collective and independent I/O. At larger scale, MPI-I/O achieves a maximum of 450 MB/s whereas PnetCDF stays in the range of 200 MB/s to 300 MB/s. Overall, HDF5 did not achieve promising results. This is due to the fact that these measurements are not tuned, i.e. no modifications to the knowledge parameters for the hints were made.

![Figure 27: Analytical\_L8: I/O bandwidth for stream-based I/O approaches.](image)

A comparison between tuned and non-tuned collective I/O performances is shown in Figure 29. Here, ROMIO’s collective buffering and data sieving have explicitly been enabled. Moreover, the number of the aggregator processes has been set to one fourth of the total number of processes and their the buffer size been increased to 25 MB such that an aggregator can hold the data of the other nodes without having to perform multiple aggregations for a single I/O request. Moreover, the Lustre striping parameters were set such that the number of OSTs equals the number of aggregator nodes with a stripe size of 1 MB. However, also in other constellations, these parameters only had little effect. For HDF5 and PnetCDF, the alignment has been set matching to the stripe size.
Naturally, these measurements use the non-aligned file sizes to calculate the overall bandwidth. For HDF5, automatic meta-data flushes were disabled. In addition, chunking was disabled as it did not achieve good results. As can be seen, the HDF5 performance increased significantly, whereas for MPI-I/O and PnetCDF the gains at 32 processes are rather small.

Figure 28: Analytical_L8: I/O bandwidth for the parallel I/O libraries MPI-I/O, HDF5 and PnetCDF.

Figure 29: Analytical_L8: Tuned I/O bandwidth for the parallel I/O libraries MPI-I/O, HDF5 and PnetCDF at smaller scales.

The measurements for the Analytical_L5 example are also shown in two separate plots. Figure 30 depicts the I/O performance of the stream-based I/O interfaces for the various distributions of blocks and fragments. With an increasing amount of blocks, the number of fragments per block decrease.

74
As a result, when locking is applied, the performance decreases with a higher number of MPI processes since more synchronization barriers are issued. On the contrary, the performance of the FPP approaches increases as more data can be stored in parallel. For SIONlib, the measurements for all constellations were rather similar. Moreover, increasing the number of OSTs and setting the stripe size to the automatically set chunk size of 4 MB in SIONlib files did not improve the performance.

Figure 30: Analytical_L5: I/O bandwidth for stream-based I/O approaches.

The collective I/O performance of the MPI-I/O based approaches are shown in Figure 31. Independent I/O has shown even more variance as in this graph and could not give any useful insights. One possible cause can be that the resolution of each fragment is too small and multiple processes access the same file system block, causing a serialization of the accesses. With collective I/O, on the other hand, the number of processes accessing the file system is the number of aggregators which has been set to a maximum of 4 with a buffer size of 8 MB. The other knowledge flags have been set as in the Analytical_L8 example. In average, MPI-I/O performed slightly better than the high-level interfaces.

As shown in Figure 32, the scaling for the two Bahamas_L5 measurements is rather similar to the one of the Analytical_L5 example when one considers its two extremities, that is "b4_b80" and "b80_b80". Furthermore, hint parameters identical to the ones of both extremities were passed. Arguably, similar results for fragment distributions like those used in the Analytical_L5 example can also be expected in the Bahamas_L5 scenario.
Another key criterion is the size of the files output on disk. A comparison of the accumulated sizes of plot files which were output over the course of the Analytical_L5 scenario can be found in Table 1. More specifically, visualization files were output fifteen times and no padding was applied. Here, the effectiveness of the aforementioned data reduction schemes is also shown. The sizes of the data and potentially separate visualization files are included in the measurements, except for the SIONlib files which only consist of plain data values and would need to be converted to another format first to enable visualization. Each file contains the fields: bath, eta, u, v and order. Bath is constant.
and can be stored once for the Xdmf printers. Note that the SIONlib example is rather artificial since these only contain raw data but no description to enable a visualization.

<table>
<thead>
<tr>
<th>Parallel I/O approach</th>
<th>no data reduction</th>
<th>nodal data reduction</th>
<th>const. data reduction</th>
<th>both reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy Vtk ASCII</td>
<td>745.448507</td>
<td>174.665447</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>XDMF FPP ASCII</td>
<td>984.118927</td>
<td>213.158977</td>
<td>561.27027</td>
<td>102.238377</td>
</tr>
<tr>
<td>XDMF FPP binary</td>
<td>443.276137</td>
<td>102.975087</td>
<td>250.516699</td>
<td>46.097269</td>
</tr>
<tr>
<td>XDMF MPI-I/O</td>
<td>442.459342</td>
<td>102.703977</td>
<td>249.775138</td>
<td>45.898495</td>
</tr>
<tr>
<td>Xdmf HDF5</td>
<td>442.694272</td>
<td>102.809187</td>
<td>249.970840</td>
<td>45.964477</td>
</tr>
<tr>
<td>ExodusII</td>
<td>442.418372</td>
<td>102.722372</td>
<td>304.746348</td>
<td>55.635948</td>
</tr>
<tr>
<td>SIONlib</td>
<td>442.393952</td>
<td>102.707552</td>
<td>249.718112</td>
<td>45.936992</td>
</tr>
</tbody>
</table>

Table 1: Total file size in MB after 15 print steps in the Analytical_L5 scenario with the fragment distribution "b10_b80".

7.5 Visualization Capabilities

In this section, the visualization capabilities of the different approaches are discussed in terms of direct methods, i.e. without having to convert it to another format.

Since MPI-I/O allows storing data in both binary and ASCII representation, it can essentially conform any visualization format. In this thesis, the XDMF format has been chosen as it provides a manifold of features and also allows for the visualization of HDF5 files. Additionally, other codes such as waLBerla and Code_Saturne have utilized MPI-I/O to enable visualization capabilities using the the XML VTK and EnSight Gold formats.

NetCDF, on the other hand, provides different meta-data conventions for which standard readers in general purpose tools like ParaView and VisIt exist. As shown in Section 6.4.2, these readers only provide rudimentary visualization capabilities in comparison to the previous approaches. All in all, for our purposes the visualization capabilities using solely meta-data conventions are not as suitable as with the previous approaches. This is due to the fact that for different mesh types, different meta-data conventions must be employed. Moreover, these require heavy user-interaction in order to visualize the results appropriately. Despite the fact that the visualization of unstructured grids via ExodusII was adequate as its API was straightforward to use and enabled visualization of different element types as well as node- and cell-centered quantities in form of scalars and vectors, it comes at the cost of a rather heavyweight library and installation process.

As already mentioned, SIONlib by itself is not directly visualizable but needs to be converted into another format first. Despite the fact that it showed good scalability, this additional post-processing step can prove to be very costly as the files first need to be defragmented and since each chunk writes contains its own geometrical, topological and field data this also requires data to be rearranged.

In the following, visual results for the newly introduced visualization interfaces are demonstrated. Figure 33 depicts the Bahamas_L5 example, including the flow velocity in x- and y-direction as well as the bathymetry and the free surface elevation, after 3000 simulated seconds. Additionally, Figure 34 depicts the free surface elevation for the artificial Analytical_L5 example. Further the differences for fields, where the nodal data reduction was applied, are demonstrated.
Figure 33: Visualization of the tidal flow at Bahamas after 3000 simulated seconds via ExodusII.

Figure 34: Visualization of the free surface elevation for the analytical example on level 5 after 3000 simulated seconds.
8 Conclusion

This section gives an overview of the results achieved over the course of this thesis. Additionally, potential avenues for further extensions are discussed.

8.1 Summary

In this work, the design and implementation of multiple interfaces not only for parallel I/O but also for visualization in scope of the code generation frameworks ExaStencils and GHODDESS were shown. For parallel I/O, a dedicated package has been developed to facilitate the coupling of I/O with its application, e.g. visualization. This package provides abstractions for different access patterns and the buffers transferred between memory and disk. Moreover, since the requirements of I/O change from application to application multiple I/O techniques and libraries were (re-)implemented and evaluated. This includes the stream-based SIONlib interface and two interfaces making use of the C++ standard I/O library and different I/O techniques, namely FPP and locking. However, these interfaces do not provide means for conveying information about the global access pattern of all involved processes to a single-shared file, but data is rather accessed independently in a process-local view. Therefore, the parallel I/O libraries MPI-I/O, HDF5 and PnetCDF were also thoroughly investigated and integrated to ExaStencils. In addition, these also provide semantics to specify the processes’ global view on the shared file in order to access the file collectively and enable optimization for such access patterns. Both collective and independent I/O was implemented for these interfaces. Additionally, this work also provides two different file layouts where data is stored in a canonical or fragment-wise order. Since these interfaces employ tunable parameters for various elements of the I/O software stack, a hinting mechanism has been developed and is exposed to ExaStencils and GHODDESS. Nevertheless, with the exception of MPI-I/O these implementations are not restricted to I/O in MPI parallel environments but can also be used in serial applications. The new I/O interfaces can be directly used for storage and retrieval of field data via a new set of DSL statements added to ExaStencils. Additionally, debugging capabilities for the different libraries were added which allows a library to inform the user about the origins of an error, e.g. if a non-existing file is read.

The main application field of the I/O interfaces is visualization. For this purpose, interfaces for different file formats which are applicable to the corresponding I/O interfaces were integrated to the compiler and tested for multiple applications. To be more concrete, ExodusII is used in conjunction with PnetCDF, whereas XDMF can be used for the other interfaces. Moreover, a more sophisticated class hierarchy for the different visualization interface types and their formats has been designed. This also includes the organization of the different types of visualization interfaces. The first type is meant for a specific kind of application and visualizes a pre-defined set of fields on unstructured grids. Accordingly, for the application domain of SWE the DSL statements for this interface type have been modified such that users can define which fields should be visualized instead of a pre-defined selection. The second type is for any kind of application and is meant for field data visualized on (block-)structured grids. For each combination of format and I/O technique, a DSL function was introduced to ExaStencils. Moreover, in this scope two reduction schemes to reduce the amounts of data output to disk were introduced. This includes a mechanism to reduce the number of discretization points output for DG applications and a referencing mechanism for constant data.

All in all, each individual I/O interface has been evaluated based on multiple criteria and have been compared to the former state of ExaStencils. Despite all these changes to the ExaStencils and GHODDESS code base, backwards compatibility has been ensured to allow existing DSL codes to be still valid and remain their functionality. Roughly speaking these criteria comprise usability for the end user, portability, file size and most importantly the performance which was evaluated in the scope of applications from the GHODDESS project on the Emmy cluster.
8.2 Future Work

Since I/O finds application for multiple occasions and comprises various techniques and libraries, there still exist subjects for the future work. All in all, this work established a framework which facilitates the current state to be extended with the following applications making use of the presented I/O package but also with new techniques for the individual I/O interfaces.

Automatic I/O Tuning

As shown in Section 6.5.1, there is a broad range of tunable I/O parameters which can have impact on the I/O performance. Currently, these are exposed to users in form of flags in the knowledge file. This generally requires knowledge not only in the I/O infrastructure but also in the underlying I/O method that is used. While there exist options to set a parameter automatically, i.e. by specifying -1 for a set of parameters, these flags are still interconnected to each other and require expertise to achieve good performance. One approach to reduce the user effort for this step is to tune these parameters automatically. One possibility is to use genetic algorithms that evaluate the performance of a miniscule I/O kernel and evolve the population of candidate solutions such that an optimal fitness for the I/O bandwidth can be achieved. This course of action has also been evaluated in an auto-tuning framework for HDF5 [8]. This framework consists of two major components. The first is H5Evolve which samples the search space of configurations by taking a pre-defined parameter space with tuning parameters from different layers of the I/O stack. The evolution begins with a random initial population and generates XML configuration files for each experiment. These are read by the second component, namely H5Tuner, which injects the parameters from the XML file and reroutes calls to the HDF5 function to an optimized implementation via dynamic linking and is therefore transparent for users. For the other interfaces, a similar approach is desirable.

Non-blocking I/O

One optimization technique which has not been implemented in the scope of this thesis is non-blocking I/O. Similar to non-blocking communication, the main idea is to overlap the time spent in I/O with computations. Calls to blocking I/O functions return first when the I/O request has finished. For the non-blocking functions, on the other hand, the I/O needs not to be finished before the functions' return. Their completion can be checked with a 'wait function'. This way, in the time interval between the initiation of the I/O requests and their completion, useful computations can take place. However, for correctness it is important to note that send buffers should not be modified and receive buffers should not be accessed until the completion. Despite the usefulness of non-blocking I/O, not each of the presented I/O libraries supply this mechanism. For instance, HDF5 does not support asynchronous I/O yet but has evaluated its benefits in [87, 12]. Here, asynchronous I/O is implemented with asynchronous background threads. SIONlib does not provide any designated functions for asynchronous I/O, but using the POSIX AIO\(^1\) on the file container could be a possible solution.

```
Request requests[nFragments];
Status status[nFragments];
openFile(..., &fh);
...
for (int fragmentIdx = 0; fragmentIdx < nFragments; ++fragmentIdx) {
    iwrite(fh, &(field[fragmentIdx][0]), ..., &request[fragmentIdx]);
    < overlap with work >
    wait(&request[fragmentIdx], &status[fragmentIdx]);
}
closeFile(&fh);
```

Listing 17: Pseudocode of asynchronous I/O statements for MPI-I/O and PnetCDF

MPI-I/O [89] and PnetCDF [56], on the other hand, provide dedicated functions for non-blocking I/O operations. The pseudo-code in Listing 17 demonstrates how asynchronous I/O is used with both libraries. Since their function signature is almost identical to the ones of the blocking I/O

\(^1\)https://man7.org/linux/man-pages/man7/aio.7.html
statements, the changes in the generator would be minimal. The main difficulty is to expose asynchronous I/O for the DSL layer. One possible solution is to introduce a `startWrite` and a `stopWrite` layer 4 statement between which any kind of work can be specified. The `startWrite` function corresponds to the open call and a fragment loop where all asynchronous I/O requests are issued. The work statements are generated as usual and for the `stopWrite` statements, the wait function for all fragments’ requests is called and the file is closed. Note that this way multiple requests are open at once, that is for each fragment. Unfortunately, in MPI-I/O this is not possible for asynchronous and collective I/O operations, also known as split collective I/O, as at most one split collective operation is allowed to be open on a file handle [60]. In other words, the work part from the DSL must be incorporated in the fragment loop as shown in the listing. One possible solution is to annotate work loops in the DSL to be incorporated into one of the new I/O statements, e.g. `writeField_mppio`. Since work in the DSL is normally represented as a fragment loop in which computations take place, the outermost loop over the fragments is then omitted and the resulting loop is incorporated as described.

Parallel I/O Libraries

In the scope of this thesis, four different I/O libraries have been integrated to ExaStencils and evaluated. However, there are still other I/O libraries that have not been covered in this thesis. For instance, the Adaptive I/O System (ADIOS) [43, 62] is a componentization of standard I/O interfaces. It employs a XML-based approach where users can specify their I/O requirements in an abstract manner. Here, users can specify data type, metadata and dimensions in a centralized description and grants the capability to use different transport methods. These include POSIX I/O, asynchronous and collective MPI-I/O, asynchronous I/O within ADIOS called DART and NULL for no I/O. Moreover, this separation of transport layers allows data to be rerouted from disk to visualization and analysis tools, effectively allowing for in-situ visualization or analysis [43]. Data is stored in its newly introduced binary packed (BP) format which, similar to SIONlib [33], is designed to minimize coordination and stores meta-data compactly. These files can then be converted into standard file formats such as netCDF and HDF5. Despite the fact that the conversion cost is linear, for large datasets this can still be a disadvantage [13].

Checkpointing

Another potential application of the new I/O interfaces is checkpointing for fault tolerance. Checkpointing is a technique where a description of the applications state is periodically put on a stable storage system. In case of failure in any process, all processes in the application are rolled back to the most recent checkpoint and the application restarts from this point. A distinction between system-level checkpointing and application-level checkpointing as an abstraction level is made here to describe the state to be saved [11]. System-level checkpointing is fine-grained and includes an application’s register, program counter and also memory. However, this approach is not suitable for HPC clusters with thousands of nodes as the aggregate amounts of data would certainly go beyond the capabilities of the underlying storage system. Application-level checkpointing means that applications are responsible for writing their own checkpointing code, including automatic restarting from the last checkpoint in case of failure.

For ExaStencils, application-based checkpointing would be the best choice. This could be implemented via an additional layer 4 statement to accept a simulation’s state as argument. In other words, the user of the DSL is responsible for specifying which fields, variables, etc. should be written to file and are restored in case of failure. While this approach requires more effort for the users, it is more practical in a sense that instead of storing the whole simulation state, e.g. including also unwanted temporary results, only the actually required data is stored on disk. This way, the data extents are reduced and less time is spent in I/O.

For the most part, the implemented I/O interfaces can be reused. Fields are directly translated into databuffers and can be directly used by one of the IR_FileAccess subclasses. For other state variables, e.g. the current time step, a convention on where and how these values are stored must first be set. Once the convention is established only the algorithmic part of the checkpointing module needs to be implemented. Moreover, there should be knowledge parameters on how frequently a checkpoint is created.
IR_InitDomainFromFile

As already mentioned, the generated counterpart of IR_InitDomainFromFile is responsible for reading geometrical and topological information from a file at startup whenever the computational domain is not generated automatically, e.g. for pure rectangular domains. Currently, the files containing this information are stored in ASCII files where one file is dedicated to a single process. These files are read via code generated from the FPP readField statement. Despite the fact that this step is done once in the beginning of the simulation, implementing it for single-shared file approaches can mitigate the management issues of FPP I/O. Moreover, storing them in a binary file format such as HDF5 can also reduce the overall file size. For this purpose, not only the interface in ExaStencils needs to be adapted but also the generator of the domain files. More specifically, the files output by the generator must conform the file format of the employed I/O interface.

Data Layout Transformations

One functionality of ExaStencils which has not been covered in the scope of this work are the layout transformations of fields. By default, higher-dimensional data types such as vectors are stored in a SoA layout. Moreover, visualization interfaces such as Xdmf provide mechanisms to join non-interleaved DataItems into interleaved DataItems, e.g. a SoA vector stored as three non-interleaved scalar components into a vector where the scalar values are stored in an interleaved fashion, i.e. Array of Structures (AoS). This mechanism is used on multiple occasions to allow higher-dimensional fields to be stored as-is in memory to mitigate data rearrangements during the simulation. However, a field layout can be transformed into another layout, e.g. SoA to AoS [47]. This does not impose a problem for the stream-based I/O interfaces, i.e. the C++ I/O library and SIONlib, as accesses to the field and its components in a loop over the discretization points are handled accordingly. On the other hand, handling for MPI-I/O, HDF5 and PnetCDF is necessary since there is only one access to a field and that is at its base address, i.e. &field[fragmentIdx][0]). The memory layout of a field fragment is described with MPI datatypes and its equivalents in higher-level libraries, e.g. hyperslabs in HDF5. Since no iteration over the discretization points of a fragment occurs and its memory layout is described as a whole, additional handling for layout transformations is necessary. At first, fields with transformed layouts need to be annotated in the generator. One possible handling of such fields is to generate statements for a copy kernel before the actual file access. Here, a copy of the annotated field with the suitable layout, i.e. SoA, is created and data from the annotated field is rearranged in the corresponding copy to match the layout. Note that this handling is not only meant for SoA to AoS transformations but allows any kind of layout transformation.

PrintField with PnetCDF

The interface for visualizing fields via PnetCDF currently only supports uniform and axis-aligned grids on rectangular domains. The visualization of non-uniform and block-structured grids in form of a scattered data plot is currently missing. However, since these visualization capabilities are too restrictive to begin with, exploring other possibilities would be more beneficial. For instance, new meta-data conventions tailored for the application domain of ExaStencils and GHODDESS can be invented and a corresponding reader in ParaView can be implemented. However, this is most likely to laborious. As an alternative, scripts to convert the netCDF files to other formats could be generated.
References


[97] Brad Whitlock. “Getting data into VisIt”. In: Lawrence Livermore National Laboratory (2010).

List of Acronyms

ADIOS Adaptive I/O System
AoS Array of Structures
API Application Programming Interface
ASCII American Standard Code for Information Interchange
AST Abstract Syntax Tree
ccNUMA cache-coherent Nonuniform Memory Access
CF Climate and Forecast
CFD Computational Fluid Dynamics
CM Computational Mechanics
CPU Central Processing Unit
CSE Computational Science and Engineering
CSV Comma-separated Values
gD Discontinuous Galerkin
dSL Domain-specific Language
FD Finite Differences
FE Finite Elements
FPP File-per-process
FV Finite Volumes
GHODDESS Generation of Higher-Order Discretizations Deployed as ExaSlang Specifications
GPL General-purpose Language
GPU Graphics Processing Unit
HDF5 Hierarchical Data Format version 5
HPC High Performance Computing
I/O Input/Output
IR Intermediate Representation
IV Internal Variable
LBM Lattice Boltzmann Method
MDS Meta Data Server
MDT Meta Data Target
NOP No Operation
NS Navier-Stokes
OSS Object Storage Server
OST Object Storage Target
PDE Partial Differential Equation
PFS Parallel File System
PnetCDF Parallel netCDF
POSIX Portable Operating System Interface
SoA Struct of Arrays
SWE Shallow Water Equation
TPDL Target Platform Description Language
UMA Uniform Memory Access
VTK Visualization Toolkit
XDMF eXtensible Data Model and Format
XML eXtensible Markup Language
### List of Listings

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Field and field layout declaration of a vector field in 3D</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>ExaSlang 4: Primary I/O statements that frequently access large amounts of data</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>XDMF example of a simple triangular mesh with two triangles in a 2D Domain</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>Callback registration for a databuffer's access pattern instantiation</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>Exemplary C++ target codes writing a 1D array via the standard I/O library</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td>MPI-I/O example for writing a field in a 2D domain partitioned into two blocks with two fragments each to a binary file</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>HDF5 example code for writing a field in a 2D decomposed domain using the fragment-wise storage layout</td>
<td>53</td>
</tr>
<tr>
<td>8</td>
<td>PnetCDF example code writing a 2D field including ghost layers in fragment-wise order via independent I/O</td>
<td>55</td>
</tr>
<tr>
<td>9</td>
<td>Exemplary target code for writing a 2D field into a SIONlib container</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>Callable layer 4 functions for the visualization of data on unstructured grids</td>
<td>60</td>
</tr>
<tr>
<td>11</td>
<td>New layer 4 functions to store field data on disk</td>
<td>63</td>
</tr>
<tr>
<td>12</td>
<td>Layer 4 functions to print visualization files for fields on (block-)structured grids</td>
<td>64</td>
</tr>
<tr>
<td>13</td>
<td>Example of specifying hints for MPI-I/O via info objects</td>
<td>66</td>
</tr>
<tr>
<td>14</td>
<td>Knowledge flags for MPI-I/O performance tuning</td>
<td>67</td>
</tr>
<tr>
<td>15</td>
<td>Tunable parameters for HDF5</td>
<td>68</td>
</tr>
<tr>
<td>16</td>
<td>Knowledge flags for PnetCDF hints</td>
<td>68</td>
</tr>
<tr>
<td>17</td>
<td>Pseudocode of asynchronous I/O statements for MPI-I/O and PnetCDF</td>
<td>80</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Multi-layered DSL design in ExaStencils (from [58])</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Hierarchical domain partitioning in ExaStencils (from [50])</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Global view of grid nodes distributed over two fragments and their assignment to the logical groups: Padding, Ghost, Duplicate and Inner (from [50])</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Structured meshes with quadrilateral elements in 2D (modified from [49])</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Unstructured triangular mesh</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Block-structured mesh</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Supported Localizations in ExaStencils</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Class hierarchy of candidates to be extended with parallel I/O</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Triangular mesh with discontinuous elements</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Code generation toolchain employed in GHODDESS (modified from [28])</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Global storage and file layouts of a 3D domain decomposed into eight fragments</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Abstract I/O infrastructure of supercomputers with external file systems (from [69])</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Lustre file system architecture (from [86])</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>General I/O patterns in parallel applications</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Presented I/O interfaces and their relation in the abstract I/O software stack</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>MPI-I/O optimization techniques</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Examples of hyperslab selections for different source and destination dataspace</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Storage model of a classic netCDF file</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Block organization of a SIONlib container (from [33])</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Linear cell types for unstructured meshes (from [91])</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>UML diagram of the new I/O package and its class hierarchy</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Global distribution of inner points in the 2D domain partitioned into a total of 4 fragments. The file is laid out in a canonical order. FX denotes the fragment id in the domain and PX the process id.</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Rough structure of the visualization interfaces for unstructured grids. To provide a better overview of the classes, the visualization classes and the traits they implement were separated.</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>XDMF class hierarchy to print fields on (block-)structured grids</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Nodal data reduction scheme: The shaded areas depict the triangles involved in the average calculation. The corresponding values of the discontinuous field are averaged over the vertices that share the highlighted grid node.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Analytical_L8: I/O bandwidth for stream-based I/O approaches</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>Analytical_L8: I/O bandwidth for the parallel I/O libraries MPI-I/O, HDF5 and PnetCDF</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Analytical_L8: Tuned I/O bandwidth for the parallel I/O libraries MPI-I/O, HDF5 and PnetCDF at smaller scales</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>Analytical_L5: I/O bandwidth for stream-based I/O approaches</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Analytical_L5: I/O bandwidth for collective I/O operations using MPI-I/O, HDF5 and PnetCDF</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>Bahamas_L5: I/O bandwidth for stream-based and MPI-based collective interfaces</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Visualization of the tidal flow at Bahamas after 3000 simulated seconds via ExodusII</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>Visualization of the free surface elevation for the analytical example on level 5 after 3000 simulated seconds</td>
<td></td>
</tr>
</tbody>
</table>