FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Generating an Interface for Parallel Multigrid Solvers and VisIt

Richard Angersbach

Bachelor Thesis

Generating an Interface for Parallel Multigrid Solvers and VisIt

Richard Angersbach

Bachelor Thesis

Aufgabensteller:	PD Dr.-Ing. habil. Harald Köstler
Betreuer:	M. Sc. Sebastian Kuckuk
Bearbeitungszeitraum:	05.02.2018 – 05.07.2018

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 Bachelor Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 5. Juli 2018 .....

Contents

1 Abstract

Scientific visualization is a branch of science directly involved with graphical illustrations which represent the data produced by a simulation. Many simulation programs calculate an approximate solution for partial differential equations (PDE) as they represent a wide variety of problems from physics and natural phenomena. One popular and efficient approach to obtain this approximation is to build a solver using the so-called multigrid methods.
Depending on the complexity, simulations are run on high performance computing (HPC) systems to distribute the problem size among the involved processors. Programs running on these systems are highly optimized towards the components they consist of in order to increase performance. These optimizations will not be optimal for different components or HPC systems, resulting in a partial revision of the code. One way to bypass this problem is the usage of domain-specific languages (DSL) where the problem is specified and code that is optimized towards the current system is generated. The code generation framework ExaStencils with its DSL ExaSlang can be used for this purpose. The framework has the ability to generate source code for parallel multigrid solvers.
Visualization tools are often used to illustrate and analyze the data they have been provided with. In many cases the transportation of the data is done by formatting and writing the data into a file. Once the simulation is finished, these files are opened by the visualization program and can be visualized. One negative aspect of this approach is the heavy usage of I/O. Nowadays HPC systems produce data at a higher rate than the bandwidth of I/O, which turns out to be a bottleneck. To reduce the impact from I/O, the integrity of the data needs to be compromised, e.g by writing the files less frequently. Another approach is to visualize the data in-situ. The visualization routines are granted direct access to the simulation's memory which nullifies the need of any I/O actions. This approach also enables the functionality of simulation steering, allowing the user to make adjustments and control the flow of the simulation. The visualization tool VisIt provides a library to visualize the data in-situ. In order to transfer the data from the simulation to VisIt, adjustments to the code have to be made. The implementation for this purpose can be quite cumbersome because it differs for serial and parallel simulations. Also, in order to provide a visualization for the whole integrity of data present, data access code for each variable and its belonging mesh has to be written.
The goal of this thesis is to bypass this problem by integrating a visualization interface into ExaStencils using VisIt's in-situ approach with the objective that the necessary changes to the DSL code are minimized.

2 ExaStencils

ExaStencils [2] is a code generation framework for highly optimized parallel multigrid solvers on block-structured grids. Within this chapter, the basics of the multigrid methods are explained, the conceptual design of the DSL ExaSlang is described and an introduction to its language elements is given. However, the focus in this chapter lies in the ExaStencils data partitioning and the variable localization since it is the most relevant information for the visualization.

2.1 Multigrid methods

In order to compute an approximation for the linear system of equations $A_h u_h = f_h$ that arises from the discretization of PDEs, multigrid solvers [10, 20] can be applied. But in many cases, these matrices can be expressed with a more compact representation, the so-called Stencils. Multigrid solvers make use of two principles. The first one is the smoothing property of iterative methods, also called relaxation methods, that solve the linear system of equation with an initial guess of the solution. Relaxation methods such as Jacobi or Gauss-Seidel are effective for reducing errors with high frequency, enabling a high convergence rate in the first steps. However, this does not apply to low frequency errors with the result that convergence rate is strongly reduced after some iterations. This is where the second principle, allowing smooth functions to be approximated on coarser grids, comes into place. Multigrid solvers make use of this principle to get an approximation of the error from lower levels. This error is used to correct the approximate solution with it, since the exact solution u_h^* equals the sum of the approximate solution u_h and the error e_h. The error is contained in the linear system of equations $A_h e_h = r_h$, also called residual equation.
Figure 1 illustrates the steps for the current multigrid cycle k to calculate a new approximation

$$u_h^{(k+1)} = MG_h\Big(u_h^{(k)}, A_h, f_h, \gamma, \nu_1, \nu_2\Big). \tag{1}$$

The parameter γ denotes the number of recursions, ν_1 the number of pre-smoothing steps and ν_2 the number of post-smoothing steps. The cycle begins at the highest level defined. Here, the current solution $u_h^{(k)}$ is smoothed, the residual r_h is calculated and restricted $\mathcal{R}$ to the residual r_H on the coarser level. Afterwards the recursion begins. In order to solve the residual equation, the initial guess e_H^0 and r_H are passed as function arguments. Once the coarsest grid has been reached, the residual equation is either directly solved or approximated with methods such as the conjugate gradient (CG) method and the result is returned to the successive finer grid. In dependency of γ the recursion can be repeated and the received result $e_h^{(j-1)}$ is passed. The prolongation or interpolation $\mathcal{P}$ of the error e_H towards the error of the current level e_h can be used to correct the approximate solution $u_h^{(k)}$. Afterwards, post-smoothing is applied. The multigrid cycle is complete when the finest level has been reached again.

if *coarsest level* **then**
  solve $A_h u_h = f_h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}\Big(u_h^{(k)}, A_h, f_h\Big)$ `pre-smoothing`
  $r_h = f_h - A_h \bar{u}_h^{(k)}$ `compute residual`
  $r_H = \mathcal{R} r_h$ `restrict residual`
  for $j = 1$ *to* γ **do**
    $e_H^{(j)} = MG_H\Big(e_H^{(j-1)}, A_H, r_H, \gamma, \nu_1, \nu_2\Big)$ `recursion`
  end for
  $e_h = \mathcal{P} e_H^{(\gamma)}$ `prolongate error`
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e_h$ `coarse grid correction`
  $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}\Big(\tilde{u}_h^{(k)}, A_h, f_h\Big)$ `post-smoothing`
end if

Figure 1: The Multigrid algorithm (from [21]).

2.2 Multi-Layered Design

In general, DSLs can be categorized as external or internal. Internal languages make use of the syntax and semantics from their corresponding host language, e.g. C++, whereas external languages introduce their own syntax and semantics. ExaSlang belongs to the external DSLs, since these are generally more flexible and expressive. It conceptually consists of four layers which, in general, can be assigned to a group of users. Here, the three groups "Engineers & natural scientists", "Mathematicians" and "Computer Scientists" are distinguished. Naturally, these groups do research on different topics and expect language elements best suited for them [1, 22].

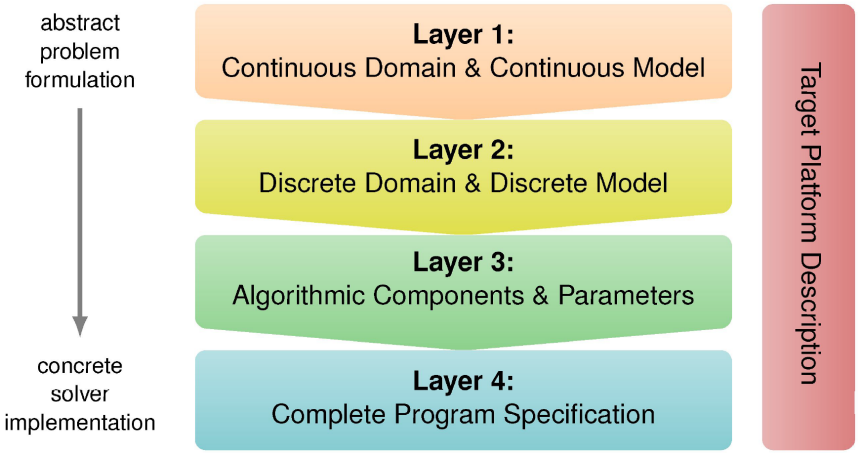


Figure 2: The four layers of ExaSlang (from [12]).

Figure 2 shows the four different layers of the DSL ExaSlang where abstraction decreases on higher layers. Layer 1 is the most abstract and suited for the "Engineers & natural scientists" group. On this layer, they are able to specify a continuous problem like an energy functional or a PDE with mathematical formulations. This also includes specifications such as the computational domain and its corresponding boundary conditions.
In Layer 2, the corresponding discretization of the problem is declared and restrictions to the grid, e.g. if the grid is structured or block-structured, are made. This layer is most likely used by "Engineers & natural scientists" and "Mathematicians".
The first appearance of the multigrid methods exists on Layer 3. The focus of this layer lies on algorithmic components, parameter values and settings. For example, smoothers with their corresponding coloring, inter-grid transfer functions such as prolongation/restriction operators and multigrid cycles. This layer is best suited for "Mathematicians" and "Computer Scientists".
In comparison to the other layers, Layer 4 is the most concrete. Here, functions and classes in a syntax similar to Scala can be integrated in order to apply various code optimization strategies, parallelization techniques such as communication patterns or to add components from external frameworks.
Another component of the ExaSlang design is the Target Platform Description Language (TPDL). Specifications for the target platforms hardware, e.g. CPUs and software components such as compilers take place here.
In addition to the various layers illustrated in figure 2, there is an intermediate representation (IR). However, it is not accessible to the user and therefore not included in the figure. Nonetheless, the IR is important for this thesis as it builds the bridge between Layer 4 and the resulting target code. In the end, the target code contains elements from both. In regards to the visualization provided in this thesis, elements from the IR are the functions which will be discussed in chapter 4. More details about the layers can be found in [1, 21].

Workflow

The core of the code generation framework is its compiler written in Scala. The compiler can either be used to translate the input in form of ExaSlang files towards a less abstract DSL layer

or in case that the most concrete layer is passed, target code for the programming languages C++ and CUDA can be generated. Besides the ExaSlang file, the generator also receives two additional inputs. The first one is the `settings` file, which contains information about paths and files, e.g. location for generated code or which ExaSlang files are passed towards the compiler. The second file is the `knowledge` file which contains domain specific parameters. The workflow of ExaStencils is to create highly optimized code by translating it from Layer 1 step by step. For each step, hardware and domain knowledge is applied. The generation of the target code begins with parsing the Layer 4 to the IR. Here, compiler-internal code transformations which describe the refinement process from the input towards the output data by either modifying, adding, removing or placing program elements, take place. In most cases, these transformations are used to apply optimization techniques. However, in regards to the implementation of this thesis, transformations are used to add the corresponding global variables and functions for the visualization to the target code. For the execution of transformations, strategies that put a group of transformations to a certain step in the generation process are needed. The generation of the visualization code is described with a default strategy, which basically applies the transformations in their given order. More details can be found in [22].

2.3 Data partitioning

In order to support data partitioning for a manifold of hardware configurations and parallelization techniques, ExaStencils introduces a three-level hierarchy [12]. The focal point of this thesis lies on the shared-memory and distributed-memory parallelization models, with the result that only the mapping between the elements in the hierarchy and the corresponding technique is explained in this section. Shared-memory parallelism is often realized with usage of the Open Multi-Processing (OpenMP) interface, where threads sharing the memory amongst themselves are run in parallel. Distributed-memory parallelism however describes the partitioning of data among the involved processors, where every processor has its own memory. In case that data from another processor is needed, communication between them takes place. For this technique, the Message Passing Interface (MPI) is used and the processors are assigned to so-called MPI ranks. Naturally, there are also hybrid approaches, where the most common is to spawn threads on each MPI rank.
Figure 3 illustrates the hierarchy's elements that are used to partition data physically. The leaf elements in the hierarchy are sets of grid points or cells and represent a part of the computational domain. A fixed number of these leaf elements are contained in so-called unit fragments. These assemble the fragments, which represent the next level in the hierarchy. Their mapping to parallelism corresponds to either one OpenMP thread per fragment or in case that the number of fragments is too high, multiple fragments are assigned to an OpenMP thread. On the top of the hierarchy are the so-called blocks, which consist of a single or multiple fragments. Here, a one by one ratio between blocks and MPI ranks is used.

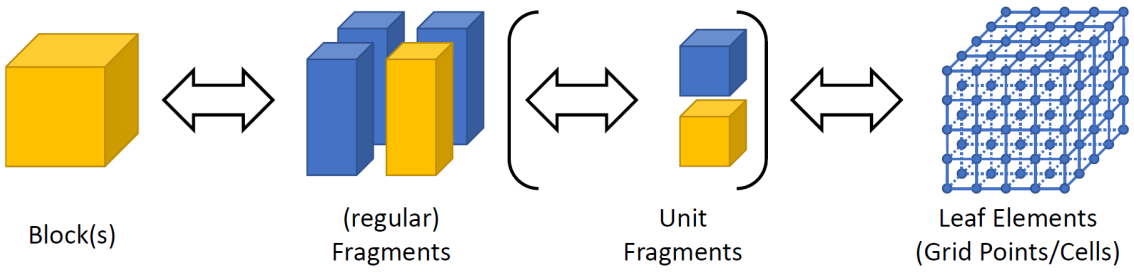


Figure 3: Hierarchy of data partitioning in ExaStencils (from [12])

In addition to the physical decomposition of the domain, a partition of leaf elements into logical groups takes place. The first elements within this partition are the regular grid nodes or cells. In a stencil code, these elements represent the values in the grid on which the stencil operator is applied. In case that the stencil code is run on different resources, e.g. when a distributed-memory parallelism is applied, grid values from the neighboring processors are required for the stencil operations. For this reason, the so-called ghost or halo layers are introduced. As shown in

figure 4, these layers contain copies of the regular elements from other fragments and are updated as soon as new data is required. Another group within the logical partitioning which requires data from other processors are the so-called duplicate layers. These layers are contained by multiple fragments as pictured in figure 4. On the contrary to the ghost layers, calculations take place on these elements. Also, their position within the domain is completely independent of the current multigrid level. The distinction between regular and duplicate elements is made because deviations of the values between adjacent fragments may occur. One possible cause for these deviations is the order in which the operands are applied. ExaStencils makes use of a synchronization strategy where a dominant instance is declared. The values from the other adjacent instances are overwritten by the ones coming from the dominant instance. The next logical group are the so-called padding layers, which can be used for optimizations such as reducing the rate of conflict cache misses. The order in which the different kinds of layers are applied can be seen in figure 4. The logical groups are denoted as (P)adding, (G)host, (D)uplicate and (I)nner layer. Around the inner layers are the duplicate layers which are also surrounded by ghost/padding layers.

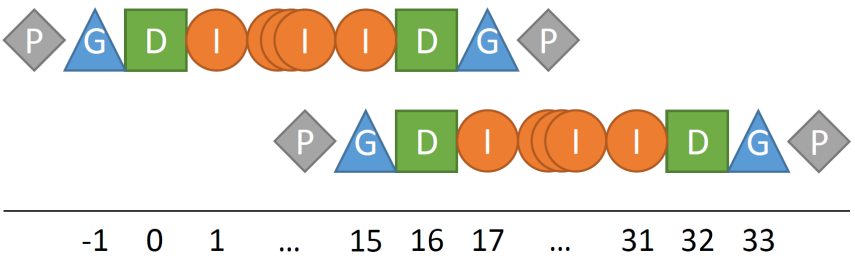


Figure 4: Logical data partitioning and global index distribution between two fragments in 1D (from [12]).

2.4 Variable Localization

In general, discretization describes the transformation of continuous data such as functions or variables into a set of discrete points. As a result, vectors containing approximate values for each discretization point arise. These vectors are declared as fields within ExaStencils. However, in dependency of the problem that is solved, different discretization approaches are used. The difference between these approaches mainly lies within the number of discretization points and their coordinates. As a result, the localization of a field's data points must be specified in its field layout. As shown in figure 5, data points can be positioned on the centers of the grids nodes, cells or faces. However, face-centered variables are mainly used in combination with cell-centered variables in a so-called staggered grid, which are in most cases used for fluid flow simulations to obtain a stable discretization [20]. Within the domain of geometrical multigrid solvers, the number of points for a node-based discretization equals $c_{dim} * 2^{level} + 1$ whereas the number of cells is $c_{dim} * 2^{level}$ [12]. Examples for the factor c_{dim} can be found in chapter 4.4. In staggered grids however, face-centered variables exist for each dimension and the localization within the grid differs for each of them. In general, these can be interpreted as node-centered in the dimension of its centering and cell-centered for the other dimensions [13].

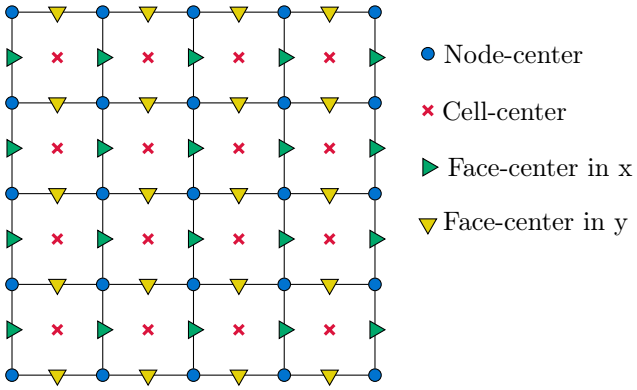


Figure 5: Variable localizations supported in ExaStencils

2.5 Language Elements

As described in chapter 2.2, components from external frameworks are added towards Layer 4. Also, it is possible to declare an application function which becomes the main function at the target code [1]. The mandatory changes to the DSL code take place in Layer 4 and therefore a short introduction for its language elements is given. However, this introduction only presents the most important language elements in regards to the changes in the DSL code. For further information see [13, 22, 21].

Level specifications

In the domain of multigrid solvers, level specifications are needed since the algorithm takes different actions on each level, e.g. direct solving on the coarsest level. ExaSlang simplifies the declaration of the levels with keywords. Functions and objects can be accessed on specific levels with these keywords. The most relevant keywords are `@all`, `@current`, `@finer`, `@coarser`, `@finest` and `@coarsest`. In addition, it is also possible to exclude certain levels with the `@but` as shown in listing 1.

```
Function mgCycle@(all but coarsest) {
    Smooth@(current) ( ) /* calculateResidual@current, ... */
    mgCycle@(coarser) ( )
    /* prolongate@current, ... */
}
Function mgCycle@(coarsest) {
    /* solve directly or with many iterations */
}
```

Listing 1: Multigrid cycle with two different level specifications

Control flow

Besides functions, ExaSlang provides language elements such as branching with if-else statements and loops which are categorized as temporal or spatial. Temporal loops are a sequence of steps whereas spatial refers to an iteration over the whole computation domain. Temporal loops can be expressed as `repeat until <condition>` or `repeat <N> times` and spatial loops have the signature `loop over <field>`. Another important language element is the application function shown in listing 2. This function corresponds to the main function in the C++ target code. In relation to the visualization code, this function plays an important role since some of the provided functions need to be added here.

```
Function Application ( ) : Unit {
    /* initialization */
    repeat 10 times {
```

```
        mgCycle@(finest) ( )
    }
    /* de-initialization */
}
```

Listing 2: Application function

Data types and variables

The data types that are available in ExaSlang are categorized into the groups *simple data type* and *algorithmic data type*. There are also so-called *aggregate data types* which represent complex numbers, but these are not addressed in this thesis. Simple data types are denoted as `Real` for floating-point numbers, `Unit` which is a functions return type with no value, and the familiar data types from other programming languages `Integer`, `Boolean` and `String`. These simple data types can be assigned to a `Variable` (in short `Var`) or to a constant `Value` (in short `Val`). Two examples are shown in listing 3.

```
Val pi : Real = 3.141
Var twoPi : Real = 2.0 * pi
```

Listing 3: Variable and value assignment

Algorithmic data types are classified as data types used in numerical calculations. The most relevant of those are the fields which were mentioned in chapter 2.4, since these represent the data which is visualized. Fields are tied to a computational domain and therefore to its physical coordinates. Another specification for them is the field layout. It consists of information such as the number of layers from the logical data partitioning, the localization and the data type of the discretization points. Additionally, boundary conditions such as `None`, `Neumann` or `Dirichlet` in form of an constant value can be chosen. Listing 4 shows a three-dimensional, node-centered variable with Dirichlet boundary conditions. It is surrounded by one duplicate and one ghost layer per dimension. These properties apply to each multigrid level defined.

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

Layout NodeWithComm< Real, Node >@(all) {
  duplicateLayers = [1, 1, 1] with communication
  ghostLayers     = [1, 1, 1] with communication
}
Field Solution< global, NodeWithComm, 0.0 >@(all)
```

Listing 4: Declaration of a field and its components

2.6 Configuration

Naturally, the generation of the visualization is strongly dependent on the input parameters passed to the compiler. For the implementation of this thesis, following knowledge parameters are taken into consideration:

- minLevel/maxLevel: coarsest and finest multigrid level.
- numLevels: equals maxLevel−minLevel+1.
- useDblPrecision: used to determine whether to use double or single floating point precision.
- mpi_enabled: determines whether MPI is used or not. In chapter 4 a simulation is referred as parallel, when it is set to true.
- mpi_defaultCommunicator: used in callback functions for the synchronization between master and slave processes.

- dimensionality: dimensionality of the problem that is solved.
- domain_numBlocks: total number of blocks within the domain. Also corresponds to the number of MPI ranks.
- domain_numFragmentsPerBlock: total number of fragments in a block.
- discr_h*: constant step size for each available level and dimension (* equals x/y/z).
- domain_rect_numFragsPerBlockAsVec: number of fragments per block for each dimension. Only relevant for rectilinear domains.
- domain_fragmentLengthAsVec: multiplier for the number of points in each direction for the fragments.
- targetCompiler: compiler used for the target code.

3 Scientific Visualization with VisIt

The application VisIt [7] is an open source visualization and analysis tool which can be used on Unix, Windows and Mac platforms. Users are granted the ability of interactive visual exploration and analysis for massive data sets. VisIt's customizable plug-in design gives the capability to add components that allow a wide set of data formats as well as data visualization and manipulation operators to be supported. This chapter describes the general architecture of VisIt, explains how data is processed in a parallel visualization pipeline and which rendering techniques are used by VisIt.

3.1 Architecture

Large data visualization

On many occasions, visualization tools such as VisIt might be overwhelmed with the data produced by the simulation, in the meaning that the whole data set cannot be processed at once. However, these simulation tools offer various solutions for this problem. One of them is out-of-core algorithm *streaming* which makes use of the property that data does not have to be sent all at once, but instead the data set is treated as a composition of pieces which are sequentially read and processed by the pipeline. Another strategy is the selection of data that is relevant for the visualization result instead of using the entire data set available. Query-driven visualization and multiresolution processing make use of this principle. The most common approach however, is to distribute the data along parallel resources and is called *pure parallelism*. More details can be found in [4, 17].

Components

VisIt embodies a distributed client-server architecture [7, 25, 27] where its component programs, which are responsible for certain tasks of data visualization and analysis, are classified as client- or server-sided. However, this does not necessarily mean that a visualization purely on the client side is not possible. Nonetheless, it is often the case that an execution on a server, often represented by a supercomputer, is required due to its computing capabilities and most of all, to handle data which would overwhelm the client. Here, the *pure parallelism* approach is applied. Each processor assembles the same so-called data flow network for the visualization pipeline and executes it on its own share of the whole data. In dependency of number of primitives, either a reduced set of the data, which is rendered on the client computer, or images rendered on the server are sent towards the client. More details can be found in the next two sections.
Before visualization requests can be made, information regarding the server's files and their contents is necessary. One important component program for this task is the **Database Reader**, also called mdserver. It is responsible for browsing the remote's file system and also shows the user which files can be opened. Depending on the properties of a file such as its format, a suitable plug-in responsible for reading its content is loaded. Once it has been opened, metadata such as the list of available variables is sent to the client and visualization requests can be made. The component responsible for the processing of visualization and analysis requests is the **Compute Engine**. It also loads a plug-in that reads the data from a file and additionally transforms it into VTK objects. These are put into the visualization pipeline and the result is transferred to the client, where it is displayed. The last server-sided component is the **VisIt Component Launcher** (VCL) and is responsible for launching the other server-sided programs.
The client-sided applications are run on a local desktop and mainly used for rendering and interaction. The **Viewer** creates plots for the current visualization and analysis request. The operations of the viewer can be controlled by various interfaces. Figure 6 shows two example interfaces. VisIt's graphical user interface (**GUI**) is built with the Qt widget set. Its main menu consists of seven sub menus that can be used to connect to remote hosts, browse their file systems with the information gained by the database reader, change the attributes of plots and the operators that manipulate them etc. For more details see [26]. The other illustrated interface is the command line interface (**CLI**). VisIt has a Python interface that fulfills the same role as the GUI, with the only difference that Python scripting is used. The CLI is basically an interpreter for the Python interface and simplifies the process of running scripts. More information can be found in [16]. In addition, both

interfaces can be replaced with custom versions. Besides the other user interfaces shown in figure 6, also Java and C++ clients can communicate with the viewer [8, 24].

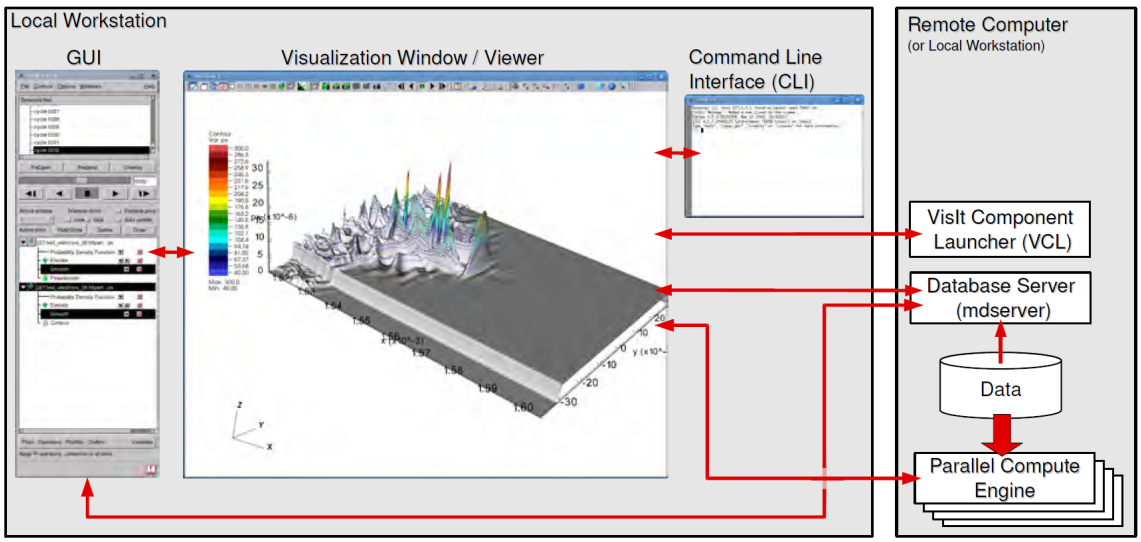


Figure 6: VisIt's client/server architecture (from [7])

3.2 Visualization Pipeline

Visualization pipelines often make use of the data flow network processing design. Data flow networks are considered as frameworks that provide an execution model, a data model that grants various representations for the data and algorithms for the transformation of the data. This design consists of the two base types data objects and components, which receive and produce data objects. Components are either sources, filters or sinks. A visualization pipeline consists of an ordered sequence of components. Typically, it begins with the source which provides an output of data objects. The most common example for the source would be a file reader. The source is followed by a number of filters, which transform the data. The output from the last filter reaches the sink, which can be a file writer or a rendering engine. The components of a pipeline are interchangeable with the condition that the output from a new component is accepted by the downstream input. For further reading see [5, 17].

The flow of a visualization pipeline can be described with its execution model. These can be categorized into event-driven or demand-driven [17]. In an event-driven execution system the pipeline is executed once data is available in the source. The source pushes the data towards the downstream components, whose execution is triggered by the source, and, once completed, the results passed towards the next component. In case that the demand-driven approach is used, the pipeline launches as soon as data is requested. Here, the first step occurs at the sink, where upstream components are asked for data until the source is reached. The second step begins once the data has been generated by the source. Here, the data is passed downstream and processed. In VisIt's pipeline the first step is called *update* and the second *execute* [5]. In comparison to the event-driven approach, which can be used when data is likely to change over time, demand-driven execution proves useful when visualization requests are made by a user.

In general, the parallel execution of a visualization pipeline is realized by executing its components concurrently. There are three different basic modes [17] that make use of this principle. One of them is the *task parallelism* approach, where the pipeline is split into independent roles. These roles are realized as sub-pipelines that are executed concurrently, making this approach applicable for every kind of algorithm as the original pipeline structure stays the same. One disadvantage of task parallelism is the low degree of parallelism that can be achieved because the number of independent roles within a pipeline is limited.

Another parallelization strategy with a higher level of concurrency is the *pipeline parallelism*. Here, different pipeline components are executed concurrently with usage of streaming. This mechanism starts with reading the first piece and proceeds to the next downstream component. With the

parallelization of the components, the next piece can be read while the component processes the the first piece. The read and processed piece can be passed downstream and with the continuation of this approach, several pieces are in the pipeline.
The most common strategy to run components concurrently is with usage of *data parallelism*, where the whole data set is also divided into a number of pieces. An identical visualization pipeline is built for each piece. This method scales well, because large data can be distributed among processes fairly easily, allowing a higher degree of parallelism. In addition, with evenly distributed pieces, the execution of the identical pipeline enables good load balancing. For task and pipeline parallelism, load balancing might be more of an issue since sub-pipelines or components do not finish their execution at the same time.
VisIt uses a modified version of the demand-driven data flow network from the Visualization Tool Kit (VTK) [23], which gives foundation for the data representations and algorithms [7]. One important modification takes place in its execution model, where contracts are introduced [4, 5]. The principle behind contracts is to grant the capability of communication between the components about their impacts on the pipeline. The initial version of the contract is located at the sink and with every *update* request the contract is modified by the filters. Once the contract is complete, the *execution* can begin. With this approach optimizations which are dependent on the pipeline's components can take place. One example is the data reduction of the data that is read when a slice filter is applied. The addition of contracts also makes the VisIt plug-in architecture possible, where unknown components developed by end-users are added to the pipelines and optimizations for those are required.

3.3 Parallel Rendering

The main task within a rendering pipeline is to determine each scene object's contribution to the pixel with a certain view given. It can be divided into transformation and rasterization, where transformation is responsible for the projection from model to screen space and rasterization for the conversion of the geometry's primitives into a raster image. Naturally, these stages are often processed in parallel. However, when these stages are parallelized, a redistribution of the data among the involved processors can occur anywhere in the pipeline. Therefore, the redistribution can be interpreted as an sorting problem [11, 19].
When sort-first is applied, every processor is responsible for processing a part of the screen and takes an arbitrary portion of the data. Each processor performs an initial transformation step to determine the screen position of each primitive and sends them to the processor with the corresponding portion of the screen. Afterwards, the execution of the pipeline with its remaining transformation and rasterization steps continues.
Sort-middle follows another strategy, where a logical or even physical distinction between transformation and rasterization processing units is made. Here, the transformation units fully execute the transformation stage and pass the results to the rasterization units with their belonging portion of the screen space.
In sort-last each processor executes the complete pipeline until the pixel values of each sub-image are determined. In the end, every sub-image stores its pixel values into the z-buffer of the final image.
When taking VisIt's client-server design into consideration, there are use cases where it is a better choice to send renderable geometry to the client and run the pipeline to completion. However, in some use cases this approach is not applicable because the client may be overwhelmed by a higher geometry count and therefore geometry must be processed on the server. VisIt provides a dynamic strategy to switch between these rendering modes. Figure 7 illustrates which steps occur on client and server side for both modes [3].
The *Send-Images Partitioning*, also called scalable rendering mode, makes use of the sort-last algorithm. The advantage of this mode is that only images travel through the network. On the other hand, it might be not applicable when it comes to interactive visualization, where a certain frame rate is desired because there is latency in frame production and in the network.
The *Send-Geometry Partitioning* illustrated below, is more feasible for interactive visualizations. In this case the geometry is sent towards the client and is additionally rendered. With this approach, parameters (e.g. for viewing) resulting to a modified visualization of the current geometry do not need any communication with the server. However it is not applicable when the memory of the

client is not sufficient for large geometries.

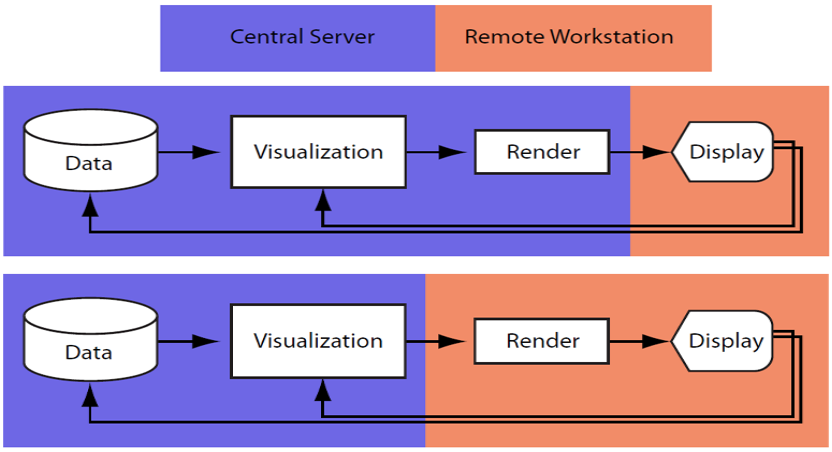


Figure 7: VisIt's pipeline partitioning (from [3])

3.4 In-situ visualization

This thesis deals with the generation of a visualization interface using the in-situ approach. In general, there are two different strategies for this approach. These are called tightly coupled and loosely coupled in-situ visualization. There are also some hybrid versions combining both strategies. However, all in-situ strategies have in common that simulation and visualization routines skip I/O. After all, with post-hoc visualization the price for I/O is paid twice since the simulation results must be written to a file and additionally read by the visualization program. With the tightly coupled approach, simulation and visualization routines share the same resources. As a result the rate at which data is accessed is higher, as also shown in figure 8. This allows visualization and analysis routines to be performed on every time step, whereas in post-processing the number of files that can be written are compromised due to the slow rate of I/O. One disadvantage that comes from the shared memory is that simulation and visualization compete for the memory with the result that the simulation may be affected by errors coming from the visualization. Another important advantage is the minimal communication between simulation and visualization. As soon as the visualization routines are integrated into the simulation code, the visualization can simply take place after each simulation step in the mainloop, which will be described in chapter 4.3, and once the visualization is done, the loop continues.
When the visualization is loosely coupled, the simulation data is sent towards dedicated visualization nodes via network. This allows the concurrent execution of simulation and visualization with the advantage that the simulation is not impacted by errors occurring at the visualization. As shown in figure 8, the bandwidth via network is higher than the rate of I/O, but it is slower in comparison to tight coupling. This brings up the issue that not as much data can be transferred, but it can be reduced with usage of filtering or compression. Also, compared to the tightly coupled approach, data needs to be duplicated when it is sent to the nodes. This might cause performance issues for the simulation. Also, the communication between simulation and visualization is not as simple as in the tightly coupled case, because it has to be ensured that data is sent and received correctly.
The hybrid approach finds use when neither pure strategies are applicable, for example when the network or the compute node's resources are exhausted. In this case, tightly coupled components are used to reduce the data before it is passed to a dedicated visualization node via loose coupling, where the visualization process is continued. With the combination of them, their advantages and disadvantages are still valid to some extent. More information about both strategies can be found at [6, 18]. A list of visualization frameworks using either of those strategies, results from use cases and also a more detailed comparison between both approaches can be found at [11].

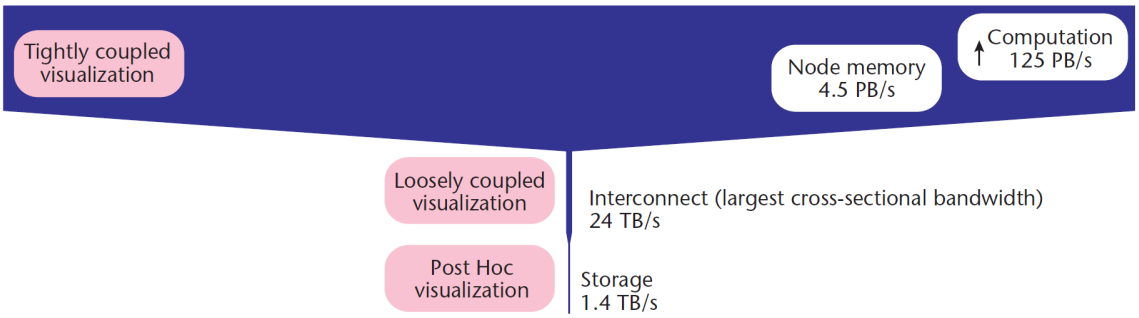


Figure 8: Comparison of bandwidth between post-processing and both in-situ approaches on the Titan supercomputer at the Oak Ridge Leadership Computing Facility (from [18]).

3.5 Libsim

VisIt provides an in-situ visualization library called Libsim, which allows to access the data from a simulation with the tightly coupled strategy. One step towards the coupling between a simulation and the visualization tool VisIt is the linking with the library and the inclusion of the headers it contains. From the perspective of a code generation framework user, this should happen automatically. In fact it does, but it uses an environment variable `SIMV2DIR` (see appendix A) that should be set by the user since the installation path may differ on each machine.
The library offers two interfaces which provide the analysis and visualization of data using the whole tool set VisIt possesses. One of them is the data interface. It is a collection of interfaces allowing to send data to a VisIt client when it is required. These interfaces offer allocation and getter/setter functions for the data that is sent to VisIt. Normally, the plain data values from a simulation are not enough to create a visualization with VisIt, as they need to be transformed into a suited format. For this reason, data objects are used. They contain references to the data and are used to create data structures that VisIt can visualize. The callback functions explained in chapter 4.4 will use this representation. These callbacks use functions from the data interface to supply the data objects with information. In order to distinguish the various kinds of data objects, handles are allocated as an identifier for the corresponding kind. The data objects are then returned and additionally transformed into VTK objects. VTK supports C data types like double, int, etc. Other data types need to be transformed into one of these supported data types by copying and casting it into a temporary variable. Libsim also offers function arguments that allow VisIt to free the variable from the memory once the calculations are finished [25, 27].
The second interface is the control interface. It has the capability to expose itself to VisIt clients (see chapter 4.2), creating a listen socket to detect connections coming from them and also to advance the connection back. In case that the connection back towards the VisIt instance was successful, the simulation can be seen on the VisIt GUI. Another key task is the handling of visualization requests, which can be made in case callback functions have been provided. This also includes the notification (see chapter 4.4.5) towards the viewer that new data was produced by the simulation [6].
Libsim consists of the two libraries illustrated in figure 9. The front-end is a lightweight static library that contains functions from the control interface and is linked with the simulation. The second is the heavyweight dynamic runtime library which is loaded by the front-end library as soon as the connection has been completed. The runtime library contains functions performing the visualization routines of a compute engine, which makes an instrumented simulation rather similar to a compute engine. The simulation data is accessed with the callback functions that are registered with the runtime library. One important feature that comes from the dynamic loading of the runtime library is that the overhead is reduced to zero when not used [6, 27].

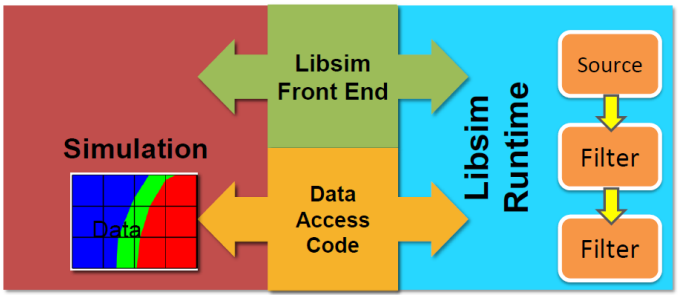


Figure 9: Connection between Simulation and Libsim (from [9]).

4 Implementation

To transport simulation results to a visualization program, changes to the simulation program need to be made. For offline visualization, the data is transformed into a format that is supported by the visualization program and is written into a file afterwards. For online visualization, a connection between the simulation and the visualization program needs to be established and data access functions that also transform the simulation data into a suitable format must be implemented. This chapter describes mandatory changes to the simulation's code to enable in-situ visualization. The integration into the code generation framework and the outcome for a manifold of input parameters declared in the DSL files will be explained in combination with these mandatory code changes.

4.1 Design

For the implementation of the visualization interface, following decisions regarding the coupling between VisIt and ExaStencils have been made:

1. VisIt emphasizes the storing of simulation data in a typedef struct which, in general, represents the state of a simulation. It should contain variables storing the number of time steps or cycles, an indicator used to distinguish a running or stopped simulation and also pointers to simulation results. The callback functions, that will be described in chapter 4.4 have a similar signature which allow structs to be passed as an argument. This design allows the exposure of the required data towards the callback functions without using global variables. This approach will not be used within the scope of this thesis because the ExaStencils framework does not provide data structures for the realization of structs and additional to that, algorithmic data types such as fields must be declared as global within the DSL code anyways [21]. Instead of the emphasized struct the callback functions receive a `nullptr` as an argument.

2. Libsim has the two different versions simV1 and simV2. Version simV1 is the older version and uses structs (other than the ones mentioned earlier) that are filled with data and passed to VisIt. This approach is very sensitive to not properly filled structs. This turned out to be a cause for crashes. This problem does not occur when using the newer version simV2 which offers a list of various functions that fill the previously mentioned data objects. Within this implementation, the version simV2 is used as it fixes crashing issues and also offers additional features like mesh types, etc. [25].

3. Libsim offers two different approaches for an in-situ visualization. The first one is the interactive visualization mode. Here, simulation results are displayed as requested by the user on an interface such as the GUI. One negative aspect is that data is only available for a limited time. The second approach is the batch processing of Libsim, which allows to store the visualization results into databases. However, in this approach the visualization steps are predetermined and require many changes to the DSL code since the user must specify which plot is applied at a certain time. Within this thesis, the interactive approach is used since it is easier to use and a specification at which exact point of time data should be visualized is often not possible [14, 18].

4.2 Initialization

After the interfaces and the static library have been added to the simulation successfully, initialization steps have to be made to establish the connection with VisIt. This chapter explains changes to the code that have to be made in case that a serial or parallel simulation is coupled with Libsim. The control interface contains the functions needed for this purpose.
The first two initialization functions are optional. The first function is `VisItSetDirectory` which takes the path to the directory containing the VisIt executable. This path is set with usage of the `VISIT_HOME` (see appendix A) environment variable. Calling it prevents version mismatches in case that the user has several versions of VisIt installed and their executables added to the path environment. It also proves to be useful if the path environment does not contain any VisIt executables. For debugging purposes the `VisItOpenTraceFile` function can be called. It records every operation that Libsim performed and dumps it into a text file. This might prove to be useful to

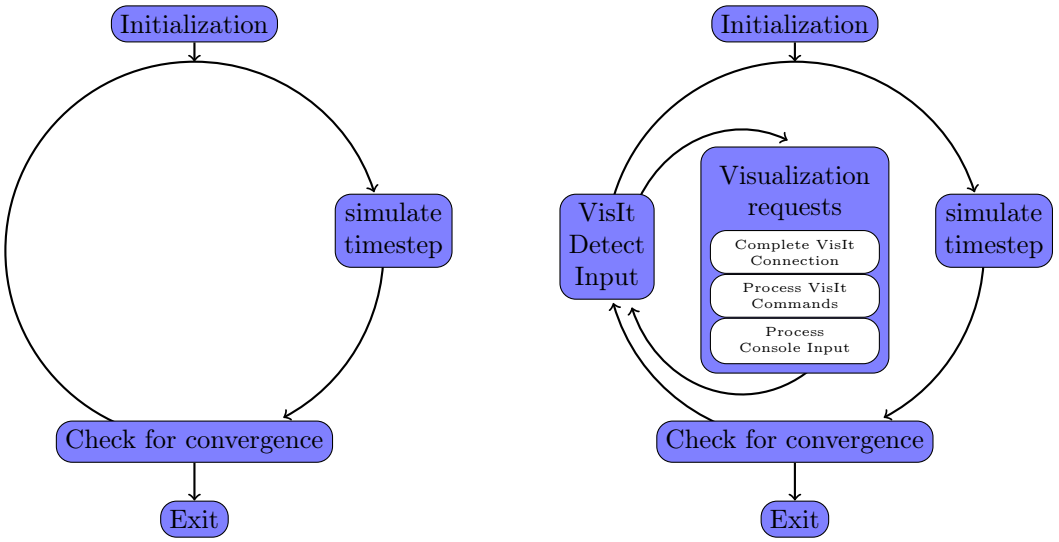
determine the origin of failures. At the de-initialization stage of a program, the trace file will be closed using the `VisItCloseTraceFile` function. In case of a parallel simulation both functions should be called from every processor [15].
Parallel simulations coupled with Libsim need to implement additional callback functions because only the root process has a TCP connection with VisIt. These callbacks are needed to ensure that each processor operates on the same task. The communication between them is realized with a simple broadcast from the root process. Libsim offers the functions `VisItSetBroadcastIntFunction` and `VisItSetBroadcastStringFunction` to register the required callback functions for Integers and Strings. As soon as these functions are registered, the functions `VisItSetParallel` and `VisItSetParallelRank` can be called to tell VisIt that the simulation is run in parallel.
Another mandatory initialization step for both serial and parallel simulations is to call the function `VisItSetupEnvironment` which extends the current environment with paths to find VisIt's plug-ins and libraries. Naturally, in a parallel simulation every processor has to call this function. There is also an additional implementation `VisItSetupEnvironment2` which is used when MPI permits the creation of processes that are identifying the environment. In this case, only the root process discovers the environment and additionally passes it towards the function to broadcast this value. Therefore for parallel simulations, the second version is used to avoid conflicts with MPI and for serial simulations, the first version is chosen [15, 25].
The last step is to call the `VisItInitializeSocketAndDumpSimFile` function. As soon as the function is called, Libsim is initialized and the simulation listens for connections coming from a VisIt client. The connection between simulation and client is realized with the opening of the .sim2 file that is created by the function. This file can be found in the ∼/.visit/simulations (Unix) or %Documents%\VisIt\simulations (Windows) directory. The file contains information such as the host name and the port that is needed to establish the connection. The function also has the option to pass a custom user interface file that opens up more possibilities for simulation steering. Parallel simulations should only call this function from the root processor because it is the only process communicating with VisIt and only one .sim2 file should be created [15, 27].

- VisItSetDirectory
- VisItOpenTraceFile
- Parallel only initializations
  - VisItSetBroadcastIntFunction
  - VisItSetBroadcastStringFunction
  - VisItSetParallel(1)
  - VisItSetParallelRank(mpiRank)
- VisItSetupEnvironment(serial), VisItSetupEnvironment2(parallel)
- VisItInitializeSocketAndDumpSimFile, only called from root process if parallel

Figure 10: Initialization steps for serial and parallel simulations

4.3 Adapting the mainloop

As mentioned in chapter 3.5, Libsim is periodically listening for connections from a VisIt client and also responds to a client's commands. Therefore changes to the code that enable this kind of interactivity must be made. Figure 11.a shows the event loop of a typical simulation. The starting point is the *Initialization*. Normally, a simulation's data will be assigned to an initial value and some pre-calculations take place. After the *Initialization* phase is finished the program enters the *mainloop*. In general, a *mainloop* performs a sequence of *simulate timestep* calls until the stopping criteria illustrated in *Check for convergence* of the algorithm are fulfilled and the simulation reaches the *Exit* state where the de-initialization and termination of the program begins.



(a) regular mainloop

(b) VisIt mainloop (from [27]).

Figure 11: Comparison of the two mainloops

The comparison between figure 11.b and figure 11.a shows that the difference between both event loops are the two states *VisItDetectInput* and *Visualization Requests*. The latter is responsible for handling requests coming from a console or viewer and completes the connection between simulation and VisIt. The *VisItDetectInput* state represents the function from the control interface, which detects input coming from a VisIt instance once the socket has been initialized. This function takes the arguments `blocking` and `consoledesc`. The argument `blocking` tells the function to wait until input arrives or to continue the simulation. This argument will be used for the commands that control the flow of the simulation. The `consoledesc` argument allows the user to use the terminal to enter commands if a valid file descriptor has been passed and in case that this feature is not needed, the value `-1` can be used. Also, when the infinite blocking until input arrives is not desired, another version `VisItDetectInputWithTimeout`, which takes an additional argument to set the time until a timeout occurs, can be used. Based on both function's return value different actions need to be taken [25, 27]:

```
int visit_input = VisItDetectInput(int blocking, int consoledesc)
```

- visit_input < 0: An error occurred or the function was interrupted. In this case the function should not be called anymore resulting to the termination of the program.
- visit_input $= 0$: A zero is returned when a timeout occurred or when the function did not block. After a timeout the simulation should regain control and is realized by performing a simulation step and additionally rejoining the mainloop again once it completed. Using the second version `VisItDetectInputWithTimeout` would enable to listen for connections while the simulation is running.
- visit_input $= 1$: An incoming connection from a VisIt client is detected and additionally accepted by the `VisItAttemptToCompleteConnection` function. It also loads the dynamic runtime library in order to use the compute engine's visualization routines. Afterwards the function tries to connect back to Visit and once the connection is complete, the simulation is visible on the viewer. Still, requesting plots of the simulation data and controlling the simulation flow by using the buttons on the GUI is not possible yet. For this reason the callback functions that will be described in chapter 4.4 are registered here.
- visit_input $= 2$: Once the callback functions are registered and visualization requests have been sent from the viewer, they have to be processed. The `VisItProcessEngineCommand` function is responsible for this task. The return value zero indicates an error that occurred during the process. In this case the simulation cuts the connection with VisIt and starts running again. The function `VisItDisconnect` should be used for this purpose. It resets Libsim completely and also allows the user to reconnect to the simulation.

- visit_input = 3: In case a valid file descriptor was passed to the `VisItDetectInput` function, commands coming from the console can be read and processed. Their implementation is almost identical to the one of the command buttons which will be explained in chapter 4.4.5.

In parallel simulations, only the root process should call this function because it is the only process communicating with VisIt. The processes should work in unison, which means that the result of `VisItDetectInput` should be broadcast among the other processes. The same should be done when it comes to the execution of control commands coming from the console. Here, the root process reads the commands with `VisItReadConsole` and writes it into a buffer. This command is broadcast to the other processes and executed. When it comes to the execution of engine commands coming from the client, the synchronization is a bit more complex. In order to tell the slave processes to execute the command, a callback function broadcasting an indicator to them must be implemented and registered. The root simply processes the command whereas the slave processes wait for input such as the indicator or the return value from the `VisItProcessEngineCommand` function executed by the root. In case that the current slave process is instructed to execute a command, it returns the value passed from the root after the command has been executed [15, 25].

4.4 Data access functions

As previously mentioned in chapter 3.5, callback functions that transform the plain simulation data into a VTK object must be implemented to enable plots to be created on demand. In addition, the implemented callbacks must be registered with the dynamic runtime library after the connection with VisIt is complete [27]. This chapter shows possible implementations for the data access functions that provide a visualization for the data requested and describes, how these implementations can vary depending on the parameters declared in the DSL files. The steps that make interactive visualization and simulation steering possible, are also explained here.

Exposing data to the pipeline

In order to supply the data objects, which additionally are transformed into VTK objects and put into the visualization pipeline, with information, functions from the data interface can be used. These functions are the foundation for the callback functions introduced here. As mentioned in chapter 3.5, VTK only supports C data types. In combination with the data types available in ExaSlang, these are floating point numbers and integers. Figure 5 shows the signature of these functions. Its arguments play an important role within this implementation. Especially the `owner` argument is relevant, since it determines which instance is responsible for the memory management, i.e. freeing the data whose pointer is passed towards the function. In the case that temporary memory is allocated and passed as an argument, the ownership should be given to VisIt with `VISIT_OWNER_VISIT` and the memory is freed after the calculations in VisIt are finished. Otherwise ownership is given to the simulation via `VISIT_OWNER_SIM` where the memory is normally freed at the de-initialization stage. The value `nComps` denotes the number of components that the values passed consist of. For example `nComps = 1` would be used for scalar fields whereas higher values can be used when vectors, tensors, etc. are visualized. The argument `nTuples` describes the number of values that are passed [25].

```
int VisIt_VariableData_setDataD(visit_handle obj, int owner,
    int nComps, int nTuples, double *);
int VisIt_VariableData_setDataI(visit_handle obj, int owner,
    int nComps, int nTuples, int *);
```

Listing 5: Function supplying the data objects

4.4.1 Metadata

Once the connection between simulation and VisIt is completed, the simulation is asked to provide metadata [6]. In general, metadata is lightweight information describing the characteristics of the actual simulation data. In this thesis, metadata will be provided to describe the simulation's status, meshes, variables, curves and custom commands. The data object returned from this function

contains a collection of the metadata that has been provided. As soon as the function has been registered, the components from this collection including the control buttons for the commands are visible on the GUI.

```
visit_handle SimGetMetaData(void* cbdata) {
    visit_handle md = VISIT_INVALID_HANDLE;
    if(VisIt_SimulationMetaData_alloc(&md) == VISIT_OKAY) {
        /* update simulation status(time, cycle, mode) */
        if(VisIt_SimulationMetaData_alloc(&md) == VISIT_OKAY) {
            int mode = (visit_runMode == true)
                ? VISIT_SIMMODE_RUNNING : VISIT_SIMMODE_STOPPED;
            VisIt_SimulationMetaData_setMode(md, mode);
            VisIt_SimulationMetaData_setCycleTime(md, sim_cycle, sim_time);
        }
                                    ⋮
```

Meshes are defined in two or three-dimensional space and required for the visualization of the field data. One important attribute to describe a mesh is its type. In general, it describes how the points in a discretized domain are connected to each other and which shapes the elements that emerged from the connection between these have. Another important feature that is supplied by this visualization interface is the illustration of one-dimensional data. VisIt offers a curve callback function for this purpose [25], but it does not support multiple sub-domains. Therefore, curvilinear meshes that consist of the variable's values and their corresponding position in the domain are used to provide a graphical illustration for one-dimensional fields on multiple sub-domains and to grant an additional representation for two-dimensional variables as shown in chapter 4.4.4. Another important metadata for meshes is the name it is given. As shown in the fraction for variable metadata in listing 6, the name is used to appoint a mesh to place the variable on.

```
                                    ⋮
        /* set mesh name and type, dimensionality and number of domains */
        visit_handle mmd =  VISIT_INVALID_HANDLE;
        if(VisIt_MeshMetaData_alloc(&mmd) == VISIT_OKAY) {
            VisIt_MeshMetaData_setName(mmd, "mesh2d");
            VisIt_MeshMetaData_setMeshType(mmd, VISIT_MESHTYPE_RECTILINEAR);
            VisIt_MeshMetaData_setTopologicalDimension(mmd, 2);
            VisIt_MeshMetaData_setSpatialDimension(mmd, 2);
            VisIt_MeshMetaData_setNumDomains(mmd, 4);

            VisIt_SimulationMetaData_addMesh(md, mmd);
        }
                                    ⋮
```

The visualization of variables can be interpreted as the mapping of values to certain elements of the mesh such as points, areas or volumes. Libsim allows the placing of values onto a mesh's nodes or zones. The differences between them will be explained in chapter 4.4.3. Besides the nodal or zonal variable centering, the variable type must also be specified. In this code fraction, metadata for a scalar variable is provided. But the visualization of vectors, tensors and arrays is also possible. With the variety of variable types, a specification for the number of components they consist of is required. This specification takes place in the variable callback function.

```
                                    ⋮
        /* specify mesh, variable type, name and centering */
        visit_handle vmd =  VISIT_INVALID_HANDLE;
        if(VisIt_VariableMetaData_alloc(&vmd) == VISIT_OKAY) {
            VisIt_VariableMetaData_setName(vmd, "zonal");
            VisIt_VariableMetaData_setMeshName(vmd, "mesh2d");
            VisIt_VariableMetaData_setType(vmd, VISIT_VARTYPE_SCALAR);
```

```
            VisIt_VariableMetaData_setCentering(vmd, VISIT_VARCENTERING_ZONE);

            VisIt_SimulationMetaData_addVariable(md, vmd);
        }
                                        ⋮
```

The commands that are added here will be displayed on the *Simulations* menu on the GUI. There is a maximum of six commands that can be displayed. This maximum can be exceeded by using a custom .ui file. As already mentioned in chapter 4.3, it is also possible to process commands which have been read from the command line. A generated target code will always provide the functionality to control the simulation with a `step`, `stop`, `run`, `switchUpdates` command which can be entered via the buttons on the GUI or the command line. In the territory of multigrid solvers the commands `level up` and `level down` are also available. A brief explanation for each command's functionality can be found in chapter 4.4.5.

```
                                        ⋮
        const char *cmd_names[] = {"step", "stop", "run", "switchUpdates"};
        for(int i = 0; i < sizeof(cmd_names)/sizeof(const char *); i++) {
            /* install commands */
            visit_handle cmd = VISIT_INVALID_HANDLE;
            if(VisIt_CommandMetaData_alloc(&cmd) == VISIT_OKAY) {
                VisIt_CommandMetaData_setName(cmd, cmd_names[i]);
                VisIt_SimulationMetaData_addGenericCommand(md, cmd);
            }
        }
    }
    return md;
}
```

Listing 6: Metadata callback function

4.4.2 Rectilinear meshes

In order to visualize two- or three-dimensional variables, a mesh callback function has to be implemented first. This function returns a data object which is filled with information necessary for the requested mesh. Depending on the mesh type, additional information or even additional callback functions must be provided.
The focal point of this thesis is to provide a visualization for multigrid solvers on rectilinear domains since their corresponding meshes are very memory efficient compared to others. These are aligned to its specified coordinate values on each axis whereas other mesh types require the coordinate values for each discretization point. Listing 7 shows a possible implementation for a callback function constructing a two-dimensional rectilinear mesh. Here, handles for the mesh and its coordinates are allocated and the coordinate arrays $coords[d]$ with $numPoints[d]$ points are passed. No temporary memory is allocated, so the simulation is responsible for the memory management. In fact, a three-dimensional implementation is rather similar. The difference is an additional handle that is allocated, filled with the third coordinate array and added to the handle returned by the callback function.

```
visit_handle SimGetMesh(int domain, const char* name, void* cbdata) {
    visit_handle h = VISIT_INVALID_HANDLE;
    if(strcmp(name, "mesh2d") == 0) {
        if(VisIt_RectilinearMesh_alloc(&h) == VISIT_OKAY) {
            visit_handle handles[2];
            VisIt_VariableData_alloc(&handles[0]);
            VisIt_VariableData_alloc(&handles[1]);
            VisIt_VariableData_setDataD(handles[0], VISIT_OWNER_SIM, 1,
            coords[0], numPoints[0]);
```

```
            VisIt_VariableData_setDataD(handles[1], VISIT_OWNER_SIM, 1,
            coords[1], numPoints[1]);
            VisIt_RectilinearMesh_setCoordsXY(h, handles[0], handles[1]);
        }
    }
    return h;
}
```

Listing 7: Callback function for a 2D rectilinear mesh

Integration to the generator

As mentioned in chapter 2.2, the code transformations for the visualization code occur at the IR. Here, the knowledge parameters described in 2.6 have been set by the input files for the compiler and additional data structures are available. With the knowledge parameters it is possible to construct a rectilinear mesh for each block. However, as shown in chapter 2.4 there are different localizations which can be specified in a field's layout. Also, in case of a staggered grid it is possible that multiple localizations are present. In chapter 4.4.3 it will be shown that variables can be either mapped onto the nodes or zones of a mesh and also that the mesh construction for node- and cell-centered variables is the same. Nonetheless, nodal and zonal variables are the only options and an adaption for face-centered variables must be made. The adaption is the creation of additional meshes consisting of the face centers' positions. In the generator, different localization types are collected and for each type present, code responsible for the initialization and de-initialization of its coordinate arrays is produced. Additionally, its corresponding metadata and mesh callback is added. At the IR, these types can be collected by iterating over the `IR_FieldCollection`, which contains an entry for each field and its levels declared in the DSL code. For each entry, the `IR_FieldLayout` and its `IR_Localization` can be retrieved. In case that a new localization type is found, it is added to the collection. The calculation of each block's number of points per dimension uses following parameters and variables:

- $numFrags[d]$: Alias for knowledge parameter domain_rect_numFragsPerBlockAsVec.
- $fragLen[d]$: Alias for knowledge parameter domain_fragmentLengthAsVec.
- $level$: Represents the current initialization level which is in range of the knowledge parameters minLevel and maxLevel.
- $isNodal[d]$: This variable is either "1" or "0" per dimension. Node- and cell-based variables can be put on the same mesh, which means that it is "1" in each direction for both. For face-centered variables it is set to "1" for its nodal component, otherwise it is "0".

For the current localization type and $level$, the number of points contained in dimension d are calculated as follows:

$$numPointsTotal[d] = numFrags[d] * fragLen[d] * 2^{level} + isNodal[d] \tag{2}$$

As soon as the number of points is known and the coordinate arrays are allocated, their corresponding coordinate values must be set. The start position of each fragment and block per dimension is required for this purpose. The IR layer has data types that contain the actual position and index of each fragment within the whole domain, in the meaning that the global position of the block is also taken into consideration. For the computation of the coordinate values, additional variables are introduced:

- $stepSize[d]$: Stores the step sizes from the knowledge parameters discr_h*.
- $fragPos[d]$: Alias for the IR data type `IR_IV_FragmentPositionBegin` which returns the global start position of the current fragment.
- $center[d]$: Depending on the localization and dimension this value is either "0" or "0.5". For face-centered variables it is "0.5" for its cell-centered components, otherwise "0".

- $fragIdx[d]$: Equals the IR_IV_FragmentIndex data type which contains the global index of the current fragment for each dimension.

The initialization of the coordinates is done by iterating over the fragments. Derived from equation 2, the number of points of fragment is calculated as follows:

$$numPoints[d] = fragLen[d] * 2^{level} + isNodal[d] \tag{3}$$

For each discretization point i within a fragment, the coordinate value must be determined. For dimension d, this value is computed in dependency of the current $level$ and localization with:

$$coords[d][i + startIdx] = fragPos[d] + (i + center) * stepSize[d] \tag{4}$$

The value $startIdx$ is used to determine the offset towards the current fragment's first grid point in direction d and can also be interpreted as the position of the duplicate layers shared by adjacent fragments. In order to get a block-local offset towards its current fragment, the global index retrieved from $fragIdx$ must be mapped to a local index first. The offset $startIdx$ towards the current fragment in direction d is determined by:

$$startIdx = (numPoints[d] - isNodal[d]) * (fragIdx \ \% \ numFrags[d]) \tag{5}$$

4.4.3 Variables

Once the data access function for the different kinds of meshes has been added and an appropriate visualization for each mesh has been provided, the data access function placing the variable values onto a mesh is ready to be implemented.
VisIt offers two approaches to place a scalar variable onto a mesh. The first one is the node-based centering where variable values are placed onto the mesh's nodes and within the zone that a number of nodes span, interpolation takes place and a color gradient is built. Depending on the dimensionality, the zone is represented as the area or volume of a mesh's element, e.g. the volume of a cube within a three-dimensional rectilinear mesh. The second approach is the zone-based centering where the variable value is put onto the whole zone without performing any interpolation, resulting that the whole zone has the same color [25].

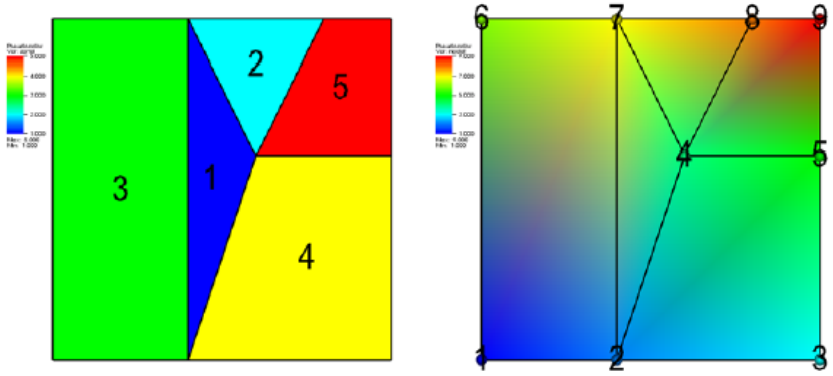


Figure 12: Zonal and nodal variable placed on a 2D unstructured grid (from [25]).

Figure 12 illustrates the different visualization results of zonal and nodal variables placed on the same mesh. It also shows that the number of values passed to the callback function is different for both representations. In consideration of rectilinear meshes being used, the number of its zones is one value less than the number of nodes per dimension and therefore the same mesh can be used to place node- and cell-centered variables on. As described in the previous chapter, face-centered variables have their own meshes consisting of the face centers' coordinates. These variables will be represented with a nodal plot.
Listing 8 shows a variable callback function for a two-dimensional zonal variable. It passes the pointer of the variable directly to the `VisIt_VariableData_setDataD` function and therefore ownership is given to the simulation. Since the mesh has been constructed with $numPoints[d]$ points per dimension, $numPoints[d] - 1$ zones must be passed.

```
visit_handle SimGetVariable(int domain, const char* name, void* cbdata) {
    visit_handle h = VISIT_INVALID_HANDLE;
    if(VisIt_VariableData_alloc(&h) == VISIT_OKAY) {
        if(strcmp(name, "u") == 0 && curLevel == 6) {
            VisIt_VariableData_setDataD(h, VISIT_OWNER_SIM, 1,
                (numPoints[0]-1)*(numPoints[1]-1), fieldData_u);
        }
    }
    return h;
}
```

Listing 8: Callback function for a 2D zonal variable

Integration to the generator

The fields declared in the DSL code may have additional layers such as ghost and pad layers. By adding these layers, the field's number of points is greater than the ones of its mesh, resulting in a wrong visualization. These layers are used for communication and optimization and will not be visualized. The points that do not belong to those layers will be referred as inner points. In order to prevent the visualization of the ghost or pad data, only the inner points are copied into a temporary memory whose pointer is passed to the `VisIt_VariableData_setDataD` function and therefore ownership must be given to VisIt.
In order to accomplish the goal of providing a visualization for the whole integrity of data, i.e. every field declared in the DSL code, the `IR_FieldCollection` is iterated over. For each field, an implementation in the metadata and variable callback is generated. Besides the fields' various properties such as localization and number of ghost/pad layers, there are also differences in their range of levels, e.g. when a CG solver is applied on the coarsest level and its results are only available on this level. In order to prevent mismatches between the field's range of levels and the current visualization level `curLevel`, it needs to be queried whether an visualization of the field is possible as shown in 8. Since the field collection contains entries for each field and its corresponding levels, data access code similar to this listing is produced for each entry. In case that a visualization request for unavailable field data is made, an empty handle is returned by the variable callback and the corresponding error message is visible on the GUI. As stated in chapter 4.4.2 the field layout provides information of the field's localization and depending on it, either zonal or nodal variable centering is chosen in metadata and variable callback.
Within ExaStencils the field data arrays are linearized. The calculation of the offset towards a variable's inner points is dependent on its dimensionality and the number of total points per dimension, in other words including ghost/padding layers. All this information is contained in the field layout. In order to assemble an equation for this problem, following variables are introduced:

- $numInnerPoints$: Stores a variable's number of discretization points per dimension. It is determined by subtracting the index of the "rightmost" duplicate layer by the index of the "leftmost" duplicate layer.
- $numOuterLayers$: Contains the number of ghost/pad layers on the "left" side per dimension in order to calculate the offset towards the first inner point. As illustrated in chapter 2.3, padding layers surround ghost and inner layers. For this reason, the number of outer layers is computed with the subtraction of the index for the "leftmost" duplicate layer with the "leftmost" padding layer.
- $numTotalPoints$: This value combines the number of inner and outer points and is calculated by subtracting the "rightmost" with the "leftmost" duplicate layer.

For two-dimensional fields, the linearized offset can be calculated as follows:

$$offset_{2d} = numOuterLayers[1] * numTotalPoints[0] + numOuterLayers[0] \tag{6}$$

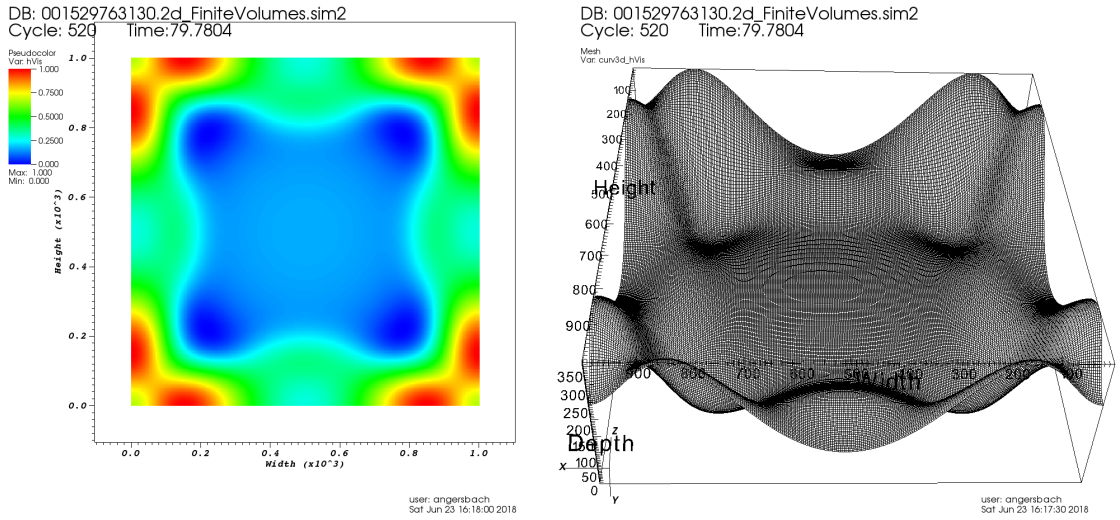
Whereas for three-dimensional cases:

$$offset_{3d} = numOuterLayers[2] * numTotalPoints[0] * numTotalPoints[1] + offset_{2d} \tag{7}$$

With the construction of a mesh containing the nodes of each fragment within a block, the same should be done with the variables. In fact, the equations described in chapter 4.4.2 are also used to forge the variable arrays from the fragments together with the important difference that $isNodal$ is zero for cell-based variables because of the zonal mapping. As a result, the temporary variable also has $numFrags[d] * (numInnerPoints[d] - isNodal[d]) + isNodal[d]$ points and each fragment's $startIdx$ equals $(numInnerPoints[d] - isNodal[d]) * (fragIdx \ \% \ numFrags[d])$. In case that the offset is zero, which basically means that no outer layers exist, and a block consists of only one fragment, no temporary memory is allocated and the pointer of the variable is directly passed to the `VisIt_VariableData_setDataD` function and ownership for the memory management is given to the simulation.

4.4.4 Curvilinear meshes

In addition to the visualization of variables mapped onto the nodes or zones of a rectilinear mesh, the visualization interface also provides the illustration of one- and two-dimensional variables as curvilinear meshes that are one dimension higher than the actual dimensionality of the problem that is solved. The curvilinear meshes consist of variable values and their corresponding coordinates of the regular mesh points. On the contrary to the rectilinear meshes which only need the coordinate values on each dimension's axis, curvilinear meshes require the coordinate values per dimension for every discretization point. Figure 13 pictures the two visualization possibilities for two-dimensional fields. On the left side the pseudocolor plot of a variable can be seen and its corresponding curvilinear mesh can be seen on the right side. However, it is often necessary to scale the variable and therefore the command `scale=<value>` can be entered in the command line to scale the variable values as they are copied. With this variable representation, a distinction between node- and cell-based variables must be made since these are used to construct the mesh now instead of being placed on a mesh. As a result, the coordinates of the cell-centers are used. The handling of face-centered variables on rectilinear meshes can also be applied.



(a) 2D pseudocolor plot

(b) 3D curvilinear mesh

Figure 13: Provided visualization possibilities for 2D field data

The strategy used to provide data access functions for these meshes is slightly different compared to the rectilinear meshes. However, the initialization stage is rather similar in the meaning that also localization types are collected and the belonging coordinate arrays for the meshes are generated. The difference lies in the implementation of the data access functions. The callback function for the curvilinear meshes can be seen as a combination of the callbacks for the rectilinear meshes and the variables. Here, the field collection is iterated over and the localization type is retrieved from the field layout and in case that ghost or pad layers are used, the field's inner points are transferred to

temporary memory and ownership of the memory management for it is given to VisIt. Depending on the localization, pointers to the corresponding coordinate arrays in addition to the values from the current field are passed to VisIt.

4.4.5 User defined commands

The commands that have been declared in chapter 4.4.1 are visible in the viewer but they cannot be used yet. To enable interactive visualization, these commands need to supply mechanisms to control the state of the simulation. In other words, a callback function has to be implemented and registered. These commands execute time steps and allow the switching between levels. Plot updates must be explicitly requested within the source code. For this purpose the functions `VisItTimeStepChanged` and `VisItUpdatePlots` can be used. `VisItTimeStepChanged` signals that the simulation advanced a number of time steps and as a result, new metadata is sent. More importantly the `VisItUpdatePlots` function tells VisIt that new data is demanded in order to update the current plot and the display for time and cycle [15].
The implementation of the actions that are performed by each command can vary, but their detection is similar. Listing 9 shows a template for a control command callback function.

```
void ControlCommandCallback(const char* cmd, const char* args, void* cbdata) {
    if(strcmp(cmd, "step") == 0) {
        /* implementation for the command */
                        ⋮
    }
    /* more commands */
            ⋮
}
```

Listing 9: Template for a control command callback

The visualization interface offers the following commands:

- switchUpdates: Used to determine whether the plot should be updated after the computation of a time step. At the beginning of the simulation auto-updates are enabled.
- step: Simulation advances by performing a single simulation step. In addition the simulation metadata will be updated automatically whereas the plots are updated depending on the switchUpdates status.
- run + stop: Toggles an internal boolean variable that is used to determine if the `VisItDetectInput` function should block or not.
- level down: This command decreases the current level used for the visualization by 1 until the value of the knowledge parameter minLevel is reached. The plots are always updated when one of the level switches is used.
- level up: This command increases the current level until maxLevel which is also the initial visualization level is reached.

4.5 Changes to the DSL code

The objective of this thesis was to enable code generation for an in-situ visualization with only minimal changes to the DSL code. This section shows which variables, functions and function calls have to be added into the DSL code. In general, the names for the variables and functions that are used in this chapter should be used as well because the generated visualization functions will in fact access the variables and call the functions with these names.
The first additions are the global variables which are used to represent the simulation's status. They cannot be generated automatically since the DSL code for the algorithm depends on them. Listing 10 shows the variables to be specified. The `sim_done` variable is used to determine the

simulation's termination and to exit the mainloop. It has to be explicitly set to true within the DSL code once the stopping criteria of the algorithm are fulfilled. As shown in chapter 4.4.1 the current cycle and time of the simulation is passed as metadata towards the VisIt client. This is used to give the user an overview of the current simulation state. For this purpose `sim_time` and `sim_cycle` are used. The user has to specify when one or both of these values are updated.

```
Globals {
    Var sim_done : Boolean = false
    Var sim_time : Real = 0.0
    Var sim_cycle : Int = 0
}
```

Listing 10: Mandatory global variables

The next step is the implementation of the `simulate_timestep` function. This function varies depending on the problem that is calculated. In fact, the same applies to the stopping criteria of the simulation, resulting that both states illustrated in figure 11 must be implemented by the user. The following time-dependent implementation forges both together and can be used as a template:

```
Function simulate_timestep@(finest) {
    if(sim_time < maxTime) {
        /* perform calculations */
                        ⋮
        /* update time or cycle */
        sim_time += dt
    } else {
        sim_done = true
    }
}
```

Listing 11: Template for a time-dependent simulate_timestep function

The following changes to the DSL code occur at the application function. As pointed out in chapter 4.2, initialization steps must be made to establish a connection between simulation and VisIt. This also includes the allocation and initialization of the data structures used to build the mesh. Therefore, it is important to call the `visit_init` function which is responsible for these steps after the initialization of the global variables because the function calls for the connection establishment require MPI related variables. Also, in case that several fragments or blocks are used, their number has to be known and for the coordinate variables the start position of the blocks/fragments in each dimension is required. The most important step is to add the `visit_mainloop` function that was discussed in chapter 4.3. The callback functions that are registered and the console commands that are implemented in the mainloop are responsible for calling the `simulate_timestep` function, when requested by the user. Additionally, once the `visit_mainloop` is finished, the `visit_destroy` function which frees the memory from the data types needed for the visualization and closes the trace files used for debugging as mentioned in chapter 4.2. At last the permission to generate the required visualization code should be granted. For this purpose the knowledge parameter `experimental_visit_enable` should be set to `true`. Also, in order to prevent some generated functions to be inlined, the `opt_maxInliningSize` parameter must be set to zero.

5 Example

To demonstrate the visualization capabilities, following three-dimensional Poisson's equation has been chosen:

$$-\Delta u(x,y,z) = f(x,y,z) \quad (x,y,z) \text{ on } \Omega \in \mathbb{R}^3 \tag{8}$$

The domain is defined as $\Omega = [0,1]^3$ and Dirichlet boundary conditions have been set:

$$u(x,y,z) = g(x,y,z) \text{ on } \delta\Omega \tag{9}$$

The right-hand side f and boundary conditions g are defined as:

$$f(x,y,z) = 0, \tag{10}$$

$$g(x,y,z) = x^2 - 0.5y^2 - 0.5z^2 \tag{11}$$

The parallel multigrid solver makes use of a V(3,3)-Cycle and a CG solver on the coarsest level. The field for the solution $u(x,y,z)$ has been declared as shown in listing 12. Additionally, global variables representing the simulation's current state are defined.

```
Globals {
  Var sim_done : Boolean = false
  Var sim_time : Real = 0.0
  Var sim_cycle : Int = 0

  Var initRes : Real = 0.0
  Var curRes : Real = 0.0
  Var prevRes : Real = 0.0
}

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

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

Field Solution< global, NodeWithComm, 0.0 >@(all but finest)
Field Solution< global, NodeWithComm, vf_boundaryPosition_x ** 2 - 0.5 *
            vf_boundaryPosition_y ** 2 - 0.5 * vf_boundaryPosition_z ** 2 >@finest
```

Listing 12: Domain, field and field layout declaration of $u(x,y,z)$

The implementation of the `simulate_timestep` function shown in listing 13 is instructed to call as many V(3,3)-Cycles until the current residual is a millionth of the initial residual. In case that this condition is met, the `sim_done` is true and the simulation terminates. The data partitioning can be retrieved from listing 14.

```
Function simulate_timestep@finest {
  if(sim_cycle < 100 &&
            curRes > 1.0E-6 * initRes) {
    sim_cycle += 1
    mgCycle ( )

    communicate Solution
    loop over Residual {
      Residual = RHS - Laplace * Solution
    }
    apply bc to Residual

    prevRes = curRes
    curRes = ResNorm ( )

  } else {
    sim_done = true
  }
}
```

Listing 13: simulate_timestep function

```
dimensionality                  = 3

minLevel                        = 2
maxLevel                        = 6

mpi_enabled                     = true
mpi_numThreads                  = 8

domain_onlyRectangular          = true
domain_rect_generate            = true
domain_rect_numBlocks_x         = 2
domain_rect_numBlocks_y         = 2
domain_rect_numBlocks_z         = 2
domain_rect_numFragsPerBlock_x = 1
domain_rect_numFragsPerBlock_y = 1
domain_rect_numFragsPerBlock_z = 2
```

Listing 14: Knowledge configuration

The domain configuration is illustrated on figure 14. Also, the corresponding mesh for the coarsest level is shown. Here, it can be seen that the step size in z-direction is half of the step size in the other directions, since two fragments in z-direction have been set.

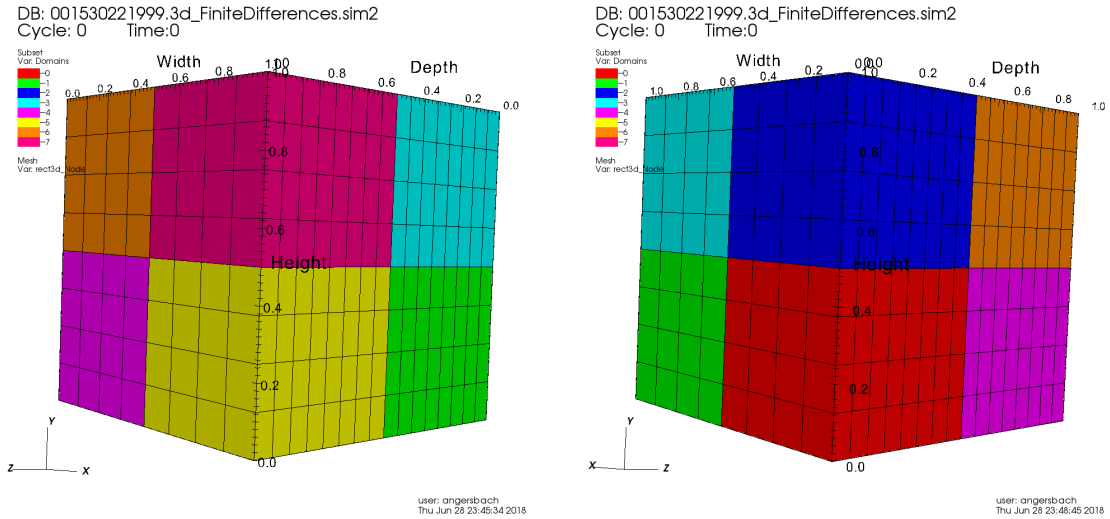


Figure 14: Domain decomposition and rectilinear mesh on coarsest level

The corresponding boundary conditions are pictured in figure 15. However, the values within the cube are more interesting since the values for the boundary conditions are already known. Therefore slice plots through the z=0.5 plane are shown in figure 16 after one, two and ten cycles. In addition to that, the corresponding value of the residual is also shown.

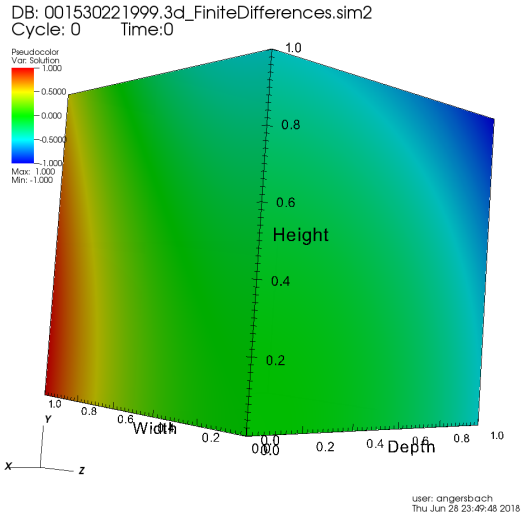
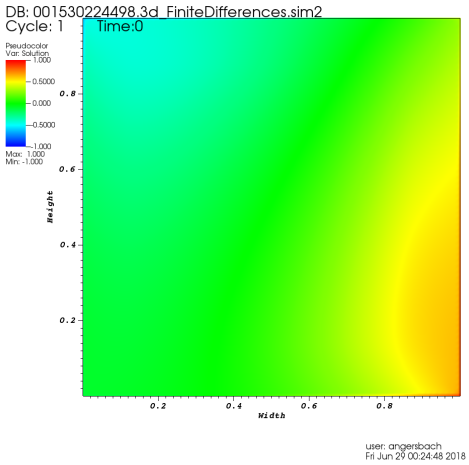
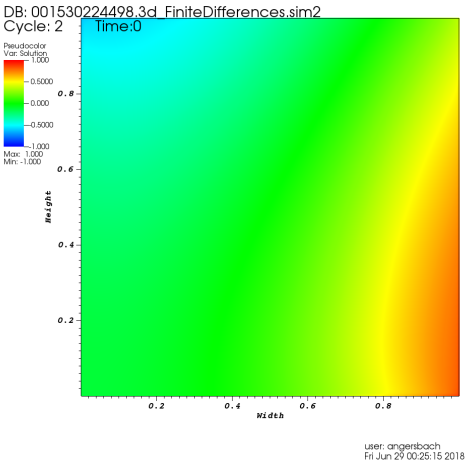


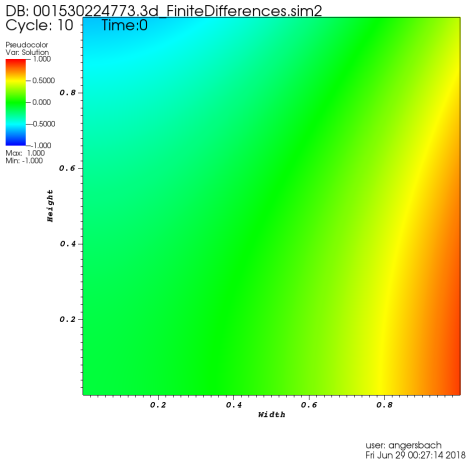
Figure 15: Boundary conditions



(a) After 1 cycle: Residual=912071



(b) After 2 cycles: Residual=230815



(c) After 10 cycles: Residual=4.76353

Figure 16: Slice plot through z=0.5 plane after a certain amount of V-cycles.

6 Conclusion

Summary

The goals that have been set up in the course of this thesis have been achieved. The main goal was the integration of a visualization interface for parallel multigrid solvers with usage of the visualization tool VisIt into the code generation framework ExaStencils. To be more precise, an in-situ visualization approach where simulation and visualization routines operate on the same resources has been chosen. This approach enables techniques for interactive visualization, allowing the user to explore the simulation results during runtime. Naturally, the integration into a code generation frame requires the support of various parameter configurations where the domain to be visualized differs for each of them. In fact, any configuration of the physical data partitioning such as multiple blocks or fragments in each dimension is supported. In addition to that, combinations of different logical groups such as ghost/padding layers and duplicate layers are also possible. Another important goal of this thesis is to provide a visualization for the whole integrity of data, i.e. every field declared in the DSL code. In general, two different graphical representations for the field data exist: variables are mapped onto either nodes or zones of a rectilinear mesh or pictured as a curvilinear mesh consisting of the variable values and their corresponding coordinates. The last achievement is related to user experience. Here, only a handful of changes to the DSL code are required to generate the desired visualization interface.

Future work

The implementation for the visualization interface developed in this thesis gives foundations for following features and optimizations:

1. Within this thesis, the computational domain is always considered as rectilinear or at least composed of rectilinear sub-domains. However, ExaStencils is capable of generating multigrid codes for block-structured grids, which allow more complex domain layouts. The support for those is yet to be integrated into the code generator.

2. An additional visualization interface for advanced users making use of the batch processing supported by VisIt. The data access functions and a portion of the initialization steps provided by the existing implementation can be used for the additional interface. However, this attempt does not make use of the same mainloop where Libsim listens for connections and visualization requests from VisIt. Since the plot requests in batch processing are predetermined, the user would either have to specify it directly in the target code or through a simplified interface in the DSL code.

3. One possible optimization for the current implementation is to introduce so-called ghost nodes or cells. These are used for parallel simulations where multiple domains are rendered. In comparison to a single domain, the external faces of each sub-domain are rendered despite the fact that these belong to the internal of the whole domain and are only visible when the plot is transparent [5, 25].

4. Another possible addition to the interface is the support for non-uniform grids. Until now, the meshes are created with the step size retrieved from the knowledge parameters, which are constant per multigrid level and dimension. However, the nodes or cells position for non-uniform grids and their corresponding spacing model can be retrieved from data structures of so-called virtual fields.

5. In the current implementation, variables are illustrated as scalar values which are either mapped onto the mesh's nodes or zones. But it is possible to visualize variables consisting of multiple components such as vectors, tensors, arrays, etc. One possible realization of this feature would be the combination of multiple fields. The user would specify which fields should be combined, e.g. velocity in u and in v direction as a two-dimensional vector.

A VisIt Installation Guide

This section provides a guide for the installation of the visualization toolkit VisIt and describes which environment variables must be added in order to get a successful linking of the target code with the in-situ visualization library Libsim.

Installation

The installation of VisIt can either be carried out by downloading a pre-built binary or building it from scratch. In general, it is recommended to use the pre-built binaries since the building process takes approx. three hours with a fast build. Nonetheless, it is useful when the addition of self-made plug-ins is desired or a parallel execution for the compute engine is desired. An overview of the possible options for the build process can be found here.

Pre-built executable

Windows and Mac users should follow these steps:

- Download a suitable executable
- Follow the instructions in the install notes

The installation for Unix systems such as Ubuntu can be simplified with following script for the newest version:

```
#!/bin/bash
# Get binary and install script, change name of tar for other Unix distributions
wget http://portal.nersc.gov/project/visit/releases/2.13.2/visit2_13_2.linux-x86_64-ubuntu14.tar.gz
wget http://portal.nersc.gov/project/visit/releases/2.13.2/visit-install2_13_2
# Execute install script, change argument of install script for other distributions
chmod 755 visit-install2_13_2
./visit-install2_13_2 2.13.2 linux-x86_64-ubuntu14 /usr/local/visit
# Add VisIt's binary to user's search path
cd
echo "set path = ($path /usr/local/visit/bin)" >> .cshrc
```

Once the installation is complete, the following environment variables must be set in order to get the correct path of the header files and libraries of Libsim:

- VISIT_HOME: Top level directory where VisIt is installed.
  Example: `export VISIT_HOME='/usr/local/visit'`.
- SIMV2DIR: Path to the V2 directory of Libsim.
  Example: `export SIMV2DIR='/usr/local/visit/2.13.2/linux-x86_64/libsim/V2'`.

As soon as these environment variables have been set, target code for the visualization can be generated, compiled and executed.

References

[1] "A Scala Prototype to Generate Multigrid Solver Implementations for Different Problems and Target Multi-core Platforms". In: *Int. J. Comput. Sci. Eng.* 14.2 (Jan. 2017), pp. 150–163. ISSN: 1742-7185. DOI: 10.1504/IJCSE.2017.082879. URL: https://doi.org/10.1504/IJCSE.2017.082879.

[2] *Advanced Stencil-Code Engineering (ExaStencils)*. [accessed June 18, 2018]. URL: http://www.exastencils.org/.

[3] E. Wes Bethel and Mark Miller. "Remote and Distributed Visualization Architectures". In: *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Ed. by E. Wes Bethel, Hank Childs, and Charles Hansen. Chapman & Hall, CRC Computational Science. http://www.crcpress.com/product/isbn/9781439875728, LBNL-6325E. Boca Raton, FL, USA: CRC Press/Francis–Taylor Group, Nov. 2012, pp. 25–48. ISBN: 9781439875728.

[4] E.W. Bethel, H. Childs, and C. Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. Chapman & Hall/CRC Computational Science. CRC Press, 2012. Chap. 2, pp. 9–23. ISBN: 9781439875735. URL: https://books.google.de/books?id=0zPOBQAAQBAJ.

[5] Hank Childs et al. "A Contract-Based System for Large Data Visualization". In: *Proceedings of IEEE Visualization 2005*. Minneapolis, Minnesota, 2005, pp. 190–198.

[6] Hank Childs et al. *In situ processing*. Tech. rep. Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2012.

[7] Hank Childs et al. "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data". In: *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Oct. 2012, pp. 357–372.

[8] *Creating a C++ Client for VisIt*. [accessed June 14, 2018]. URL: https://www.visitusers.org/index.php?title=CreatingNewClientWithC%2B%2B.

[9] Jean M. Favre. *Grid data visualization and VisIt*. [accessed June 12, 2018]. 2016. URL: https://comp-phys-2016.sciencesconf.org/conference/comp-phys-2016/pages/Grid_data_visualization_with_Visit_1.pdf.

[10] Harald Köstler. "A multigrid framework for variational approaches in medical image processing and computer vision". In: 2008.

[11] James Kress. "In Situ Visualization Techniques for High Performance Computing". In: 2017.

[12] Sebastian Kuckuk and Harald Köstler. "Automatic Generation of Massively Parallel Codes from ExaSlang". In: *Computation* 4.3 (2016). ISSN: 2079-3197. DOI: 10.3390/computation4030027. URL: http://www.mdpi.com/2079-3197/4/3/27.

[13] Sebastian Kuckuk et al. "Towards generating efficient flow solvers with the ExaStencils approach". In: 29 (May 2017).

[14] *Libsim: Batch processing*. [accessed Juli 2, 2018]. URL: https://www.visitusers.org/index.php?title=Libsim_Batch.

[15] *Libsim: Control interface documentation*. [accessed June 10, 2018]. URL: https://fossies.org/dox/visit2.13.2/VisItControlInterface__V2_8h_source.html.

[16] VisIt Python Interface Manual. *Getting Data Into VisIt*. [accessed June 13, 2018]. URL: http://visit.ilight.com/svn/visit/trunk/releases/2.10.0/VisItPythonManual.pdf.

[17] K. Moreland. "A Survey of Visualization Pipelines". In: *IEEE Transactions on Visualization and Computer Graphics* 19.3 (Mar. 2013), pp. 367–378. ISSN: 1077-2626. DOI: 10.1109/TVCG.2012.133.

[18] Kenneth Moreland. "The Tensions of In Situ Visualization". In: *IEEE Comput. Graph. Appl.* 36.2 (Mar. 2016), pp. 5–9. ISSN: 0272-1716. DOI: 10.1109/MCG.2016.35. URL: http://dx.doi.org/10.1109/MCG.2016.35.

[19] Carl Mueller. "The Sort-first Rendering Architecture for High-performance Graphics". In: *Proceedings of the 1995 Symposium on Interactive 3D Graphics*. I3D '95. Monterey, California, USA: ACM, 1995, 75–ff. ISBN: 0-89791-736-7. DOI: 10.1145/199404.199417. URL: http://doi.acm.org/10.1145/199404.199417.

[20] Christoph Pflaum. *Lecture notes in scientific computing II*. [accessed June 30, 2018]. 2017. URL: https://www10.informatik.uni-erlangen.de/media/filer_public/ae/ae/aeae0eea-9cc8-4256-851c-14e3d694dd2e/siwirii_script.pdf.

[21] Christian Schmitt et al. "Systems of Partial Differential Equations in ExaSlang". In: *Software for Exascale Computing - SPPEXA 2013-2015*. Ed. by Hans-Joachim Bungartz, Philipp Neumann, and Wolfgang E. Nagel. Springer International Publishing, 2016, pp. 47–67. ISBN: 978-3-319-40528-5.

[22] C. Schmitt et al. "ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers". In: *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. Nov. 2014, pp. 42–51. DOI: 10.1109/WOLFHPC.2014.11.

[23] *The Visualization Tool Kit*. [accessed June 18, 2018]. URL: https://www.vtk.org/.

[24] *VisIt Java Client API*. [accessed June 14, 2018]. URL: http://www.visitusers.org/visit/2.3.0/javadocs/index.html.

[25] Brad Whitlock. *Getting Data Into VisIt*. [accessed June 10, 2018]. 2010. URL: https://wci.llnl.gov/codes/visit/2.0.0/GettingDataIntoVisIt2.0.0.pdf.

[26] Brad Whitlock. *VisIt User's Manual*. [accessed June 13, 2018]. 2005. URL: https://wci.llnl.gov/codes/visit/1.5/VisItUsersManual1.5.pdf.

[27] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. "Parallel in Situ Coupling of Simulation with a Fully Featured Visualization System". In: *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*. EGPGV '11. Llandudno, UK: Eurographics Association, 2011, pp. 101–109. ISBN: 978-3-905674-32-3.

List of Listings

List of Figures