HiFlow^3 – A Flexible and Hardware-Aware Parallel Finite Element Package

H. Anzt, W. Augustin, M. Baumann, H. Bockelmann,
T. Gengenbach, T. Hahn, V. Heuveline, E. Ketelaer, D.
Lukarski, A. Otzen, S. Ritterbusch, B. Rocker, S. Ronnås,
M. Schick, C. Subramanian, J.-P. Weiss, F. Wilhelm

www.emcl.kit.edu

HiFlow³ – A Flexible and Hardware-Aware Parallel Finite Element Package

Hartwig Anzt, Werner Augustin, Martin Baumann, Hendryk Bockelmann,
Thomas Gengenbach, Tobias Hahn, Vincent Heuveline, Eva Ketelaer,
Dimitar Lukarski, Andrea Otzen, Sebastian Ritterbusch, Björn Rocker,
Staffan Ronnås, Michael Schick, Chandramowli Subramanian,
Jan-Philipp Weiss, Florian Wilhelm

Engineering Mathematics and Computing Lab (EMCL)
Karlsruhe Institute of Technology, Germany
{Hartwig.Anzt,Werner.Augustin,Martin.Baumann,Hendryk.Bockelmann,
Thomas.Gengenbach,Tobias.Hahn,Vincent.Heuveline,Eva.Ketelaer,
Dimitar.Lukarski,Andrea.Otzen,Sebastian.Ritterbusch,Bjoern.Rocker,
Staffan.Ronnas,Michael.Schick,Chandramowli.Subramanian,
Jan-Philipp.Weiss,Florian.Wilhelm}@kit.edu

Abstract. This paper details the concept and implementation of the parallel finite element software package HiFlow³. HiFlow³ is driven by application requirements and aims at the solution of large-scale problems obtained by means of the finite element method for partial differential equations. By utilizing object-oriented concepts and the full capabilities of C++ the HiFlow³ project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism – as the basis for high performance simulations on modern computing systems – is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends. Modern numerical schemes in HiFlow³ are built on top of both levels of parallelism. This paper describes the project, its concept, and application scenarios in detail and outlines our hardware-aware cross-platform portable approach that benefits from various emerging technologies like GPU acceleration in a unified and user-friendly manner.

Keywords: Parallel Finite Element Software, High Performance Computing, Numerical Simulation, Hardware-aware Computing, Multi-core, GPGPU, hp-adaptive FEM

1 Introduction

Numerical simulation based on finite element discretizations of partial differential equations describing physical processes is a powerful means for gaining deeper

scientific insight. It aims at delivering accurate simulation results, allowing to replace costly experimentation, enabling comprehensive parameter studies, and allowing for optimization of production processes and development cycles. Complexity of geometries and models and the demands for high accuracy typically result in a huge number of modeled degrees of freedom (DoF) with complex couplings. In this setting short solving run times require both powerful hardware and efficient parallel numerical methods compliant with multi-level parallelism and hierarchical memory systems of modern computing platforms. Due to the shift towards the multi- and many-core era both aspects need to be considered in the big picture following the approach of hardware-aware computing. The wide variety of available platforms and the associated challenges of software portability result in a strong need for portable and self-adaptable concepts providing high performance and moderate programming effort for domain specialists and software users.

Scientific computing and numerical simulations are very important tasks in research, engineering and development. Mathematics combined with new software design dedicated to state-of-the-art hardware technologies result in high complexity but also break new grounds in the area of scientific computing. HiFlow^3 is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). HiFlow^3 is based on finite element methods (FEM). Its implementation utilizes objected-oriented and template concepts in C++. It is actively developed by a team of seventeen people at the *Engineering Mathematics and Computing Lab* (EMCL) at Karlsruhe Institute of Technology (KIT). HiFlow^3 is based on ten years of experience and development and combines state-of-the-art programming techniques with aspects of high performance computing and modern hardware platforms. Due to multi-core-shift and new paradigms in parallel computing the whole project is currently re-engineered and re-structured in order to fully utilize recent hardware technologies, the associated potential of hybrid computing platforms, but also to account for bottlenecks in the communication infrastructure.

The vision of HiFlow^3 is to synergize numerical simulation, numerical optimization, and high performance computing (HPC) for the solution of complex problems with high relevance in science and impact on society for instance in the fields of meteorology, climate prediction, energy research, medical engineering, and bioinformatics. All these problems have in common a high complexity in interaction of physical and chemical effects in the system under study, a mathematical model with strong couplings, and large requirements of computer resources such as memory and CPU speed. Due to the wide variety of models, assumptions, requirements, and purposes of the problem settings usually different aspects need to be considered in the simulation cycles. To handle this, HiFlow^3 aims at maximal flexibility by its modular approach.

The paper is structured as follows. First, the design of HiFlow^3 is outlined and the modules and their respective challenges are introduced. Then the main modules Mesh, DoF/FEM and Linear Algebra are described in details and the

concept for assembly of the system matrices is presented. Finally, a show case demonstrates the capabilities and the workflow of the HiFlow^3 parallel finite element package.

## 2 Motivation, Concepts and Structure of HiFlow^3

2.1 Fields of Application

A typical application incorporating the full complexity of physical and mathematical modeling is the *United Airways* project [28]. Within an interdisciplinary framework its objective is the simulation of the full human respiratory tract including complex and time-dependent geometries with different length scales, turbulence, fluid-structure interaction (e.g. fine hairs and mucus), and effects due to temperature and moisture variations. Due to the complex interaction of all effects it is a great challenge to construct robust numerical methods giving accurate simulation results within a moderate time. As a further vision, interactive and real time simulations should give medical advice on personalized data during examination and surgery. Another example is the project *Goal Oriented Adaptivity for Tropical Cyclones* (see 8.1). Many weather phenomena such as the development and motion of Tropical Cyclones are influenced by processes on scales that may range from hundreds of meters to thousands of kilometers. Due to computational requirements and memory limitations it is often impossible to resolve all relevant scales on globally refined meshes. Application dependent requirements and varying demands with respect to the quantity which is of interest lead to specific treatment of modern adaptive numerical methods that need to be adapted to the associated problem settings.

2.2 Flexibility

The conceptual goal of HiFlow^3 is to be a flexible multi-purpose software package that can be adapted to most user scenarios. To this end, the core of HiFlow^3 is divided into three main modules: Mesh, DoF/FEM and Linear Algebra; see Figure 1. These three core modules – further extended by a suite of other modules – are essential for the solution procedure based on finite element methods for partial differential equations. They offer the main functionality for mapping mathematical problem scenarios into parallel software.

Other building blocks for HiFlow^3 are routines for numerical integration of element integrals, assembling of system matrices, inclusion of boundary conditions, setting up nonlinear and linear solvers, providing output data for visualization of solutions, error estimators, and others. These routines are added to the core modules. This modular structure ensures the flexibility to employ the library to solve a wide variety of problems. Another aspect of modularity is to enable extensibility of the HiFlow^3 package. There exist two basic ways to extend HiFlow^3. Firstly further modules and methods for user-defined applications such as modules for stochastic finite elements or chemical reactions can
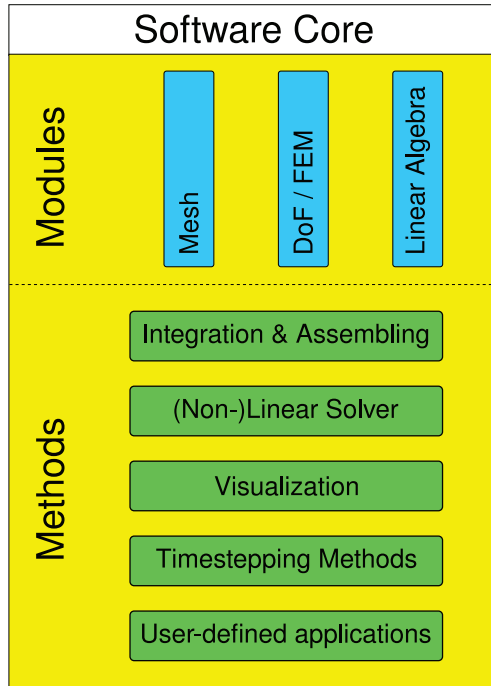
Fig. 1. Structure of the HiFlow^3 core divided into modules and methods.

be added to the HiFlow^3 core. Secondly existing modules and methods can be augmented by adding further functionality. The possibility to extend HiFlow^3 by static or dynamic polymorphism provides a basis for future development. For instance in the mesh module a new implementation inherited from the abstract mesh base class can be implemented, the DoF/FEM module can be extended by further finite elements spaces such as $H(div)$ or $H(curl)$, and in the Linear Algebra module more data structures for matrices could be added – all in the sense of dynamic polymorphism. Available linear algebra data structure can be extended through the use of templates. The basic principle of HiFlow^3 is the aim for a generic implementation in order to be able to use different modules and methods for a variety of problems following the approach of a multi-purpose software. Of course there is a trade-off when dealing with problems that need particular implementation and highly-specialized methods for efficient, robust and accurate simulations.

2.3 Performance, Parallelism, Emerging Technologies

Another object of HiFlow^3 is the full utilization of all available resources on any platform – from large HPC systems to any stand-alone workstation or coprocessor-accelerated machine. To achieve this goal all three modules (Mesh, DoF/FEM, Linear Algebra) provide distributed data structures. The communication between different nodes and processors is realized by means of an MPI-layer. On the linear algebra level we use concepts of hardware-aware computing for acceleration of basic routines like local matrix-vector and vector-vector op-

erations. Here, the main challenge is to fully utilize the available computing power of emerging technologies like multi-core CPUs, graphics processing units (GPUs), multi-GPUs, and any other coprocessor-accelerated or heterogeneous platform. The advantage of the structure provided by the Linear Algebra module is the potential to build solvers without having detailed information on the underlying platform. All methods are layed out with respect to scalability in a generic sense. Of course, best performance conflicts with flexibility. In that case HiFlow[3] favors performance over flexibility – which could mean to offer a solver which is only efficient for a special problem. Another important issue is the usability and convenience. HiFlow[3] provides the user with a transparent structure enabling him to set up and get a simulation for his application running with minimal effort.

2.4 Hardware-aware Computing

The huge demand on fast and accurate simulation results for large-scale problems poses considerable challenges on the implementation on modern hardware. Supercomputers and emerging parallel hardware like GPUs offer impressive computing power in the range of Teraflop/s for desktop supercomputing up to Petaflop/s for cutting edge HPC machines. The major difficulty is the development of efficient parallel code that is scalable with respect to exponentially increasing core counts and portable across a wide range of computing platforms. With the advent of the many-core era new platforms like the GPUs, the Cell BE, FPGA-based systems like Convey's Hybrid Core Computer or Intel's tiled architectures (Polaris, Rock Creek) or the Larrabee incarnation Knight Ferry have emerged [21, 37, 40, 44, 49, 54, 55]. These technologies come along with impressive capabilities but different programming approaches, different processing models, and different tool chains. Moreover, all numerical methods need to be compliant with multiple levels of parallelism within hybrid systems and with hierarchical memory subsystems. Typically, manual tuning and parameter variation is necessary for optimal performance on a dedicated system. In this context, hardware-aware computing is a multi-disciplinary approach to identify the best combination of applications, physical models, numerical schemes, parallel algorithms, and platform-specific implementations that is giving the fastest and most accurate results on a particular platform [17]. Design goals are maximal flexibility with respect to software capabilities, mathematical quality, efficient resource utilization, performance, and portability of concepts. Hardware-aware computing not only comprises highly-optimized platform-specific implementations, design of communication-optimized data structures, and maximizing data reuse by temporal and spatial blocking techniques it also relies on the choice and development of adequate mathematical models that express multilevel-parallelism and scalability up to a huge number of cores. The models need to be adapted to a wide range of platforms and programming environments. Data and problem decomposition on heterogeneous platforms is a further objective. Memory transfer reduction strategies are an important concept for facing the bottlenecks of current platforms. Since hardware reality and mathematical cognition give rise

to different implementation concepts, hardware-aware computing means also to find a balance between the associated opponent guiding lines while keeping the best mathematical quality (e.g. optimal work complexity, convergence order, error reduction and control, accuracy, robustness, efficiency). All solutions need to be designed in a reliable, robust and future-proof context. The goal is not to design isolated solutions for particular configurations but to develop methodologies and concepts that preferably apply to a wide range of problem classes and architectures or that can be easily extended or adapted. The HiFlow[3] project has implemented related concepts in the framework of the local multi-platform LAtoolbox [33, 35] within the Linear Algebra module.

2.5   Object-oriented Programming Approach

In order to be able to handle the complexity of solving such a great variety of problems HiFlow[3] is implemented in C++. The object-oriented paradigm of C++ with its support for polymorphism and inheritance allows all involved developers to contribute with their individual specialized knowledge. End-user and researcher can handle opaque and problem-oriented objects, computer scientists and hardware specialists produce optimized low-level implementations, while mathematicians provide new numerical solver algorithms. Furthermore, the intensive use of the template features of C++ gives the compiler a lot of opportunities for compile-time optimizations. Our modular approach with utilization of the object-oriented concepts of C++ is the best way to handle the complexity of such a sophisticated software project with many contributors. The approaches of data abstraction, encapsulation and clear interfaces between various modules not only ease maintainability of the code but also enable reusability of specific parts for the extension of functionalities and features. An additional benefit of this approach is the possibility to use parts of the software as standalone libraries or modules for other projects like e.g. the LAtoolbox. In such a way, our project provides building blocks for the development of new solvers and the extension to new problem domains.

2.6   Modeling and Workflow

Many physical problems are modeled by means of partial differential equations. Typical examples are the Poisson equation (e.g. for electric or gravity fields), stationary or time-dependent convection diffusion equations (e.g. transfer of particles or energy), and the Navier-Stokes equation (e.g. fluid flow around an obstacle, simulation of a tropical storm, air flow in the lung). A classical approach for solving associated problems relies on discretization of the equations in the domain of interest by means of finite element methods [14, 15, 19] on spatial grids. Searching for approximations in finite dimensional function spaces by means of weak formulations results in linear or nonlinear systems of equations that are typically solved by Newton-type methods, time-stepping schemes and iterative solvers. Coupling between degrees of freedom is mainly expressed by nearest neighbor interaction with a high degree of locality. Due to the required accuracy

of the discrete approximations large numbers of degrees of freedom are required resulting in huge and sparse matrices with bad condition numbers in general.

For the finite element solution procedure the following steps have to be performed for mapping a problem modeled by PDEs to HiFlow[3]. First, the physical problem has to be expressed in terms of PDEs in an adequate way. The next theoretical step is to derive a weak formulation in a variational sense. In a pre-processing step the domain of interest is discretized by means of a finite element triangulation and converted to a format readable by the mesh module. Once the spatial grid is read in it can be adjusted e.g. by means of a refinement procedure. Then, a problem-adapted finite element space with appropriate ansatz functions has to be chosen. For the Navier-Stokes equations Taylor-Hood elements are an appropriate choice. Once these two informations are provided the degrees of freedom can be determined first locally (this corresponds to the FEM part) on each cell, and then globally for the whole mesh (this corresponds to the DoF part). In the case of distributed memory platforms the DoF-partitioner is used to create a global degree of freedom (DoF) numbering throughout the whole computational domain by only using its local information and MPI communication across the nodes. Once all DoFs are identified the matrix can be assembled by local integration over all elements within the triangulation. Data structures for matrices and vectors are provided by the Linear Algebra module. In case of nonlinear problems a Newton method is combined with a linear solver for the problem solution. Finally, output in either sequential or parallel format is provided for the visualization. This is an important aspect of the simulation cycle for an assessment and exploration of simulation data. There exist several visualization tools which can be used for this postprocessing step. Within HiFlow[3] there are e.g. back ends for HiVision [36] and Paraview [31].

2.7 Modules in HiFlow[3]

The implementation of a parallel and flexible finite element software package aiming at general purpose deployment and portability across a wide range of platforms comes along with various challenges in the respective modules.

The mesh module is responsible for the interaction with the discretized computational domain. It is designed primarily for unstructured meshes, and can handle several different cell types in different dimensions. In order to support adaptive algorithms, the Mesh module can work with both nonconforming meshes and meshes containing cells of mixed types. Additionally, it provides functionality to refine and coarsen a mesh, both globally and locally, and to retrieve the history of these modifications. Furthermore, the use of meshes distributed over several processors is supported, in order to reduce the memory requirements for large-scale simulations running on a high-performance cluster. For this purpose, the module also contains functionality for dealing with partitioning and communication of the mesh data. The link between the local mesh and its neighbors is a layer of ghost cells that are shared between each pair of processors.

The DoF/FEM module treats the problem of numbering and interpolating the degrees of freedom. In the first step the local DoFs of each cell for a chosen finite element space need to be determined. Then the local DoFs of each cell have to be numbered globally. We are using a generic approach handling all cell types and finite element spaces. In the case of distributed data structure the DoF module further handles the neighborhood relations between cells which are distributed on different processors. To this end, the module takes advantage of ghost cells created by the mesh module. Another task is to distribute the DoFs in a balanced way such that each processor is responsible for almost the same number of degrees of freedom. Here, different complexities and work load contributions need to be considered since each cell might use different ansatz functions.

The Linear Algebra module provides distributed and platform-optimized data structures for vectors and matrices as well as implementations for corresponding matrix-vector and vector-vector operations. Due to localized interactions between finite element basis functions resulting matrices are typically sparse. For that purpose, adequate sparse matrix formats are provided. The structure of the Linear Algebra module is based on two levels: the inter-node level communication layer utilizes MPI and the intra-node communication and computation model is based on platform-specific programming environments that are accessed by generic and unified interfaces. In order to reduce the cost of communication on the upper layer the Linear Algebra module takes advantage of ghost cells provided by the Mesh module (and also used in the DoF module). On the local intra-node layer the focus is set to methods of hardware-aware computing aiming at best choice of platform-specific implementations. The concept of the Linear Algebra module allows to compute local matrix vector operations on hybrid systems and accelerators such as multi-core-CPUs, GPUs, and OpenCL-enabled devices. By providing unified interfaces with back ends to different platforms and accelerators the module allows seamless integration of various numerical libraries and devices. The user is freed from any particular hardware knowledge – the final decision on platform and chosen implementation is taken at run time. Naturally, scalability plays an important role in this module.

2.8   Short Survey on Existing FEM-Software

There exist reams of other non-commercial and commercial parallel finite element software packages beside HiFlow[3]. We do not want to give an complete and exhaustive overview on state-of-the art finite element software but we want to mention in alphabetical order at least a few of them. Alberta [48] 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. COMSOL Multiphysics [20] is a commercial simulation software environment. It includes the definition of the geometry, meshing, specifying the physics, solving as well as the visualization.

For good usability many common problem types are predefined as templates. The C++ programming library deal.II [8, 9] is an Open Source project and uses adaptive finite elements for solving partial differential equations. It supports one, two, and three space dimensions, and is restricted to intervals, quadrilaterals and hexahedrons. DUNE [24] 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. FEAST [26] is a software package designed to solve FE problems. For better floating point performance and memory utilization it can use different co-processors platforms like GPU and Cell BE. It is based on a specific structure grid which lead to sparse banded matrices. Fluent [4] is a commercial Computation Fluid Dynamics (CFD) software package provided from Ansys Inc. It is one of the most mature software on the CFD market. NASTRAN 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 old legacy languages. UG [51] stands for unstructured grids and is a software tool for numerical solutions of partial differential equations. To achieve numerical efficiency UG uses adaptivity on the grid level, multi-grid methods and parallelism in form of distributed dynamic data. In most cases based on the FE discretization, useful packages for solving the obtained linear system are PETSc [5–7] and Trilinos [32] 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.

3   Mesh Module

In the finite element method, the computational domain is approximated by a mesh, which allows it to be represented in a discrete form on a computer. The Mesh module provides support for input and output from files, iteration over the entities in the mesh, refinement and coarsening of cells, and communication between meshes existing on different processors.

This section first describes some basic mathematical ideas concerning the geometry and topology of meshes. This is followed by an explanation of the abstract interfaces that the module provides for interacting with meshes and mesh entities, as well as the first implementation of these interfaces, which is provided in the module. The three last sub-sections deal with three aspects of the Mesh module in more detail: the generic description of cell types, the support for refinement and coarsening, and the functionality for managing distributed meshes.

3.1   Geometry and Topology

In the context of HiFlow[3], we consider a mesh as being a partitioning of a domain into cells of a limited number of predefined shapes. The list of possible cell types is extensible, and there are already implementations for triangles and quadrilaterals in 2d, and tetrahedrons and hexahedrons in 3d. These shapes and the vertices are referred to as entities in the following. The cells are non-overlapping in the sense that their mutual intersections are lower-dimensional shapes, but the mesh does not need to be conforming, which means that the intersection of two neighboring cells does not have to be a sub-entity of both cells.

The geometry of the mesh is considered separately from its topology, in order to simplify the development of simulations involving for instance moving meshes. Currently, the geometry representation in the Mesh module is simply an assignment of some coordinates to each vertex. This is the natural choice, as the vertex coordinates is the basic information provided by all mesh generators.

The topology of the mesh is described via the incidence relations (also called connectivities by some authors) between its entities. For the computation and representation of incidence relations, we closely follow the approach described in [39]. As in that paper, the incidence relation between entities of dimension d_1 and those of dimension d_2 in a mesh is denoted by $(d_1 \rightarrow d_2)$. What this incidence relation means depends on whether d_1 is larger than, smaller than, or equal to, d_2.

- $d_1 > d_2$: the d_2-entities contained in each d_1-entity (e.g. $3 \rightarrow 1$ the edges of a 3d-cell).
- $d_1 < d_2$: the d_2-entities containing each d_1-entity (e.g. $1 \rightarrow 3$ the cells sharing an edge).
- $d_1 = d_2$, d_1, $d_2 > 0$: the d_2-entities sharing at least one vertex with each d_1-entity

Starting from the input connectivity $D \rightarrow 0$, where D is the topological dimension of the cells of the mesh, all connectivities can be computed through the combination of three basic algorithms:

- `build`: computes $d \rightarrow 0$ and $D \rightarrow d$ from $D \rightarrow D$ and $D \rightarrow 0$, for $0 < d < D$.
- `transpose`: computes $d_1 \rightarrow d_2$ from $d_2 \rightarrow d_1$, for $d_1 < d_2$
- `intersect`: computes $d_1 \rightarrow d_2$ from $d_1 \rightarrow d_3$ and $d_3 \rightarrow d_2$, for $d_1 \geq d_2$. $d_3 = 0$ if $d_1, d_2 > 0$

These algorithms, together with the algorithm that combines them to compute a given connectivity is described in [39]. The special case $0 \rightarrow 0$ has been omitted since it has not been implemented in HiFlow[3]).

3.2   Abstract Interface Classes

The use of abstract interfaces is the cornerstone of modular programming, which enables a decoupling between different parts of a program, allowing each to vary

separately. The Mesh module provides an abstract interface to its services via a collection of classes with separate responsibilities.

The `Mesh` class represents a computational mesh. It is an abstract base class, and hence cannot be instantiated. Instead, different concrete mesh implementations can be created, possibly with different performance characteristics, and different levels of generality. The use of dynamic polymorphism instead of static polymorphism based on templates, which is the current trend in scientific C++ programming, is motivated through the higher degree of flexibility (the implementation can be chosen at runtime), and the simpler code that results. The price to pay is the overhead of virtual function calls, which the authors believe to be small for typical use cases. As a comparison, an example of the use of static polymorphism for mesh handling can be found in the dune-grid module of the DUNE project [24].

An important characteristic of the `Mesh` class is that its public interface contains almost exclusively `const` functions, meaning that a `Mesh` can be considered to be an immutable object. This design choice was based on the considerable simplifications that it allows when reasoning about the validity of the state of the `Mesh` objects. The only time at which a `Mesh` can be modified is when it is constructed. This responsibility is given to implementations of the `MeshBuilder` interface, which in accordance to the Builder design pattern [27], lets the user build the mesh by providing its vertices and entities incrementally, before obtaining the finished `Mesh` object via the `build()` method.

An exception to the immutability of the `Mesh` object is the attributes mechanism. Similar to the `MeshFunction` class described in [39], the `Attribute` class provides a way for the user to associate named data of different types to the entities of the `Mesh`. This concept is also used to store results associated with different mesh operations, such as refinement.

The `Mesh` abstract base class represents an entire mesh. In many algorithms, however, it is useful to work with a local view of individual entities. This is provided through the `Entity` class, which uses the `Mesh` interface to obtain the data associated with a single entity of the mesh. Even though it is a concrete class and can be instantiated, it only depends on the `Mesh` interface, and can thus be used with all possible `Mesh` implementations.

Random access to the entities of a `Mesh` is provided through the function `Mesh::get_entity()`, but it is also possible to iterate over entities in two ways. Forward iteration over all entities of a given dimension in the `Mesh` is provided through the class `EntityIterator` and the functions `Mesh::begin()` and `Mesh::end()`. Iteration over the entities incident to a given entity is provided through the class `IncidentEntityIterator`, which is obtained from the pair of functions `Entity::begin_incident()` / `Entity::end_incident()` . Like the `Entity` class, the iterator classes are independent of the concrete `Mesh` implementation used. More complex types of iteration can be implemented through the use of the boost::Iterator framework [1], which these iterator classes support.

3.3 Mesh Implementation

There is at the moment one implementation of the interface defined by the `Mesh` class, which is provided by the `MeshDbView` and `RefinedMeshDbView` subclasses. The second class is derived from the first, and represents a mesh that is a refinement of another mesh. In addition to the data stored in `MeshDbView`, `RefinedMeshDbView` also has the refinement history associated to it, and overloads some of the functions to use this refinement history.

Most of the functionality is however provided by the `MeshDbView` class, which heavily depends on a third class, `MeshDatabase`. The `MeshDatabase` class manages a unique numbering for all entities over a set of meshes. A single `MeshDatabase` object is shared between all meshes in a refinement hierarchy, and between a mesh and its boundary mesh. In accordance with its name, a `MeshDbView` object represents a limited view of the `MeshDatabase`: either a level in the refinement hierarchy, or the boundary of a mesh.

The `MeshDatabase` class manages the entity-vertex and vertex-entity connectivities $(d \rightarrow 0)$ and $(0 \rightarrow d)$ for all existing entities. On demand, the `MeshDbView` class can compute and cache the restriction of these connectivities to the entities belonging to a particular mesh. It can also compute the other connectivities upon request.

The `MeshDatabase` class provides set semantics for its entities, meaning that if an entity is added several times, it only exists once and always receives the same id. This is implemented with the help of an additional structure, the `VertexSearchTable`, which makes it possible to search for a vertex by its coordinates. The underlying data structure is simply an array of the vertex id:s sorted by their distance to the origin. The search for a given vertex consists of first finding all vertices which are at the same distance from the origin (within a small tolerance), and then performing a linear search over these vertices. Although the worst-case efficiency of this structure is not optimal, it has proven to be quite fast when used together with typical simulation data.

The VertexSearchTable makes it easy and efficient to enforce set semantics for vertices. When a vertex is added to the `MeshDatabase`, a search is performed first to see whether it already exists. If so, the id of the existing vertex is returned instead of creating a new vertex. For the entities of dimension larger than 0, the corresponding search is performed using the $0 \rightarrow d$ connectivities, which, unlike the other connectivities, are sorted. Looking up an entity, specified by its vertex ids, consists in computing the intersection of the sets of entities connected to each of its vertices. If the entity does not exist, this intersection will be empty, and otherwise it will be simply the set containing the id of the sought entity.

Building new `MeshDbView` objects is done through the `MeshDbViewBuilder` class, which implements the interface defined by the `MeshBuilder` class. The `MeshDbViewBuilder` holds a reference to the `MeshDatabase` to which new entities should be added. The fact that the `MeshDatabase` provides set semantics for adding vertices and entities, makes the implementation of the `MeshDbView-Builder` extremely simple: the same vertex or entity can be added several times,

but will always receive the same id, which saves the `MeshDbViewBuilder` class from having to keep track of which entities have already been added.

3.4   Cell Types

The description of the different types of cells (lines, triangles, quadrilaterals, tetrahedrons and hexahedrons) is done through the CellType class hierarchy. The base class, CellType, stores information about the local $D \to d$ connectivities and the possible refinements of a cell type. By combining these two pieces of information, the connectivities of the refined sub-cells are also derived, which is the basis for the handling of non-conforming meshes. Each subclass of CellType simply implements some callback functions that return the specific information for that cell type, and it is only used for initialization.

A refinement is specified in three steps. Firstly, so-called "refined vertices" are added to the cell type. A refined vertex is simply the barycenter of a set of existing vertices. In a second step, one can then define the "sub-cells" by specifying the set of vertices (regular or refined) that it contains. Finally a refinement is defined as the set of sub-cells that should be produced by the refinement.

The connectivities for the sub-cells and its sub-entities are computed automatically using the CellType definitions of the sub-cells. The central idea is that "refined" entities are numbered consecutively, starting after the corresponding "regular" entities. In each cell type, cell 0 is the cell itself, and cells $(1,\dots)$ are the sub-cells that can be included in a refinement. Similarly if vertices $(0,\dots,N)$ are the vertices of cell 0 (i.e. the regular vertices), one can add refined vertices $(N+1,\dots)$, which can be used to define the sub-cells.

In this way, it is possible to compute the local connectivities between sub-cells, and extract all information necessary both for refinement and for dealing with non-conforming meshes. Numbering the entities consecutively makes it easier to deal with entities and sub-entities in a uniform way.

3.5   Refinement and Coarsening

Refinement and coarsening of a mesh is provided through the `Mesh::refine()` function. It takes as input an array indicating for each cell if it should be coarsened, refined (and if so, how), or left as it is. The function builds the refined mesh, and returns it to the caller.

In the `MeshDbView` implementation, `RefinedMeshDbView`, a specialized subclass, is used to represent refined meshes. This class keeps track of the refinement history, by storing an array of pointers to all ancestor meshes, as well as two attributes that indicate for each cell in which ancestor mesh the parent cell lives, and what its index is local to that mesh. This makes it possible to access the parent of a cell, even if one only has access to the refined mesh.

In addition to these two attributes, we have also found it useful to store the "sub-cell number", i.e. the index of the sub-cell in the parent's cell type. This information is used both for boundary extraction and handling of non-conforming meshes.
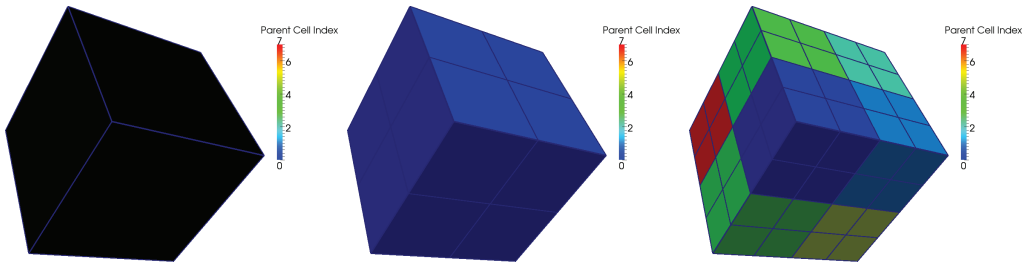
Fig. 2. Refinement of a cube with the parent cell index attribute.
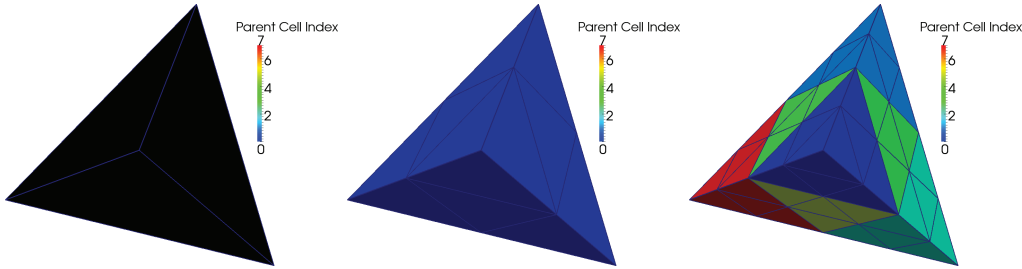
**Fig. 3.** Refinement of a tetrahedron with the parent cell index attribute.

During refinement, some cells can also be coarsened. This means that they are replaced by their parents. There is some ambiguity about what to do when one cell is marked to be coarsened, while another cell sharing the same parent (a sibling) is not marked to be coarsened. One could for instance either force a coarsening of all sibling cells, or require all sibling cells to be marked for coarsening, before it can take place. We follow the second path, by searching for "permitted ancestor" cells for all cells that are marked to be coarsened. A "permitted ancestor" is a cell in an ancestor mesh such that all its children in the current mesh are marked to be coarsened. If no "permitted ancestor" is found, the coarsening mark on the cell is removed, and it is left untouched. If one or several "permitted ancestors" are found, all their children are coarsened. This means that coarsening of large areas can be performed during one refinement step.

3.6   Distributed Meshes

The Mesh module provides the possibility to work with distributed meshes. In the current implementation, a distributed mesh is a set of `Mesh` objects, one on each process in the communicator group. The communication is handled by functionality external to the `Mesh` classes, which themselves are not aware that they are part of a larger, global mesh. The advantage of this is that all `Mesh` functions are local to a process and do not require communication, which facilitates their implementation and use. Code reuse is also possible, since the communication code works with the `Mesh` interface, and not the individual implementations.

In order to communicate between processes, it is necessary to know how the parts of the global mesh are connected. This is done via the `SharedVertexTable`
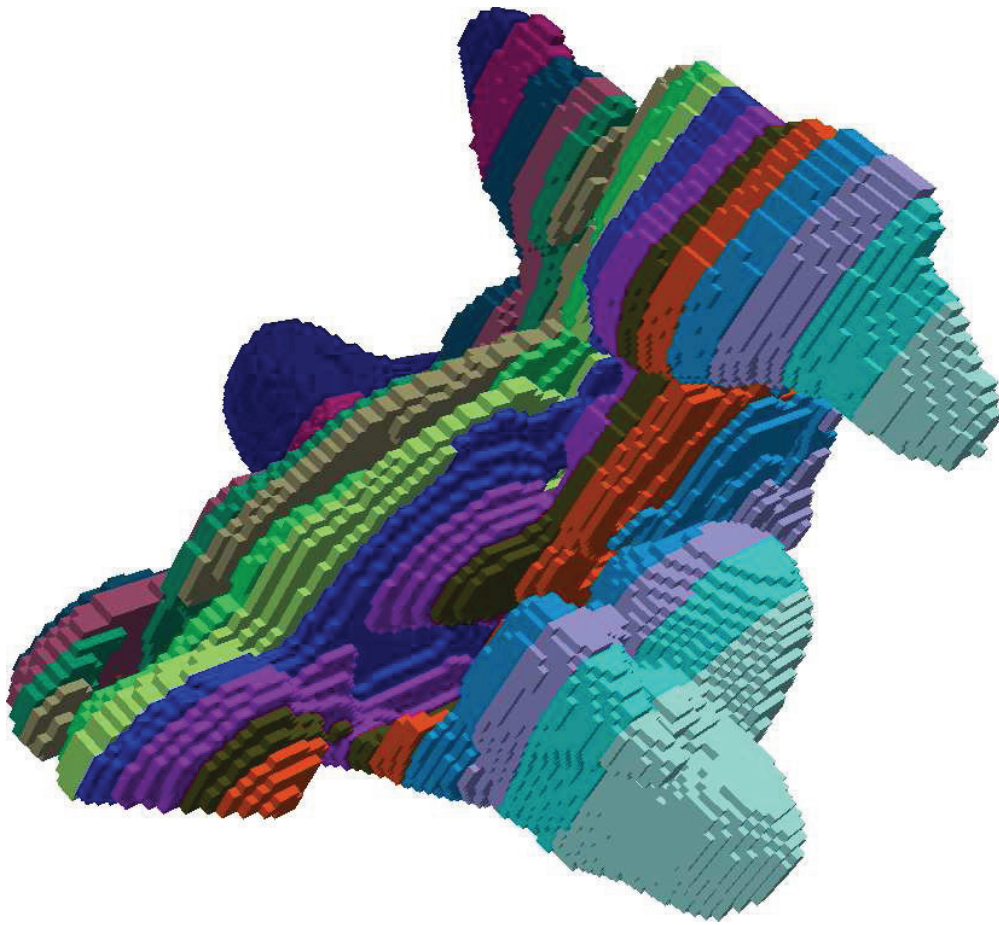
Fig. 4. A mesh of a human nose distributed in 16 stripes. One stripe plus one layer of ghost cells exists on each process.

class, which contains the information about which of the local vertices are shared with what other processes, and what the remote id is of each shared vertex. Having this information for vertices, it is possible to also identify shared entities of higher dimension.

If the mesh has been read in from a parallel VTK file [53], the information provided by the `SharedVertexTable` is not available, and thus has to be computed in a global communication. This is done by the `update_shared_vertex_table()` function which exchanges the coordinates of all vertices of a mesh with all other processes, looks up the id:s of the received vertex coordinates locally via the `MeshDatabase`, and communicates this information back to all other processes. This is a potentially expensive operation, but necessary in the case that no other information is available.

The communication of mesh entities is performed via a simple serialization procedure. A set of entities defined as an `EntityIterator` range can be packed into an `EntityPackage` object, which is then communicated. Two modes of communication, scatter and non-blocking point-to-point, have been implemented in the `MpiScatter` and `MpiNonBlockingPointToPoint` classes, respec-

tively. On the receiving end, the `EntityPackage` object is rebuilt, and can then be used together with a `MeshBuilder` to reconstruct the mesh on the remote process. Again, for the `MeshDbView` implementation, the set semantics of the `MeshDatabase` greatly simplify this procedure.

The computation and communication of ghost cells has been built on top of this framework. A higher-level function takes a local mesh on each process, and creates a new mesh containing one level of ghost cells from the neighboring processes. All cells are marked with attributes that indicate the owning process and the index of the cell on that process.

There is also support for computing the partitioning of a mesh, i.e. deciding how it is to be distributed over the processes. The GraphPartitioner abstract base class provides an interface, for which there are currently two implementations. One is provided through an interface to the well-known library METIS, and the second is a "naive" implementation which partitions based on the numbering of the cells. At the moment, these partitioners work on the local mesh only, and in practice one reads in the mesh on one process, computes the partition, and then distributes the parts to the other processes.

4 DoF/FEM Module

In the context of the finite element method solutions are expressed in terms of linear combinations of some chosen shape functions defined on mesh cells. The degrees of freedom (DoF) represent the finite number of parameters that define such a discrete function. The number of DoFs can easily count up for millions of unknowns and can typically be associated with locations which are distributed over the mesh. In the following, the DoF/FEM module will be described, representing the finite element ansatz (submodule FEM) and the challenging treatment of the degrees of freedom (submodule DoF).

4.1 FEM Submodule

This submodule is dedicated to represent a finite element ansatz in a generic way to ensure extensibility at low memory costs, but still to enable high performance assembly. The basic concept lies in defining three major classes `FEType`, `FEManager` and `FEInstance`. The first one, `FEType`, is an interface class and deals with representing a specific continuous or discontinuous finite element ansatz on a defined reference cell. For instance a Lagrangian element `FELagrange:FEType` is a specialized implementation and defines the necessary parameters, such as the polynomial degree. Also, since each ansatz has a specific number of degrees of freedom, these are initially defined on this cell and stored in a lexicographically order (see figure 5). Further, interfaces to compute the values of the shape functions, which indices correspond to the local cell numbering, etc. are provided. To add a new finite element ansatz, a new class needs to be derived from `FEType` and the few needed functions must be implemented. It is guaranteed that the new elements can be used all over the library, which makes extensibility easy.
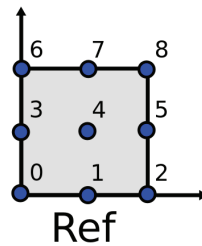
**Fig. 5.** Lexicographical numbering of the DoFs on the reference cell.

On every mesh cell and for every variable a specific finite element ansatz is prescribed. Therefore, a design pattern from software engineering known as "singleton" is implemented, which means that every ansatz which occurs more than once is represented and stored in only one element, managed by `FEInstance`. For establishing the mapping between a tuple of a mesh cell and a variable to their corresponding singleton, only *references* are stored. This mapping and all interfaces to out-of-module classes are managed by the `FEManager`. Especially, each mesh cell has a mapping from physical space to the reference cell, depending on its geometry and finite element ansatz, which is also stored within `FEManager`. To give an example, for Lagrangian elements on hexahedrons, this results in a trilinear transformation. Since for a new ansatz a new mapping is needed, it also can be implemented as a derived class from `CellTransformation`.

This separation of tasks leads to a generic and efficient way of representing a finite element ansatz. At this stage of HiFlow[3] not only arbitrary degrees of Lagrangian elements can be incorporated, but also their usage in an assembly routine is with high performance, since each shape function can be evaluated and stored for each singleton and therefore does not need to be computed on the fly.

4.2 DoF Submodule

Considering as an example typical fluid problems in three dimensions, one needs to deal with a finite element ansatz for four scalar variables (three velocities and one pressure). For a complex unstructured geometry, this easily can lead to a very high number of unknowns ($\gg 10.000.000$) alias degrees of freedom. These DoFs need to be numbered and interpolated in an unique way depending on arbitrary combinations of finite element ansatz on neighboring cells. For a continuous ansatz, given the local numbering strategy provided by the submodule FEM on each cell, one major task is to create a mapping between a local DoF Id and a global (mesh wide) DoF Id. The second major task is to interpolate those DoFs, which are restricted due to conditions provided by FEM and cannot be identified. Such cases can occur for example in the case of h-refinements (hanging nodes), p-refinements or hp-refinements (see figure 6).

As mentioned in the previous subsection, the DoFs in any cell of the considered mesh are determined by a transformation that maps the reference cell to

18
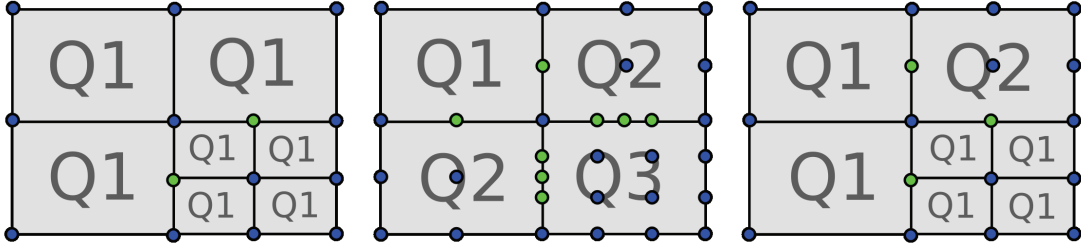
Fig. 6. Degrees of freedom in h-refined (left), p-refined (middle) and hp-refined (right) setting. Green marked DoFs are interpolated due to continuity constrains.

the chosen physical cell and thereby defines the location of the DoF points (see figure 7). At that state the DoF Ids can be defined and numbered consecutively in the order of a given iteration through the mesh cells and the local FEM numeration scheme. Further a mapping that maps a cell index of the mesh and a local DoF index (key) to the corresponding DoF Id (value) is created. This is done for each variable of the underlying problem. If no continuity constraints are set for the global DoF numbering procedure, the resulting numeration is appropriate for discontinuous finite element methods.

In case of constraints (i.e. continuity constraints) an interface approach is realized for the calculation of the interpolation and identification of DoFs on neighboring cells. In this context *interface* denotes the common cell boundary between two neighboring cells. By this, it is possible to handle all types of neighboring finite elements in a generic way, and as expensive local calculations are made only once for each occurring type of interface, the overall performance is still very good. Through an iteration over all interfaces of the mesh, a so-called *interface pattern* is determined that includes the geometrical information (orientation of cells, h-refinement status, etc.) as well as the finite element ansatz of the participating cells such that all information needed to characterize the interface is included. With the information contained in the pattern the corresponding DoF interpolation and identification in terms of the local DoFs are calculated. Therefore the general transfer operator of Schieweck [47] is used that allows interpolation between different finite element spaces. An important step within this phase is the transformation of DoFs of a neighboring cell next to the reference cell (see figure 7). Once this local evaluation is done, the tuple of the pattern description (key) and the interpolation and identification information (value) are stored in a map structure that allows for fast access given a pattern description. Using this map the interpolation and identification of DoF Ids can be realized efficiently even in the complex hp-refined context using an equivalence class generator. Later on the numbering of the DoF Ids can be changed easily by applying some user defined permutation.

4.3 Partitioning

In a domain decomposition setup, i.e. several processes are used for the solution of one single domain, each part of the domain (subdomain) is dedicated
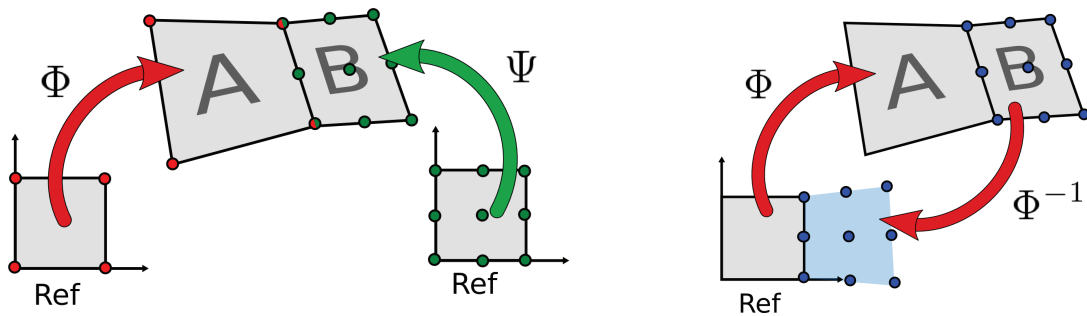
Fig. 7. Transformation of DoF points to a cell in mesh (left). Transformation of DoFs of neighboring cell next to the reference cell (right).

to one process using MPI. Again a unique DoF numeration of all DoFs in the global domain must be determined. For good scaling properties, a parallel and distributed handling of the DoFs is needed, i.e. each process manages the DoFs that are connected to the cells lying in its domain and the class `DofPartition` is used to create the correspondence with other subdomains from other processors via MPI communication. Each subdomain has information of the neighboring domains by the *ghost cells* (one *ghost layer*), which are represented by the red cells in figure 8. To create a DoF numbering with (domain-) global Ids, each process determines in a first step a consecutive numbering of the DoFs within its subdomain, whereas also the ghost layer is treated as if it would belong to the subdomain. Next, the antiquated information stored in this layer needs to be updated via communication. Hereby, a decision needs to be made, whether a DoF lying on the skeleton of the domain belongs to a subdomain or not, i.e. this DoF is lying on two subdomains, which are sharing it. The implemented procedure states, that the subdomain represented by a unique lower subdomain Id will own the DoF. Hence, identification and interpolation is possible. At the final stage, every subdomain has a unique numbering, containing the correct DoF Ids in its ghost layer.
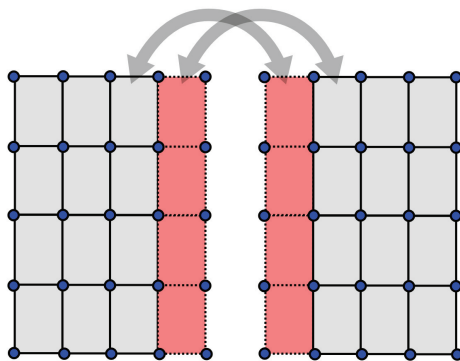
Fig. 8. Two domains with ghost cells

Algorithm 1 Distributed matrix vector multiplication $y = Ax$

---

    function DISTR_MVMULT(A, x, y)
        Start asynchronous communication: exchange ghost values;
        $y_{\text{int}} = A_{\text{diag}}\, x_{\text{int}}$;
        Synchronize communication;
        $y_{\text{int}} = y_{\text{int}} + A_{\text{offdiag}}\, x_{\text{ghost}}$;
   **end function**

---

5 Linear Algebra Module

The HiFlow[3] finite element toolbox is based on a modular and generic framework in C++. The module LAtoolbox handles the basic linear algebra operations and offers linear solvers and preconditioners. It is implemented as a two-level library. The upper (or global) level is an MPI-layer which is responsible for the distribution of data among the nodes and performs cross-node computations, e.g. scalar products and sparse matrix-vector multiplications. The lower (or local) level takes care of the on-node routines offering an interface independent of the given platform.

5.1 The Global Inter-Node MPI-level

The MPI-layer of the LAtoolbox takes care of communication and computations in the context of finite element methods. Given a partitioning of the underlying domain (handed over by the mesh module, see Section 3) the DoF module (see Section 4) distributes the degrees of freedom (DoF) according to the chosen finite element approach. This results in a row-wise distribution of the assembled matrices and vectors. Each local sub-matrix is divided into two blocks: a diagonal block representing all couplings and interactions within the subdomain, and an off-diagonal block representing the couplings across subdomain interfaces.

In Figure 9 we find a domain partitioning into four subdomains. In order to determine the structure of the global system matrices, first, each subdomain has to be associated with a single process on a node. Then, each process not only detects couplings within its own subdomain, but also couplings to the so-called *ghost DoF*, i.e. neighboring DoF which are owned by a different process. These identifications are performed simultaneously and independently on each node since the mesh module offers a layer of ghost DoF and hence no further communication is necessary. DoF i and j interact if the matrix has a non-zero element in row i and column j.

The distributed (sparse) matrix vector multiplication is given in Algorithm 1. While every process is computing its local contribution of the matrix vector multiplication an asynchronous communication for exchanging the ghost values is initiated. After this communication phase has been completed, the local contributions from coupled ghost DoF are added accordingly.
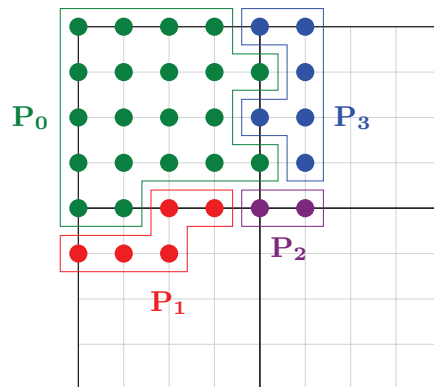
Fig. 9. Domain partitioning: DoF of process $\mathbf{P_0}$ are marked in green (interior DoF in the diagonal block); the remaining DoF represent inter-process couplings for process $\mathbf{P_0}$ (ghost DoF in the off-diagonal block)


Fig. 10. Distributed matrix vector multiplication

5.2 Local On-Node Level

The highly optimized BLAS 1 and 2 routines are implemented in the local multi-platform linear algebra toolbox (lmpLAtoolbox) which is acting on each of the subdomains. After the discretization of the PDE by means of finite element or finite volume methods typically a large sparse linear system with very low number of non-zeros is obtained. Therefore, the Compressed Sparse Row (CSR) data structure [11] is the favorable sparse matrix data format. On the NVIDIA GPU platforms we use a CUDA implementation with CSR matrix-vector multiplication based on a scalar version as it is described in [12, 13]. Our library supports several back-ends including multi-core CPUs and NVIDIA GPUs. Currently, we are developing OpenCL [45] modules for further back-ends like IBM Cell BE, ATI GPUs and x86 multi-core CPUs.

The LAtoolbox provides unified interfaces as an abstraction of the hardware and gives easy-to-use access to the underlying heterogeneous platforms. In this respect the application developer can utilize the LAtoolbox without any knowledge of the hardware and the system configuration. The final decision on which specific platform the application will be executed can be taken at run time. A data parallel model is used for the parallelization of the basic BLAS 1 and 2 routines which by their nature provide fine-grained parallelism. From a theoretical point of view the data parallel model results in good load balancing and scala-

bility across several computing units. These expectations have been confirmed by our practical experiments across a variety of platforms and systems.
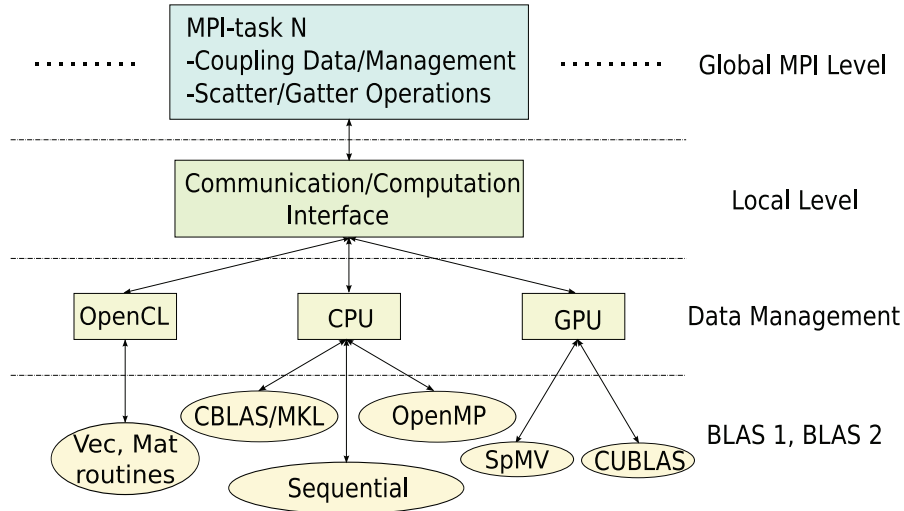
Fig. 11. Structure of the lmpLAtoolbox and LAtoolbox for distributed computation and node-level computation across several devices in a homogeneous and heterogeneous environment.

The layered structure and organization of both the LAtoolbox and the lmpLAtoolbox is depicted in Figure 11. It shows a high-level view of distributed communication and computation across nodes and the node-level computation across several devices in a heterogeneous environment.

A cross-platform makefile generator (cmake) identifies all software libraries (i.e. OpenMP, CUDA, MKL, and etc) available on the underlying computing platform. After compilation the decision of final platform can be taken at runtime e.g. by providing information in a configuration file or from user input. All data structures (matrices and vectors) are distributed or offloaded accordingly. On a lean platform like a netbook or even a smartphone the project and all modules can be used in their sequential version. Within this setting there is further room for autotuning mechanisms, e.g. when several devices are available.

The code fragment in listing 1.1 exemplifies handling of the lmpLAtoolbox. It details declaration of two vectors and a matrix. The CPU performs all input and output operations. To this end, the CPU matrix type is declared. Later conversion between two different platforms is based on a copy-like function. For avoiding unnecessary transfers to the same device a simple platform check is typically used.

The PCIe bus between host and device is a known bottleneck for attached accelerators. In particular, bandwidth limitations occur for updates of interprocess couplings between subdomains kept on different devices. Advanced data packaging and transfer techniques are utilized for mitigation of delays due to these inevitable cross-device data transfers. The underlying idea for maximized

throughput is to manage and reorganize irregular data structures on the host CPUs and to transfer repackaged data in huge and continuous buffers to the accelerators. Moreover, sophisticated re-ordering techniques for the arrangement and redistribution of DoF like e.g. (reverse) Cuthill-McKee ordering [22] and graph-coloring algorithms [46] have been included for optimizing data access patterns, matrix structure and cache reuse.

Due to the full abstraction within the libraries there is limited or no access to the platform-specific data buffers. In certain cases – e.g. for nested loop iterations or irregular data access patterns – there is the option for defining device-specific data structures (vectors and matrices) with direct access to all data buffers.

```
CPU_lMatrix<double> mat_cpu;
// Read matrix from a file
mat_cpu.ReadFile('matrix.mtx');

lVector<double> *x, *y ;
// init a vector for a specific platform and implementation
x = init_vector<double>(size,"vec x", platform, impl);
// clone y as x
y = x->CloneWithoutContent();

lMatrix<double> *mat;
// init empty matrix on a specific platform
// (nnz,nrow,ncol,name,platform,implementation,format)
mat = init_matrix<double>(0,0,0,"A",platform,impl,CSR);
// Copy the sparse structure of the matrix
mat->CopyStructureFrom(mat_cpu);
// Copy only the values of the matrix
mat->CopyFrom(mat_cpu);

...

// Usage of BLAS 1 routines
y->CopyFrom(*x);          // y = x
y->Axpy(*x, 2.3 );        // y = y + 2.3*x
x->Scale(6.0);            // x = x * 6.0
// print the scalar product of x and y
cout << y->dot(*x);

// Usage of BLAS 2 routines
mat->VectorMult(*y, x); // x = mat*y

...

delete x, y, mat;
```

Listing 1.1. Example code for lmpLAtoolbox

Due to the modular setup and the consistent structure, the lmpLAtoolbox can be used as a standalone library independently of HiFlow[3]. It offers a com-

plete and unified interface for many hardware platforms. Hence, the LAtoolbox not only offers fined-grained parallelism but also flexible utilization and cross-platform portability.

5.3 Linear and Nonlinear Solvers

The LAtoolbox offers a high-level of abstraction by providing unified interfaces for basic matrix and vector routines. Platform-specific implementations are transparent to the user. Therefore, linear and non-linear solvers can be implemented easily and generically without any information on the underlying hardware platform while keeping platform-adapted and tuned code.

This module provides two Krylov subspace methods as iterative linear solvers, namely CG and GMRES. Additionally, the module offers parallel preconditioners based on a blockwise multilevel incomplete LU factorization [41–43]. Furthermore, advanced techniques for exploiting fine-grained parallelism based on multi-coloring algorithm are deployed in the library. This kind of preconditioners are applicable to a wide variety of problems and yield performance advantages on most parallel platforms. Applied methodologies and a performance analysis on multi-core x86 CPUs and NVIDIA GPUs are presented in [33].

Moreover, a nonlinear Newton solver is included in the framework of the LAtoolbox. It utilizes the iterative solvers for the solution of the linearized problem and features interfaces for damping and forcing strategies for generalization and extension to Newton-like methods.

6 Assembly Tools

The assembly tools in HiFlow³ combine the functionality in the Mesh, DoF/FEM and Linear Algebra modules, to provide a mechanism that efficiently produces the global system matrix and right-hand-side vector. As is common in finite element literature, the global assembly algorithm is element-oriented, with an outer iteration over the cells in the mesh that performs a local assembly on each cell separately. This section first describes the global assembly algorithm, which is independent of the PDE, and then explains how the local assembly can be implemented based on the variational formulation of the PDE.

6.1 Global Assembly

The global assembly algorithm is quite simple, consisting of a loop over all cells, the computation of the element matrix or vector by the local assembly, and finally an addition of the result into the global matrix or vector.

In order to enable optimizations in the local assembly, the order of iteration of the cells is according to their type and the polynomial degree of the associated element. This minimizes the number of times the local basis functions and their gradients must be evaluated on the reference element.

The local assembly is defined through a user-defined class, on which the global assembly functions are templated. Inside the loop over the cells, the global assembly functions first call the function `initialize_for_element`, passing the element information and quadrature object as parameters. The local matrix or vector is then computed with a call to `assemble_local_matrix` or `assemble_local_vector`, respectively.

The addition of the local element matrix or vector into the corresponding global object uses the DoF numbering established by the DoF/FEM module. This numbering is also used to compute the graph of the global matrix, which is needed for the initialization of the sparsity structure of this object. The function which does this has the same structure as the global assembly, and is therefore also a part of the assembly tools.

6.2   Local Assembly

The task of the local assembly is to compute the local element matrices and vectors. The local assembly is where the variational formulation of the PDE comes into play. In HiFlow[3], a strategy similar to that of deal.II [9] is followed, where the user of the library is responsible for choosing a variational formulation, and implementing the corresponding local assembler functions in a class, which we will hereafter call `LocalAssembler`, although the name can be freely chosen in the code. This is in contrast to the approach described in [3], where a separate language, UFC, is used to describe the variational formulation. The UFC code is then compiled to a target language, e.g. C++, and can thereafter be included in the finite element program. The advantage of such an approach is a simplified syntax for users of the library, and a potential for interoperability between finite element libraries. The drawback is, as with any abstraction layer, that the user has less control over the code, which could lead to performance penalties.

HiFlow[3] supports the user in the creation of the local assembler through a helper class, which provides a large part of the necessary functionality. The helper class computes and caches the values of the local shape functions and their gradients on the reference element as well as transformed to the physical element. It provides the information related to the current quadrature rule, and the cell transformation. Furthermore, it aids in the evaluation of existing finite element functions defined through coefficient vectors, which is useful for nonlinear and time-dependent problems. Support for assembly over facets is also being developed, which is needed e.g. for inhomogeneous Neumann boundary conditions. The helper class works without user interference on meshes with mixed cell types, varying polynomial degrees and hanging nodes.

In order to illustrate the construction of a local assembler, Listing 1.2 shows C++ code for the Poisson equation

$$-\Delta u = x + y$$

with variational formulation

$$\int_\Omega \nabla u \cdot \nabla v \mathrm{d}\boldsymbol{x} = \int_\Omega (x + y) v \mathrm{d}\boldsymbol{x}$$

```cpp
const int DIM = 2;
class LocalLaplaceAssembler : private AssemblyAssistant<DIM> {
public:
  void initialize_for_element(const Element& el,
                               const Quad<double>& quad) {
    AssemblyAssistant<DIM>::initialize_for_element(el, quad);
  }

  void assemble_local_matrix(const Element& el,
                              LocalMatrix& lm) const {
    const int num_q = num_quadrature_points();
    const int num_local_dofs = num_dofs(0);

    for (int q = 0; q < num_q; ++q) {
      const double wq = w(q);
      const double dJ = detJ(q);

      for (int i = 0; i < num_local_dofs; ++i) {
        for (int j = 0; j < num_local_dofs; ++j) {
          lm(dof_index(i, 0), dof_index(j, 0)) +=
            wq * dot(grad_phi(i, q), grad_phi(j, q)) * dJ;
        }
      }
    }
  }

  void assemble_local_vector(const Element& el,
                              LocalVector& lv) const {
    const int num_q = num_quadrature_points();
    const int num_local_dofs = num_dofs(0);

    for (int q = 0; q < num_q; ++q) {
      const double wq = w(q);
      const double dJ = detJ(q);

      for (int i = 0; i < num_local_dofs; ++i) {
        lv[dof_index(i, 0)] +=
          wq * (my_f(x(q)) * phi(i, q)) * dJ;
      }
    }
  }

  double my_f(Vec<DIM> pt) const { return (pt[0] + pt[1]); }
};
```

**Listing 1.2.** Local assembler for the Poisson problem

As illustrated in this code, the helper class `AssemblyAssistant` provides most of the functions needed for the assembly. The user implements the compu-

tation of the local matrix and vector in the functions `assemble_local_matrix()` and `assemble_local_vector()`, respectively. In this case, the implementation uses the `AssemblyAssistant` class to obtain the values of the local shape functions and their gradients `phi()` and `grad_phi()`, at the quadrature points of the current quadrature rule transformed to the physical element. The function `dof_index()` provides the position in the matrix or index where the degrees of freedom corresponding to a given variable (in this case variable 0) should be inserted. `w(q)` gives the weight associated with quadrature point `q` in the quadrature rule, and `detJ(q)` gives the Jacobian of the element transformation at that point. The local vector assembly illustrates the use of a user-defined function `my_f()`, which is evaluated at the point `x(q)` on the physical cell that corresponds to quadrature point `q`. More complex settings, with different models coupled to each other, can be also be handled in this framework.

7 Numerical Simulation in Urban Environments

As an illustration of the methodology of applying mathematical modeling, numerical simulation and scientific visualization for complex simulations made possible by HiFlow[3], we present the Karlsruhe Geometry project which performs numerical simulations in an urban environment in collaboration with the city council of Karlsruhe, Germany. Using the increasing detail of three dimensional city models, more reliable simulations are feasible, for example concerning airflow, fine-dust or noise distribution in the very environment that we live in.

This section describes an example of airflow around the Department of Mathematics of the Karlsruhe Institute of Technology, in order to illustrate how the components of HiFlow[3] are combined into a simulation.
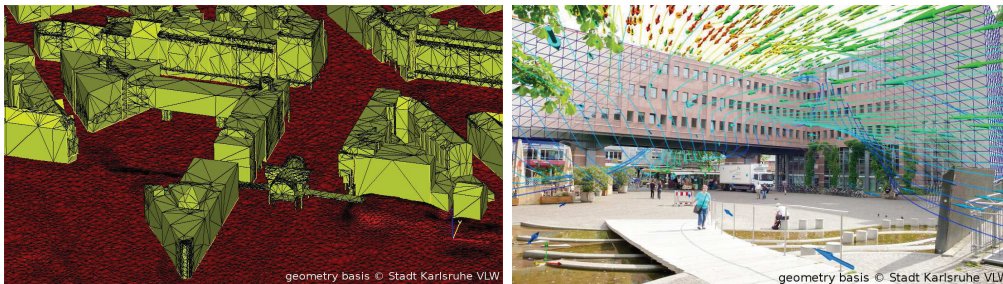
Fig. 12. Geometry used for the simulation and a visualization of simulation results.

7.1 Problem Formulation

The instationary Navier-Stokes equations are solved in a box surrounding the buildings of the Kronenplatz square. As boundary conditions, a simple Poiseuille profile is used on one side Γ_{in}, and natural do-nothing boundary conditions are

used on the opposite side Γ_{out}. On the remaining boundaries, including the walls of the buildings, the velocity is set to zero. The model is formulated as an initial boundary value problem for the velocity $\boldsymbol{u}\left(\boldsymbol{x}, t\right)$ and the pressure $p\left(\boldsymbol{x}, t\right)$ in Equation 1.

$$
\begin{aligned}
\partial_t \boldsymbol{u} - \nu \Delta \boldsymbol{u} + \left(\boldsymbol{u} \cdot \nabla\right) \boldsymbol{u} + \nabla p &= 0 & \left(\boldsymbol{x}, t\right) &\in \Omega \times \left(0, T\right) \ , \\
\nabla \cdot \boldsymbol{u} &= 0 & \left(\boldsymbol{x}, t\right) &\in \Omega \times \left(0, T\right) \ , \\
\boldsymbol{u} &= \boldsymbol{u}^{in} & \left(\boldsymbol{x}, t\right) &\in \Gamma_{\mathrm{in}} \times \left(0, T\right) \ , \\
\left(-\mathcal{I} p + \nu \nabla \boldsymbol{u}\right) \cdot \boldsymbol{n} &= 0 & \left(\boldsymbol{x}, t\right) &\in \Gamma_{\mathrm{out}} \times \left(0, T\right) \ , \\
\boldsymbol{u} &= 0 & \left(\boldsymbol{x}, t\right) &\in \Gamma \times \left(0, T\right) \ , \\
\boldsymbol{u}\left(\boldsymbol{x}, 0\right) &= \boldsymbol{u}_0\left(\boldsymbol{x}\right) & \boldsymbol{x} &\in \Omega \ .
\end{aligned}
\tag{1}
$$

7.2 Implementation of the Navier-Stokes Solver

Equation 1 is discretized first in time, and then in space. For the time discretization, the fractional step θ-scheme is used, and in space $Q_2 - Q_1$ ("Taylor-Hood") finite elements [16]. Due to the nonlinearity $\left(\boldsymbol{u} \cdot \nabla\right) \boldsymbol{u}$ each timestep is resolved with the exact Newton algorithm. This involves solving a linearized problem iteratively, which is done using the GMRES method preconditioned by the ILU++ [41] preconditioner. Overall, the computation is structured as in Algorithm 2, with the supporting modules indicated in parentheses.

Algorithm 2 Overall structure of the instationary Navier-Stokes simulation

---

    function SOLVE_INSTATIONARY_NAVIER_STOKES(mesh_filename)
        Read mesh and distribute to all processors (Mesh).
        Number the DoFs using a Q2-Q1 finite element space (DoF/FEM).
        Create the system matrix, residual vector and solution vector (Linear Algebra).
        Initialize the solution vector with initial solution 0.
        Set the DoF values on the inflow boundary to the values prescribed by $\boldsymbol{u}^{in}$.
        while $t < T$ **do**
            for $s = 1, 2, 3$ **do**
                Solve nonlinear problem for substep s to obtain $\left(\boldsymbol{u}^{n,s}, p^{n,s}\right)$.
            end for
            $t = t + \Delta t$
            Visualize the solution at timestep t
        end while
    end function

The nonlinear solution algorithm is encapsulated in the class `Newton` which implements an abstract interface `NonlinearSolver`. This has a reference to an object which implements the functions `EvalGrad` and `EvalFunc` that compute the system matrix and the residual vector, respectively. These two functions use the assembly functionality described in Section 6. The `NonlinearSolver`

contains a reference to a linear solver of type `GMRES`, which is used in each step to compute the update for the current linearization point.

The local assembly of the linearized variational formulation is implemented in a class `InstationaryFlowAssembler`. The state of this class depends on the current substep in the timestepping method, as well as the solutions corresponding to the linearization point in the Newton method and the previous timestep. This problem-specific code is relatively long, but easy to write once the corresponding variational form has been derived analytically.

This example illustrates the interplay between reusable software components, provided in the HiFlow[3] library through the three core modules, and the problem-specific code that must be implemented specifically for each individual solver. In this case, the Mesh module provides the functionality for reading in and distributing the computational mesh; the DoF/FEM module computes the DoF numbering, and is also used in the assembly of the system matrix and residual; and the Linear Algebra module provides management of matrices and vectors, as well as an iterative method to solve the linear system. Abstract interfaces are used in several places, both for implementing callbacks for the components in the library, such as for the nonlinear solver; and for facilitating the exchange of components, as with the concrete linear and nonlinear solvers (GMRES and Newton).

8 Current Projects

The combination of the progress in high performance computing and modern software developments for numerical simulation enables us to solve more complex and relevant problems than ever before. By the increasing realism of simulation results and accessibility of improved scientific visualization, scientific computing is continuously raising its impact on our every-day lives. This is especially true, for example, in the application fields of environmental sciences, meteorology, medical engineering, energy research and computations fluid dynamics. In this section we present current projects working with HiFlow[3].

8.1 Goal Oriented Adaptivity for Tropical Cyclones

In this project adaptive numerical methods are developed in the context of meteorological multi-scale problems. Many meteorological and environmental phenomena, such as the dynamics of Tropical Cyclones, are influenced by processes on a large range of scales in space and time. For such problems the numerical modelling and solution is challenging since not all scales can be resolved adequately due to memory or CPU time restrictions. Often a certain physical quantity of the solution is of interest. In such cases goal-oriented adaptivity is a promising approach as only features that are relevant for the determination of the quantity of interest need to be considered. A major task is the solution of the dual problem in higher-order accuracy, which contains information of the sensitivity with respect to the quantity of interest. For the discretization space

time finite elements are used which allow for the estimation of local error contributions by means of a class of a posteriori error estimators. The temporal and spatial mesh and the finite element ansatz can be adapted appropriately to improve the quality of the approximate solution with respect to the chosen goal.

8.2 United Airways

The goal of the United Airways project [28] is the full description of the flow behavior in the human respiratory system by means of numerical simulations on supercomputers. The considered approach relies on an integrative methodology allowing to analyze the interaction of the nose, paranasal sinuses, larynx and lungs in a coupled way. The interdisciplinary approach relies on the expertise of principle investigations in the areas of medicine, biotechnology, numerical simulation, numerical optimization, high performance computing and visualization.

HiFlow^3 is used to numerically simulate the airflow and particle deposition in the complex geometries of the nose and the lungs. It is possible to analyze airflow patterns supporting physicians to diagnose and treat diseases of the human respiratory system with the help of models for the bronchioles [29]. Another, more abstract, intention is gaining better insights in the functionality of the human respiratory system. Particle depositions give evidence about the impact of respirable dust on the lungs and can be used to optimize and control the dosage of pharmaceutical drugs given through the respiration system. Studies are carried out to decide whether particle simulations have to be done patient-specific or statistical models can be used.

8.3 Stochastic Finite Elements

Dealing with finite elements, one often assumes that the underlying partial differential equation has deterministic parameters, e.g. the kinematic viscosity of a fluid, or certain specified boundary and initial conditions. In practice, however, the exact knowledge of critical parameters might often not be known in an exact manner, hence, a stochastic model could be used to represent the uncertainty resulting in a stochastic partial differential equation, where the solution does not only depend on space and time but also itself is a random process, which probability distribution is not known a priori. The classical way of addressing this problem is to use sampling techniques such as the popular Monte Carlo method. Yet, this technique relies on a large number of realizations driving the computational cost (even in the trivial parallelization case) into not feasible dimensions. An alternative approach is to use a spectral expansion in stochastic terms, whereas a decomposition of the solution in a random and a space / time part is achieved. Utilizing a Galerkin projection onto the random space of square integrable functions, a coupled system of deterministic partial differential equations is obtained, which is solved employing the finite element method. The procedure is also known as the polynomial chaos expansion [30]. As a result one can calculate the stochastic moments of interest, which are usually the expectation and variance of the solution.

8.4 hp-adaptive FEM

As the complexity of models being solved using FEM increases, the need for adaptive algorithms that can construct accurate approximations using a small number of degrees of freedom becomes apparent. hp-adaptivity is one of the most powerful approaches to adaptive finite element discretization [23, 50, 52]. By combining local mesh refinements close to irregularities of the solution with the use of higher-order elements where the solution is smooth, this technique often yields very high convergence rates and reduces the computation time significantly.

In order to support this type of adaptive methods, it is planned to extend the library to be able to handle non-conforming meshes and local variations of the polynomial degree of the elements. The central difficulty lies in being able to treat degrees of freedom that are constrained due either to a non-conformity in the mesh (a so-called "hanging node") or a mismatch of element degrees over an interface. These degrees of freedom should be detected, and the local interpolation operator, which expresses the constraints that these DoF must fulfill, computed. These extra constraints must then be incorporated in the global system of equations, in order to reduce it to the constrained subspace where the solution is to be sought. An overview of the necessary steps is presented in [10].

As support for controlling an adaptive algorithm, we also plan to implement functionality for a posteriori error estimation. There are several different approaches in this domain that would be worth including in the library, for instance the Element Residual Method [2] and the duality-based methods [34].

8.5 Science to Go: Numerical Simulation on the Spot

This project is dedicated to the development of new technologies in order to allow and simplify access to scientific computation, to facilitate interaction with numerical simulations, and to deliver the results to where they are needed by support of mobile devices.

Scientific computing and numerical simulation is essential to all scientific areas, but the use of software and exploitation of the results usually require dedicated expertise. Simulations may give answers to real life problems, but are not easily available at where they turn up. The vision to give simplified access to high performance computing based on the versatility of HiFlow[3] using mobile devices is a key concept for decision makers. It aims to raise the public impact of numerical simulations.

The concept for simplified access to numerical simulation and support of mobile devices for visualization touches many aspects of scientific computing, interactive visualization, infrastructure and human perception. The project plans to develop a unified and automated simulation and visualization framework. It aims at supporting interactive visualization over networks with low bandwidth and high latency by model reduction with minimal visual impact and exploring new visualization concepts for numerical simulation by augmented reality and adapted data representations.

9   Future Work

Mesh Module In the future, extensions are planned for the mesh module to support a more complete description of the geometry, e.g. for domains with curved boundaries. Algorithms for parallel partitioning and re-distribution, which is necessary to improve the scalability of the library, are also under consideration. For this task, there are existing parallel graph partitioning libraries such as ParMETIS [38] and PT-SCOTCH [18], whose services could be leveraged. Support for moving meshes, which can be implemented using attributes, and quality criterions for meshes are planned to be included in the module as well.

DoF/FEM Module The generic structure allows for the extension to further finite element ansatz. Especially $H(div)$ and $H(curl)$ conforming vector valued finite elements will be implemented [25]. Further transformations that map the reference cell to arbitrary cells within the mesh are planned, which provide isoparametric elements [16].

Linear Algebra – Advanced Preconditioning Techniques As condition numbers are increasing polynomially with problem size sophisticated preconditioning techniques are essential building blocks. In the era of multi-core and many-core processors like GPUs there is a strong need for scalable and fine-grained parallel preconditioning approaches. Currently we are working on fine-grained parallel preconditioners based on multi-coloring decomposition approach. In our ongoing work we consider an extended set of preconditioners (e.g. Gauss-Seidel, ILU(p), Chebyschev, non-symmetric cases with GMRES). Furthermore, preconditioning techniques in a cluster of multi-core-CPU and GPU nodes with parallelization across several devices will be investigated.

10   Conclusion

The creation of a multi-purpose finite element software package that is portable across a wide variety of platforms including emerging technologies like hybrid CPU and GPU platforms is a challenging and multi-facetted task. By our modular approach that utilizes the concepts of object-orientation, data abstraction, dynamic polymorphism and inheritance we have created a highly capable piece of software that provides a powerful means for gaining scientific cognition. By utilizing several levels of parallelism by means of a two-level communication and computation model and following the concepts of hardware-aware computing HiFlow³ is a flexible numerical tool for solving bleeding-edge scientific problems on the basis of finite element methods optimized for high performance computers. Furthermore, the modules Mesh, DoF/FEM and Linear Algebra complemented by auxiliary methods provide a broad suite of building blocks for development of modern numerical solvers and application scenarios. The user is freed from any detailed knowledge of the hardware – he only has to familiarize

with the provided interfaces and needs to customize the available modules in order to adapt HiFlow³ to his domain-specific problem settings. As an open source project HiFlow³ further supports extensibility of modules and methods. Within large scale projects like e.g. the United Airways project HiFlow³ has already proven its potential. In the next steps the efficiency of the methods considered in the different modules especially related to the scalability will be evaluated. Based on these results HiFlow³ will improve, following further the path of object oriented techniques in software design for scientific computing.

Acknowledgements

References

1. D. Abrahams, J. Siek, and T. Witt. The Boost.Iterator Library. http://www.boost.org/doc/libs/release/libs/iterator/.
2. M. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis.* Wiley and Sons, New York, 2000.
3. M. S. Alnaes, A. Logg, K.-A. Mardal, O. Skavhaug, and H. P. Langtangen. Unified Framework for Finite Element Assembly. *International Journal of Computational Science and Engineering*, 4(4):231–244, 2009.
4. Ansys - Fluent. http://www.fluent.com/.
5. S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.
6. S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2009. http://www.mcs.anl.gov/petsc.
7. S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

8. W. Bangerth, R. Hartmann, and G. Kanschat. `deal.II` *Differential Equations Analysis Library, Technical Reference.* http://www.dealii.org.

9. W. Bangerth, R. Hartmann, and G. Kanschat. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw.*, 33(4):24/1–24/27, 2007.

10. W. Bangerth and O. Kayser-Herold. Data Structures and Requirements for *hp* Finite Element Software. *ACM Trans. Math. Softw.*, 36(1):4/1–4/31, 2009.

11. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

12. M. M. Baskaran and R. Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. Technical report, IBM, 2009.

13. N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

14. D. Boffi, F. Brezzi, L. F. Demkowicz, R. G. Durán, R. S. Falk, M. Fortin, D. Boffi, and L. Gastaldi. *Mixed Finite Elements, Compatibility Conditions, and Applications*, volume 1939. 2008.

15. D. Braess. *Finite Elemente.* Springer-Verlag GmbH; Auflage: 1, 2007.

16. S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods.* Texts in applied mathematics; 15. Springer, New York, 2. ed. edition, 2002.

17. R. Buchty, V. Heuveline, W. Karl, and J.-P. Weiss. A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators. 2010.

18. C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318, 2008.

19. P. G. Ciarlet. *The finite element method for elliptic problems.* Classics in applied mathematics; 40. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2002.

20. COMSOL Multiphysics. http://www.comsol.de/products/multiphysics/.

21. Convey Computer Corporation. http://www.conveycomputer.com/.

22. E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM '69: Proceedings of the 1969 24th national conference*, pages 157–172, New York, NY, USA, 1969. ACM.

23. L. Demkowicz, J. Kurtz, D. Pardo, M. Paszynski, W. Rachowicz, and A. Zdunek. *Computing with hp-adaptive finite elements.* Chapman and Hall applied mathematics and nonlinear science series. Chapman and Hall, London, 2008.

24. DUNE: Distributed and Unified Numerics Environment. http://dune-project.org/.

25. A. Ern and J.-L. Guermond. *Theory and practice of finite elements.* Springer, NY, USA, 2004.

26. FEAST: Finite Element Analysis and Solutions Tools. http://www.feast.uni-dortmund.de/.

27. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

28. T. Gengenbach, T. Henn, W. Heppt, V. Heuveline, M. J. Krause, and S. Zimny. United Airways: Numerical Simulation of the Human Respiratory System, 2010. http://www.united-airways.eu.

29. T. Gengenbach, V. Heuveline, and M. Krause. Numerical Simulation of the Human Lung: A Two-scale Approach. In *BMT 2010 - Reguläre Beiträge (BMT 2010 Reguläre Beiträge)*, Rostock-Warnemünde, Germany, 2010.

30. R. Ghanem and P. Spanos. *Stochastic finite elements: A spectral approach.* Springer-Verlag, New York, NY, USA, 1991.

31. A. Henderson Squillacote. *The ParaView guide: a parallel visualization application.* Kitware, [Clifton Park, NY], 2007.

32. M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

33. V. Heuveline, D. Lukarski, and J. P. Weiss. Scalable Multi-Coloring Preconditioning for Multi-core CPUs and GPUs .

34. V. Heuveline and R. Rannacher. Duality-Based Adaptivity in the hp-Finite Element Method. *Journal of Numerical Mathematics*, 11(2):95–113, 2003.

35. V. Heuveline, C. Subramanian, D. Lukarski, and J. P. Weiss. A Multi-Platform Linear Algebra Toolbox for Finite Element Solvers on Heterogeneous Clusters. In *PPAAC'10, IEEE Cluster 2010 Processing Workshops*.

36. HiVision. http://www.hivision-project.org/.

37. S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 51–60, New York, NY, USA, 2006. ACM.

38. G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.

39. A. Logg. Efficient Representation of Computational Meshes. *International Journal of Computational Science and Engineering*, 4(4):283–295, 2009.

40. D. Lukarski. Specific aspects of a parallel implementation of a 3D CFD solver on the Cell architecture. Master's thesis, University of Karlsruhe, Germany, 2008.

41. J. Mayer. ILU++ software package. http://www.iluplusplus.de/.

42. J. Mayer. A multilevel Crout ILU preconditioner with pivoting and row permutation. *Numerical Linear Algebra with Applications*, 14(10):771–789, 2007.

43. J. Mayer. Symmetric permutations for I-matrices to delay and avoid small pivots during factorization. *SIAM J. Sci. Comput.*, 30(2):982–996, 2008.

44. F. Oboril. Parallel Multigrid Methods on the Cell Broadband Engine. Master's thesis, Karlsruhe Institute of Technology, 2010.

45. OpenCL. http://www.khronos.org/opencl/.

46. Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

47. F. Schieweck. A General Transfer Operator for Arbitrary Finite Element Spaces, 2000.

48. A. Schmidt and K. G. Siebert. *Design of adaptive finite element software: the finite element toolbox ALBERTA.* Lecture notes in computational science and engineering; 42. Springer, Berlin, 2005.

49. M. Schmidtobreick. Numerical Methods on Reconfigurable Hardware using High Level Programming Paradigms. Master's thesis, Karlsruhe Institute of Technology, 2010.

50. C. Schwab. *P- and hp- finite element methods: theory and applications in solid and fluid mechanics.* Numerical mathematics and scientific computation. Clarendon Press, Oxford, repr. edition, 2004.

51. UG - Unstructured Grids. http://atlas.gcsc.uni-frankfurt.de/ ug/index.html.

52. P. Šolín, K. Segeth, and I. Doležel. *Higher order finite element methods.* Studies in advanced mathematics. Chapman and Hall, CRC, Boca Raton, Fla., 2004.

53. VTK - The Visualization Toolkit. http://www.vtk.org/.

54. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

55. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing Kernels on the cell processor. *Int. J. Parallel Program.*, 35(3):263–298, 2007.

Preprint Series of the Engineering Mathematics and Computing Lab