HiFlow^3 – Technical Report on Release 2.0

Gawlok, S., Gerstner, P., Haupt, S., Heuveline, V., Kratzke, J., Lösel, P., Mang, K., Schmidtobreick, M., Schoch, N., Schween, N., Schwegler, J., Song, C., Wlotzka, M.

Preprint No. 2017-06

$\frac{\partial u}{\partial t} - \nu\Delta u + (u\cdot\nabla)u + \frac{1}{\rho}\nabla p$
$\frac{\partial u}{\partial t}$

Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)
ISSN 2191–0693
Preprint No. 2017-06

Affiliation of the Authors

Gawlok, S.[a], Gerstner, P.[a], Haupt, S.[a], Heuveline, V.[a], Kratzke, J.[a], Lösel, P.[a], Mang, K.[a], Schmidtobreick, M.[a], Schoch, N.[a], Schween, N.[a], Schwegler, J.[a], Song, C.[a], Wlotzka, M.[a]

[a]*Engineering Mathematics and Computing Lab (EMCL), Interdisciplinary Center for Scientific Computing (IWR), Heidelberg University, Germany*

Impressum

Heidelberg University
Interdisciplinary Center for Scientific Computing (IWR)
Engineering Mathematics and Computing Lab (EMCL)

Im Neuenheimer Feld 205,
69120 Heidelberg
Germany

HiFlow^3 – Technical Report on Release 2.0

Gawlok, S., Gerstner, P., Haupt, S., Heuveline, V., Kratzke, J., Lösel, P., Mang, K., Schmidtobreick, M., Schoch, N., Schween, N., Schwegler, J., Song, C., Wlotzka, M.

November 22, 2017

Abstract

HiFlow^3 Version 2.0 continues the path as a multi-purpose finite element software, which provides powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). New features and functionalities, which allow to run numerical simulations with more advanced solution algorithms and discretizations in comparison to previous releases of HiFlow^3, have been implemented. These comprise Uncertainty Quantification (UQ), energy-efficient multigrid techniques, Schur complement solvers for saddle-point problems, extended third-party library support, and adaptive local mesh refinement in a parallel computing environment. Furthermore, HiFlow^3 has been successfully integrated into advanced and state-of-the-art simulation environments by means of the Medical Simulation Markup Language (MSML), for example.

The presented new algorithms and features as well as general under-the-hood improvements have enabled excellent and relevant research activities in the fields of both medical engineering and meteorology and environmental sciences. The described show cases demonstrate the potential and advantages, which HiFlow^3 can offer in performing a numerical simulation by means of finite element methods. Especially, the high performance computing capabilities of HiFlow^3 – not only in the mentioned fields of applications, but also in general – have been significantly improved in Version 2.0.

1 Introduction

Complementing theory and experiment, computer simulation can be postulated as the "third pillar" of scientific research. In seeking the answers of today's research challenges, modeling and simulation provides for many applications valuable insights for the understanding of the underlying processes and their impact. Translating the resulting formulas and equations into computer simulations enables to quantify the quantities of interest. With that, numerical simulations can be used as an extension, a preparation or even instead of real experiments. The strongly growing amount of available data can be used in addition to calibrate computer simulations.

Partial differential equations are usually used in order to model physical processes. Typical examples are the Poisson equation, the Navier-Stokes equations (e.g. simulation of a cyclone-cyclone interaction or blood flow simulation in the aorta) as well as the elasticity equations (e.g., for soft tissue simulation). The weak formulations of the underlying equations can be discretized by means of finite elements [28, 35]. This results in a linear or nonlinear system of equations which is typically solved by Newton-type methods and associated iterative solvers (e.g. GMRES). For instationary problems, time-stepping schemes need to be considered. Applications in medical engineering, meteorology and environmental sciences usually require a high accuracy resulting in huge sparse matrices with, in general, bad condition numbers. Therefore, appropriate preconditioning techniques are needed for solving the discretized systems.

For dealing with uncertain parameters arising in mathematical models, uncertainty quantification is needed. Hereby, classical methods like the Karhunen-Lòeve expansion and the polynomial chaos expansion can be used.

HiFlow^3 is a multi-purpose finite element software providing powerful tools for efficient and accurate solving of a wide range of problems modeled by partial differential equations. Currently, HiFlow^3 is mainly used in numerical scenarios occurring in the areas of medical engineering, meteorology, environmental physics and energy systems. In the field of *medical engineering*, patient-individual functional modeling and numerical simulation can provide (virtual) insight into the operated tissues and their behavior and functionality. Finite Element Method (FEM) based elasticity simulations may reveal the behavior of soft tissues subjected to external forces and momentums, e.g., during or after surgical manipulation of the mitral valve. Computational fluid dynamics or fluid structure interaction simulations can depict the flow behavior of, e.g., blood in the aorta, and hence allow, e.g., for risk analysis in aortic aneurysms. In the field of *meteorology*, numerical simulation with different physical models – i.e., different sets of underlying PDEs – allows to investigate the effect of specific physical phenomena on the numerical solution in a unified and mathematically sound framework. Based on these information, in a concretely given application scenario the best model can be chosen in terms of accuracy, physical realism or compute time.

Based on the discussed aspects, the general focus of HiFlow^3 is as follows: HiFlow^3 is designed as a library with high modularity, which allows to address a broad range of application scenarios, where each scenario can be treated with a variety of methods in order to determine and choose the solution methodology, that is best-suited for the considered problem. Best-suited is not only understood in the sense of approximation quality, but also in the sense of the metric time-to-solution. The latter implies, that each component of the software needs to stand out in good performance as well as parallel scaling behavior. The importance of scalability arises from the fact, that many challenging problems of current research in the considered fields of application require huge amounts of compute power.

In order to cope with the tasks and problems stated above, the HiFlow^3 framework offers many features. A first design feature is a *generic modular structure*, which reflects a typical simulation cycle based on finite elements. This modular structure ensures the flexibility and maintainability of the software and is presented in-depth in Section 2. Furthermore, the gained flexibility is a key-prerequisite in order to adapt HiFlow^3 to different mathematical and algorithmic needs as well as to different hardware and computer architectures. The importance of the latter can not be underestimated, because high performance compute clusters very often rely on highly heterogeneous hardware nowadays [127]. This observation motivates a second feature of the HiFlow^3 framework, namely the ability to map *hardware-aware computing* [11, 12]. By this term we understand the ability of the software to provide and utilize specialized implementations of – especially performance critical – functions, which are adapted to the underlying hardware of the current compute system in order to unleash as much of the available compute power as possible. In the current state of the software, this is extensively done in the linear algebra module of HiFlow^3 and the realization is described in detail in the Subsections 2.3 and 2.5, respectively. A third feature is the awareness of the HiFlow^3 framework to the topic of *energy consumption* and *energy efficiency*, which plays an important role in high performance computing [126]. In the context of HiFlow^3, this is on the one hand tightly connected to the hardware-awareness and on the other hand currently respected in the design and implementation of a specific algorithm, e.g. in the context of a geometric multigrid method. The details about the implementation of this algorithm in HiFlow^3 are described in Subsection 2.3.3. Last but not least, the integration of generalized Polynomial Chaos Expansion (gPCE) and specific solvers for stochastic methods allow to easily include techniques of uncertainty quantification (UQ).

Beside these more technical aspects, HiFlow^3 is regarded as a platform for the implementation of state-of-the-art methods in order to perform research in the field of scientific computing, where the tackled problems are modeled by partial differential equations. Therefore, the new HiFlow^3 version 2.0 release contains many new algorithms and possibilities to solve the resulting discrete problems. One focus of the new release is on linear solvers, where the capabilities of HiFlow^3 are heavily extended by a whole bunch of newly implemented algorithms and methods. Within the HiFlow^3 code base, the aforementioned geometric algorithm as well as a Schur complement linear solver and preconditioner (cf. Subsection 2.3.2) have been implemented. Furthermore, HiFlow^3 version 2.0 features new interfaces to several third-party libraries – *PETSc* [17–19] and *hypre* [33, 42], for example –, which are commonly regarded as well-established libraries in their field of applications and methods (cf. Subsection 2.5). Another focus of the new release is on the topic of adaptivity in a parallel setup. With the aid of the *p4est* library [30, 63], HiFlow^3 is now capable to perform adaptive mesh refinement in two and three space dimensions in a fully parallel manner, cf. Subsections 2.1 and 2.5. A third core aspect of the new release are the features and functionalities in the field of *uncertainty quantification*, cf. Subsection 2.4.

It has to be mentioned, that there exists a numerous number of other non-commercial and commercial parallel finite element software packages besides HiFlow^3. In the context of this work we are not able to give an exhaustive overview on the state-of-the art of finite element software, but we would like to mention in alphabetical order at least a few of them: Abaqus 2017 - Unified FEA [105] is a commercial FEM software which is sold by Dassault Systèmes Simula. It comprises three software packages: Abaqus/CAE, Abaqus/Standard and Abaqus/Explicit. Abaqus/CAE is applied for modeling, analyses of mechanical components and visualizing the finite element analysis. The FEM analysis itself is either done by Abaqus/Standard (a traditional implicit integration scheme) or Abaqus/Explicit (explicit integration scheme to solve non-linear systems). The ADINA System [3] is a commercial FEM software package, which is sold by the ADINA R & D Inc. It is used to solve structural, fluid, heat transfer, and electromagnetic problems and is capable to deal with multiphysics problems including fluid-structure interactions. The latest version is 9.3.3. and it was released in September 2017. ANSYS Fluent [10] is a commercial Computational Fluid Dynamics (CFD) software package provided from Ansys Inc. It is one of the most mature software on the CFD market. Alberta 3.0 [95], last updated in March 2014, is an adaptive hierarchical finite element toolbox. Only simplices are used for the triangulation. Refinement and coarsening is restricted to bisection. The main focus lies on adaptivity which is achieved by different mesh modification algorithms and a data structure which stores the mesh in a binary tree. In March 2017 the company Autodesk stopped selling the software Autodesk Simulation Mechanical [15], though, it is still a widely used finite element analysis software. Since then the finite element parts of the software distributed by Autodesk are included in Autodesk Nastran In-CAD and Fusion 360 [16]. COMSOL Multiphysics [58] is a commercial simulation software environment which is actively developed. It's capabilities range from defining the geometry, meshing, specifying the physics and solving, up to the problem's visualization. For good usability many common problem types are predefined as templates. The C++ programming library deal.II [13, 20] (Version 8.5.0) is an Open Source project and uses adaptive finite elements for solving partial differential equations. It's last major release was in April 2017. DUNE [23] is a modular toolbox for solving PDEs with grid based methods. It is not restricted to Finite Elements but also discretizes with Finite Volumes and Finite Differences. The main principles are abstract interfaces, generic programming techniques and reuse of existing finite element packages.The current version is 2.5.1. It was released in July 2017. FEAST [24] is a software package designed to solve finite element problems. For better floating point performance in terms of speed it can use different co-processing platforms like graphic cards (GPU) or Cell BE, for example. It is based on a specific structure grid which leads to sparse banded matrices. It seems as if there is no further active development: the webpage's last update was in 2013. The FEniCS Project [91] combines different free and open-source software components to provide an accessible environment for automated solution of differential equations. Its FIAT (Finite element Automatic Tabulator) [69] component is a Python module that allows for the generation of arbitrary order finite element basis functions on simplices and, thus, it is the FEM backend of FEniCs. The latest stable release of FEniCS is version 2017.1.0, which was released on May 12 2017. NASTRAN [82] is a program for general Finite Element Analysis which was originally developed in the late 1960s for the NASA. The whole software package is quite large and powerful but due to its age it is written in FORTRAN. OpenFOAM [53] is an open source CFD software, which primarily uses finite volume methods, but also includes a small amount of finite element analysis. The latest release is OpenFOAM v1706, which was made available to the public in June 2017. In most cases based on the finite element discretization, useful packages for solving the obtained linear system are PETSc [18] and Trilinos [56] which are highly scalable linear algebra libraries providing data structures and routines for large scale scientific applications modeled by partial differential equations. Their mechanisms are optimized for parallel application codes, such as parallel matrix and vector assembly routines and linear solving routines, and allows the user to have detailed control over the solution process.

1.1 License

With the *European Union Public Licence (EUPL) v1.2* [41], the release 2.0 of HiFlow^3 comes under a new open-source software license. The EUPL grants free usage and distribution of derivative works under a copyleft clause. Previously, HiFlow^3 was licensed with the GNU Lesser General Public License (LGPL). As opposed to the LGPL, the EUPL is legally valid in 23 official languages and is conform with the copyright laws of each of the member states of the European Union. For distribution with other frameworks, the EUPL is compatible to a number of similar, widely used open-source software licenses.

2 Features and Functionalities of the Open-Source FEM Software Toolkit HiFlow^3

HiFlow^3 features a modular structure, which reflects a typical simulation cycle based on the finite element method [11, 12]. As the focus of HiFlow^3 is on offering efficient and dedicated methods for solving challenging problems in an interdisciplinary context, the underlying geometries are assumed to be generated by a give mesh generator. Therefore, it is assumed that the mesh data is given by a pre-processed external file, e.g., VTK [102] file format. Consequently, the steps, that typically need to be taken for a simulation inside HiFlow^3, are the following:

1. The mesh has to be read from a given mesh file. If a higher spatial accuracy is required, the given mesh can be further refined. And in the context of locally adaptive finite element methods, this can be done iteratively by several runs of the whole simulation cycle, where the mesh is adapted in each run due to measures, which are evaluated for the solution of the previous run. Furthermore, in order to exploit distributed memory parallelism, the mesh needs to be distributed among several parallel processes. The classes and methods, which are needed to fulfill these demands, are implemented in the *mesh module* and are described in Subsection 2.1.
2. Based on the geometry, which is defined and managed by the mesh module, the finite element trial (or shape) functions are defined by means of the *finite element method (FEM) module.* The shape functions are uniquely determined by appropriately chosen functionals, which are called *degrees of freedom (DoF)* in the context of FEM. The pair of DoF/FEM defines the trial space for an unknown function in a PDE and these pairs are implemented in the *DoF/FEM module.* For vector-valued problems, this pair can be defined *per variable.* The management of the DoF/FEM for all unknown functions in a (possibly system of) PDE is done by the `VectorSpace` class of the *space module.* In Subsection 2.2 the functionalities of the DoF/FEM module are described.
3. Inserting the shape functions into the weak formulation of PDE yields a finite-dimensional (possibly non-) linear system of equations, where each entry in the solution vector is associated with a DoF of the finite element trial space. These finite dimensional systems of equations are represented by means of finite-dimensional vectors and, in the case of linear systems, matrices. The implementation of (parallel) data structures for these objects is accomplished in the *linear algebra module.* Based on the matrix/vector representations, the discrete version of the underlying PDE has to be solved by means of (non-) linear solvers. These algorithms are implemented in the *solver module.* The linear algebra and solver modules are described in Subsection 2.3.
4. The discrete solutions are represented by vectors, where the entries are associated with the DoF of the finite element discretization. In order to inspect a solution visually, a *visualization* of the solution of the PDE is implemented. HiFlow^3 does not offer a visualization functionality directly, but is capable of providing tools for writing appropriate output files in the well-established VTK data format [102]. The input/output (I/O) capabilities of HiFlow^3 are described in the context of the extensibility by means of third-party software packages in Subsection 2.5.
5. A specialty of the HiFlow^3 framework are data structures and algorithms for performing *intrusive uncertainty quantification (UQ)* on PDEs with stochastic parameters. These methods require specific implementations of vectors and matrices – and, consequently, solution algorithms – based on the combined discretizations of the physical and stochastic spaces. The resulting discrete problems become usually very large, such that the full capabilities of HiFlow^3 in terms of scalability and performance need to be utilized in order to solve such problems in a feasible amount of time. The specific data structures and algorithms for UQ problems are implemented in the *UQ module*, which is described in Subsection 2.4

2.1 Mesh

In many applications, the solution of a PDE exhibits a structure of varying spatial scales in different parts of the physical domain. Adaptive mesh refinement is a common way of obtaining a finite element space which is capable of resolving many different scales with a minimal amount of degrees of freedom. To this end, the new release of HiFlow^3 provides a mesh module which realizes a distributed, locally refined mesh.

Similar to the mesh implementation `mesh_dbview` introduced in the very first version of HiFlow^3, the new module decouples the main mesh routines like refining, coarsening and ghost cell computation, from the underlying data structures. These main routines are implemented in the object `mesh_pXest` which contains a pointer to the database object `mesh_database_pXest`. It is important to note, that there exists only one database object at any time, but several instances of `mesh_pXest`. The initial one of these instances is built after reading in the coarse mesh from a file (e.g. in .inp format). All further instances are created by modifying an existing mesh instance and do always point to the same database object.

`mesh_database_pXest` can be roughly divided into two parts: The object `mesh_database` has already been part of previous HiFlow^3 versions [11] and is responsible for storing and managing geometrical data such as physical coordinates of mesh vertices, see also the section below. The second part of `mesh_database_pXest` takes care of topological information between cells in potentially different meshes which evolve from the same initial coarse mesh by adaption routines. This part makes heavy use of the external software package *p4est* [30] which is based on a forest of quadtrees and octrees in case of 2D and 3D, respectively. `mesh_pXest` is limited to meshes consisting of either quadrilaterals or hexahedra, since *p4est* only supports these cell types. *p4est* provides routines for creating, storing and manipulating tree based data structures that are distributed over several parallel processes and was primarily developed for local mesh refinement.

Figure 1 gives a schematic overview of the reworked mesh module.

2.1.1 Data Structures

We now give a brief overview of the involved data structures which are stored in `mesh_database_pXest`. These structures are distributed over several parallel processes according to the principle of domain decomposition with an overlap due to ghost cell layers. In the following, we use the notion quadtree and quadrant for both the 2D and 3D case.

Topological Data Structures

The main part of topological information is stored in the object `p4est_forest` in form of a forest of quadtrees. The root node of each tree refers to a cell in the initial mesh (in the following called coarse mesh). All further nodes (in the following called quadrants) are created by successive refinement of individual quadrants. In 2D, each parent quadrant has 4 child quadrants, whereas in 3D, each parent quadrant has 8 child quadrants. Each `p4est_forest` consists of several `p4est_tree` objects which itself contain an array of `p4est_quadrant` objects. There is always exactly one instance of `p4est_forest` whose respective structure represents the most recent instance of `mesh_pXest`. In particular, `p4est_forest` stores only those quadrants which do not have child quadrants (i.e. leaf quadrants) and it is exactly these quadrants which are assigned to the cells in the current physical mesh.

Concerning parallelization, the set of all tree objects is distributed over all involved parallel processes according to an initial partition of the coarse mesh. Each `p4est_quadrant` stores a small amount of integer data for describing its position within the complete forest:

- tree index $t \in \{0, \ldots, N_t\}$
- refinement level $l \in \{0, \ldots, N_l\}$
- Morton index $m \in \{0, \ldots, 2^{N_l}\}$

Here, the Morton index basically describes how to reach a specific quadrant starting from the root quadrant of the corresponding tree. In addition, *p4est* offers the possibility to store a pointer to arbitrary user data in each leaf quadrant object. Below, we describe how we make use of this option.

Besides `p4est_forest`, there are two more crucial *p4est* objects to note: `p4est_ghost` is used for keeping track of ghost cells and for exchanging data between neighboring processes. Below, this object will be described in more detail. Furthermore, the object `p4est_connectivity` represents the coarse

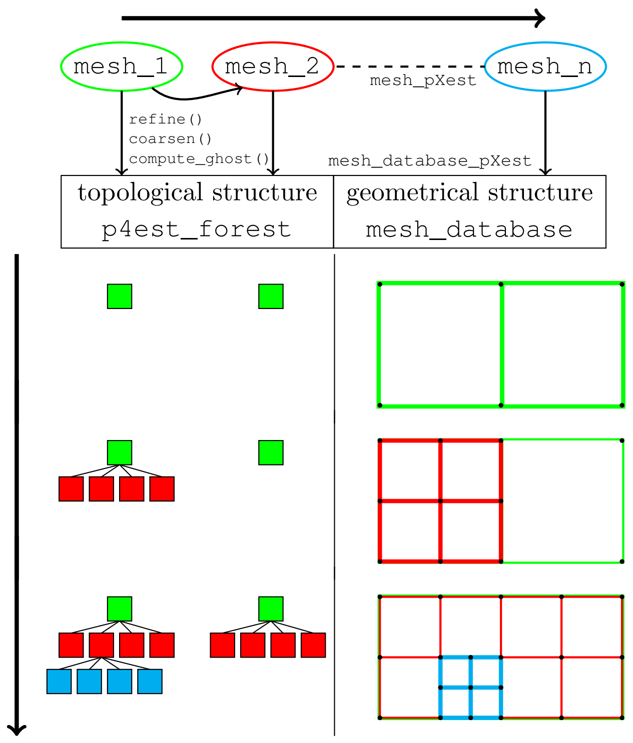


Figure 1: Overview of mesh concept. Top: several instances of the class `mesh_pXest` which are created successively by adaption routines and do have access to the same `mesh_database_pXest` object. Bottom left: schematic view on a `p4est_forest` object consisting of two trees whose roots refer to the cells of the initial coarse mesh. The leaf quadrants (without children) represent the cells being present in the respective physical mesh. Bottom right: schematic view on the corresponding physical mesh after two steps of local refinement.

tdim	2D	3D
0	point	point
1	edge, (facet)	edge
2	cell	face, (facet)
3		cell

Table 1: Definition of entities.

mesh and has to be set up before an initial p4est_forest can be created. In contrast to p4est_forest and p4est_ghost, this object is not distributed across processes. Instead, each process owns an identical copy.

Geometrical Data Structures
Since mesh_database has already been part of previous versions, we only give a very brief summary about its functionality. More details can be found in [11]. As noted before, the main task of mesh_database consists in storing data describing the geometrical properties of the mesh. To this end, it stores an array containing the physical coordinates of all mesh vertices which lie in the local subdomain assigned to the respective process. Moreover, for all entities of each topological dimension *tdim* (see also Table 1), the indices of those vertices, forming the corresponding entity, are stored. Indexing of all entities of different topological dimensions in mesh_database is given in the form

$$j(\text{tdim}) \in \mathcal{J}(\text{tdim}) =: \{0, \dots, N_{\text{tdim}}\}. \tag{1}$$

These indices are defined locally and are the same for all mesh_pXest instances sharing the same database.

Mapping between Topological and Geometrical Data Structures
In order to establish a connection between topological and geometrical information, we use the data structures quad_data and quad_coord.

An instance of quad_data is stored as user data in each leaf p4est_quadrant, resulting in a mapping from forest quadrant to physical cell

$$\begin{aligned} \mathcal{J}_{tg} \colon \{0, \dots, N_t\} \times \{0, \dots, N_l\} \times \{0, \dots, 2^{N_l}\} &\to \mathcal{J}(\text{cell}), \\ (t, l, m) &\mapsto j(\text{cell}). \end{aligned} \tag{2}$$

Among other data, quad_data stores the cell index $j(\text{cell})$ of that physical cell which is represented by the corresponding p4est_quadrant. In addition, quad_data stores the indices of all cells which are represented by all ancestor quadrants of p4est_quadrant.

In order to define a mapping in reverse direction, i.e.

$$\mathcal{J}_{gt} = \mathcal{J}_{tg}^{-1}, \tag{3}$$

for each cell $j(\text{cell})$ in mesh_database there exists an instance of type quad_coord. In particular, this structure contains the coordinates (t, l, m) of that p4est_quadrant that represents cell $j(\text{cell})$. Now, inter-cell relationships can be obtained by combining the previously defined data structures and mappings. E.g., computing the parent cell for some given child cell c is done by algorithm 1.

Mesh Class
The object mesh_pXest serves as an interface to access the data stored in mesh_database_pXest. In this way, it provides all functionality which is required by other modules in HiFlow^3. An instance m of this object is defined by a subset $\mathcal{K}(m) \subset \mathcal{J}(\text{cell})$ of all physical cells stored in the underlying database. This set contains locally owned cells, as well as ghost cells, i.e.

$$\mathcal{K}(m) = \mathcal{K}_l(m) \,\dot{\cup}\, \mathcal{K}_g(m). \tag{4}$$

Algorithm 1 Parent cell computation

```
Let child cell with index $c$ be given.
    **Get** corresponding p4est_quadrant $q_c = \mathcal{J}_{gt}(c)$.
    **Search** leaf quadrants in p4est_forest for a descendant quadrant $q_l$ of $q_c$. If such a
    quadrant is not found, search p4est_ghost. Searching can be done efficiently by using
    internal routines of *p4est*.
    **Retrieve** parent cell index $p_c$ from quad_data stored in $q_l$.
```

Data Partitioning
To sum up, each process stores information on all cells in its local subdomain and on those cells, which are located in the associated ghost layer with a user defined width, see below. Moreover, all cells that are ancestors of any ghost cell are stored and the coarse mesh is present in each process in form of `p4est_connectivity`.

2.1.2 Main Routines

In this subsection, we give an overview of the main mesh routines that are usually needed in every FEM application and are meant to be called by the user.

Creation and Initial Partitioning
The process of creating an initial instance of `mesh_pXest` is implemented outside of this object. In the first step, the master process builds the coarse mesh based on an input file in .inp format. To this end, a preliminary, sequential `mesh_database` object is created and filled with those vertex coordinates and entities which are defined in the input file. Afterwards, this data is used to create the `p4est_connectivity` object representing the coarse mesh. Optionally, the coarse mesh can be refined uniformly, e.g. if the number of parallel processes is higher than the number of coarsest cells.

In order to obtain an initial partition, any graph partitioner can be applied to the coarse mesh. In particular, HiFlow^3 provides an interface to METIS (see section 2.5.1) for accomplishing this task. To propagate this partition to the subsequently created `p4est_forest`, the cell and vertex indices in `p4est_connectivity` are permuted accordingly. After broadcasting this object and the initial partitioning to all involved processes, the initial `p4est_forest` object can be built by invoking the corresponding routines provided by *p4est*. This and all further steps are executed in parallel on all processes. Afterwards, the master process distributes the data stored in its `mesh_database` object to all other processes according to the previously computed partitioning. Now, every process is able to create its own, local `mesh_database_pXest` and `mesh_pXest` objects which, at this point, do only contain locally owned cells. The initialization routine is finalized by setting up the `quad_data` and `quad_coord` structures for all root quadrants and coarse mesh cells, respectively. Figure 2 visualizes the complete procedure.

Mesh Adaption
Adaption of an existing `mesh_pXest` is done by applying the routine `refine(marker)` for an existing `mesh_pXest` object. In the course of this routine, a new instance of `mesh_pXest` is created and a pointer to this object is given back as return value. Here, `marker` denotes an integer array whose size is equal to the number of cells in the local subdomain. The respective values of this array denote different options on how to adapt a specific cell, see also table 2. Here, a family $\mathcal{F}$ of cells (i.e. cells with the same parent cell) is coarsened if and only if the following conditions hold

$$\texttt{marker}[c] \leq 0 \text{ for all } c \in \mathcal{F},$$
$$\frac{1}{|\mathcal{F}|}\sum_{c\in\mathcal{F}} \texttt{marker}[c] \leq -1.$$

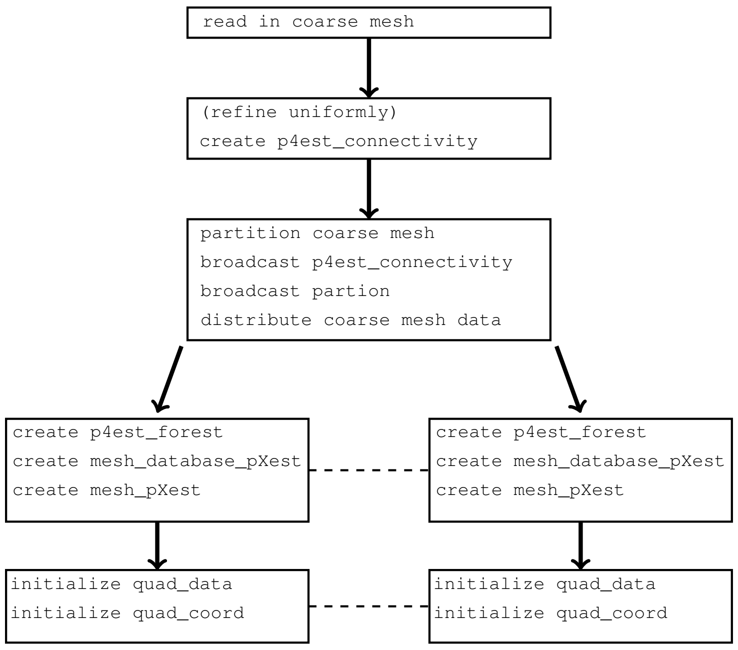


Figure 2: Procedure for reading in, partition and distribution of coarse mesh and building local mesh objects.

`marker`	Action
0	keep cell
1	refine cell
< 0	potentially coarsen cell

Table 2: Definition of adaption markers.

This allows the user to decide how many cells per family should be marked for coarsening to actually coarsen the corresponding cells. Remember that the mesh includes locally owned cells as well as ghost cells. Therefore, the user has to provide adaption markers for ghost cells as well. However, ghost cells always enter the newly created mesh without being refined or coarsened. In this way, the user does not have to check whether a cell is locally owned or not when defining adaption markers in the application. Keeping track of the cells that should be contained in the adapted mesh is realized by an array `new_cells` of cell indices $j(\text{cell}) \in \mathcal{J}(\text{cell})$. In the first step, all cells marked for refinement are refined by executing algorithm 2. Afterward, the coarsening step is realized by algorithm 3.

After these two adaption steps, it might be possible that neighboring cells differ in more than one level of refinement. Such a case is problematic for the DOF/FEM module which requires that at most one hanging node is present on each facet. In order to resolve this issue, *p4est* offers a balancing routine which successively refines additional quadrants until neighboring quadrants differ in at most one level of refinement. Here, several options of neighboring are possible and it is up to the user to select one of these, see Table 3. All options result in having at most one hanging node per facet. A high `balance_mode` leads to more cells in the resulting mesh and a smoother transition of cell diameters. Balancing is accomplished by means of algorithm 4.

For specific applications it might be desirable to have a mesh which is uniformly coarsenable, e.g. when using a patch interpolation as higher order approximation in the Dual Weighted Residual (DWR) method. In this case, the user has the option to enable the so called `patch_mode`. Then, non-coarsenable families of cells are identified by another call to `p4est_coarsen_ext()` (without actually coarsen any

Algorithm 2 Refine mesh

Let those cells be given that should be refined. The integer array `new_cells` is empty.

```
    **Compute** coordinates and indices $j(\text{cell}) \in \mathcal{J}(\text{cell})$ of corresponding child cells, done by mesh_database.
    **Pass** indices of refined cells and child cells are passed to p4est_forest.
    **Call** p4est_refine_ext() to refine p4est_forest accordingly. In this step, quad_data structures are created for child quadrants and are initialized by using quad_data from their respective parent quadrant. In addition, the previously defined cell index $j(\text{cell})$ is added to this structure.
    **Add** child cell indices to new_cells.
```

Algorithm 3 Coarsen mesh

Let adaption markers be given for each cell which was not refined in algorithm 2. Let the integer array `new_cells` be given by algorithm 2.

```
    **Pass** coarsening markers to p4est_forest.
    **Call** p4est_coarsen_ext() to coarsen p4est_forest accordingly. In this step, the quad_data structure of the parent quadrant (now a leaf) is initialized by using quad_data of one of the later on removed child quadrants. The cell index assigned to the parent cell is stored in p4est_forest.
    **Add** cell indices stored in p4est_forest to new_cells.
    **Add** indices of all remaining cells with marker $= 0$ to new_cells.
    **Create** intermediate mesh object $m$ of type mesh_pXest with $\mathcal{K}(m) =$ new_cells.
```

`balance_mode`	Two cells are neighbors if they share a common
0	edge(2D), face (3D)
1	vertex (2D), edge (3D)
2	vertex (3D)

Table 3: Definition of `balance_mode`.

Algorithm 4 Balance mesh

Let `balance_mode` be defined and the integer array `new_cells` be given by algorithm 3.

```
    **Call** p4est_balance_ext() to balance p4est_forest. In this step, new quadrants are created whose quad_data structures are initialized by the corresponding parent data. However, at this stage, no cell index is available yet. Instead, all refined quadrants and their child quadrants are stored in p4est_forest.
    **Refine** the respective cells $\mathcal{K}_p$ in mesh_database based on the previously stored quadrants.
    **Put** the newly defined cell indices $\mathcal{K}_c$ into the unfinished quad_data structures.
    **Create** a new mesh_pXest object $m + 1$ with
        $$\mathcal{K}(m + 1) = (\mathcal{K}(m) \setminus \mathcal{K}_p) \cup \mathcal{K}_c. \tag{5}$$
```

Algorithm 5 Compute ghost cells

```
Let a distributed mesh_pXest object be given and a ghost layer width be defined.
    Call p4est_ghost_new() and p4est_ghost_expand() to create a p4est_ghost object
    of desired layer width.
    Retrieve indices $\mathcal{K}_m(m)$ of all mirror cells from the corresponding quad_data structures by
    iterating through all mirror p4est_quadrant objects stored in p4est_ghost.
    Extract geometrical data and quad_data structures for all mirror cells from
    mesh_database and store them in p4est_ghost.
    Call p4est_ghost_exchange_custom() to exchange mirror cell data between processes.
    Retrieve exchanged data from p4est_ghost.
    Create ghost cells from exchanged geometrical data and put them into mesh_database.
    This step yields indices $\mathcal{K}_g(m+1)$ for the current ghost cells.
    Initialize ghost p4est_quadrant objects in p4est_ghost by using exchanged quad_data
    structures and indices $\mathcal{K}_g(m+1)$.
    Create a new mesh_pXest object $m+1$ with $\mathcal{K}(m+1) = \mathcal{K}_l(m) \dot{\cup} \mathcal{K}_g(m+1)$.
```

cells). Afterward, `refine(patch_marker)` is called for the balanced mesh created by algorithm 4 with

$$\texttt{patch\_marker}[c] = \begin{cases} 1, & c \text{ is member of a non-coarsenable family,} \\ 0, & \text{else.} \end{cases} \tag{6}$$

Note that one iteration of this process is sufficient, if `balance_mode` is set to 1 (2D) or 2 (3D).

Ghost Cell Computation

Computation of ghost cells for each local subdomain can be done after a distributed `mesh_pXest` object is created. In particular, it is possible to apply several calls to `refine()` before the layer of ghost cells is updated. Similar to the `refine()` routine, a new instance of `mesh_pXest` is created every time `compute_ghost()` is called and a pointer to the corresponding object is returned. Moreover, the user can specify up to which distance from locally owned cells, measured in number of neighboring relationships, cells are considered to be ghost. The default value is one. Here, two cells are treated as neighbors if they share a common vertex.

In order to recognize hanging nodes on facets that are adjacent to ghost cells, it is necessary that each process contains all cells in its local `mesh_database` that are ancestors of any ghost cell in the current mesh. Therefore, cells might be exchanged although they are not part of the actual ghost layer. Figure 3 gives a schematic overview of this issue.

The actual ghost cell computation is based on the object `p4est_ghost`. Among other data, this object contains an array of mirror `p4est_quadrant` objects ($\rightarrow$ local cells that are ghost cells for another process) and an array of ghost `p4est_quadrant` objects ($\rightarrow$ foreign cells that are adjacent to locally owned cells). In order to compute a new `mesh_pXest` object $m+1$ from object m, algorithm 5 is performed.

2.2 DoF/FEM

The new release of HiFlow[3] offers an increased variety of different finite element spaces. In addition to continuous Lagrange finite elements that allow different element types ($\mathbb{P}$ or $\mathbb{Q}$) and different polynomial degrees to be employed, discontinuous Lagrange elements and pyramid elements $\mathbb{G}$ are available as well.

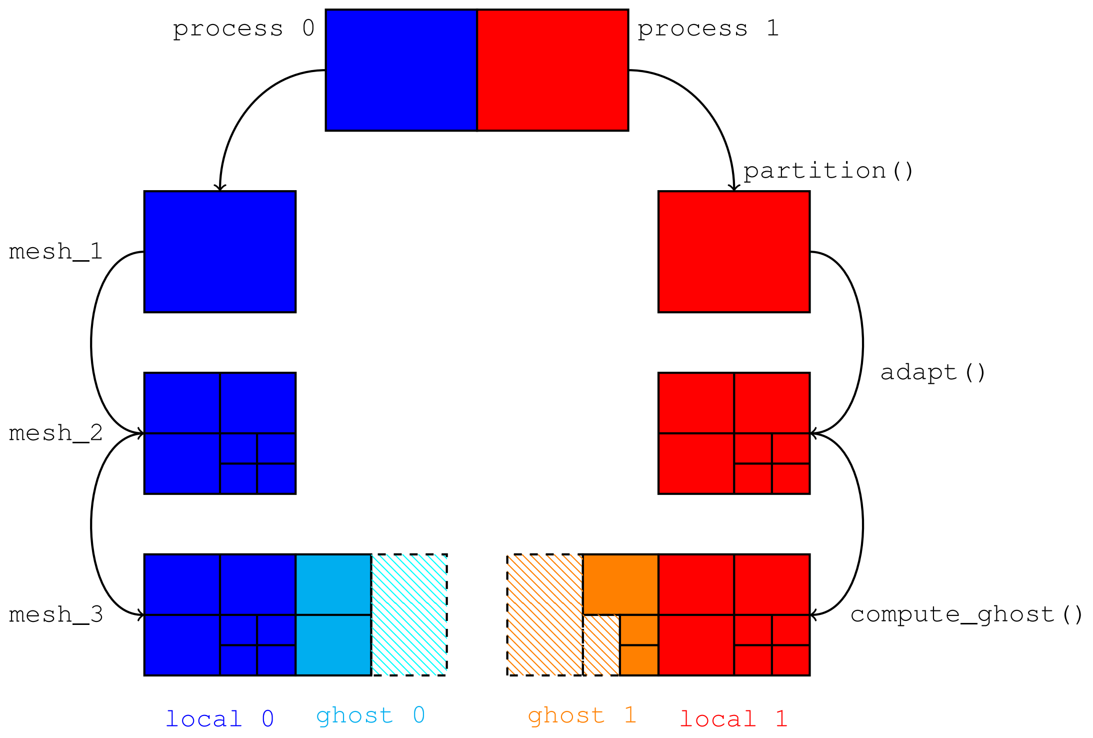


Figure 3: Schematic view of ghost cell computation for a locally refined mesh that is distributed over two processes. Cells surrounded by dashed lines denote parent cells of ghost cells (bright color, surrounded by solid lines) and are stored in `mesh_database`, but not part of current `mesh_pXest` object.

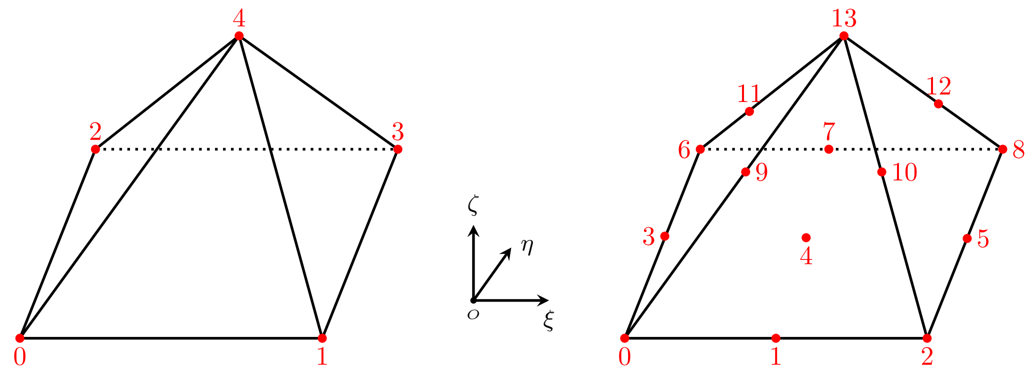


Figure 4: location and numbering of degrees of freedom for 1st and 2nd order pyramid element. 1st order case DOF numbering overlap also the numbering of vertices.

2.2.1 Pyramid Element

Besides hexahedral and tetrahedral elements, we also implemented a pyramid element in HiFlow^3 for the 3D case. Although the two former elements can already deal with most of complex geometries, the pyramid element can play a role as "glue element" ($\mathbb{G}$) to connect tetrahedral and hexahedral grids.

Nevertheless, it has been proven that it is impossible to construct conforming continuously differentiable basis functions on a pyramid element [118]. Therefore, the shape functions are built by using composite elements, it means that we decompose the pyramid into 4 tetrahedrons and the basis functions are constructed separately.

Figure 4 indicates first the geometry of a pyramid element, which consist of 4 triangles and 1 Quadrilateral. In HiFlow^3, only the 1^{st} and 2^{nd} order elements are implemented. The numbering of the degree of freedoms (DOFs) are also shown in Figure 4, the Gaussian quadrature points and corresponding weights are provided in [32, 75].

2.2.2 Discontinuous Galerkin

Discontinuous Lagrange finite elements of the form read:

$$V_h^{dG} = \{v \in L^2(\Omega),\ v_{|_K} \in \mathbb{L}(K)_{l(K)}\ \forall K \in \mathcal{T}_h\}, \tag{7}$$

which are available for a given triangulation $\mathcal{T}_h$ of the physical domain Ω. When using the mesh module `mesh_dbview`, it is possible to mix different types of cells within a single mesh. In this case, different cell-wise elements $\mathbb{L}(K) \in \{\mathbb{P}, \mathbb{Q}, \mathbb{G}\}$ are used. When using `mesh_pXest`, only $\mathbb{L}(K) = \mathbb{Q}$ is a valid option. In both cases, different polynomial degrees $l(K) \in \mathbb{N}_0$ in different cells are possible.

When applying a discontinuous Galerkin formulation for a given PDE, one usually has to evaluate integrals of the form:

$$\begin{aligned} a_e(u_h, v_h) &:= \int_e f(u_h^M, u_h^S, v_h^M, v_h^S, n_e)\,\mathrm{d}\sigma, \\ b_e(v_h) &:= \int_e g(v_h^M, v_h^S, n_e)\,\mathrm{d}\sigma, \end{aligned} \tag{8}$$

for an interface $e = \overline{M(e)} \cap \overline{S(e)}$ between two cells $M(e), S(e) \in \mathcal{T}_h$ (master and slave), unit normal $\boldsymbol{n}_e$, discontinuous functions $u_h, v_h \in V_h^{dG}$ and some scalar functions f, g. e might also denote a boundary facet, implying $M(e) = S(e)$. Here, we set for $x \in e$:

$$\begin{aligned} v_h^M(x) &:= \lim_{h \downarrow 0} v_h(x - h\,\boldsymbol{n}_e(x)), \\ v_h^S(x) &:= \lim_{h \downarrow 0} v_h(x + h\,\boldsymbol{n}_e(x)). \end{aligned} \tag{9}$$

The `DGGlobalAssembler` object was developed to compute the additional contributions from DG method to the corresponding system matrix by means of Algorithm 6. Contributions to the global system

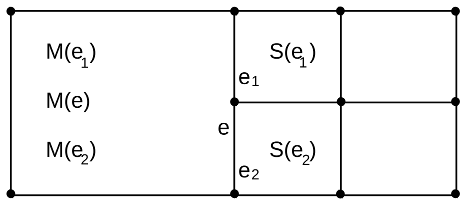


Figure 5: Interface with hanging node shared between one master and two slave cells.

Algorithm 6 Interface integral (contribution to global system matrix A)

Let $\{\phi_l^K, l \in \mathcal{I}(K), K \in \mathcal{T}_h\}$ be a basis of V_h^{dG} with an index set $\mathcal{I}(K)$ of local basis functions with support in K and $i(l,K)$ denoting the global dof index of basis function ϕ_l^K.

```
**for all** inner interfaces $e$
    **for all** $K \in \{M(e), S(e)\}$, **for all** $K' \in \{M(e), S(e)\}$
        **for all** $l \in \mathcal{I}(K)$, **for all** $j \in \mathcal{I}(K')$
            Call local LocalDGAssembler to compute $I := a_e(\phi_l^K, \phi_j^{K'})$.
            Add to global system matrix: $A_{i(j,K'),i(l,K)}$ += $I$.
**for all** boundary interfaces $e$
    **for** $K = M(e)$, **for** $K' = M(e)$
        **for all** $l \in \mathcal{I}(K)$, **for all** $j \in \mathcal{I}(K')$
            Call local LocalDGAssembler to compute $I := a_e(\phi_l^K, \phi_j^{K'})$.
            Add to global system matrix: $A_{i(j,K'),i(l,K)}$ += $I$.
```

vector are computed in an analogue way, see Algorithm 7. Moreover, the `DGGlobalAssembler` is capable of dealing with interfaces that involve hanging nodes, see Figure 5. In this case, the portions e_1 and e_2 of a common parent interface e do both occur in the outer loop of Algorithm 6 and 7 instead of e.

2.3 Linear Algebra and Solvers

HiFlow^3 Version 2.0 still features the same two-level linear algebra toolbox as described in [11, 12]. The robustness, scalability properties and capabilities of this toolbox have been shown in several application areas, cf. [57, 96, 100], for example.

Based on these capabilities, further abstractions, structures and algorithms have been developed and implemented, which are described in detail in the following.

2.3.1 Abstract Matrix and Vector Interfaces

In previous versions of HiFlow^3, the linear algebra toolbox and its capabilities and functionalities were tied closely to the `CoupledMatrix` and `CoupledVector` classes for the representation of parallelly distributed vectors. While `CoupledMatrix` and `CoupledVector` – in conjunction with the implementations of local matrices and vectors on the intra-node level – provide excellent performance and scalability in terms of the Basic Linear Algebra Subroutines (BLAS), the ability to interact with third-party libraries necessitates a more general interface.

To be more specific, third-party linear algebra libraries, e.g. hypre [33, 42] or PETSc [17–19], which themselves employ parallel concepts and data structures and implement customized algorithms based on their own data structures. These data structures provide an own Application Programming Interface (API), which not necessarily coincides with the one, that is defined and used by HiFlow^3. In order to facilitate the interoperability in terms of matrix and vector assembly or a mixture of our own HiFlow^3

Algorithm 7 Interface integral (contribution to global system vector b)

Let $\{\phi_l^K, l \in \mathcal{I}(K), K \in \mathcal{T}_h\}$ be a basis of V_h^{dG} with an index set $\mathcal{I}(K)$ of local basis functions with support in K and $i(l,K)$ denoting the global dof index of basis function ϕ_l^K.

```
    for all inner interfaces $e$
        for all $K' \in \{M(e), S(e)\}$
            for all $j \in \mathcal{I}(K')$
                Call local LocalDGAssembler to compute $I := b_e(\phi_j^{K'})$.
                Add to global system vector: $b_{i(j,K')}$ += $I$.
    for all boundary interfaces $e$
        for $K' = M(e)$
            for all $j \in \mathcal{I}(K')$
                Call local LocalDGAssembler to compute $I := b_e(\phi_j^{K'})$.
                Add to global system vector: $b_{i(j,K')}$ += $I$.
```

algorithm with those provided by third-party software packages, abstract base classes for (parallel) matrices and vectors as well as linear solvers and preconditioners have been defined which are used throughout the whole HiFlow^3 library.

These abstract base classes – namely `Matrix`, `Vector`, `Preconditioner`, `LinearSolver` and `Preconditioner`, respectively – allow an easy implementation of wrapper classes to the data structures and algorithms of external software packages by means of inheritance from the corresponding abstract base classes. The wrapper class agitates as translator between the HiFlow^3 API and the respective API calls of the wrapped third-party data structure or algorithm. Especially, the explicit and costly conversion between our own HiFlow^3 data structures and those of third-party packages is avoided, because the third-party data structures are created and assembled directly. Therefore, the difference in runtime for creating data structures – especially, parallelly distributed matrices and vectors – is almost the same for using the HiFlow^3 and third-party data structures.

Examples, where this concept is employed, can be found in the wrappers to the hypre library, see the `HypreMatrix`, `HypreVector`, `HypreBiCGSTAB`, `HypreBoomerAMG`, `HypreCG`, `HypreGMRES`, `HypreLinearSolver` and `HyprePreconditioner` classes, for example.

2.3.2 Schur Complement Preconditioner and Solver

Schur complement preconditioners and solvers are a well-established choice to solve the saddle-point problems, that arise in the solution of the incompressible Navier-Stokes equations or in multi-physics problems, see [27, 64, 103], for example.

First, we describe the overall idea of the Schur complement preconditioner. Let a linear system $\mathcal{A}\xi = \mathbf{b}$, $\mathcal{A} \in \mathbb{R}^{N\times N}$, in block matrix form

$$\mathcal{A}\xi = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right) \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix} \tag{10}$$

be given and assume, that A is regular. By performing a block Gaussian elimination on (10), this linear system is equivalent to the following two equations:

$$\left(D - CA^{-1}B\right)\mathbf{y} = \mathbf{g} - CA^{-1}\mathbf{f}, \tag{11}$$

$$\mathbf{x} = A^{-1}\mathbf{f} - A^{-1}B\mathbf{y}. \tag{12}$$

The matrix $\Sigma := D - CA^{-1}B \in \mathbb{R}^{N_1\times N_1}$, $0 \le N_1 \le N$, is called the *Schur complement* of A in the block matrix $\mathcal{A}$ and (11) is called the *Schur complement equation* for y. The strategy to solve equations (11) and (12) is given in Algorithm 8.

Algorithm 8 Solution strategy for Schur complement equation

Let an initial solution $\xi_0 \in \mathbb{R}^N$, a right hand side vector $(f, g)^\top \in \mathbb{R}^N$, a system matrix $\mathcal{A} \in \mathbb{R}^{N\times N}$, a relative tolerance $\varepsilon_{rel} > 0$, an absolute tolerance $\varepsilon_{abs} > 0$, a maximum iteration number $I_{max} \in \mathbb{N}$ and preconditioning matrices $M_j^{-1} \in \mathbb{R}^{N_1\times N_1}$, $j \in \mathbb{N}$ for the Schur complement matrix Σ be given.

1. Solve Schur complement equation (11) for y by the Flexible Generalized Minimum Residual (FGMRES) method [94] and the given parameters ε_{rel}, ε_{abs}, I_{max} and M_j^{-1}.
2. Compute x via (12).

For the computation of the multiplication of A^{-1} with a vector, any linear solver routine that is available in HiFlow^3 can be used. The implementation of Algorithm 8 is straightforward and only a specialized routine to compute the matrix-vector multiplication with the matrix Σ is needed.

The main difficulty is to compute the partitioning of the *system matrix* $\mathcal{A}$ into the *block matrices* A, B, C and D as well as the partitionings of ξ and $\mathbf{b}$, respectively. The objective is to create *parallelly distributed* matrices A, B, C and D such that their row and column numberings are consecutive in parallel in order to be able to apply any – possibly provided by a third-party software package – linear solver to compute the multiplication with A^{-1}. The procedure to compute the partitionings is described in detail in the following.

To illustrate the concepts, we consider the example of a three-dimensional flow problem with velocity field $\mathbf{v} := (u, v, w)$ and pressure p. Furthermore, let the variable numbers in the `VectorSpace` object be $(u, v, w, p) = (0, 1, 2, 3)$. Then, a classical splitting according to (10) is

$$\mathcal{A} = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right) := \left(\begin{array}{c|c} A_{\mathbf{v},\mathbf{v}} & A_{\mathbf{v},p} \\ \hline A_{p,\mathbf{v}} & A_{p,p} \end{array}\right).$$

The `Init` function of the `SchurComplement` class expects three arguments in order to compute the partitionings: the `VectorSpace` object of the current finite element simulation as well as two `std::vector<int>` objects, which describe the variable numbers of the blocks in (10). In the case of the three-dimensional flow example, this yields the vectors

$$\texttt{block\_one\_variables} = (0, 1, 2),$$
$$\texttt{block\_two\_variables} = (3),$$

i.e., `block_one_variables` is associated with the variables of the test and trial functions of the block matrix A and `block_two_variables` with those of the block matrix D. Consequently, block matrix B consists of the couplings

$$\texttt{block\_one\_variables} \times \texttt{block\_two\_variables}$$

and block matrix C of the couplings

$$\texttt{block\_two\_variables} \times \texttt{block\_one\_variables},$$

respectively. With these notions, the `Init` function takes the following steps:

1. Compute the sparsity structure of the block matrices A, B, C and D in the *DoF numbering of the system matrix* $\mathcal{A}$. This is done precisely in the same way as for the matrix $\mathcal{A}$ during the setup phase of the simulation, but the considered couplings are restricted to those associated with the individual four block matrices. These sparsity structures are stored in `std::map<int, SortedArray<int> >` data structures. Concerning to a by-product, whose key-values of the maps for block A and block D precisely coincide with the DoFs associated with these blocks in the *system* DoF numbering of the system matrix $\mathcal{A}$. Let these sets of DoF indices be denoted by $\{s_{a_i}\}_{i=1}^{N_A}$ for block A and by $\{s_{d_j}\}_{j=1}^{N_D}$ for block D, respectively. Then, it holds
$$\{s_{a_i}\}_{i=1}^{N_A} \cap \{s_{d_j}\}_{j=1}^{N_D} = \emptyset.$$
This step is done on every parallel process, which is independent of all other processes in the *global* DoF numbering of matrix $\mathcal{A}$.

2. Compute the *global* offsets of the sets $\{s_{a_i}\}_{i=1}^{N_A}$ and $\{s_{d_j}\}_{j=1}^{N_D}$. These offsets denote, how many DoFs in these blocks are owned by the individual processes.
3. Based on the results of the two previous steps, consecutive *global* block numberings for the blocks A and D can be computed, i.e., maps
$$\{s_{a_i}\}_{i=1}^{N_A} \leftrightarrow \{1, \dots, N_A\}$$
and
$$\{s_{d_j}\}_{j=1}^{N_D} \leftrightarrow \{1, \dots, N_D\}$$
are created.
4. Determine the index numbers of the ghost DoFs in the individual block numberings as well as maps from the system to the block numbering for these DoFs.
5. Translate the sparsity structures, that have been determined in step 1, from the *system* to the *block* numbering by applying the maps created in the previous steps.
6. Create the block matrices A, B, C and D as well as the block-vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{f}$ and $\mathbf{g}$ based on the *block* sparsity structures and numberings determined above.

Then, the `SetupOperator` function copies the entries from the system matrix $\mathcal{A}$ to the individual blocks based on the mappings and sparsity structures determined during the `Init` routine.

2.3.3 Energy-efficient Geometric Multigrid Solver

With Version 2.0, HiFlow^3 expands its capabilities to the area of energy-aware high performance computing by means of a new energy-efficient geometric multigrid solver and related techniques. HiFlow^3's geometric multigrid framework is available in the new namespace `hiflow:la:gmg` comprising the structures for mesh hierarchy, grid transfer operators, smoothers and coarse grid solvers. In this subsection, we present the new features related to the energy-efficient multigrid solver. However, we first give an overview of the basic terms and concepts of energy-aware HPC since Version 2.0 is the first HiFlow^3 release which expands into that field. Following the overview on energy efficiency, we introduce the new geometric mutligrid framework.
Energy consumption of high performance computing centers has gained significant attantion in recent years due to the increasing amounts of energy necessary to feed the computer hardware and building infrastructure. The recent developments in computer architecture, especially in multi-, many-core and accelerator technology, have triggered considerable performance gains in computing, allowing the continued increase of computational power. To this end, hardware is being rapidly adopted in computing facilities. Nevertheless, further performance improvements attained from a substantial increase in the number of cores, is constrained by the aggregated energy budget necessary for large-scale high performance computing systems. In particular, power consumption has a direct impact on the operation and maintenance costs of these centers, compromising their existence and impairing the installation of new ones. Already today, the electricity costs for many computing centers exceed the hardware acquisition costs in just a few years. Furthermore, energy consumption results in carbon dioxide emissions, a hazard for the environment and public health. Also, heat is produced, which reduces the reliability and lifetime of hardware components. Therefore, the concerns about the rise of an energy crisis, climate change and fault-tolerance in large-scale systems lead to a well justified call for energy efficiency in high performance computing.

Energy metrics, tracing and analysis for scientific applications The traditional performance metric in HPC, floating point operations per second (Flop/s), is used to rank the most powerful supercomputers in the world by means of the Linpack benchmark performance in the bi-annual Top500 list [127]. The most common metric for energy efficiency is the Flop/s metric divided by the power draw, yielding floating point operations per second and per Watt (Flop/sW), or equivalently floating point operations per Joule. This is used by the Green500 list [126], which ranks the Top500 candidates by the Flop/sW metric. However, there is a discussion in the scientific community on other metrics, see for example [26]. Besides the relative quantities of Flop/s and Flop/sW, there are the absolute metrics of total time to solution T and total energy to solution

$$E = \int_0^T P\,\mathrm{d}t \tag{13}$$

for a specific application run, where P is the instantaneous power draw. Clearly, these absolute metrics are connected to the relative Flop/s and Flop/sW metrics by the total number of floating point operations performed. From (13) we see that optimization for performance may imply optimization for energy if, at the same time, the power draw is not increased too much. In addition, energy-aware high performance computing seeks dedicated energy optimizations which reduce the power draw without affecting, or at least not impairing too much, the performance.

Proper tracing and analysis of applications is required in order to understand the performance and energy characteristics, to identify inefficiencies, and to find opportunities for improvement. There exists a rich set of performance tracing and analysis tools, including well-known packages such as the HPCToolkit [2], TAU [104], Scalasca [129], HPCView [80], Vampir [70], Paraver [90], Score-P [71], and more may be found in [83]. In recent years, also for power and energy measurement, tracing and analysis tools and devices have been developed including PowerMon 2 [25], PowerPack [50], Extrae in conjunction with Paraver [6], pmlib [21, 22], and a more extensive overview is given in [116]. Power and performance traces are typically obtained from instrumented application code or from built-in hardware sensors and counters obtained from machine specific registers. However, power information of hardware sensors is often not gained through actual measurements, but rather estimated from performance counters by means of a power model, as it is the case with Intel's Running Average Power Limit (RAPL) [60]. This has driven the attempt to employ dedicated power meters to obtain accurate measurements. Power meters like ZES Zimmer LMG450, Yokogawa WT300 or Tektronics PA5000 can be used as external devices to measure the total power draw between the external power source and the power supply unit of the computer. Individual power lines between the power supply unit and other components inside the computer such as mainboard, CPU, memory, or accelerators can be measured with internal devices like ArduPower [38].

Energy-efficient techniques for multi-core processors In embedded and mobile systems, energy consumption of the hardware has ever been an issue due to limited power supply or thermal restrictions. Performance states and power states, two important developments for improving energy efficiency, have been defined in the Advanced Configuration and Power Interface Specification (ACPI) standard [4], which most current processors adhere to. The standard defines means to configure the processor according to the workload, which can be used to adjust the power consumption of the processor.

Performance states A performance state, or P-state, is defined by a pair of values for the processor operating voltage V and frequency f. Since the dynamic power P of a processor is related to voltage and frequency as $P \sim V^2 f$, lowering operating power and frequency reduces the energy consumption of the processor. The state P0 defines the nominal operating voltage and frequency yielding the nominal performance (not considering "boost modes", overclocking etc.), while the deeper states P1, P2,... define successively reduced voltages and frequencies. The P-states featured by processors are usually vendor- and product-specific, often customized for particular purposes and applications. An example of the P-states for two processors is given in Table 4. Some processors allow to set individual P-states for each core on the socket, while others force all cores to the same P-state. Furthermore, in some processors the memory bandwidth is linked with their frequency. The common technique of changing the P-states during runtime of an application in order to adjust the processor performance to the needs and to save energy is called dynamic voltage and frequency scaling (DVFS). On Linux systems, the P-states can be controlled from user space by means of governors. The `user-space` governor can be used to set a fixed P-state, and the `on-demand` governor employs a heuristic to adjust the P-state depending on the workload. Changing P-states is affected with a performance and energy cost due to the latency of the transition between the states. The usual approach is to use higher P-states, i.e. higher frequencies and voltages, for compute-bound operations to minimize the execution time. In contrast, switching to a deeper P-state may yield energy savings for memory-bound operations if execution time is not increased. However, optimizing for energy-efficiency through DVFS is often a tedious task due to the complex dependencies between hardware properties and application characteristics.

	Intel E5504 (4 cores)		AMD 6128 (8 cores)	
	voltage [V]	frequency [GHz]	voltage [V]	frequency [GHz]
P0	1.04	2.00	1.23	2.00
P1	1.01	1.87	1.17	1.50
P2	0.98	1.73	1.12	1.20
P3	0.95	1.60	1.09	1.00
P4	n/a	n/a	1.06	0.80

Table 4: Performance states (P-states) for the Intel E5504 and the AMD 6128 processor.

Power states The power states, or C-states, define what parts of a processor are switched off during idle time. While C0 defines the active working state, the sleeping C-states C1,C2,... can be adopted when the processor is idle. In the sleeping states, certain components like the floating point unit or caches are switched off, thus saving energy, but the processor cannot perform any computation. Deeper C-states can potentially yield larger energy saving, but the latency for the transition to the active state is longer. Similar to the P-states, also the C-states are vendor- and product-specific. Since the C-states are controlled by the hardware and operating system (OS), programmers can try to create conditions in their program where the OS will promote the processor to a sleeping state when idle.

Shared memory systems and energy-aware runtimes Multi-core processors, including many integrated cores (MIC) devices like the Intel Xeon Phi [62], represent shared memory systems, which are often programmed by means of threads to exploit task parallelism. The threads are assigned to tasks and mapped to processor cores for execution. Threads may depend on each other through shared data objects, thus requiring proper synchronization. Usually, the programmer defines parallel regions and data dependencies in the program source code by means of specific compiler directives. The actual assignment of threads to tasks and the mapping to cores is done by the underlying runtime. One widely used shared memory programming model for Fortran and C/C++ is the Open Multi-Processing (OpenMP) [88] application programming interface (API).

Considerable research has been dedicated to the development of runtimes with features like decomposing high-level operations into tasks, analyzing task dependencies, and optimized out-of-order scheduling of the tasks. Examples include Cilk [59], Harmony [37], Kaapi [48], Mentat [52], StarPU [14], and OmpSs [39]. Runtimes and libraries dedicated to numerical schemes, in particular to linear preconditioners and solvers, include SuperMatrix [31] an ILUPACK [5].

For current multi-core processors, usually larger execution time results in increased energy consumption. Therefore, optimizations towards energy-efficiency mostly imply, or are even equivalent to, performance optimizations. However, researchers seek to apply further energy optimizations without hurting the performance by exploiting the P-states and C-states. A typical approach is to identify idle threads, e.g. waiting for their task dependencies to get fulfilled, and letting them wait in a blocking mode instead of a busy wait, thus preventing frequent polling. The goal of this technique is to fulfill the conditions upon which the operating system would promote cores to an energy-saving deeper C-state. One can also try to leverage the P-states in order to save energy, but this is often far more difficult due to the complexity mentioned above. Clearly, measures to reduce the power draw are only beneficial if they do not increase the execution time too much, so that the positive effect on the energy consumption in not impaired.

Offloading to co-processors Another typical energy-efficient technique is to offload computations to co-processors, also called accelerators, like many integrated core (MIC) devices, graphics processing units (GPU), or field programmable gate arrays (FPGA). The goal of offloading computations to co-processors is to leverage their superior Flop/s and/or Flop/sW metrics for specific tasks. MIC are often built on the same architecture as multi-core processors. Therefore, they are usually programmed using the shared memory thread parallelism as discussed above. In contrast, GPU and FPGA require dedicated programming techniques due to their different architecture. While FPGA are not yet common place in HPC systems and therefore not considered in this work, today many of the most powerful machines are equipped with GPU. The prevalent programming techniques for GPU are the CUDA extension to the C programming language [87], OpenACC directives [111], and OpenCL [112].

The first task when offloading computations is to identify those parts of the algorithm at hand which can run efficiently on the co-processor. Typical candidate code sections are regular workloads comprising mostly integer or floating point arithmetic, as they appear for example in linear algebra operations. To increase performance and to decrease energy consumption through offloading, the ability of an algorithm to exploit the massive parallelism of GPU and MIC is often a necessary prerequisite. Also, time and energy for data transfer between host and device memory, and any overhead for co-processor function and kernel invocation need to be considered. Furthermore, it is essential to avoid unnecessary energy consumption of the host system during co-processor computations. To this end, one option is to overlap host and device computations as far as possible to let both host and device do useful work. Another strategy is to aggregate co-processor kernel invocations. This can either be done by kernel merging where the work of several small kernels is united in one larger kernel, or by invoking kernels directly from the device without host interaction. The benefit of both methods for aggregating co-processor kernels is that invocations take longer time to return to the host system, so that the host can potentially spend longer time in an energy-saving idle state.

Distributed memory systems and clusters Distributed memory means the separation of the available memory in distinct address spaces. This may be inherent to the computer platform, e.g. the individual compute nodes of a cluster each have their own memory address space which is not accessible from other nodes. A distributed memory setup may also be induced by the co-existence of several host processes with individual private address spaces within the same node. This is a fundamentally different situation compared to shared memory platforms, where all host processes or threads can access the same memory using a common address space. Unless the application is “embarrassingly parallel”, which denotes a situation without any dependency between the processes, a technique for making data from one process available to other processes is needed. The widely used standard technique in HPC is explicit data transfer between processes using the message passing interface MPI [81]. Recently, also partitioned global address space (PGAS) approaches such as Unified Parallel C (UPC) [115], Co-array Fortran (CAF) [92] or the Global Address Space Programming Interface (GASPI) [47] have received considerable attention.

Generally, all energy-aware techniques for shared memory systems, including offloading to co-processors, also apply to the node level in distributed memory systems. Furthermore, data transfer over the network between nodes can become a major issue not only with respect to performance, but also with respect to energy consumption. Therefore, the traditional performance optimizations like overlapping computation and communication, seeking lean communication patterns, and avoiding communication as far as possible, are likewise essential for energy efficiency.

Energy-efficient geometric multigrid solver The new HiFlow^3 namespace `hiflow:la:gmg` comprises the geometric multigrid framework introduced with Version 2.0. Table 5 lists the relevant classes. Opportunities for making multigrid solvers energy-efficient can be sought in all building blocks of the method, including the grid hierarchy setup, grid transfer operators, smoothers, and coarse grid solvers [124]. Due to their different roles in the multigrid algorithm, these building block usually need to be addressed by individual measures for performance and energy optimization.

The smoother's purpose is to remove high frequency error contributions. Besides this smoothing property, it does usually not need to yield an accurate solution. This gives space for the choice and optimization of smoothers. Traditional smoother choices include the Jacobi or Gauss-Seidel iteration and their damped variants. For smoothers it might however be acceptable to modify these schemes by removing synchronizations. This leads to asynchronous iterations which can benefit from massively parallel architectures like many-core devices or graphics processing units (GPUs) due to their relaxed synchronization requirements. Asynchronous schemes can easily be adapted to fit the hardware at hand, e.g. by aggregating the components into blocks and mapping them to the cores of a multi-core processor or to the thread blocks of a CUDA GPU. Thus the adaption of the classic, synchronized relaxation scheme allows to efficiently exploit the parallelism of modern hardware, and in particular it offers an opportunity to benefit from the superior flops per Watt characteristics of GPUs. HiFlow^3 offers a (block-)asynchronous iteration solver/smoother dedicated to GPU-accelerated platforms in the class `hiflow::la::AsynchonoursIterationGPU`. This has been proved very efficient as a linear solver in conjunction with a mixed precision iterative

`BasicLevel`	single level in a grid hierarchy holding a mesh, a finite element space, a system matrix, a solution and a right-hand-side vector
`ConnectedLevel`	inherits from `BasicLevel`; holds connections to the next coarser and finer grids
`GMGLevel`	inherits from `ConnectedLevel`; holds an additional vector for the residual computation in multigrid methods and the facilities needed for GPU offloading of matrix and vectors, i.e. the memory transfer operations between host and device
`AbstractHierarchy`	defines a hierarchy of objects of arbitrary type, access functions and interators through the hierarchy
`BasicHierarchy`	inherits from `AbstractHierarchy`; hierarchy of `BasicLevel` objects and initialization of finitel element space, matrix and vectors
`AsymmetricDataTransfer`	transfers vectors between grid levels which may use arbitrary parallel distributions
`DofIdentification`	identifies finite element degrees of freedom across the grid hierarchy
`BasicConnection`	sets up the `DofIdentification` and the `AsymmetricDataTransfer` between two grid levels
`GMGConnection`	inherits from `BasicConnection`; provides functions for solution, right-hand-side and residual transfer between two `GMGLevels`
`LinearInterpolation`	linear interpolation of fine grid degrees of freedom from the neighbouring coarse grid degrees of freedom
`LinearRestriction`	restriction of fine grid degrees of freedom to the neighbouring coarse grid degrees of freedom, adjoint operator to `LinearInterpolation`
`SmootherHierarchy`	inherits from `AbstractHierarchy`; defines smoothers for the fine grid levels in a multigrid hierarchy
`GeometricMultiGrid`	inherits from `LinearSolver`; the geometric multigrid solver using above features

Table 5: Classes of the geometric multigrid framework.

refinement routine in [121]. Furthermore, asynchronous smoothers greatly improved the energy efficiency both on shared memory platforms using OpenMP, and on (multi-)GPU platforms, compared to classic Jacobi smoother [123]. The `GeometricMultiGrid` solver uses a `SmootherHierarchy` to devise appropriate smoothers for all but the coarsest grid level.
In contrast, the residual and the grid transfer usually need to be computed accurately, because otherwise the overall convergence of the multigrid solver cannot be guaranteed. Nevertheless, performing residual and grid transfer computations on co-processors like many-core accelerators or GPUs might still prove beneficial, but care must be taken to ensure consistency in distributed systems [123]. The `LinearInterpolation` and `LinearRestriction` classes offer both cell-based grid transfer operators and a preprocessing facility to assemble global matrix operators. The `GMGLevel` class is enabled to offload the matrices and vectors to GPU for accelerated residual computation and grid transfer operators.
The coarse grid error correction solver is also expected to provide an accurate result. It nonetheless offers space for optimization, both in the choice of the method and its implementation. One usually employs Krylov subspace methods such as CG or GMRES methods, or direct methods like LU or QR decompositions. The coarse grid solver can itself be subject to energy and performance optimization, and the overall multigrid solver would then inherit the benefits. The `GeometricMultiGrid` solver can employ any linear solver from the `hiflow:la` namespace as coarse grid solver.
Another direction for energy and performance optimization, which is particularly relevant for distributed memory platforms, affects the parallel setup of the grid levels in the multigrid hierarchy. The problem sizes in the hierarchy often differ in several orders of magnitude from the largest problem size on the finest level to the smallest problem size on the coarsest level. A simple parallelization where all grid levels are distributed to all available processors may turn out to scale poorly for a large number of processors. This is due to the communication overhead becoming significant and diminishing the efficiency of the computations on coarse levels with small problem sizes. Instead, parallel setups where coarser levels use only a subset of the available processors can be beneficial. Balanced setups can maintain the overall performance of the parallelization while reducing the communication, such that the overall efficiency is conserved. A fraction of the available processors can be temporarily deactivated while the multigrid algorithm operates on coarse levels, and activated again to use the full computing power on finer levels. It is crucial to keep communication patterns local between neighboring sub-domains, both within each grid level as well as between grid levels for the grid transfer. Well-configured parallel setups in the multigrid hierarchy can yield substantial energy savings by deactivating processes and reducing communication while maintaining the overall performance. These techniques are implemented in our multigrid framework in the relevant classes including the `BasicLevel`, `BasicHierarchy`, `DofIdentification` and `AsymmetricDataTransfer`. We impose restrictions on the general parallelization scheme to prevent from inefficient setups. First, a process which is once deactivated on a certain grid level, will stay inactive on all coarser grids. Second, any subdomain on a coarse grid coincides exactly with the union of possibly several subdomains on the next finer grid. This reduces the necessary communication for a prolongation or restriction to a minimum. For applying the prolongation or restriction operator, any process of a fine grid needs to exchange data with only one process of the next coarser grid. This yields independent local communication patterns between fine and coarse grid processes as indicated in Fig. 6 [122]

2.4 Uncertainty Quantification (UQ)

Quantification of uncertainty is highly desired to make statements about reliability and accuracy.

Uncertainty Quantification (UQ) focuses on understanding, quantifying and propagating uncertainty in the computational simulation of models. It connects simulation models with probability theory and deploys a wide variety of tools and methods to quantify these probabilistic models.

Uncertainties arise from different sources and are often grouped in the categories aleatoric and epistemic uncertainties. Aleatoric uncertainties are inherent to the model itself and cannot be reduced. Epistemic uncertainties are due to model assumptions, parameterizations and discretization [106, 108]. One fundamental goal of UQ is to understand and quantify how uncertainty or rather variability of model parameters propagates to uncertainty in resulting model outputs. Monte Carlo methods are standard methods addressing this task, as they are applicable to almost every model, with convergence being independent of the problem dimension. However, the slow convergence rate w.r.t the number of samples is a major drawback, resulting in high computational costs. Consequently, there is a need for alternative and

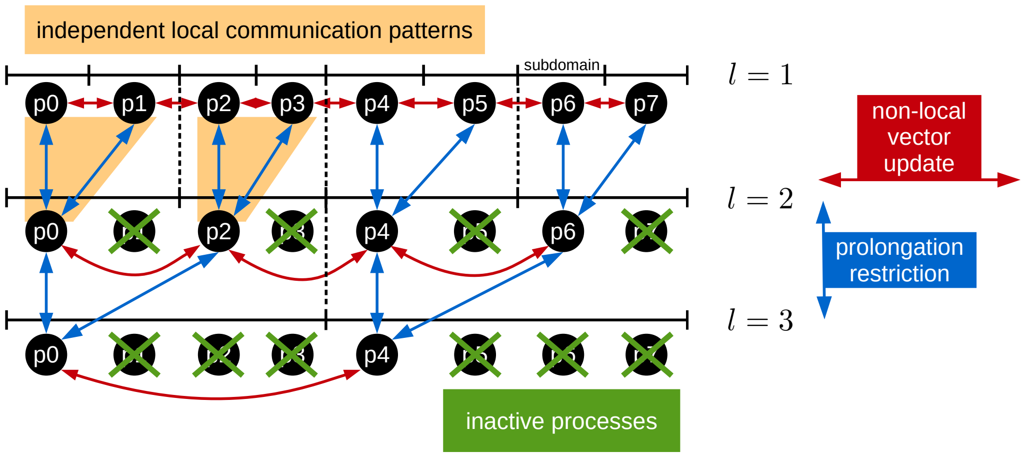


Figure 6: Hierarchy with 3 grids and 8 processes. All processes are active on the fine grid. Every second process is set to sleep on the middle grid, and again every second of the remaining processes is set to sleep on the coarse grid. Non-local vector updates affect only processes of neighboring subdomains. Prolongation and restriction use independent local communication patterns.

especially cheaper methods. Recently, polynomial chaos expansions [40, 117, 128] have gained popularity to build faithful model surrogates, that are cheaper to evaluate than the original model. This leads to the need of methods that compute the expansion coefficients efficiently. These methods are grouped into intrusive [77] and non-intrusive methods [84, 85].

In HiFlow^3, an intrusive Stochastic Galerkin method is implemented and applied to an Blood Pump simulation, see Subsection 4.4. Non-intrusive methods are not implemented in HiFlow^3, as they do not require adaption of existing solvers. However, the use of non-intrusive methods with HiFlow^3 is also possible by utilizing other software like Dakota [1] or ChaosPy [43], as an outer loop. In Subsection 4.2 a non-intrusive method is applied to an Aortic Aneurysm.

As mentioned in Section 1, we focus on the intrusive Galerkin method [77] in HiFlow^3. The intrusive method relies on the Galerkin projection technique, which is based on a weighted residual formulation of the stochastic system equations. The first step of employing the intrusive Galerkin method is to express the random variables by using spectral expansion, e.g., Karhunen-Loève (KL) [65], homogeneous chaos [117] or polynomial chaos (PC) expansion [51]. More specifically, in HiFlow^3, we mainly work with the generalized Polynomial Chaos Expansion (gPCE) [128].

The gPCE attempts to express a random variable by a specific polynomial regarding to its probability distribution. Table 6 shows two kinds of polynomials implemented in HiFlow^3, which correspond to the Gaussian and Uniform distribution, respectively. Therefore, a random quantity U can be written as:

$$U = \sum_{i=0}^{\infty} u_i(\boldsymbol{x})\psi_i(\boldsymbol{\xi}) \approx \sum_{i=0}^{P} u_i(\boldsymbol{x})\psi_i(\boldsymbol{\xi}) \ . \tag{14}$$

Here, $\{u_i\}$ is the set of solution modes only depending on spacial information, $\{\psi_i\}$ is the orthogonal Polynomial Chaos basis and $\boldsymbol{\xi}$ denotes the vector of random variables. The random quantity, more precisely the stochastic solution, can be approximated by a truncation up to a certain order of polynomial degree N_o, result in P PC modes as shown above. The dimension P of the PC basis is obtained by:

$$P := \frac{(M + N_o)!}{M!N_o!} \ , \tag{15}$$

where M is the dimension of random variables, i.e. $dim(\boldsymbol{\xi}) = M$.

Moreover, the stochastic solution and random input can be written in a similar fashion as the random quantity in (14), and are inserted into the governing equations. Afterward, the stochastic equation, which

Distribution	**pdf**	**Polynomials**	**Support**
Uniform	$\frac{1}{2}$	Legendre	$[-1, 1]$
Gaussian	$\frac{e^{-s^2/2}}{\sqrt{2\pi}}$	Hermite	$[-\infty, +\infty]$

Table 6: Chaos Polynomials implemented in HiFlow^3. (pdf: probability density functions)

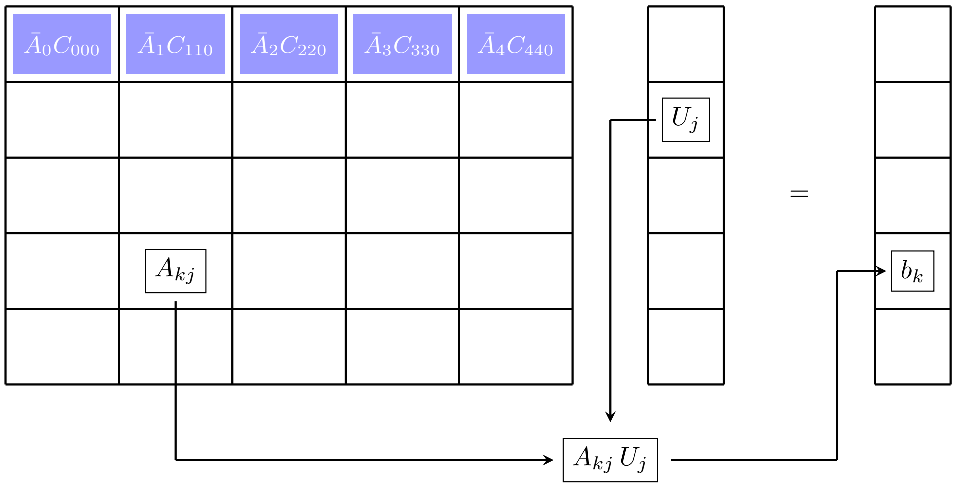


Figure 7: Illustration of the Stochastic Galerkin matrix-vector multiplication.

is obtained from last step, has to be projected onto the same PC space in order to formulate the desired stochastic system. The resulting stochastic system can be written in a general way:

$$AU = b\ , \tag{16}$$

where,

$$A = \begin{bmatrix} A_{00} & \cdots & A_{0P} \\ \vdots & \ddots & \vdots \\ A_{P0} & \cdots & A_{PP} \end{bmatrix}, U = \begin{bmatrix} U_0 \\ \vdots \\ U_P \end{bmatrix}, b = \begin{bmatrix} b_0 \\ \vdots \\ b_P \end{bmatrix}\ . \tag{17}$$

Without going into detail, the stochastic system is expressed in a general form in (17). U_i is the solution mode corresponding to PC basis function ψ_i and has same the dimension as the solution vector of the deterministic problem. A_{ij} denotes the sub-block of the global Galerkin matrix A, describing the coupling between modes i and j. However, storing the global matrix A is very expensive, as a sub-block A_{ij} has the same dimension as the system matrix in the deterministic case. Therefore, in HiFlow^3, only the sub-blocks on the first row in matrix A (Figure 7) are stored and matrix-vector multiplication is achieved by using (18):

$$A_{kj}U_j := \sum_{i=0}^{P} C_{ijk}\bar{A}_i U_j = b_k\ , \quad j,k = 0, ..., P\ . \tag{18}$$

Here, C_{ijk} is a third order Galerkin tensor [1)], which is defined by the projection process. In HiFlow^3, the polynomial chaos basis is defined under `hiflow::polynomialchaos::PCBasis` and the Galerkin tensor is found in `hiflow::polynomialchaos::PCTensor`.

[1)]Note that, we give here only an example for third order Galerkin tensor, higher order tensors can be obtained by using C_{ijk}[77].

2.5 Extensibility: Third-party packages

While it is the credo of the HiFlow^3 developers to provide at least a naive implementation for all core functionalities of a FEM software, doing research necessitates the ability to use state-of-the-art algorithms and implementations. Therefore, HiFlow^3 provides abstract base classes for the performance-critical parts of a simulation-cycle – meshes, graph-partitioners, matrices and vectors, for example – in order to wrap these functionalities from third-party libraries while maintaining the HiFlow^3 interface to these functionalities. As a consequence, much of the application development can be done without any additional software installed.

In the following, those third-party libraries, where HiFlow^3 provides a ready to use interface, are listed per module.

2.5.1 Mesh module

The features and functionalities of the mesh module can be extended in two different setups by external third-party libraries.

The first aspect is the (re-) distribution of meshes in a parallel computing environment. In order to produce high-quality partitionings with a high volume-to-surface ratio of the individual subdomains on the parallel processes, HiFlow^3 is capable to call the specific partitioning routines of the METIS [66] and ParMETIS [67] software packages.

The second aspect is the management of local mesh refinements in a parallel computing environment. To accomplish this task for quadrilateral meshes in 2D and hexahedral meshes in 3D, HiFlow^3 relies on the p4est [30, 63] library.

2.5.2 Linear Algebra and Solvers

The capabilities of the linear algebra and solvers module can be extended by various third-party libraries. These interfaces aim, on the one hand, at providing optimized basic linear algebra subroutines (BLAS) for different compute architectures and, at the other hand, at providing state-of-the-art solution algorithm for linear systems of equations.

To be specific, HiFlow^3 is capable of using features and functionalities from the cuBLAS [86], LAPACK [9], hypre [33, 42], ILU++ [78, 79], Intel Math Kernel Library (MKL) [61], MUltifrontal Massively Parallel sparse direct Solver (MUMPS) [7, 8], PETSc [17–19], SLEPc [54, 55, 93], and UMFPACK [36] libraries.

2.5.3 I/O

Conducting large-scale and massively parallel simulation requires efficient strategies for managing the input and output (I/O) of simulation data. Especially, the runtime of a single job on cluster systems is typically restricted to several hours which necessitates the ability to reload the last state of a simulation run. In order to write and read large data efficiently, HiFlow^3 interfaces the HDF5 [110] library for binary data I/O.

In order to visualize the finite element solution functions and possibly derived/evaluated functions – e.g., L^2 and H^1 error norms – HiFlow^3 is capable to write visualization data in the well-established (parallel) VTK data format [102]. This functionality is implemented in the `CellVisualization` and `ParallelCellVisualization` classes, respectively. As the names of the classes indicate, the visualization output is written in a *cell-wise* manner, which means that the DoF values are written per cell. As a result, DoF values may be written redundantly several times in the context of continuous finite element methods, whenever a DoF is shared by several cells. The big advantage of this methodology is the capability to write true *discontinuous* visualization output for *discontinuous finite element methods*. Furthermore, it allows precise visualizations in the context of *hp*-FEM.

3 Interfaces and Environments

HiFlow^3 can be set in the context of larger approaches by automating and integrating it into an entire simulation workflow taking into account pre- and post-processing steps.

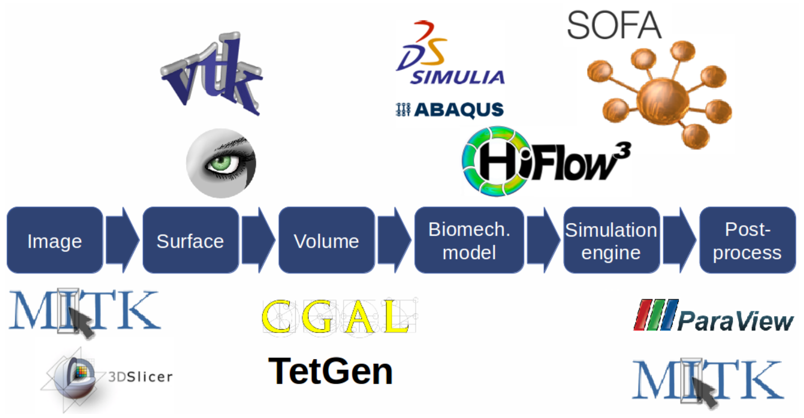


Figure 8: Schematic representation of the complex biomechanical modeling workflow from tomographic images via simulation to post-processing, showing the miscellaneous specific tools and software components used for the different stages.

As an example, a simulation pre- and post-processing scheme for surgery simulations, called Medical Simulation Markup Language (MSML), is presented in order to analyze and process patient-individual medical data. Then, this information is used as input for the fully automated HiFlow^3 simulation setup.

Further, we developed an interface between HiFlow^3-based simulations and machine learning approaches. Hereby, machine learning algorithms are used in order to calibrate parameters which are necessary for the HiFlow^3-based simulations. As an example, soft tissue material parameters are calibrated patient-individually for elasticity simulations.

3.1 Simulation Pre- and Post-Processing and Cognitive Application Pipeline

In the field of medical engineering, mathematical soft tissue modeling and FEM-based elasticity simulation allow to predict the behavior of soft tissue subjected to external forces and momentums, e.g., during or after surgical manipulation. Similarly, computational fluid dynamics (CFD) and fluid structure interaction (FSI) simulations can depict the flow behavior of, e.g., blood in the aorta, and hence allow, for risk analysis, e.g., in aortic aneurysms.

Using the FEM simulation toolkit HiFlow^3 and describing a respective surgery simulation scenario by means of a HiFlow^3 *Extensible Markup Language (XML)* input file, we can run a simulation and obtain results, which can be provided to surgeons in order to assist them during surgery. However, there is a complex biomechanical modeling workflow which precedes the actual simulation (simulation preprocessing), and which is to compose the above-mentioned XML scenario input files along with the respectively needed specifications, like mesh geometries, material parameters, boundary conditions, etc.

This biomechanical modeling workflow usually covers many different steps, see Figure 8. It commonly starts from tomographic data, goes via image segmentation and FE mesh generation, and also includes the definition of boundary conditions, material parameters and other model and simulation specifications. Despite of being a seemingly simple task, the composition of an appropriate biomechanical model as a result of passing through these workflow steps can be very time-consuming in practice, and often requires manual interaction.

With the modeling approach of the Medical Simulation Markup Language (MSML) [109], we describe and integrate the entire workflow, and facilitate the patient-specific construction of biomechanical models for subsequent HiFlow^3-based simulation. Namely, using the MSML and a set of dedicated MVR simulation preprocessing operators, we allow for the comprehensive analysis and processing of patient-individual medical data and for fully automatically setting up HiFlow^3 XML input files, which can directly be executed through the HiFlow^3-based simulation application [101].

For more information, please see [96].

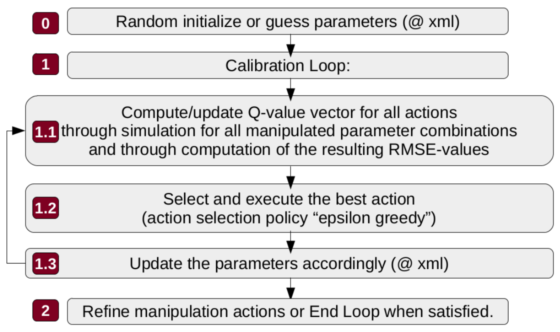

0
Random initialize or guess parameters (@ xml)
1
Calibration Loop:
1.1
Compute/update Q-value vector for all actions through simulation for all manipulated parameter combinations and through computation of the resulting RMSE-values
1.2
Select and execute the best action (action selection policy "epsilon greedy")
1.3
Update the parameters accordingly (@ xml)
2
Refine manipulation actions or End Loop when satisfied.

Figure 9: Draft of the basic structure and setup of the implementation of our reinforcement learning algorithm for simulation calibration [97].

3.2 Machine Learning-based Calibration of Soft Tissue Simulation Material Parameters

Based on our works on cognition-guided and patient-individual soft tissue simulation for surgery assistance [98], we further investigated on an interface between HiFlow^3-based soft tissue simulation and Machine Learning algorithms. We developed a machine learning framework which allows to patient-individually calibrate soft tissue material parameters for subsequent HiFlow^3-based elasticity simulations [97].

A Q-Learning (reinforcement learning) algorithm, which assesses the quality of the preliminarily obtained simulation result by means of comparison with real patient data, allows to iteratively approximate the default material parameters to the real patient-individual parameters. Figure 9 gives a structural draft of the algorithm.

An evaluation by means of a simple soft tissue-filled beam on the one hand, and of a coarse-grained liver morphology on the other hand showed the usability and the efficacy of this promising approach. Further, training data sets have been created for setting up and optimizing a neural network in order to obtain calibration results in a more efficient and time-saving way.

For more details, please see the EMCL preprint [97]. We emphasize that this work – rather than being a closed research work with an in-depth theory backup and a complete evaluation – represents a technical report and some interesting experimental works that are to serve for further research and development.

3.3 Multiphysics model coupling using OpenPALM

"Simulations that couple multiple physical phenomena are as old as simulations themselves" [68]. In this section, we outline the usage of HiFlow^3 in multiphysics simulations in conjunction with OpenPALM [29, 76, 89]. OpenPALM is a dynamic parallel software coupler tool supporting the controlled execution of simulation models in a coupling algorithm, and providing a communication mechanism for data exchange between coupled models.

3.3.1 OpenPALM terms and concepts

Following [125], we briefly introduce the terms and concepts of OpenPALM. Its main idea is to consider multiphysics applications as a composition of units which can be coupled by means of a data transfer mechanism. OpenPALM's goal is to ease the coupling of new and of existing codes written in Fortran, C or C++ with minimal effort, even if they were not meant to be coupled in the first place. OpenPALM consists of three main components: a graphical user interface (GUI) named *PrePALM*, the *driver* and the *library*. The user can compose a coupled application in a pre-processing step with the help of the PrePALM graphical user interface. Its main feature is a canvas where the user can describe the coupling algorithm in a graphical form. This is done by defining execution paths, named *branches* in OpenPALM, scheduling the units by arranging them on the branches, and by connecting the units to indicate data transfer.

OpenPALM allows dynamic control flows in the coupling algorithm. This includes the conditional execution of units where it is not known a priori if and when conditions are fulfilled, repeated execution of units in loops where it is not known a priori if and how often the loop will be executed, or execution switches with multiple alternative paths. Complex control flows can be defined using an arbitrary number of branches.
OpenPALM features two levels of parallelism. On the one hand, units can run concurrently on separate sets of processors when they are scheduled to separate execution branches. On the other hand, OpenPALM is able to couple units which are internally parallelized supporting both shared and distributed memory parallel units, which may internally use message passing, multi-threading and/or accelerators.

A key feature is the communication mechanism. Owing to OpenPALM's philosophy of coupling individual units, it is necessary to facilitate data transfer between units and at the same time keeping them general and independent from any particular application. Models are viewed as entities which produce and/or consume data and perform certain computational tasks, so that generality and reusability for any purpose is maintained. Therefore, units cannot know about communication partners. Instead, they need a way to request for input data or to announce the availability of output data without information about source or target of the communication. OpenPALM offers communication routines for sending and receiving data, which fulfill these requirements. These routines, among others, are implemented in the *library*. Developers can use them in the unit's source code and link against the library. The communication routines are independent from the specific application at hand by using an abstract description of the data to be exchanged.

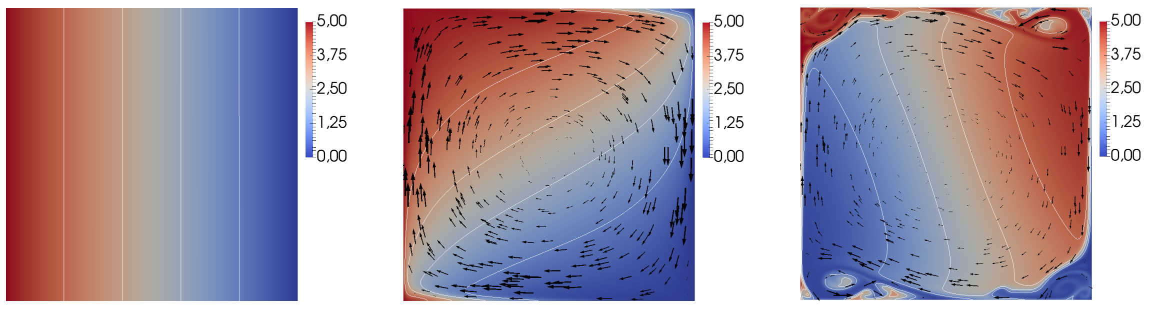
These library routines used in the units are complemented by the OpenPALM *driver*. The driver is a special entity which is automatically adjoined to any coupled application. It has two main purposes: to orchestrate the execution of the branches and units, and to act as a broker for the data transfer between the units. The driver starts, stops and monitors the execution of the branches, and controls the units' access to resources such as files, memory, or processors. It also forms the counterpart to the communication routines used in the units. Since units do in general not know their communication partners, they announce data transfer requests to the driver. The driver then deduces the correct matching of source and target, and arranges a connection between the corresponding units.

3.3.2 Example application for fluid-heat-coupling in a natural convection scenario

In [120], we demonstrated the use of OpenPALM in natural convection simulations. We implemented a Navier-Stokes solver and a heat equation solver with HiFlow^3, and used them as units in the OpenPALM coupling. Figure 10 shows the solution of the natural convection flow at selected time steps. The fluid and the heat model update each other after each time step with the new velocity and temperature fields using OpenPALM's communication mechanism. Figure 11 (left) shows the coupling setup in the PrePALM graphical user interface. The models are executed on concurrent execution branches with data exchange defined by connections of the input/output plugs. Both models used HiFlow^3's MPI parallelization with individual domain decompositions. Although the non-linear fluid model has a far larger demand for computing power than the linear temperature model, we achieved a balanced time-to-solution for both models by allocating appropriate resources to each model using OpenPALM's dynamic parallel coupling techniques. Furthermore, we achieved tremendous performance increases for specific parallel setups owing to the coupling-level parallelism using a concurrent operator splitting time stepping scheme [119, 120] while maintaining accuracy, as the Figures 11 (center) and (right) show.

4 Show Cases

HiFlow^3 is developed at the *Engineering Mathematics and Computing Lab (EMCL)* at Heidelberg University and in the *Data Mining and Uncertainty Quantification (DMQ)* research group at Heidelberg Institute for Theoretical Studies (HITS gGmbH). It is the workhorse of EMCL for many of its research activities, where the numerical solution of PDE is involved. EMCL is particularly noted for its openness to multidisciplinary research in the field of engineering and scientific computing. At EMCL and DMQ, interdisciplinarity characterizes both daily activities and the long-range direction of research. The challenge of EMCL and DMQ is to enable world-class scientific research by employing cutting-edge supercomputing



(a) $t = 0$ **(b)** $t = 5$ **(c)** $t = 10$

Figure 10: Fluid temperature (color) and velocity (arrows) at selected time steps [120].

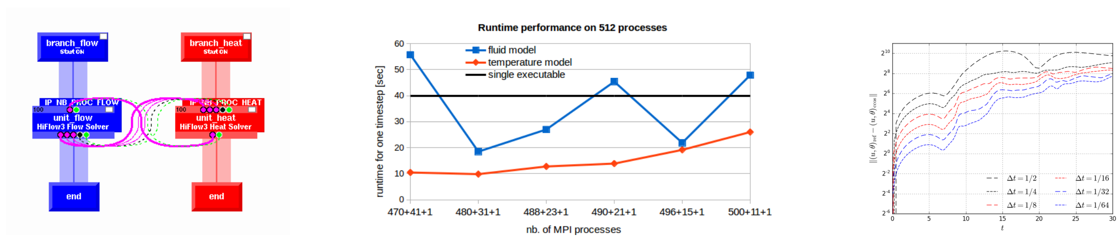


Figure 11: PrePALM coupling setup of the fluid model and the heat model (left), simulation runtime concurrent operator splitting scheme vs. reference (center), integrator accuracy concurrent operator splitting scheme vs. reference [119, 120].

technologies.

Many of the (new) features and functionalities, which have been described in Section 2, have been developed in the context of actual and current research activities. This also holds true for the configurations in Section 3. In the following, selected results of these research activities, which have been published before, are presented as *show cases* in order to give an impression, what kind of numerical simulations are enabled by HiFlow^3 Version 2.0.

Readers, who are new to HiFlow^3 or numerical simulation by means of finite elements in general, are kindly referred to Section A in the Appendix, where an overview of available tutorials on HiFlow^3 is given. These tutorials comprise general theory on the tackled problems in general and, particularly, the implementation of a numerical solver by means of finite elements with the aid of HiFlow^3.

4.1 Show Case: Cyclone-Cyclone Interaction (CCI)

A show case for the demonstration of the capabilities of HiFlow^3 in the context of meteorology and High Performance Computing (HPC) is the simulation of two interacting tropical cyclones scenarios with different physical models. These models are expressed in the form of systems of PDEs. The models — including their derivation and discretization as well as further numerical results — are presented in detail in [49]. For further information on the background, history and state-of-the-art in this field, please refer to [49] and the references therein. The task of forecasting the motion and evolution of the interaction of tropical cyclones is a challenging and computationally expensive task. The underlying physical processes interact in complex ways on a wide range of spatial and temporal scales, which needs to be considered in the discretization of the respective models.

Typically, tropical cyclones have diameters on the scale of several 100 km. In the considered scenario, two cyclones of this type, which interact with each other, are placed in the computational domain with an initial distance of the storm centers of 400 km. Therefore, the horizontal extend of the dynamic evolution of the two cyclones easily reaches the scale of 1000 km. Consequently, the domain Ω needs to be chosen large enough such that the cyclones are still fully contained within the domain on the considered time-

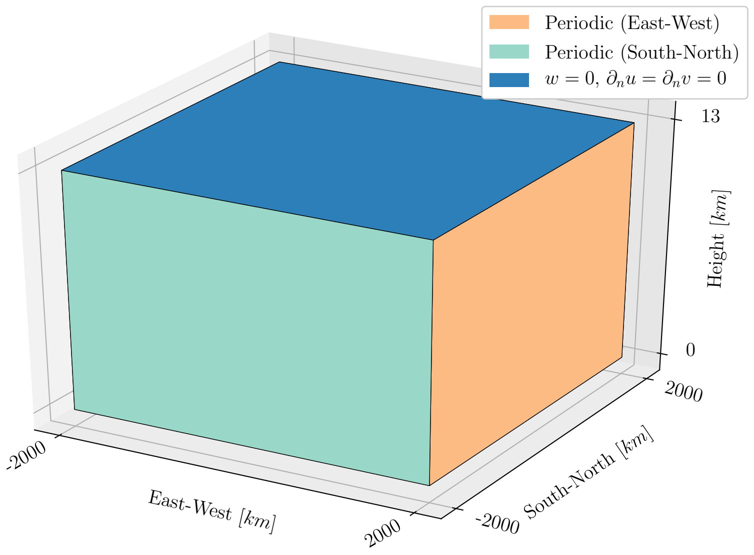


Figure 12: Domain Ω and boundary conditions for CCI scenario.

interval. In the presented case, the domain extends over 4000 km in both horizontal directions, and 13 km in the vertical. Horizontally, the domain is centered around the origin of the coordinate system, i.e., the domain Ω is defined as

$$\Omega := [-2,000,000; 2,000,000] \times [-2,000,000; 2,000,000] \times [0; 13,000], \tag{19}$$

where the boundaries of the intervals are given in meters $[m]$. The domain Ω as well as the applied boundary conditions for Problems 4.1 and 4.2, respectively, are depicted in Figure 12.

The physical models for the evolution of the fluid dynamics, which are considered here, are the *compressible Navier-Stokes equations* for a dry atmosphere as well as a so-called *Low-Mach* number approximation, cf. [49]. The governing equations of these models are given as follows:

Problem 4.1 (Compressible Navier-Stokes model [49])

Let $\Omega \subset \mathbb{R}^3$ be as in (19) and $T \geq 0$ a final point in time. Find a velocity field $\mathbf{v} := (u, v, w)^\top : [0,T) \times \Omega \to \mathbb{R}^3$, a density perturbation $\rho^* : [0,T) \times \Omega \to \mathbb{R}$, a temperature perturbation $\theta_v^* : [0,T) \times \Omega \to \mathbb{R}$ and a pressure perturbation $p^* : [0,T) \times \Omega \to \mathbb{R}$ satisfying

$$\partial_t \mathbf{v} + (\mathbf{v} \cdot \nabla)\mathbf{v} + \frac{1}{\rho}\nabla p^* - \nu_a \Delta \mathbf{v} = \left(fv, -fu, -\frac{\rho^*}{\rho} g \right)^\top \tag{20}$$

$$\partial_t \rho^* + w \partial_z \rho_0 + \mathbf{v} \cdot \nabla \rho^* + \rho \operatorname{div} \mathbf{v} = 0 \tag{21}$$

$$\partial_t \theta_v^* + w \partial_z \theta_{v,0} + (\mathbf{v} \cdot \nabla)\, \theta_v^* = 0 \tag{22}$$

$$\left[\frac{\left(-\frac{g\kappa}{\theta_z} \ln\left(1 + \frac{\theta_z z}{\theta_0}\right) + R' \right) \rho^* \left(\theta_v^* + \theta_{v,0}\right)}{p_0} + 1 + \frac{\theta_v^*}{\theta_{v,0}} \right]^{\frac{1}{1-\kappa}} p_0 - p_0 = p^* \tag{23}$$

$$w = 0, \quad \partial_{\mathbf{n}} u = 0, \quad \partial_{\mathbf{n}} v = 0 \quad \text{on} \quad [0,T] \times \Gamma \tag{24}$$

$$\mathbf{v}(0,x) = \mathbf{v}_0(x), \quad \rho^*(0,x) = \rho_0^*(x), \quad \theta_v^*(0,x) = \theta_{v,0}^*(x), \quad p^*(0,x) = p_0^*(x) \tag{25}$$

as well as periodic boundary conditions in both horizontal directions for all variables $\mathbf{v}$, ρ^*, θ_v^* and p^*,

$$p_0^* := \left(\left(\rho_0^* + \rho_0\right) R' \left(\theta_{v,0}^* + \theta_0\right) \right)^{\frac{1}{1-\kappa}} (1000hPa)^{\frac{-\kappa}{1-\kappa}} - p_0$$

and

$$\rho := \rho^* + \rho_0.$$

Equations (20)-(23) are required to hold on $(0,T) \times \Omega$ and (25) is asked to hold on $\{t = 0\} \times \Omega$.

Problem 4.2 (Low-Mach model [49])
Let $\Omega \subset \mathbb{R}^3$ be as in (19) and $T \geq 0$ a final point in time. Find a velocity field $\mathbf{v} := (u, v, w)^\top : [0,T) \times \Omega \to \mathbb{R}^3$, a density perturbation $\rho^* : [0,T) \times \Omega \to \mathbb{R}$, a temperature perturbation $\theta_v^* : [0,T) \times \Omega \to \mathbb{R}$, a pressure perturbation $p^* : [0,T) \times \Omega \to \mathbb{R}$ and a thermodynamic pressure $p_{th} : [0,T) \to \mathbb{R}$, which fulfills

$$\int_\Omega p^* \mathrm{d}x = 0, \tag{26}$$

satisfying

$$\partial_t \mathbf{v} + (\mathbf{v} \cdot \nabla)\mathbf{v} + \frac{1}{\rho}\nabla p^* - \nu_a \Delta \mathbf{v} + \left(-fv, fu, \frac{\rho^*}{\rho} g\right)^\top = 0 \tag{27}$$

$$\partial_t p_{th} + w\partial_z p_0 + \frac{p_{th} + p_0}{1 - \kappa} \operatorname{div} \mathbf{v} = 0 \tag{28}$$

$$\partial_t \theta_v^* + w\partial_z \theta_{v,0} + (\mathbf{v} \cdot \nabla)\, \theta_v^* = 0 \tag{29}$$

$$\frac{\left(\frac{p_0}{p_{th}+p_0}\right)^\kappa p_{th}\theta_{v,0} + \left[\left(\frac{p_0}{p_{th}+p_0}\right)^\kappa - 1\right] p_0\theta_{v,0} - p_0\theta_v^*}{\left(-\frac{g\kappa}{R'\theta_z} \ln\left(1 + \frac{\theta_z z}{\theta_0}\right) + 1\right) R'\left(\theta_{v,0} + \theta_v^*\right)\theta_{v,0}} = \rho^* \tag{30}$$

$$\partial_t p_{th} - \frac{\int_\Omega \kappa w \partial_z p_0 \mathrm{d}x}{(1-\kappa)\,|\Omega|} = 0 \tag{31}$$

$$w = 0, \quad \partial_\mathbf{n} u = 0, \quad \partial_\mathbf{n} v = 0 \quad \text{on} \quad [0,T] \times \Gamma \tag{32}$$

$$\mathbf{v}(0,x) = \mathbf{v}_0(x), \quad \rho^*(0,x) = \rho_0^*(x), \quad \theta_v^*(0,x) = \theta_{v,0}^*(x), \quad p^*(0,x) = p_0^*(x), \quad p_{th}(0) = 0 \tag{33}$$

as well as periodic boundary conditions in both horizontal directions for all variables $\mathbf{v}$, ρ^*, θ_v^* and p^*,

$$\rho_0^* := \frac{(1000\ hPa)^\kappa\,(p_{th}(t) + p_0(x))^{1-\kappa}}{R'\left(\theta_{v,0}^* + \theta_0\right)} - \rho_0$$

and

$$\rho := \rho^* + \rho_0.$$

Equations (27)-(30) are required to hold on $(0,T) \times \Omega$ and (33) is asked to hold on $\{t = 0\} \times \Omega$.

In both the Compressible Navier-Stokes model and the Low-Mach model, the unknown functions $\mathbf{v}$, ρ^*, θ_v^* and p^* are discretized in space by means of finite elements and by means of finite differences in time [49]. The domain Ω is triangulated admissibly in congruent hexahedra. Based on this triangulation, finite elements of Lagrange type with trilinear basis polynomials are chosen for all six unknown functions, i.e., a $\mathbb{Q}_1/\mathbb{Q}_1/\mathbb{Q}_1/\mathbb{Q}_1/\mathbb{Q}_1/\mathbb{Q}_1$ discretization is chosen in space [49]. Also, all finite dimensional test function spaces are chosen to be defined by the $\mathbb{Q}_1$ discretization of the domain Ω by hexahedra.

For the discretization in time, in the momentum equation of both the Compressible Navier-Stokes and the Low-Mach model all terms are treated in a Crank-Nicolson manner except for the pressure part p^*, which is treated in an implicit Euler manner [49]. The continuity equation is in both cases discretized by the implicit Euler scheme in time, whereas the thermodynamic energy equation is discretized by the Crank-Nicolson time-stepping scheme [49].

The resulting discrete nonlinear systems of equations are solved with an inexact Newton method, where the linearized systems in each Newton step are solved with the GMRES algorithm [94] – preconditioned by the BoomerAMG preconditioner of the hypre library [42] – in the case of the compressible Navier-Stokes model and with the FGMRES algorithm [94] – preconditioned by a nested Schur complemented approach

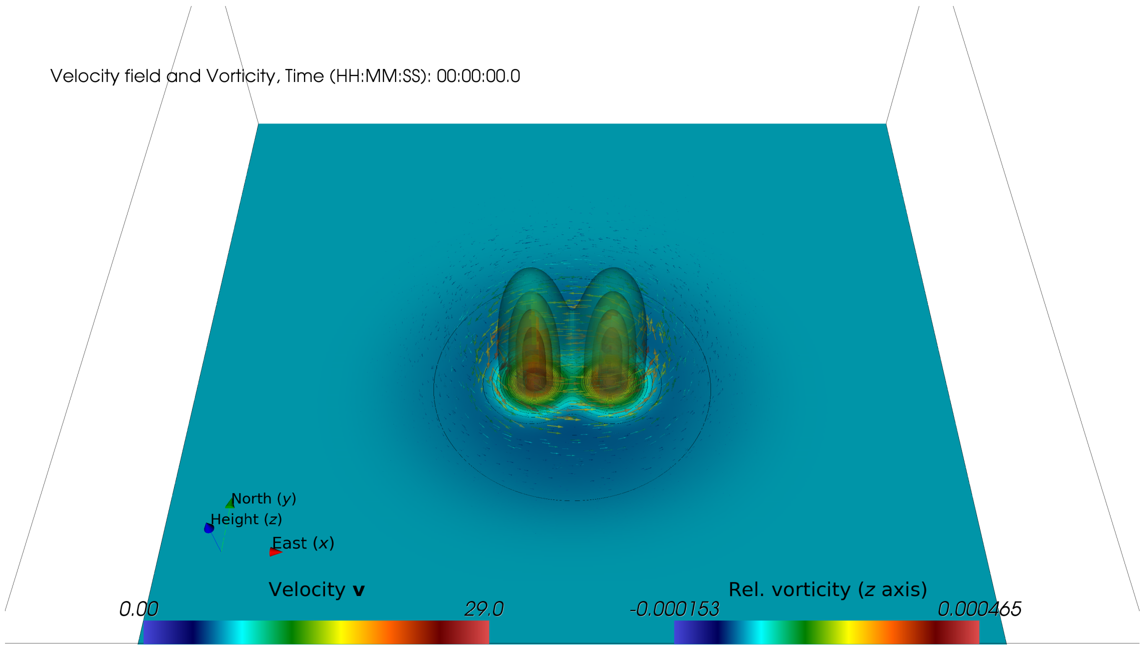


Figure 13: Initial velocity field $\left[\frac{m}{s}\right]$ and vertical vorticity component $\left[\frac{1}{s}\right]$ [49].

[49] – in the case of the Low-Mach number approximation. Please refer to [49] for further details about the solution process.

The initial conditions for the CCI scenario are depicted in Figure 13. Please refer to [49] for further visualizations. To demonstrate the capabilities of HiFlow^3 in the context of this showcase in terms of scalability, the parallel scaling behavior of the two models with the respective solvers has been investigated and the results are shown in Figures 14 and 15, respectively.

The results show that both models scale very well up to 1024 processes and achieve over 80% of parallel efficiency, while the differences between both models are small. For the configuration of 2048 processes, there is a notable decrease in efficiency measure as the speed-up from 1024 to 2048 processes is notably smaller than for the lower processor numbers. On 2048 processors, the Compressible Navier-Stokes model performs considerably better by providing an efficiency of over 60%, whereas the efficiency of the Low-Mach model decreases to approximately 45%. For further comparative measures of the two models and their solvers, please refer to [49].

Figure 16 shows a comparison of the solutions of the two models after approximately 12 h of simulated physical time and Figure 17 shows the solution of the Low-Mach model after 96 h of simulated physical time. For an in-depth comparison, please refer to [49].

4.2 Show Case: Fluid-Structure Interaction for Aortic Blood Flow

The numerical simulation of aortic blood flow can help to get a deeper understanding of the bio-mechanics of aortic physiology and disease. Pre-surgical risk parameters computed by numerical simulation are discussed in the literature. They can enhance the risk assessment before aortic surgery. Furthermore, simulations can give information on the dynamics of blood flow after different surgery scenarios. At the current state of the art, however, simulation methods have to be refined and validated in a broad way [34].

The three-dimensional and time-dependent bio-mechanics of the aorta are mainly governed by the fluid flow of the blood and the vessel wall movement. The fluid flow can be modeled by the Navier-Stokes equations. The vessel wall displacement can be modeled by the momentum conservation equations for visco-elastic materials. At the boundaries of the considered vessel section, the three-dimensional model can be embedded in a zero-dimensional lumped parameter model of the full cardiovascular system [45].

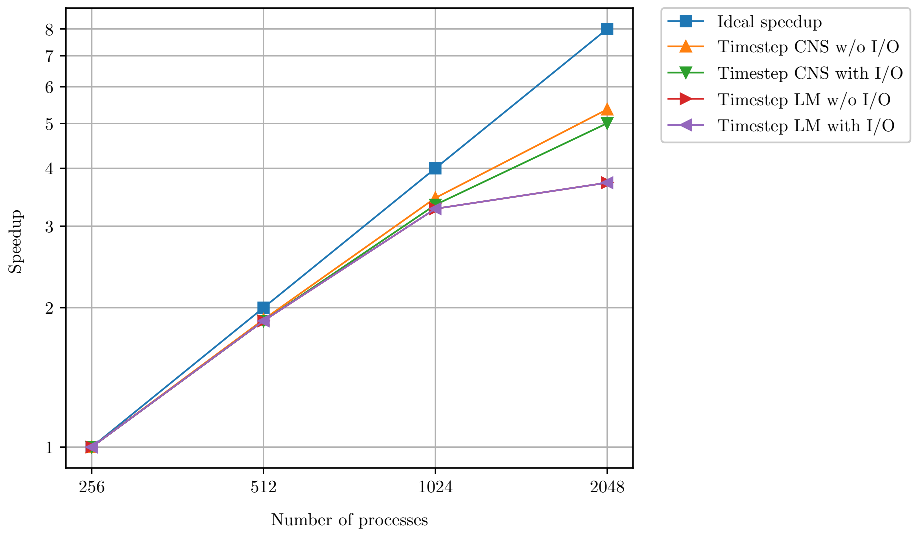


Figure 14: Speedup in strong scaling test for whole time-step on bwForCluster MLS & WISO (Production) relative to 256 processes [49].

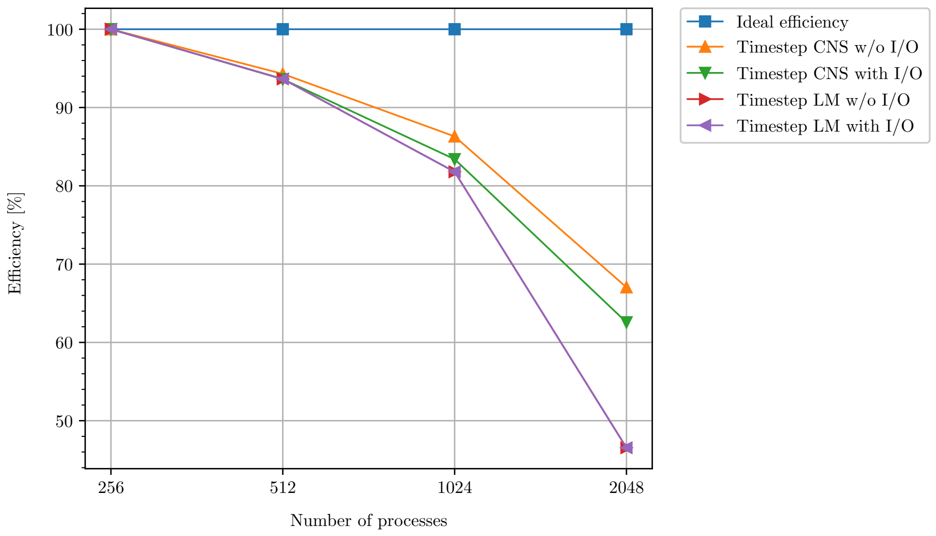
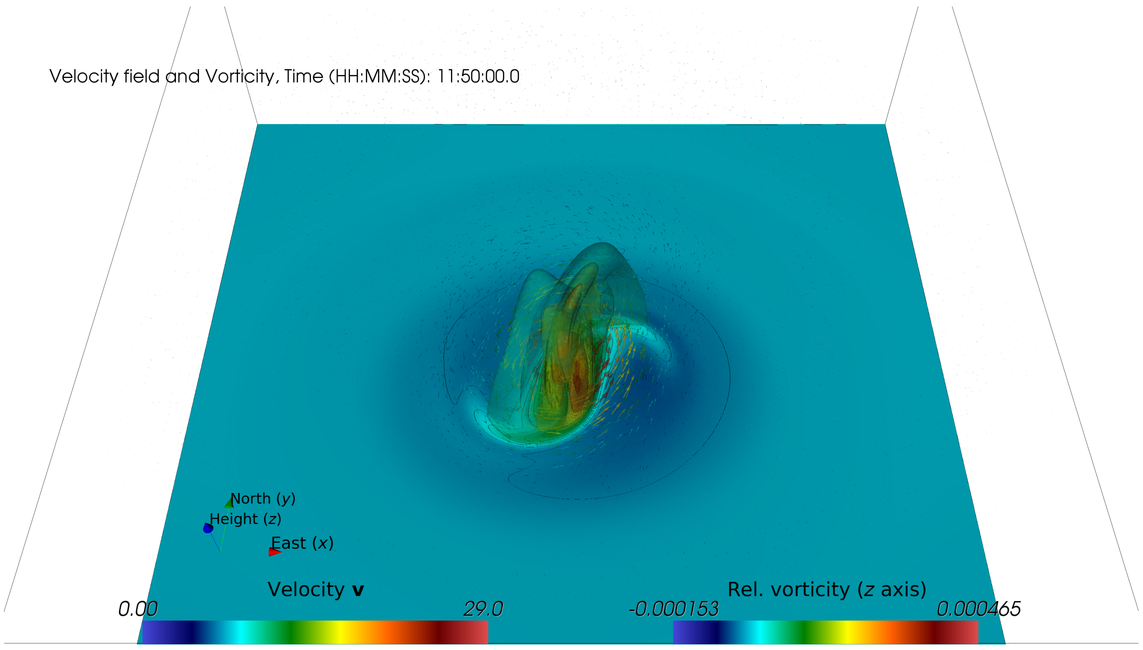
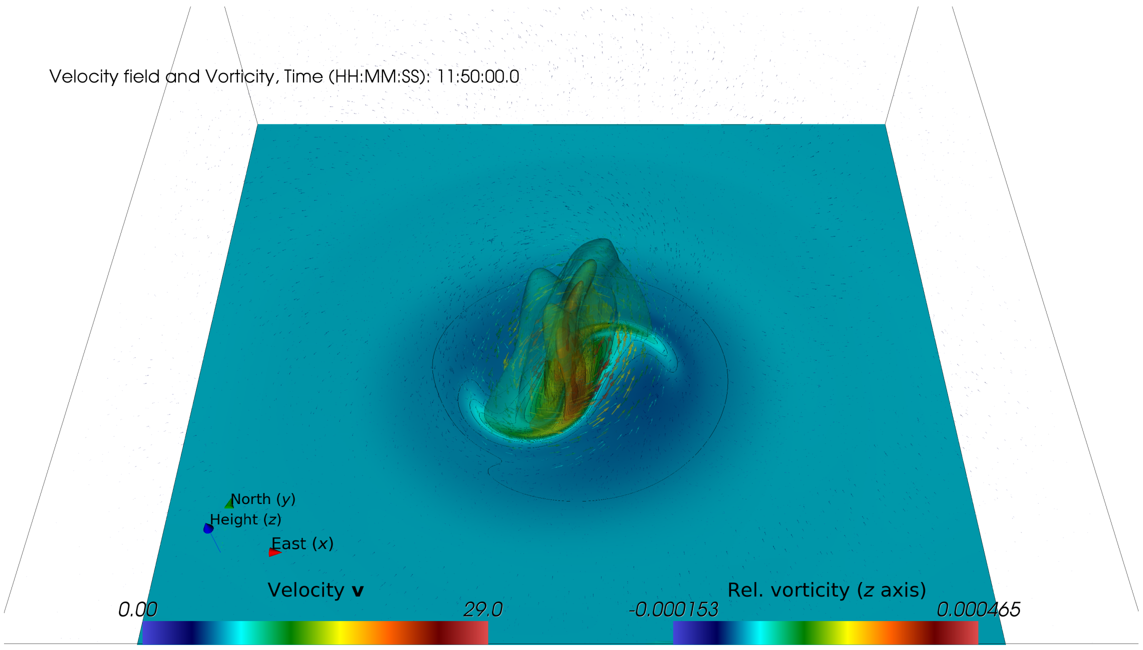


Figure 15: Efficiency in strong scaling test for whole time-step on bwForCluster MLS & WISO (Production) relative to 256 processes [49].



(a) Compressible Navier-Stokes



(b) Low-Mach

Figure 16: Velocity field and vertical vorticity component at common final time.

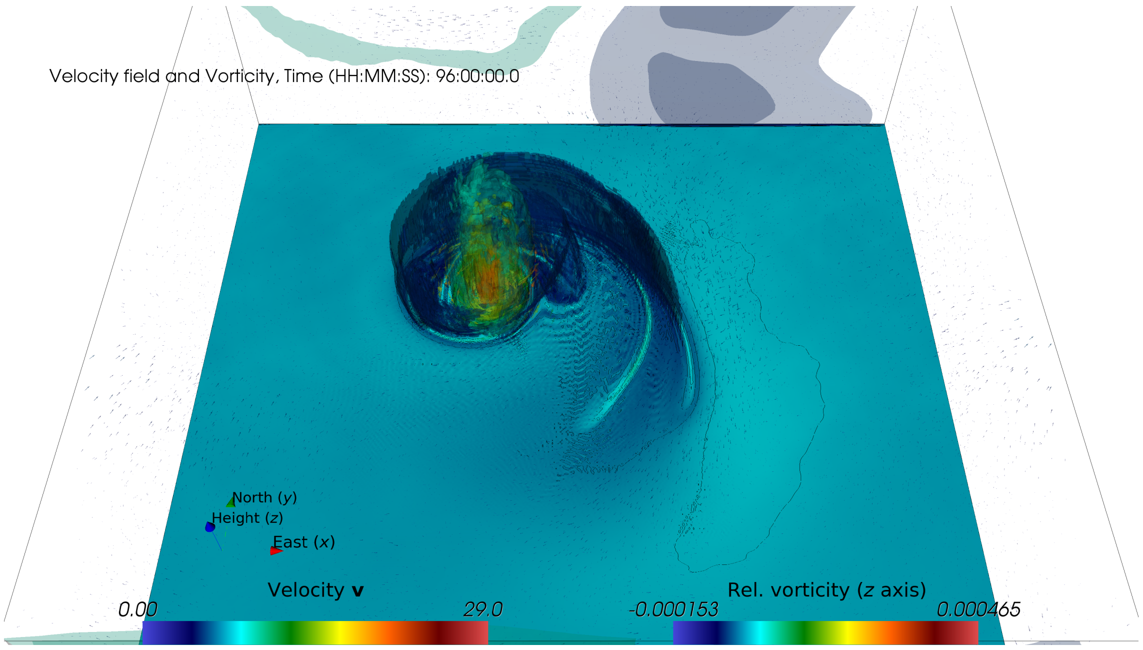


Figure 17: Final velocity field and vertical vorticity component at $T = 96$ h, computed with Low-Mach model.

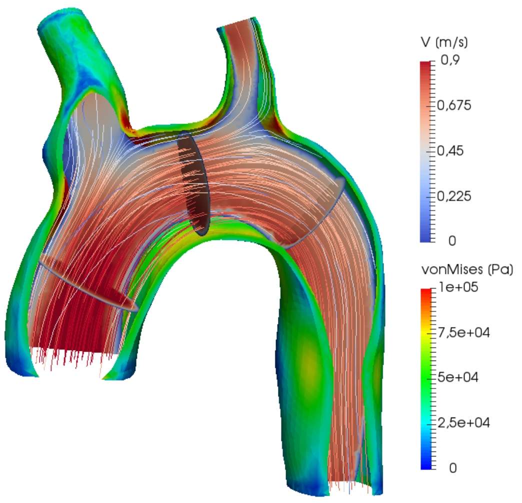
Considering blood vessel dynamics, which include the elastic vessel wall movement, involves the coupling of fluid flow and wall displacement. A monolithic solver for the resulting fluid-structure interaction (FSI) problem is implemented in HiFlow^3-based on the Arbitrary Eulerian-Lagrangian (ALE) method. The mathematical foundation is described for example in [46].

In the release 2.0 of HiFlow^3, the FSI-solver is available for the computation of the FSI benchmark proposed by [113]. The computed benchmark quantities are well within the range of the reference values provided in [114].
Furthermore, an analytically solvable benchmark problem for fluid-structure interaction including the uncertainty of input parameters has been defined. It is based on a two-dimensional Couette-flow with an additional layer of structure material surrounding the fluid phase. The analytical solution is point-symmetric for the modes of the Polynomial Chaos expansion. The implementation of the FSI-solver in combination with the intrusive UQ-approach described in 2.4 has been verified by means of this benchmark in [72].

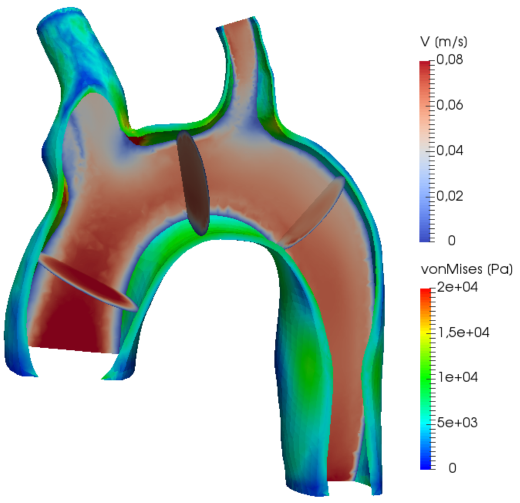
For further validation purposes, the implemented FSI solver has been used to simulate the fluid flow through a prototypical aortic phantom. The simulation results were compared to the velocity field of the fluid flow through the silicone vessel measured by phase-contrast magnetic resonance imaging (PC-MRI) [74]. Considering the uncertainties in the stiffness of the vessel wall material, the simulation results are in well accordance to the measurements [73].

The FSI simulation can be embedded in a pre- and post-processing workflow enabling the usage of patient-specific geometries as described in Section 3.1. The simulation of a proband is visualized in Figure 18.

Possible future extensions of the FSI application are given by data assimilation (cf. Section 3.2) with PC-MRI measurements and the usage of complex material laws for blood flow and the aortic wall. The developments aim to enhance medical imaging of aortic blood flow and to provide additional information for the pre-operative risk assessment of aortic surgery.



(a) Mean value: Highest von Mises stress values appear at the bifurcations of the aortic bow.



(b) Standard deviation: The highest influence of the uncertainty of the input parameters can be located within the domain.

Figure 18: Simulation of aortic blood flow with elastic vessel wall movement. The visualization shows the velocity magnitude V and the von Mises stress at maximal mid-systolic inflow. The inflow velocity and the stiffness of the vessel wall were modeled as uncertain parameters.

4.3 Show Case: Mitral Valve

In the framework of the collaborative research center (SFB TRR 125) *Cognition-guided Surgery* [2)], we model and simulate the biomechanical behaviour of the elastic mitral valve soft tissue in the human heart, subjected to surgical manipulation during and after a minimally-invasive mitral valve reconstruction through annuloplasty surgery [100].

Using the MSML (see 3.1), a segmented mitral valve (MV) geometry is meshed and annotated for the HiFlow^3-based FEM simulation, and boundary conditions are defined to model the natural forces during the cardiac cycle and the external forces through annuloplasty.

Our HiFlow^3-based simulation application implements via the elasticity equations (cf. the respective HiFlow^3 Tutorial on Soft Tissue Simulation [99]) a biomechanical model for the numerical simulation of the MV behavior, see Figure 19 for an illustration. Specifically, it is capable of simulating the post-surgical behavior of the mitral valve, both with respect to the closing behavior and with respect to the resulting von Mises stress distribution which may indicate potential tissue rupture (see Figure 20).

Looking at the performance of the entire mitral valve repair (MVR) surgery assistance system, we separately evaluated the MSML-based MVR simulation preprocessing components on the one hand, and the HiFlow^3-based MVR simulation component on the other hand.

For the MSML-based simulation preprocessing component, we employ Intel Core i7-4600U notebook PCs with 8 GB RAM, which have access to our medical patient data bases, and thus fully automatically yield results after 72 ± 4 seconds.

The HiFlow^3-based MVR simulation application [100] is executed on the bwUniCluster [3)], a High-Performance Computing cluster based in Germany. We run the simulation on 128 cores, distributed over 16 nodes, each with 64 GB RAM. Space discretization with approximately 120 000 tetrahedral cells and 70 000 DoFs corresponds to the original TEE-ultrasound-based imaging resolution, such that the usage of the HiFlow^3-based CG solver along with its internal Symmetric Gauss-Seidel and the external hypre BoomerAMG preconditioners yields simulation results (as shown in Figure 20) in just under three minutes of computation time [98].

[2)] Cognition-Guided Surgery: `www.cognitionguidedsurgery.de/`

[3)] bwUniCluster: `https://bwunicluster.urz.uni-heidelberg.de/`

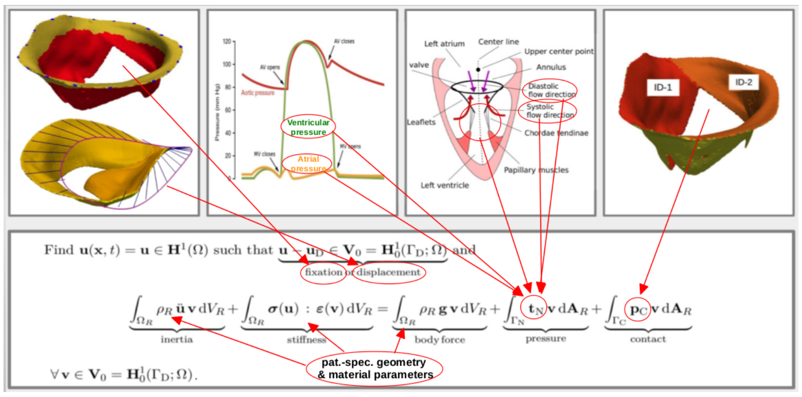

Left atrium
Center line
Upper center point
Annulus
valve
Diastolic flow direction
Systolic flow direction
Leaflets
Chordae tendinae
Papillary muscles
Left ventricle
Ventricular pressure
Atrial pressure
ID-1
ID-2
Find $\mathbf{u}(\mathbf{x},t) = \mathbf{u} \in \mathbf{H}^1(\Omega)$ such that $\mathbf{u} - \mathbf{u}_D \in \mathbf{V}_0 = \mathbf{H}^1_0(\Gamma_D;\Omega)$ and
fixation
displacement
$\underbrace{\int_{\Omega_R} \rho_R \ddot{\mathbf{u}} \mathbf{v} \, dV_R}_{\text{inertia}} + \underbrace{\int_{\Omega_R} \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) \, dV_R}_{\text{stiffness}} = \underbrace{\int_{\Omega_R} \rho_R \mathbf{g} \mathbf{v} \, dV_R}_{\text{body force}} + \underbrace{\int_{\Gamma_N} \mathbf{t}_N \mathbf{v} \, d\mathbf{A}_R}_{\text{pressure}} + \underbrace{\int_{\Gamma_C} \mathbf{p}_C \mathbf{v} \, d\mathbf{A}_R}_{\text{contact}}$
$\forall \mathbf{v} \in \mathbf{V}_0 = \mathbf{H}^1_0(\Gamma_D;\Omega)$.
pat.-spec. geometry & material parameters

Figure 19: Illustration of the biomechanical model specification for describing the MV behavior and the procedure of MVR annuloplasty surgery.

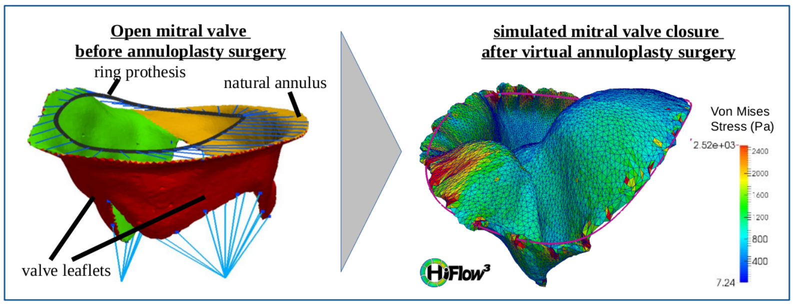

Open mitral valve before annuloplasty surgery
ring prothesis
natural annulus
valve leaflets
simulated mitral valve closure after virtual annuloplasty surgery
Von Mises Stress (Pa)
2.52e+03
2400
2000
1600
1200
800
400
7.24
HiFlow3

Figure 20: Visualization of the biomechanical model and of the simulated post-surgical MV behavior after MVR annuloplasty surgery.

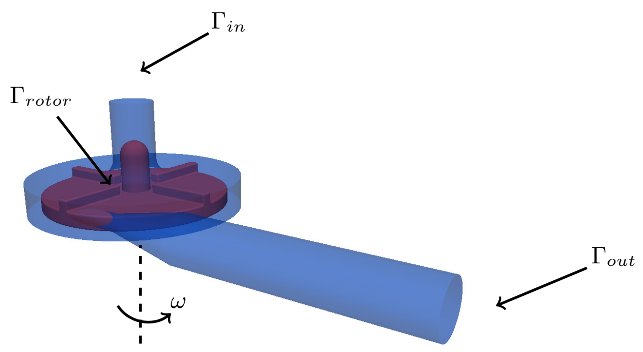


Figure 21: Illustration of the boundaries and the axis of rotation on the blood pump geometry.

Inflow maximal speed (m/s)	0.5	Inflow speed variation (σ_1)	10%
Dynamic viscosity $(N \cdot s/m^2)$	0.0035	Viscosity variation (σ_3)	10%
Angular speed (rad/s)	261.8	Angular speed variation (σ_2)	10%
RPM	2500	Density (Kg/m^3)	1035

Table 7: Model parameter values.

4.4 Show Case: Blood Pump with UQ

One UQ application in HiFlow^3 by using the intrusive Galerkin method is a FDA blood pump [44] simulation. Figure 21 indicates here the geometry of FDA blood pump, it consists of a pump housing (blue) and a rotor (red). The rotor operates under a high rotating speed and pushes the blood from inlet to the outlet through the pump chamber.

In order to simulate the blood flow within the pump chamber, the Variational Multiscale method is employed, the moving mesh is modeled with a shear layer update approach [107]. We consider three uncertain parameters: inflow boundary condition $\boldsymbol{g}$, rotating speed $\boldsymbol{\omega}$ of the rotor and dynamic viscosity μ. We model each of those uncertain parameters with independent, uniformly distributed random variables $\xi_i \sim U(-1,1), i = 1,2,3$, it reads:

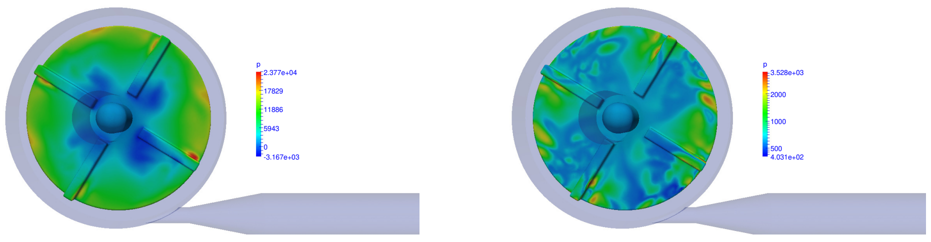
$$\boldsymbol{g} = \boldsymbol{g}_0 + \boldsymbol{g}_1\xi_1 \ , \tag{34a}$$

$$\boldsymbol{\omega} = \boldsymbol{\omega}_0 + \boldsymbol{\omega}_2\xi_2 \ , \tag{34b}$$

$$\mu = \mu_0 + \mu_3\xi_3 \ . \tag{34c}$$

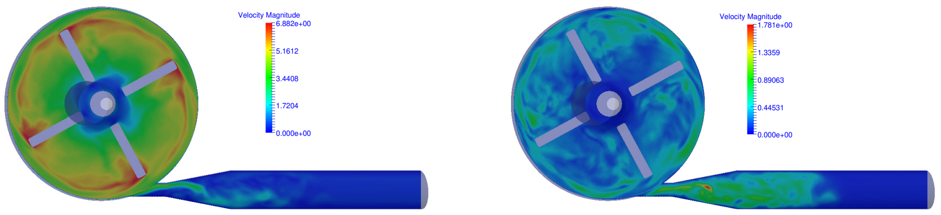
Where $\boldsymbol{g}_1 = \sigma_1\boldsymbol{g}_0$, $\boldsymbol{\omega}_2 = \sigma_2\boldsymbol{\omega}_0$ and $\mu_3 = \sigma_3\mu_0$. σ_i are the decay factors with respect to the mean value, thus $0 < \sigma_i < 1$. Due to the three uncertain inputs, velocity and pressure can be expressed by using the Polynomial Chaos (PC). After inserting the PC expansion into the governing equation, a Galerkin projection can be performed in order to obtain a generalized PCE system. For the sake of simplicity, the detailed procedure can be found in [107].

Before showing the numerical results, some physical parameters are presented in Table 7. We consider 10% deviation from the mean value, and the rotating speed is 2500 revolution per minute (RPM). Figure 22 shows the mean value and standard deviation of the pressure distribution on the rotor. The standard deviation follows slightly the magnitude of the mean value, since it is lower at the center and higher on the hub of the blade. But the uncertainty distribution arises also at certain locations where the pressure is less important. Figure 23 represents the mean value and the standard deviation of the velocity. The standard deviation becomes higher after the flow at the outlet, it might be due to the acceleration after the nozzle structure.



(a) Mean value. **(b)** Standard deviation.

Figure 22: The mean value and standard deviation of the pressure at time step = 500.



(a) Mean value. **(b)** Standard deviation.

Figure 23: The mean value and standard deviation of the velocity at time step = 500.

5 Conclusion and Outlook

HiFlow^3 Version 2.0 continues the path as a multi-purpose finite element software, which provides powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). New features and functionalities, which allow to run numerical simulations with more advanced solution algorithms and discretizations in comparison to previous releases of HiFlow^3, have been implemented and described. These comprise fully adaptive mesh refinement in a distributed environment, a new implementation of Schur complement solvers, uncertainty quantification based on chaos polynomials, energy-aware multigrid schemes and many more.

The presented new algorithms and features as well as general under-the-hood improvements have leveraged new research activities in the fields of medical engineering, meteorology and environmental sciences. The described show cases demonstrate the advantages, which HiFlow^3 can offer in performing a numerical simulation by means of finite element methods. Especially, the high performance computing capabilities of HiFlow^3 – not only in the mentioned fields of applications, but also in general – have been significantly improved.

The development process is automated and continuously checked by a Jenkins installation run at EMCL. Here at each commit into the source repository, the code syntax is checked, the library is built and automated notifications are sent to the developers. The continuous integration guarantees a high quality standard of HiFlow^3 already during the development phase.

HiFlow^3 is subject to strong development efforts in order to increase both, the quality and the ease of use of the package even for complex applications. Current development aims at establishing a standard interface in order to be able to use HiFlow^3 by means of a software as a service.

Acknowledgements

The authors acknowledge support by the state of Baden-Württemberg through bwHPC and the German Research Foundation (DFG) through grant INST 35/1134-1 FUGG. The authors gratefully acknowledge the data storage service SDS@hd supported by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK) and the German Research Foundation (DFG) through grant INST 35/1314-1 FUGG. The authors gratefully acknowledge the computing time granted by the John von Neumann Institute for Computing (NIC) and provided on the supercomputers at Jülich Supercomputing Centre (JSC).

A Tutorials

The following Table 8 shows a list of the current HiFlow^3 tutorials.

Table 8: List and description of current HiFlow^3 tutorials

Title	**Version**	**Year**	**Description**
Poisson equation	1.4	2014	Introductory Tutorial to the basic functions of HiFlow^3
Incompressible Navier-Stokes equation	1.4	2014	An application of HiFlow^3 to the Navier-Stokes equation
Linear algebra with GPU	1.4	2014	Explanation of the HiFlow^3 LA Toolbox
Applying an Inexact Newton Method	1.4	2014	A tutorial to solve non-linear problems with the inexact Newton Method
Distributed Control Problem for Poisson Equation	1.4	2014	An optimization problem coupled with the solution of the Poisson Equation
Stabilization Schemes for Advection Dominated Stationary Convection-Diffusion Equation	1.4	2014	An introduction to the coupled problem of Convection and Diffusion
Time-Discretization Methods Based on Convection-Diffusion Equation	1.4	2014	An explanation of time-discretization
Boundary Value Problem for Incompressible Generalized Porous Media Equation	1.4	2012	Porous Media Modeling
Direct and Inverse Problem in Electrostatics	1.4	2014	HiFlow^3 applied to a standard physic's problem
Elasticity for Soft Tissue Simulation	1.5	2015	An introduction to structure simulation
Poisson equation with uncertain parameters using the Spectral-Stochastic-Finite-Element-Method	1.5	2015	A study of Uncertainty Quantification
Aortic Blood Flow Simulation	2.0	2017	A patient-specific CFD-simulation of an aortic bow.
Error Estimation on Convex Bent Domains for the Poisson Equation	2.0	2017	Error estimation and local refinement for Poisson equation with different error estimators

References

[1] Brian M Adams et al. "Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.4 User's Manual". In: (2014).

[2] L. Adhianto et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs". In: *Concurr. Comput. Pract. Exp.* (2010), pp. 1–7.

[3] Inc. ADINA R & D. *ADINA Web Page.* 2017. URL: `www.adina.com`.

[4] *Advanced Configuration and Power Interface Specification Version 5.0.* URL: `http://www.acpi.info/`.

[5] J.I. Aliaga et al. "Encyclopedia of Parallel Computing". In: ed. by David Padua. Springer, 2012. Chap. ILUPACK, pp. 917–926.

[6] P. Alonso et al. "Tools for Power-Energy Modelling and Analysis of Parallel Scientific Applications". In: *Proceedings of Int. Conf. Parallel. Proc.* 2012.

[7] P. R. Amestoy et al. "A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling". In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[8] P. R. Amestoy et al. "Hybrid scheduling for the parallel solution of linear systems". In: *Parallel Computing* 32.2 (2006), pp. 136–156.

[9] E. Anderson et al. *LAPACK Users' Guide.* Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[10] ANSYS Inc. *ANSYS Fluent Web page.* 2017. URL: `http://www.ansys.com/Products/Fluids/ANSYS-Fluent`.

[11] H. Anzt et al. "HiFlow^3 – A Hardware-Aware Parallel Finite Element Package". In: *Tools for High Performance Computing 2011* (2012), pp. 139–151.

[12] Hartwig Anzt et al. "HiFlow^3 – A Flexible and Hardware-Aware Parallel Finite Element Package". In: *Preprint Series of the Engineering Mathematics and Computing Lab* 0.06 (2013). ISSN: 2191-0693. DOI: `10.11588/emclpp.2010.06.11675`. URL: `https://journals.ub.uni-heidelberg.de/index.php/emcl-pp/article/view/11675`.

[13] D. Arndt et al. "The `deal.II` Library, Version 8.5". In: *Journal of Numerical Mathematics* (2017). DOI: `10.1515/jnma-2016-1045`.

[14] C. Augonnet et al. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Proceedings of Euro-Par Parallel Processing.* 2009.

[15] Autodesk. *Autodesk Simulation Mechanical Web Page.* The Autodesk Simulation is not sold anymore. 2017. URL: `https://www.autodesk.de/products/simulation-mechanical/overview`.

[16] Inc. Autodesk. *Autodesk Simulation Web Page.* This is the current website describing the FE parts of Autodesk software. 2017. URL: `https://www.autodesk.de/solutions/simulation/overview`.

[17] Satish Balay et al. “Efficient Management of Parallelism in Object Oriented Numerical Software Libraries”. In: *Modern Software Tools in Scientific Computing.* Ed. by E. Arge, A. M. Bruaset, and H. P. Langtangen. Birkhäuser Press, 1997, pp. 163–202.

[18] Satish Balay et al. *PETSc Users Manual.* Tech. rep. ANL-95/11 - Revision 3.8. Argonne National Laboratory, 2017. URL: `http://www.mcs.anl.gov/petsc`.

[19] Satish Balay et al. *PETSc Web page.* `http://www.mcs.anl.gov/petsc`. 2017. URL: `http://www.mcs.anl.gov/petsc`.

[20] W. Bangerth, R. Hartmann, and G. Kanschat. “deal.II – a General Purpose Object Oriented Finite Element Library”. In: *ACM Trans. Math. Softw.* 33.4 (2007), pp. 24/1–24/27.

[21] S. Barrachina et al. “An Integrated Framework for Power-Performance Analysis of Parallel Scientific Workloads”. In: *ENERGY 2013: The Third Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies.* 2013.

[22] M. Barreda et al. “Automatic detection of power bottlenecks in parallel scientific applications”. In: *Comput. Sci. Res. Dev.* (2013).

[23] Peter Bastian et al. *DUNE Web page.* 2017. URL: `https://www.dune-project.org/community/people/`.

[24] Dr. Christian Becker et al. *DUNE Web page.* 2013. URL: `http://www.feast.tu-dortmund.de/`.

[25] D. Bedard et al. *PowerMon 2: Fine-grained, Integrated Power Measurement.* Tech. rep. RENCI Technical Report Series TR-09-04, 2009.

[26] C. Bekas and A. Curioni. “A new energy aware performance metric”. In: *J. Comput. Sci. Res. Dev.* 25 (2010), pp. 187–195.

[27] Michele Benzi, Gene H. Golub, and Jörg Liesen. “Numerical solution of saddle point problems”. In: *Acta Numerica* 14 (May 2005), pp. 1–137. ISSN: 0962-4929. DOI: `10.1017/S0962492904000212`.

[28] D. Braess. *Finite Elemente: Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie.* Springer-Verlag, 2007.

[29] S. Buis et al. “PALM: A computational framework for assembling high-performance computing applications”. In: *Concurr. Comput. Pract. Exp.* 18 (2006), pp. 231–245.

[30] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. “`p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees”. In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: `10.1137/100791634`.

[31] E. Chan et al. “SuperMatrix: A Multithreaded Runtime Scheduling System for Algorithms-by-Blocks”. In: *Proceedings of ACM Symposium on Principles and Practices of Parallel Programming (PPoPP).* 2008.

[32] Chuanmiao Chen, Michal Křížek, and Liping Liu. “Numerical integration over pyramids”. In: *Advances in Applied Mathematics and Mechanics* 5.03 (2013), pp. 309–320.

[33] Edmond Chow, Andrew J. Cleary, and Robert D. Falgout. “Design of the hypre Preconditioner Library”. In: *In SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing.* Ed. by Mike Henderson, Chris Anderson, and Steve Lyons. SIAM, 1998, pp. 106–116.

[34] Bongjae Chung and Juan Raul Cebral. "CFD for evaluation and treatment planning of aneurysms: review of proposed clinical uses and their challenges". In: *Annals of biomedical engineering* 43.1 (2015), pp. 122–138.

[35] P. G. Ciarlet. *The Finite Element Method For Elliptic Problems.* SIAM, 2002.

[36] T. A. Davis. "Algorithm 832: UMFPACK, an unsymmetric-pattern multifrontal method". In: *ACM Trans. Math. Softw.* 30.2 (2004), pp. 196–199.

[37] G. Diamos and S. Yalamanchili. "Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems". In: *Proceedings of High-Performance Distributed Computing (HPDC).* 2008.

[38] M.F. Dolz et al. "ArduPower: A Low-cost Wattmeter to improve Energy Efficiency of HPC Applications". In: *Proceedings of Int. Green and Sustainable Computing Conference (IGSC).* 2015.

[39] A. Duran et al. "OmpSs: A proposal for programming heterogeneous multi-core architectures". In: *Parallel Process. Lett.* 21 (2011).

[40] Oliver G Ernst et al. "On the convergence of generalized polynomial chaos expansions". In: *ESAIM: Mathematical Modelling and Numerical Analysis* 46.2 (2012), pp. 317–339.

[41] *European Union Public Licence (EUPL) v1.2.* URL: https://joinup.ec.europa.eu/page/eupl-text-11-12.

[42] Robert D. Falgout and Ulrike Meier Yang. "hypre: A Library of High Performance Preconditioners". In: *Lecture Notes in Computer Science.* Ed. by Peter M. A. Sloot et al. Vol. 2331. Springer Berlin Heidelberg, 2002, pp. 632–641. ISBN: 978-3-540-43594-5. DOI: 10.1007/3-540-47789-6_66. URL: http://link.springer.com/10.1007/3-540-47789-6%20http://link.springer.com/chapter/10.1007/3-540-47789-6%7B%5C_%7D10%20http://link.springer.com/10.1007/3-540-47789-6%7B%5C_%7D66.

[43] Jonathan Feinberg and Hans Petter Langtangen. "Chaospy: An open source tool for designing methods of uncertainty quantification". In: *Journal of Computational Science* 11 (2015), pp. 46–57.

[44] U.S. Food and Drug Administration (FDA). *FDA's "Critical Path" Computational Fluid Dynamics (CFD)/Blood Damage Project.* 2013. URL: https://nciphub.org/wiki/FDA_CFD.

[45] Luca Formaggia, Alfio Quarteroni, and Alessandro Veneziani. *Cardiovascular Mathematics: Modeling and simulation of the circulatory system.* Vol. 1. Springer Science & Business Media, 2010.

[46] Giovanni Paolo Galdi and Rolf Rannacher. *Fundamental trends in fluid-structure interaction.* Vol. 1. World Scientific, 2010.

[47] *GASPI: Global Address Space Programming Interface - Specification of a PGAS API for communication, Version 16.1.* GASPI Forum.

[48] T. Gautier, X. Besseron, and L. Pigeon. "KAAPI: A thread scheduling runtime system fo data flow computations on cluster of multi-processors". In: *Proceedings of Parallel Symbolic Computation (PASCO).* 2007.

[49] Simon Gawlok. "Numerical Methods for Compressible Flow with Meteorological Applications". PhD thesis. Ruprecht-Karls-Universität Heidelberg, 2017. DOI: 10.11588/heidok.00023532. URL: http://www.ub.uni-heidelberg.de/archiv/23532.

[50] R. Ge et al. “PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.5 (2010), pp. 658–671.

[51] Roger G. Ghanem and Pol D. Spanos. *Stochastic Finite Elements: A Spectral Approach.* New York, NY, USA: Springer-Verlag New York, Inc., 1991. ISBN: 0-387-97456-3.

[52] A.S. Grimshaw, J.B. Weissmann, and W.T. Strayer. “Portable Run-Time Support for Dynamic Object-Oriented Parallel Procesing”. In: *ACM Transactions on Computer Systems* (1993).

[53] OpenCFD Ltd. (ESI Group). *OpenFOAM Web Page.* 2017. URL: `http://www.openfoam.com/`.

[54] V. Hernandez, J. E. Roman, and V. Vidal. “SLEPc: Scalable Library for Eigenvalue Problem Computations”. In: *Lect. Notes Comput. Sci.* 2565 (2003), pp. 377–391.

[55] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. “SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems”. In: *ACM Trans. Math. Software* 31.3 (2005), pp. 351–362.

[56] Michael A. Heroux et al. “An overview of the Trilinos project.” English. In: *ACM Trans. Math. Softw.* 31.3 (2005), pp. 397–423. ISSN: 0098-3500; 1557-7295/e. DOI: `10.1145/1089014.1089021`.

[57] Vincent Heuveline et al. “Scalability Study of HiFlow^3 based on a Fluid Flow Channel Benchmark”. In: *Preprint Series of the Engineering Mathematics and Computing Lab* 0.05 (2012). ISSN: 2191-0693. DOI: `10.11588/emclpp.2012.05.11704`. URL: `https://journals.ub.uni-heidelberg.de/index.php/emcl-pp/article/view/11704`.

[58] COMSOL Inc. *COMSOL Multiphysics Web page.* 2017. URL: `https://www.comsol.de/comsol-multiphysics`.

[59] *Intel Cilk Plus.* URL: `https://www.cilkplus.org/`.

[60] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B: System Programming Guide, Part 2.* 2016.

[61] *Intel Math Kernel Library. Reference Manual.* ISBN 630813-054US. Santa Clara, USA: Intel Corporation, 2009.

[62] *Intel Xeon Phi Coprocessor Datasheet.* Tech. rep. Intel, 2015.

[63] Tobin Isaac et al. “Recursive algorithms for distributed forests of octrees”. In: *SIAM Journal on Scientific Computing* 37.5 (2015), pp. C497–C531. DOI: `10.1137/140970963`.

[64] V. John et al. “A comparison of three solvers for the incompressible Navier-Stokes equations”. In: *LSSC.* 1999.

[65] M. Kac and A. J. F. Siegert. “An Explicit Representation of a Stationary Gaussian Process”. In: *The Annals of Mathematical Statistics* 18.3 (1947), pp. 438–442. ISSN: 00034851. URL: `http://www.jstor.org/stable/2235740`.

[66] George Karypis and Vipin Kumar. “A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs”. In: *SIAM Journal on Scientific Computing* 20.1 (1999), pp. 359–392.

[67] George Karypis and Vipin Kumar. “Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs”. In: *SIAM Review* 41.2 (1999), pp. 278–300.

[68] D.E. Keyes et al. *Multiphysics Simulations: Challenges and Opportunities.* Tech. Rep. ANL/MCS-TM-321. 2011.

[69] Robert C. Kirby. “Algorithm 839: FIAT, a New Paradigm for Computing Finite Element Basis Functions”. In: *ACM Transactions on Mathematical Software* 30.4 (2004), pp. 502–516. DOI: 10.1145/1039813.1039820.

[70] A. Knüpfer et al. “Tools for High Performance Computing”. In: ed. by M. Resch et al. Springer, 2008. Chap. The Vampir Performance Analysis Tool-Set, pp. 139–155. URL: www.vampir.eu.

[71] A. Knüpfer et al. “Score-P - A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU and Vampir”. In: *Proceedings of 5th Parallel Tools Workshop*. 2012, pp. 79–91.

[72] Jonas Kratzke and Vincent Heuveline. “An analytically solvable benchmark problem for fluid-structure interaction with uncertain parameters”. In: *Preprint Series of the Engineering Mathematics and Computing Lab* 0.2 (2016). ISSN: 2191-0693. DOI: 10.11588/emclpp.2016.2.34579. URL: https://journals.ub.uni-heidelberg.de/index.php/emcl-pp/article/view/34579.

[73] Jonas Kratzke, Michael Schick, and Vincent Heuveline. “Fluid-Structure Interaction Simulation of an Aortic Phantom with Uncertain Young’s Modulus Using the Polynomial Chaos Expansion”. In: *Applied Mechanics and Materials*. Vol. 807. Trans Tech Publ. 2015, pp. 34–44.

[74] Jonas Kratzke et al. “In vitro flow assessment: From PC-MRI to computational fluid dynamics including fluid-structure interaction”. In: *SPIE Medical Imaging*. International Society for Optics and Photonics. 2016, pp. 97835C–97835C.

[75] Ethan J Kubatko, Benjamin A Yeager, and Ashley L Maggi. “New computationally efficient quadrature formulas for pyramidal elements”. In: *Finite Elements in Analysis and Design* 65 (2013), pp. 63–75.

[76] T. Lagarde, A. Piacentini, and O. Thual. “A new representation of data-assimilation methods: The PALM flow-charting approach”. In: *Q. J. R. Meteorol. Soc.* 127 (2001), pp. 189–207.

[77] O. Le Maître and O.M. Knio. *Spectral Methods for Uncertainty Quantification: With Applications to Computational Fluid Dynamics*. Scientific Computation. Springer Netherlands, 2010. ISBN: 9789048135202. URL: https://books.google.de/books?id=lOFCquL6SxYC.

[78] Jan Mayer. “A multilevel crout ilu preconditioner with pivoting and row permutation”. In: *Numerical Linear Algebra with Applications* 14.10 (2007), pp. 771–789.

[79] Jan Mayer. “Symmetric permutations for i-matrices to delay and avoid small pivots during factorization”. In: *SIAM J. Sci. Comput.* 30.2 (Mar. 2008), pp. 982–996.

[80] J. Mellor-Crummey, R. Fowler, and G. MarG. Marin. Tallent. “HPCView: A tool for Top-down Analysis of Node Performance”. In: *Supercomputing* 23.1 (2002), pp. 81–104.

[81] *MPI: A Message-Passing Interface Standard, Version 3.0.* Message Passing Interface Forum, 2012.

[82] MSC Software Corporation. *NASTRAN Web page*. 2017. URL: http://www.mscsoftware.com/de/product/msc-nastran.

[83] C. Niethammer et al., eds. *Tools for High Performance Computing 2014*. Springer, 2015.

[84] Fabio Nobile, Raúl Tempone, and Clayton G Webster. “A sparse grid stochastic collocation method for partial differential equations with random input data”. In: *SIAM Journal on Numerical Analysis* 46.5 (2008), pp. 2309–2345.

[85] Fabio Nobile, Raul Tempone, and Clayton G Webster. “An anisotropic sparse grid stochastic collocation method for partial differential equations with random input data”. In: *SIAM Journal on Numerical Analysis* 46.5 (2008), pp. 2411–2442.

[86] nVidia. *CUBLAS Library User Guide.* v5.0. nVidia. Oct. 2012. URL: `http://docs.nvidia.com/cublas/index.html`.

[87] NVIDIA Corporation. *CUDA Toolkit Documentation v6.5.* 2014.

[88] OpenMP Architecture Review Board. *OpenMP Application Program Interface.* 2013.

[89] A. Piacentini and the PALM Group. “PALM: A Dynamic Parallel Coupler”. In: *Lecture Notes in Computer Science* 2565 (2003), pp. 479–492.

[90] V. Pillet, J. LabJ. Labarta. Cortes, and S. Girona. “Paraver: A Tool to Visualize and Analyse Parallel Code”. In: *CEPBA/UPC Report No. RR-95/03* (1995).

[91] FEniCS Project. *FEnics Web page.* 2017. URL: `https://fenicsproject.org/`.

[92] J. Reid. “Co-arrays in the next Fortran Standard”. In: *ISO/IEC JTC1/SC22/WG5 N1824* (2010).

[93] J. E. Roman et al. *SLEPc Users Manual.* Tech. rep. DSIC-II/24/02 - Revision 3.7. D. Sistemes Informàtics i Computació, Universitat Politècnica de València, 2016.

[94] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* 2nd ed. Society for Industrial and Applied Mathematics, 2003. ISBN: 0898715342. DOI: `10.2113/gsjfr.6.1.30`. arXiv: `0806.3802`. URL: `http://www.stanford.edu/class/cme324/saad.pdf%20http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1231631%7B%5C%%7D5Cnhttp://www.stanford.edu/class/cme324/saad.pdf`.

[95] Alfred Schmidt and Kunibert G. Siebert. *Design of adaptive finite element software. The finite element toolbox ALBERTA. With CD-ROM.* English. Berlin: Springer, 2005, pp. xii + 315. ISBN: 3-540-22842-X/hbk.

[96] N. Schoch. “Towards Cognition-Guided Patient-Specific Numerical Simulation for Cardiac Surgery Assistance (PhD Thesis)”. In: *Heidelberg Document Server HeiDOK* (2017). DOI: `10.11588/heidok.00022579`.

[97] N. Schoch and V. Heuveline. “Towards an Intelligent Framework for Personalized Simulation-enhanced Surgery Assistance: Linking a Simulation Ontology to a Reinforcement Learning Algorithm for Calibration of Numerical Simulations”. In: *Preprint Series of the Engineering Mathematics and Computing Lab* 0.5 (2017).

[98] N. Schoch and V. Heuveline. “Towards Cognition-Guided Patient-Specific FEM-based Cardiac Surgery Simulation”. In: *Springer's LNCS Lecture Notes in Computer Science, Proceedings of FIMH 2017* 10263 (2017), pp. 115–126. DOI: `10.1007/978-3-319-59448-4_12`.

[99] N. Schoch and F. Kissler. “HiFlow^3 Tutorial: Elasticity Tutorial for Soft Tissue Simulation”. In: (2015).

[100] N. Schoch et al. “High Performance Computing for Cognition-Guided Cardiac Surgery: Soft Tissue Simulation for Mitral Valve Reconstruction in Knowledge-based Surgery Assistance”. In: *Modeling, Simulation and Optimization of Complex Processes, In: Proc. High Performance Scientific Computing (HPSC) 2015* (2015).

[101] N. Schoch et al. “Comprehensive patient-specific information preprocessing for cardiac surgery simulations”. In: *Int J CARS* 11(6) (2016), pp. 1051–1059. DOI: 10.1007/s11548-016-1397-0.

[102] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit.* 4th ed. Kitware, 2006. ISBN: 978-1-930934-19-1.

[103] A. Segal, M. ur Rehman, and C. Vuik. “Preconditioners for Incompressible Navier-Stokes Solvers”. In: *Numerical Mathematics: Theory, Methods and Applications* 3.3 (2010), pp. 245–275. ISSN: 10048979. DOI: 10.4208/nmtma.2010.33.1. URL: http://www.global-sci.org/nmtma/readabs.php?vol=3%7B%5C&%7Dno=3%7B%5C&%7Ddoc=245%7B%5C&%7Dyear=2010%7B%5C&%7Dppage=275.

[104] S.S. Shende and A.D. Mallony. “The TAU Parallel Performance System”. In: *J. High PJ. . Comput. Appl.* 20.2 (2006), pp. 287–311.

[105] Dassault Systèmes Simulia. *Abaqus Web Page.* 2017. URL: https://www.3ds.com/de/produkte-und-services/simulia/produkte/abaqus/aktuelle-version/.

[106] Ralph C Smith. *Uncertainty quantification: theory, implementation, and applications.* Vol. 12. Philadelphia, PA, USA: SIAM Computational Science & Engineering Series, 2013.

[107] Chen Song and Vincent Heuveline. “Multilevel preconditioner of Polynomial Chaos Method for quantifying uncertainties in a blood pump”. In: *International Conference on Uncertainty Quantification in Computational Sciences and Engineering (UNCECOMP), Greece, 2017.* Scopus, Elsevier, 2017.

[108] Timothy John Sullivan. *Introduction to uncertainty quantification.* Vol. 63. Cham: Springer, 2015.

[109] S. Suwelack et al. “The Medical Simulation Markup Language – Simplifying the Biomechanical Modeling Workflow”. In: *Journal on Studies in Health Technology and Informatics* 196 (2014), pp. 394–400.

[110] The HDF Group. *Hierarchical Data Format, version 5.* http://www.hdfgroup.org/HDF5/. 1997-2017.

[111] *The OpenACC Application Programming Interface, Version 2.5.* OpenACC-Standard.org, 2015.

[112] *The OpenCL Specification, Version 2.2.* Khronos OpenCL Working Group, 2016.

[113] Stefan Turek and Jaroslav Hron. “Proposal for numerical benchmarking of fluid-structure interaction between an elastic object and laminar incompressible flow”. In: *Lecture notes in computational science and engineering* 53 (2006), p. 371.

[114] Stefan Turek et al. “Numerical benchmarking of fluid-structure interaction: A comparison of different discretization and solution approaches”. In: *Fluid Structure Interaction II.* Springer, 2011, pp. 413–424.

[115] *UPC Language Specifications Version 1.3.* UPC Consortium, 2013.

[116] S. Wang, H. Chen, and W. Shi. “SPAN: A software power analyzer for multicore computer systems”. In: *Sust. Comput. Inform. Sys.* (2011).

[117] Norbert Wiener. “The Homogeneous Chaos”. In: *American Journal of Mathematics* 60.4 (1938), pp. 897–936. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2371268.

[118] Christian Wieners. “Conforming discretizations on tetrahedrons, pyramids, prisms and hexahedrons”. In: *Preprint, University of Stuttgart* (1997).

[119] M. Wlotzka. “Parallel Numerical Methods for Model Coupling in Nutrient Cycle Simulations”. PhD thesis. Ruprecht-Karls-Universität Heidelberg, 2017. DOI: `10.11588/heidok.00023544`. URL: `http://www.ub.uni-heidelberg.de/archiv/23544`.

[120] M. Wlotzka and V. Heuveline. “A parallel solution scheme for multiphysics evolution problems using OpenPALM”. In: *EMCL Prepr. Ser.* 1 (2014). DOI: `10.11588/emclpp.2014.01.13758`.

[121] M. Wlotzka and V. Heuveline. “Energy-aware mixed precision iterative refinement for linear systems on GPU-accelerated multi-node HPC clusters”. In: *Proceedings of GI/ITG Workshop Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS)*. 2015.

[122] M. Wlotzka and V. Heuveline. “An energy-efficient parallel multigrid method for multi-core CPU platforms and HPC clusters”. In: *EMCL Prepr. Ser.* 3 (2017).

[123] M. Wlotzka and V. Heuveline. “Energy-efficient multigrid smoothers and grid transfer operators on multi-core and GPU clusters”. In: *J. Parallel Distrib. Comput.* 100 (2017), pp. 181–192. DOI: `j.jpdc.2016.05.006`.

[124] M. Wlotzka et al. “ICT-Energy Concepts for Energy Efficiency and Sustainability”. In: ed. by G. Fagas et al. InTech, 2017. Chap. Energy-Aware High Performance Computing. DOI: `10.5772/66404`.

[125] M. Wlotzka et al. “New features for advanced dynamic parallel communication routines in OpenPALM: Algorithms and documentation”. In: *EMCL Prepr. Ser.* 4 (2017).

[126] www.green500.org. *The Green500 List.* June 2017.

[127] www.top500.org. *TOP500.* June 2017.

[128] Dongbin Xiu and George E Karniadakis. “The Wiener-Askey polynomial chaos for stochastic differential equations”. In: *SIAM Journal on Scientific Computing* 24.2 (2002), pp. 614–644.

[129] I. Zhukov et al. “Tools for High Performance Computating 2014”. In: ed. by C. Niethammer et al. Springer, 2015. Chap. Scalasca v2: Back to the Future, pp. 1–24.

$\frac{\partial u}{\partial t} = \nu\Delta u + (u \cdot \nabla) u + \frac{1}{\rho}\nabla p$
∂u
∂t