A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE

Peter Bastian* Markus Blatt* Andreas Dedner† Christian Engwer* Robert Klöfkorn† Ralf Kornhuber‡ Mario Ohlberger⋆ Oliver Sander‡

*Institut für Parallele und Verteilte Systeme, Universität Stuttgart, Universitätsstraße 38, D-70569 Stuttgart

†Abteilung für Angewandte Mathematik, Universität Freiburg, Hermann-Herder-Str. 10, D-79104 Freiburg

⋆Institut für Numerische und Angewandte Mathematik, Universität Münster, Einsteinstr. 62, D-48149 Münster

‡Institut für Mathematik II, Freie Universität Berlin, Arnimallee 6, D-14195 Berlin, DFG Research Center MATHEON

September 17, 2007

Abstract

In a companion paper [5] we introduced an abstract definition of a parallel and adaptive hierarchical grid for scientific computing. Based on this definition we derive an efficient interface specification as a set of C++ classes. This interface separates the applications from the grid data structures. Thus, user implementations become independent of the underlying grid implementation. Modern C++ template techniques are used to provide an interface implementation without big performance losses. The implementation is realized as part of the software environment DUNE [10]. Numerical tests demonstrate the flexibility and the efficiency of our approach.

1 Introduction

Partial Differential Equations (PDEs) are abundant in science and engineering. There is a large body of methods to numerically solve PDEs, such as the finite element, finite volume, and finite difference method as well as various gridless methods. For each of these methods, many implementations in computer codes exist, see e. g. the list provided by [29]. Each of these codes has been designed with a particular set of features in mind. Extending a code beyond this set of features is usually hard and time-consuming, because each code is based on a particular data structure.

In a companion paper [5] to this article we introduced and formally defined a generic grid interface for parallel scientific computing. Here, we will describe its implementation as a software system [10] written in C++ and present example applications which illustrate the main design principles.

The first section will give an overview of the underlying design decisions of the grid interface. Next we present the programming interface as it results from the application of the design principles in Section 2 to the abstract definitions in [5]. We then provide several example applications to give an idea of the current possibilities of the DUNE system. These examples will emphasize our design goals.

DUNE is organized as a modular system. Release 1.0 includes the core modules `dune-common` (foundation classes), `dune-grid` (grid interface and implementations), and `dune-istl` (iterative solver template library), [6, 7]. The supplementary module `dune-grid-howto` serves as an introduction to the grid interface. There are also several application modules built upon the DUNE libraries like groundwater flow, multiphase flow in porous media, inviscid fluid flow, and linear elasticity. The implementation of the grid interface, as it is described in this paper, is publicly available as part of the 1.0 release of DUNE in the `dune-grid` module.

2 Design Principles

The implementation of the abstract definitions in [5] is based on several design goals. They lead to the design principles described in this section.

Flexibility: Users should be able to write general components, which can run on any grid implementing the DUNE grid interface (Section 3).

Efficiency: Scientific computing has an unlimited demand for computing power. Users will not accept a big performance loss as the price for a clean interface.

Legacy Code: Users must be able to incorporate existing code and libraries into their new applications.

Existing frameworks are often based on a particular data model; this limits their flexibility. The main design idea for the DUNE grid interface is the separation of data structures and algorithms by abstract interfaces. This separation offers flexibility for codes based on DUNE. It ensures maintainability and extendibility of the framework and allows the reuse of existing finite element packages with a large body of functionality.

The grid interface is restricted to be slim and offers only little more than what is absolutely necessary. Therefore, more grid implementations can fulfill this interface. Extended methods and algorithms can be built on this slim interface and hence work on every grid. Furthermore, generic programming techniques allow optimized implementations of these extended methods or algorithms for a certain grid, while still offering a compatible interface. This specialized implementation can then benefit from grid features beyond the slim interface. Not all features of the interface are required, some features are optional and don't have to be implemented by every grid. Their presence can be queried at compile time using a traits class.

Dune features dimension-independent programming, using templates [2]. Dimension-independent programming reduces code bloat and improves maintainability, both of DUNE and the applications.

The container classes which can be found in DUNE follow a *view concept* modelled after [20, 25]. Data can be accessed via different views, which cannot alter the underlying container or the data. Each view offers access to a distinct subset of the container. The strict separation of read-only views and read-write access facilitates a clear design. Read-only views allow the compiler to apply various optimization strategies. Also read-only views allow to generate objects on the fly. This can dramatically reduce memory consumption and speed up execution time if certain information are only used rarely.

High-level interfaces allow to create applications without knowledge of the underlying implementation. These additional layers of abstraction usually add an overhead, leading to a performance penalty. An efficient implementation of the interface is obtained using generic programming techniques, such as static polymorphism and traits [23].

The use of generic programming techniques for the efficient separation of data structures and algorithms has been pioneered by the Standard Template Library (STL) [16], which later became part of the C++ standard library. The most important aspect of generic programming with respect to performance is that dynamic polymorphism, realized with virtual functions in C++, is replaced by static (or compile-time) polymorphism. This allows the compiler to inline interface implementation methods and to apply its full range of optimization techniques. As a consequence the abstract interface is effectively eliminated at compile time and "small" methods (consisting of only a few machine instructions) do not imply a performance penalty. This means that interfaces may be defined on any level of program design, e. g., even for the access to individual elements of a vector.

Generic programming is realized with templates in the C++ programming language. Many of the techniques used in DUNE, such as static polymorphism, traits, or template metaprogramming are, e. g., explained in the book by Vandevoorde and Josuttis [23]. Template programming techniques in scientific computing have been promoted in the Blitz++ library [24] for multidimensional arrays and for linear algebra in the Matrix Template Library [21]. Pflaum [17] concentrates on the use of expression templates (one particular template programming technique) in the numerics of PDEs. The same techniques are used in the Iterative Solver Template Library [6], which is also part of DUNE.

3 Interface Realization

The grid interface in DUNE is realized by a direct translation of the abstract definition of a grid given in [5] using the interface design principles discussed in Sec. 2. Here we will present a few of the main classes. A complete up-to-date documentation can be found at [10]. In this section, text in typewriter font denotes actual class or method names.

3.1 The Grid and Grid Entities

A `Grid` class is a container for the set $\mathcal{E}|_p$ of entities that are processed on processor p. Implementations of this class may be parametrized statically. Frequently these parameters are the grid dimension or the world dimension [5, Def. 14].

The grid class provides various iterators for the access to its entities. These iterators provide read-only access. The only way to modify the grid is through methods of the grid class itself (see the paragraph on the view concept in Sec. 2). This avoids problems related to the `const`-ness of C++ types. Grids can be changed by grid refinement (Sec. 3.3), or, if the grid implementation supports parallel processing, by load balancing (Sec. 3.5). The iterators follow the conventions of STL iterators [16]. For a given codimension c and a grid hierarchy level l an iterator of the class `LevelIterator` iterates over the sets E_l^c of entities on this level. Iterators of the class `LeafIterator` iterate over the sets $\tilde{L}^c$ of entities on the leaf grid. In parallel computations, they can be restricted to a certain processor p and `partitionType` π which is one of following five types: `interior`, `border`, `overlap`, `front`, or `ghost` [5, Def. 23].

Unlike many existing grid managers, DUNE does not store the data needed for computations in the grid itself. The mechanism used to associate external data with entities of the grid is explained in Sec. 3.2.

The interface separates the topological from the geometrical aspects of a grid hierarchy. All topological information about an entity $e \in E^c$ of codimension c is encapsulated by the class `Entity⟨c⟩`. All objects of type `Entity` know their reference element $R(e)$ (i.e., simplex, cube, etc.), their level in the grid hierarchy, and their affiliation to one of the previously mentioned partition types.

The class `Entity` is specialized for entities of codimension 0 (elements). This specialization contains several methods which are only available for elements. The `HierarchicIterator` iterates over all descendant entities of a given entity $e \in E^0$. A `LevelIntersectionIterator` is provided, which traverses the set $\mathcal{I}_e$ of intersections with elements on the same level (see [5, Section 4]). If the element is part of the leaf grid, then there is also a `LeafIntersectionIterator` which iterates over the set $\tilde{\mathcal{I}}_{[e]}$ of intersections with elements on the leaf grid. The methods `wasRefined()` and `mightBeCoarsened()` determine whether an entity was refined or might be removed during the grid adaptation step.

The geometrical information (geometric realization, [5, Def. 11]) of grid entities is provided by the class `Geometry`. The geometry object corresponding to a given object of type `Entity` is available using the member method `geometry()`. The `Geometry` class provides the geometric realization map m_e from the reference element onto the entity as described in [5, Def. 11] along with its inverse. It also provides $\nabla(m_e^{-1})^T$ for the assembly of finite element stiffness matrices and det $\sqrt{|(\nabla m_e)^T \nabla m_e|}$ which is needed for numerical quadrature. For convenience and efficiency reasons the `Geometry` class provides additional methods for the volume of the entity and the number and positions of the entity corners.

The set of reference elements $\mathcal{R}$ exists once for all grid implementations. Each reference element is implemented as asingleton [12] and can be accessed via its dimension and type through the `ReferenceElementContainer`.

3.2 Attaching Data to a Grid

The formal grid specification describes three *index maps* as the means to attach data to the grid [5, Sec. 6]. For fast access to data on a fixed grid there are the level index map $\kappa_j^{c,r}|_p : E_j^{c,r}|_p \to \mathbb{N}_0$ and the leaf index map $\lambda^{c,r}|_p : L^{c,r}|_p \to \mathbb{N}_0$ for given level j, codimension c, reference element type r, and process p. Both map their respective domains injectively onto a consecutive range of natural numbers which starts with zero. Hence the image of these maps can be used to access standard arrays with constant-time access.

The level and leaf index maps are implemented as the classes `LevelIndexSet` and `LeafIndexSet`, respectively. These two classes implement the DUNE `IndexSet` interface. Given an entity `e`, the method `index(e)` evaluates the index map, while method `subIndex<codim>(e,i)` yields the index of the `i`-th subentity of codimension `codim` for a given entity `e` with codimension 0. Furthermore, index maps implementing the `IndexSet` interface have to provide a method `size()`.

To keep data while a grid is changing the specification contains the *persistent index map* $\mu^c : \mathcal{E}^c \to \mathbb{I}$. It maps entities onto a general index set $\mathbb{I}$, which needs to be totally ordered. μ^c is persistent in the sense that indices for an entity do not change if this entity is not changed during a grid modification (see [5, Def. 27] for details). In DUNE this functionality is described by the `IdSet` interface. Each grid implementation provides an implementation of this interface which is called `GlobalIdSet`. The set $\mathbb{I}$ can be any C++ type for which the operator "`<`" exists. Hence, in general the indices returned by the `GlobalIdSet` cannot be used to address regular arrays, and associative arrays must be used instead. In analogy to the `IndexSet` classes the `GlobalIdSet` provides methods `id(e)` and `subId<codim>(e,i)`, which evaluate the persistent index map for a given entity `e` or one of its subentities. Furthermore, such a persistent index map allows to create arbitrary new index maps, for example a periodic index map.

Since indices which are unique across all processes may be very costly to obtain for specific grid implementations, the DUNE interface also provides a class `LocalIdSet`. The indices returned by this class are only unique within each process, but can in general be created more efficiently.

3.3 Grid Adaptation

According to [5, Definition 24] adaptive mesh refinement can be used to enhance accuracy and reduce cost of the simulation. The grid interface provides several methods that allow to modify the grid via refinement and coarsening procedures, if provided by the grid implementation.

The method `mark(ref, e)` is used to mark an entity `e` for refinement (`ref = 1`) or coarsening (`ref = -1`). Once entities of a grid are marked, the adaptation is done in the following way:

1. Call the grid's method `preAdapt()`. This method prepares the grid for adaptation. It returns `true` if at least one entity was marked for coarsening.
2. If `preAdapt()` returned `true`, any data associated with entities that might be coarsened (see `mightBeCoarsened()`, Sec. 3.1) during the following adaptation cycle has to be projected to the father entities.
3. Call `adapt()`. The grid is modified according to the refinement marks.
4. If `adapt()` returned `true`, new entities were created. Existing data must be prolonged to newly created entities (see `wasRefined()`, Sec. 3.1).
5. Call `postAdapt()` to clean up refinement markers.

As the data management is the user's responsibility, he or she has to take care of restriction and prolongation of data attached to the grid. This is possible using the persistent index maps (see [5, Section 6]), i. e., `LocalIdSet` and `GlobalIdSet`.

3.4 Parallel Communication

According to [5, Remark 4], the `Grid` interface method `communicate()` is introduced to organize data exchange between entity sets Σ_p and Δ_q on the processes p and q respectively. `communicate(dataHandle, interface, direction, j)` exchanges data attached to the parallel grid for all entities on level j, i.e. $E_j|_p$, `communicate(dataHandle, interface, direction)` does the same for all leaf entities $L|_p$. A pair (Σ_p, Δ_q) is called communication interface and may be specified via the parameter `interface`. Σ_p and Δ_q describe which `partitionType`s are involved on the sender side and the destination side, respectively.

The `direction` of an interfaces is either `ForwardCommunication` (communicate as given), or `BackwardCommunication` (reverse communication direction). If `communicate(dataHandle, interface, direction, j)` is called for a given communication interface (`interface, direction`), and grid level `j`, then all data attached to grid entities $e \in \Sigma_{p,q} = \Sigma_p \cap \Delta_q$ should be sent to the message buffer, and all data attached to entities $e \in \Delta_{p,q} = \Delta_p \cap \Sigma_q$ should be received and unpacked from the message buffer. In order to select the data associated to the entities in $\Sigma_{p,q}$, $\Delta_{p,q}$ and to prescribe the packing and unpacking mechanisms, a `DataHandle` object has to be provided by the user.

The `DataHandle` class provides the methods `gather(buffer, e)` and `scatter(buffer, e, size)` to pack and unpack data to and from a message buffer. On invocation of `communicate()`, `gather(buffer, e)` is called for each $e \in \Sigma_{p,q}$ and the data is stored in the message buffer. On the target process `scatter(buffer, e, size)` is called for all $e \in \Delta_{p,q}$ to retrieve data from the message buffer.

Apart from parallel communication via `communicate()`, additional parallel operations are necessary for the implementation of numerical methods. For example, it may be necessary to set barriers to synchronize the processes, or to implement some kind of master-slave communication. For such tasks DUNE offers the `CollectiveCommunication` class, an abstraction to the basic methods of parallel communication, following the message-passing paradigm. `CollectiveCommunication` provides status informations, e.g. `size()`, the number of processes, and `rank()`, the rank of the process. It offers basic communication methods, e.g. `barrier()` and `broadcast(data, length, root)` and `gather(inData, outData, length, root)` for distribution and collection of data. Also advanced communication methods, like `sum(data)`, `prod(data)`, `min(data)`, and `max(data)` are available. These methods perform certain mathematical operations on global data structures, using local operations.

A reference to an instance of `CollectiveCommunication` is returned by the `Grid` interface method `comm()`. It is important to note that the collective communication on a grid does only involve the process set of the grid object which may be a subset of all available processes. Thus it is possible to have several grid objects in one application assigned to different (possibly overlapping) sets of processes.

3.5 Load Balancing

When local grid adaptation is used in parallel computations it may be necessary to redistribute the grid in order to keep the load balanced that each processor has to handle. The grid interface provides two methods to activate this process: `loadBalance()` and `loadBalance(dataHandle)` calculate the load of the grid $\mathcal{E}|_p$ and repartition the parallel grid, if necessary. When a `DataHandle` object is passed, also the data associated with the object is redistributed. The `gather()` method is called to pack user data before an entity is sent to an other process and `scatter()` unpacks the data on the destination process.

3.6 Existing Implementations of the DUNE Grid Interface

At the present state of development, the `dune-grid` module contains six implementations of the DUNE grid interface. Three of them, `YaspGrid`, `OneDGrid`, and `SGrid` are full grid implementations, while the others are wrappers for legacy code which has to be obtained and installed separately.

`AlbertaGrid`: The grid manager of the ALBERTA toolbox [18]. ALBERTA supports simplicial grids in one, two, and three space dimensions with bisection refinement.

`ALUGrid`: A parallel 2d and 3d hexahedral and tetrahedral grid with nonconforming refinement, and dynamic load balancing [1, 8].

`OneDGrid`: A 1d grid with local mesh refinement.

`UGGrid`: The grid manager of the UG toolbox [3]. UG provides a parallel grid manager in two and three space dimensions that supports hybrid meshes with red–green or nonconforming refinement.

`YaspGrid`: A structured parallel grid in n space dimensions.

DUNE release 1.0 includes a prototype implementation of the Grid Interface. Unlike all other grids currently available it implements all optional methods of the grid interface. It is not tuned for efficiency and should be used for debugging and educational purposes only.

`SGrid`: Prototype implementation of an n-dimensional structured grid in an m-dimensional world.

4 Applications

In this section we will present three DUNE applications. Each of them acts as an example for one of the three design goals.

The first example will show a grid-independent discretization, which runs on all grids available in the `dune-grid` module. The discretization can take advantage of certain grid features, like local mesh refinement, if available. This allows to directly compare the speed and accuracy of different grid managers.

The second example examines the overhead introduced by the abstract interface. It compares an explicit finite volume scheme implemented on the

DUNE grid interface with an implementation based directly on the underlying grid. We have chosen an algorithm which consists to a large part of grid operations and should therefore suffer a lot from additional overhead. The example will show that even in this very challenging case the performance loss is within an acceptable range, compared to the benefits one can gain from the abstract interface.

In the last example we will show a second use of legacy code through the abstract interface. In fact, not only does the interface separate the grid implementation from the application, but it also cleanly separates the different grid implementations from each other. This way it is possible to combine several legacy grid managers and newly-implemented DUNE grids in a single application. This opens new possibilities, for example, for multi-physics and domain-decomposition applications.

4.1 Grid Independent Programming – Generic Discretization of an Elliptic PDE

We consider the second order elliptic model problem

$$-\Delta u = f \qquad \text{in } \Omega = (-1/2, 1/2) \times (0,1) \times (0,1), \tag{1a}$$

$$-\nabla u \cdot \mathbf{n} = 0 \qquad \text{on } \Gamma_N = \{(x,0,z) \mid -1/2 < x < 0,\ 0 < z < 1\}, \tag{1b}$$

$$u = g \qquad \text{on } \Gamma_D = \partial\Omega \setminus \Gamma_N \tag{1c}$$

where the right hand side f and Dirichlet boundary conditions g have been chosen such that the solution u is

$$u(r,\varphi,z) = r^{\frac{1}{2}} \sin\left(\frac{\varphi}{2}\right) 4z(1-z)$$

in cylindrical coordinates. The solution is depicted graphically in Figure 1. It has a singularity along the line $(1/2, 0, z)$.

Eqs. (1a)-(1c) are solved numerically using standard conforming P_1 finite elements on adaptively refined grids using a residual-based error estimator $\|u - u_h\|_1 \leq C\sqrt{\sum_{e \in L^0} \eta_e^2}$ with the local estimators

$$\eta_e = h_e \|f\|_{0,e} + \frac{1}{2} h_e^{1/2} \|[\nabla u \cdot \mathbf{n}]\|_{0,\partial e \setminus \Gamma_D}.$$

The generic implementation of the adaptive finite element method works on grids of all element types and space dimensions, as well as with conforming and nonconforming refinement (hanging nodes). Figure 1 shows two such grids. One is a simplicial grid with bisection refinement (generated with

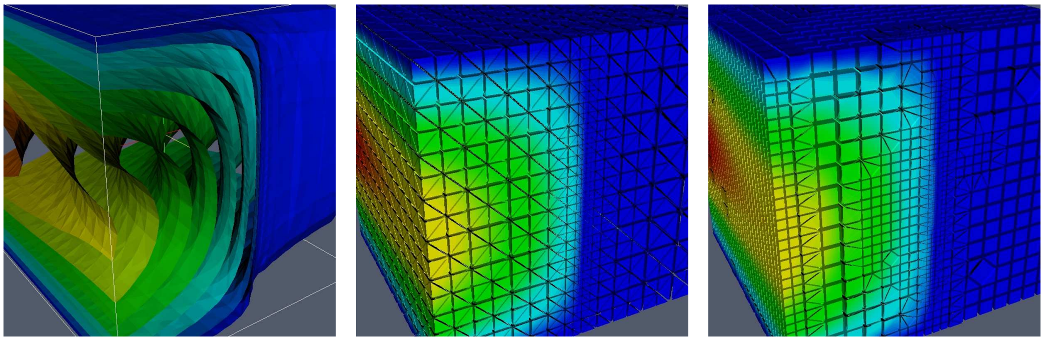

Figure 1: Interpolated analytical solution for the elliptic model problem and adaptively refined grids generated with `AlbertaGrid` and `UGGrid`.

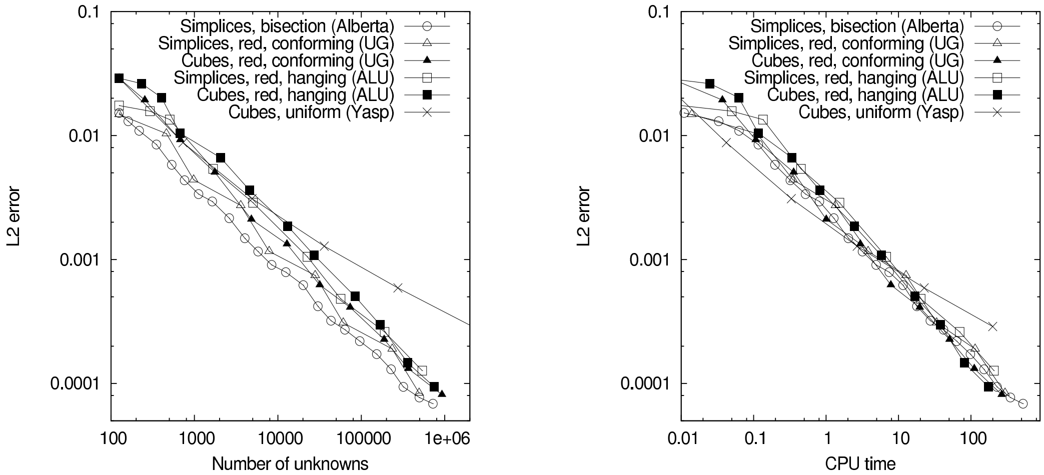

Figure 2: Comparison of error versus number of degrees of freedom and CPU time for various grids in three space dimensions.

`AlbertaGrid`), the other is a grid consisting of hexahedra, pyramids and tetrahedra with red/green type refinement (generated with `UGGrid`).

Figure 2 shows the L_2 norm of the discretization error with respect to the number of degrees of freedom using various types of grid refinement. The refinement strategy refines a fraction $0 < \alpha \leq 1$ of the elements with largest local indicators η_e. In the computations we have chosen $\alpha = 0.14$ for the grids subdividing one element into eight elements and $\alpha' = (1 + 7\alpha)^{1/3} - 1$ for the bisection grid (this results in approximately the same number of elements after three bisection steps).

The generic implementation now allows for a comparison of different grid refinement techniques. The graph in Fig. 2 shows that all adaptive grids provide the same asymptotic convergence order (as indicated by the slope

Table 1: Wall-time and time per degree of freedom for different grid implementations, the number of degrees of freedom (N), relative time for various components of the adaptive algorithm and the L_2 error. Meaning of the abreviations: MAT: construction of the sparsity pattern, ASS: the matrix assembly, SLV: the linear solver (CG with Gauß-Seidel), EST: the error estimator, ADP: the adaptation (consisting of grid refinement REF and vector reorganization but excluding error estimation), REF: the grid refinement.

Grid	N	$T[s]$	$\frac{T}{N}[\mu s]$	Relative Times [%]						Error
				MAT	ASS	SLV	EST	ADP	REF	
s, Alberta	496304	117.8	237	11	14	4.8	39	32	7.9	$7.7 \cdot 10^5$
s, UG	493030	175.3	356	11	17	6.1	29	37	33	$8.3 \cdot 10^5$
s, ALUGrid	537515	134.8	251	24	24	6.2	28	18	3.9	$12.7 \cdot 10^5$
c, UG	365891	59.6	163	14	25	8.4	26	26	22	$13.3 \cdot 10^5$
c, ALUGrid	360118	42.2	117	26	30	10	22	12	2.4	$14.7 \cdot 10^5$
c, YaspGrid	274625	19.7	72	22	34	14	25	5.1	0.0	$59.0 \cdot 10^5$

in the log-log plot) but the constants are different. Uniform mesh refinement (`YaspGrid`) is clearly asymptotically worse. The best results are obtained with simplices and bisection refinement (generated by `AlbertaGrid`) followed by simplicial and cube grids with conforming closure (generated with `UGGrid`). Last are the simplicial and cube grids with hanging nodes (generated with `ALUGrid`).

Table 1 shows timings for different parts of the adaptive algorithm on the different grids. All times are given in seconds and have been measured on a Laptop-PC with an Intel T2500 Core Duo processor with 2.0 GHz, 667 MHz FSB and 2 MB L2 cache using the GNU C++ compiler in version 4.0 and -O3 optimization. For solving the linear system a conjugate gradients solver preconditioned with symmetric Gauß-Seidel was used (residual norm reduction 10^{-3}). The following things can be observed:

- Simplicial grids with bisection refinement (`AlbertaGrid`) are fastest in terms of error versus number of unknowns. Note, however, that three times as much adaptation steps are necessary unless bisection is applied multiple times without intermediate computation. Therefore, the cube grids are typically more efficient in terms of error versus CPU time (see Fig. 2).

- For the accuracy 10^{-4} `ALUGrid` with cubes is fasted followed by `UGGrid` with cubes and `AlbertaGrid` with simplicial bisection refinement.

- Evaluation of the residual-based error estimator is more costly on the simplicial meshes as compared to the cube meshes due to the larger

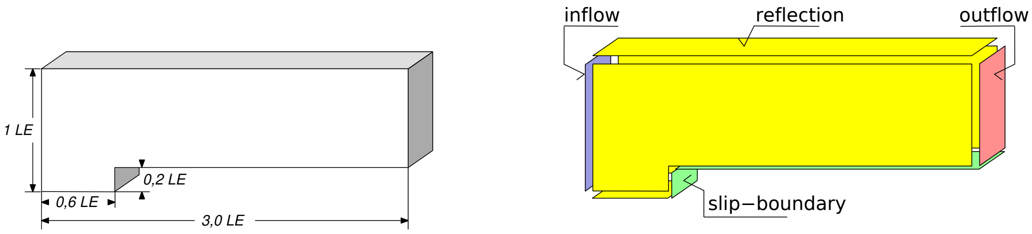

Figure 3: Setting for the forward-facing step problem.

number of faces relative to vertices.

- In the implementations based on ALBERTA and `ALUGrid` refinement of the grid is cheap and most of the time for grid adaptation is spent in the reorganization of the vector of unknowns. For `UGGrid` it is exactly the opposite.

- The structured grid is about three times faster than the unstructured `AlbertaGrid` for the same number of unknowns. Memory requirements are about four times lower for the structured grid (the memory required is the memory for the sparse matrix).

4.2 Efficiency of the Grid Interface – Forward Facing Step

In this example we examine the efficiency of the grid interface. We will measure the performance loss caused by the DUNE interface layer. An implementation of the model problem using the DUNE interface will be compared with one using the underlying grid manager directly.

The governing equations are the compressible Euler equations of gas dynamics (see [15, Section 5.1]). The forward-facing step benchmark problem [28] for a perfect gas law with $\gamma = 1.4$ is used. The domain is shown in Figure 3. The initial data is $\mathbf{U}_i^0 = (\rho_0, (\rho u)_0, 0, 0, e_0)$ with the initial density $\rho_0 = 1.4$, the initial product of density and velocity $(\rho u)_0 = 4.2$, and the initial energy $e_0 = 8.8$. The Dirichlet inflow boundary condition, described by the initial value, remains constant over time. This leads to a Mach three flow in the “wind-tunnel”.

The numerical scheme is a time-explicit Riemann-solver-based locally adaptive finite volume scheme, described in [9, 19]. Note that the implementation of the flux functions describing this Riemann solver is the same for both implementations as we want to study only the performance loss introduced by the grid interface.

For each time step the algorithm consists of five parts:

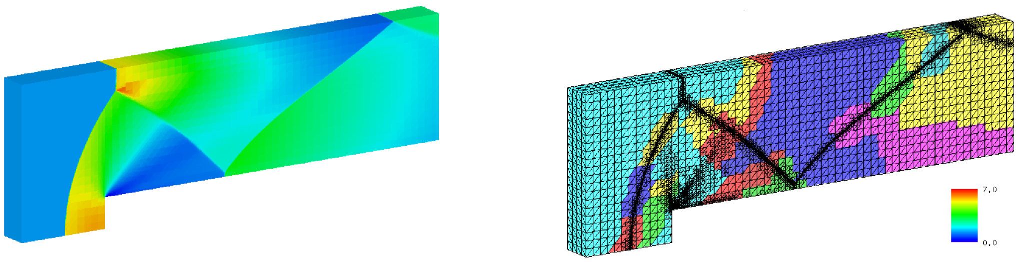

Figure 4: Density distribution and corresponding grid at computational time $T = 3$ on 16 processors using `ALUGrid`.

Communication of the Solution: Distribute the old solution among all processes.

Flux Evaluation: For each leaf entity evaluate the flux from the neighboring leaf entities. During this step the maximal admissible local time step size is computed.

Communication of Global Time Step: Calculate the global time step from the local time steps computed during the flux evaluation.

Evolution: Compute the conservative quantities at the next time step by evolving the current ones according to the flux.

Adaptation and Load Balancing: Refinement and coarsening of the grid as well as re-partitioning is done. A detailed description can be found in [19].

Flux evaluation, evolution, adaptation and load balancing strongly involve grid operations. For comparison the run-times for these three steps will be measured.

The described algorithm was implemented once using `ALUGrid` only via the DUNE interface and again using the `ALUGrid` legacy methods and data structures directly. The simulations were performed on the HP XC6000 Linux Cluster at the SSC Karlsruhe using $P = 4, 8, 16,$ and 32 processors. In Figure 4 one can see the density distribution for the 16 processor run with `ALUGrid` in three space dimensions.

Figure 5 shows the average total run-time per time step as well as the run-times for the computation of the fluxes, the evolution step, and the grid adaptation per time step averaged in the time interval $[1.5, 2.0]$. The results from the start of the simulation were excluded since at that stage the grid is too coarse to reach meaningful conclusions on 32 processors. Our results

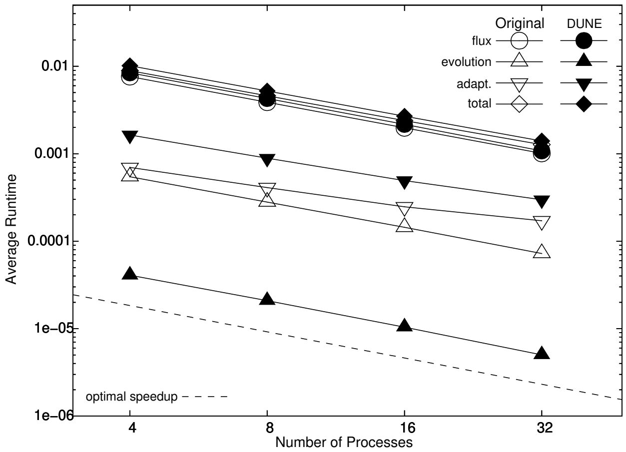

Figure 5: Total run-time and run-time for selected parts of the algorithm. Computations are done for $P = 4, 8, 16, 32$ processors.

	Relative performance loss [%]			
P	flux	evolve	adapt.	total
4	7.8	-5.0	9.3	12
8	7,5	-5.0	9.2	12
16	6.9	-5.0	9.2	11
32	4.9	-5.0	9.1	9

Table 2: Performance loss due to the DUNE grid interface. Total loss and loss within selected parts of the algorithm with respect to the total run-time of the DUNE implementation are shown.

demonstrate that the DUNE interface hardly reduces the efficiency of the numerical scheme, which confirms the observations from [4].

Table 2 shows the relative contribution to the performance gap from the grid-related parts of the algorithm. Although the explicit finite volume scheme is very challenging for a general grid interface, the difference between the original code and the DUNE code in the overall run-time is small (about 9% – 12%). While the DUNE code is inferior especially in the adaptation and flux computation it is more efficient than the legacy code when evolving the quantities to the next time step.

The major difference between the implementation of the scheme in DUNE compared to using `ALUGrid` directly concerns the storage of the data. In the `ALUGrid` implementation, the data is stored directly in the objects representing the grid entities. As during the flux computation and the grid adaptation phase the data is accessed along with the grid entities and their geometric information the data access is very efficient since it is already loaded into

original code					DUNE		
P	$T\,[s]$	$S_{4\to P}$	$E_{4\to P}$	P	$T\,[s]$	$S_{4\to P}$	$E_{4\to P}$
4	0.0089			4	0.0101		
8	0.0046	1.93	0.97	8	0.0052	1.95	0.97
16	0.0024	3.72	0.93	16	0.0027	3.78	0.94
32	0.0013	7.01	0.88	32	0.0014	7.26	0.91

Table 3: Speedup and efficiency of the original code and the DUNE implementation, measured with respect to a run with four processors using a fixed-size problem.

the cache. This is true for other time explicit schemes, too. Furthermore the reorganization of the grid during the adaptation process is very efficient since storage space for the data is automatically allocated together with the geometric information for the new entities. Since grid adaptation is performed in each time step, the time needed for the grid modification is comparable to the cost of the numerical scheme (about 20% of the overall time for both implementations).

When updating the current solution to the new time step, data storage in a consecutive vector separate from the grid in DUNE becomes a significant advantage. The legacy `ALUGrid` implementation is forced to do a grid traversal, which reduces the bandwidth available for the vector update. The DUNE implementation mainly accesses a vector and can fully utilize the memory bandwidth. Here DUNE can compensate a part of the performance loss. On 32 processors the loss in flux step and the benefit in the update step cancel each other. Note that for implicit schemes this cache efficiency advantage will be even bigger in the linear algebra used.

Performance tests show that for this problem both codes have a parallel efficiency which is close to optimal. In addition we also demonstrate the parallel efficiency of the code using the definitions of speedup $S_{4\to P}$ (speedup from 4 to P processes) and $E_{4\to P}$ (efficiency from 4 to P processes) from [8]. Since we study a fixed size problem, the parallel overhead increases with the number of processors while the cost of the numerics decreases. Hence we cannot expect optimal efficiency in this case.

The corresponding values for the original code and the DUNE code are shown in Table 3 (left) and Table 3 (right), respectively. We observe that the efficiency is quite high (around 90%) and that the values are approximately the same for both implementations of the algorithm.

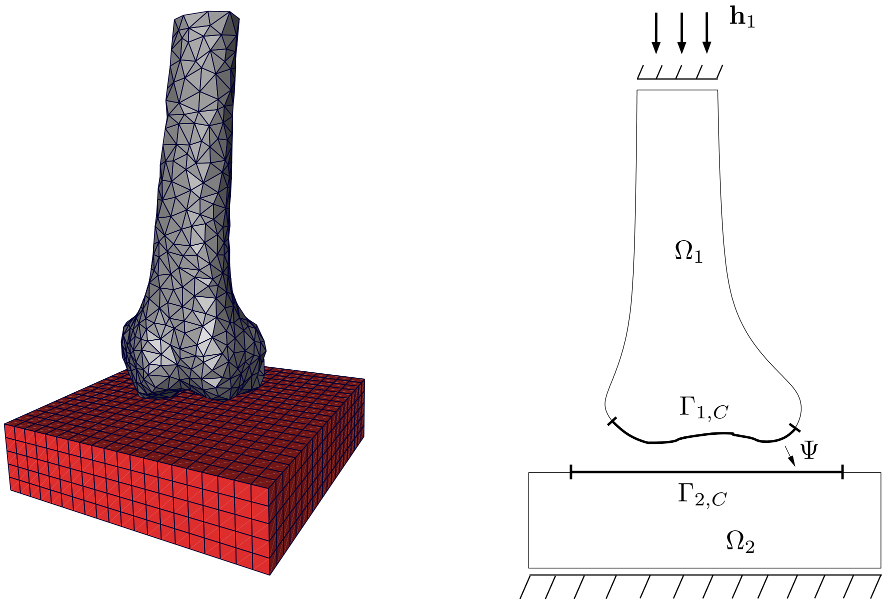

Figure 6: Two-body contact problem. Left: coarse grids, right: schematic view.

4.3 Coupling Different Grid Implementations – A Contact Problem

In this last example we will show the use of more than one grid manager in a single application. We use a two-body contact problem from linear elasticity. It models the mechanical behaviour of two elastic bodies which undergo small deformations and possibly come into contact with each other. More formally, consider two disjoint domains Ω_1, Ω_2 in $\mathbb{R}^d, d \in \{2,3\}$. The boundary $\Gamma_i = \partial\Omega_i$, $i \in \{1,2\}$, of each domain is decomposed in three disjoint parts $\Gamma_i = \Gamma_{i,D} \cup \Gamma_{i,N} \cup \Gamma_{i,C}$. Let $\mathbf{f}_i \in (L_2(\Omega_i))^d$ be body force density fields, $\mathbf{h}_i \in (H^{1/2}(\Gamma_{i,D}))^d$ be prescribed boundary displacements, and $\mathbf{t}_i \in (H^{-1/2}(\Gamma_{i,N}))^d$ be fields of surface tractions. Then we look for functions $\mathbf{u}_i \in (H^1(\Omega_i))^d$ which fulfill

$$-\operatorname{div}\sigma(\mathbf{u}_i) = \mathbf{f}_i \tag{2a}$$

$$\mathbf{u}_i = \mathbf{h}_i \qquad \text{on } \Gamma_{i,D}, \tag{2b}$$

$$\sigma(\mathbf{u}_i)\mathbf{n} = \mathbf{t}_i \qquad \text{on } \Gamma_{i,N}, \tag{2c}$$

where $\mathbf{n}$ is the outward unit normal, the stress tensor σ is defined as $\sigma(\mathbf{u}) = \frac{E}{1+\nu}(\epsilon + \frac{\nu}{1-2\nu}\operatorname{tr}\epsilon I)$, and $\epsilon(\mathbf{u}) = \frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T)$ is the linear strain tensor. In addition, the following contact condition is stated. When modelling contact in linear elasticity it is usually assumed that the areas where contact

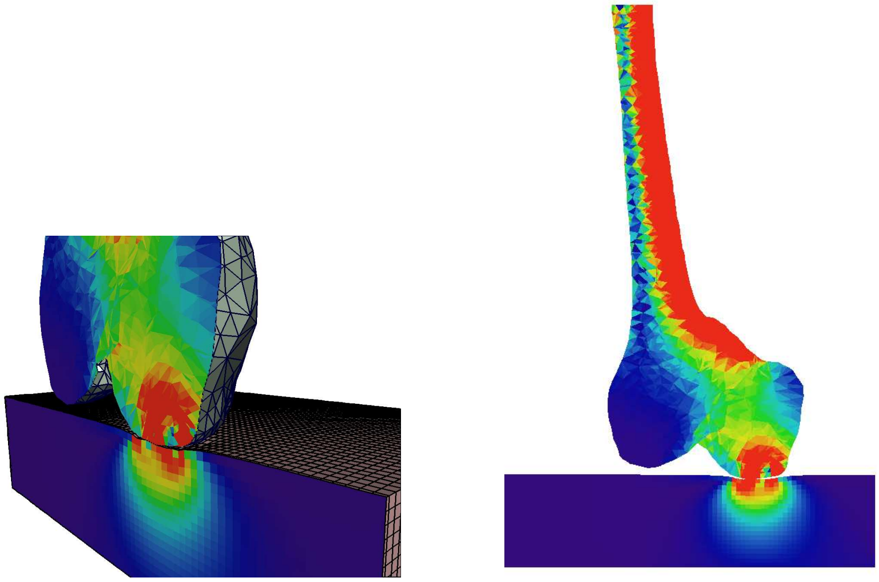

Figure 7: Two-body contact problem. Left: close-up view of the deformed solution, right: vertical cut through the von-Mises stress field.

occurs will be subsets of parts of the boundary $\Gamma_{1,C}$, $\Gamma_{2,C}$, chosen a priori. These two contact boundaries are then identified using a homeomorphism $\Psi : \Gamma_{1,C} \to \Gamma_{2,C}$. With this identification it is possible to define an initial distance function $g : \Gamma_{1,C} \to \mathbb{R}$, $g(x) = \|\Psi(x) - x\|$. The contact condition then states that the relative normal displacement of any two points x, $\Psi(x)$ should not exceed this normal distance, in formulas

$$\mathbf{u}_1|_{\Gamma_{1,C}} \cdot \mathbf{n}_1 + (\mathbf{u}_2 \circ \Psi)|_{\Gamma_{2,C}} \cdot \mathbf{n}_2 \leq g. \tag{3}$$

This condition can be derived as a linearization of the actual nonpenetration condition and is reasonable to use in the context of linear elasticity [11].

For the discretization of the problem we use first order Lagrangian elements. In order to retain optimal error bounds even in the presence of the contact condition, we use mortar elements for its discretization. That is, (3) is discretized not by its node-wise equivalent but in a weak form requiring

$$\int_{\Gamma_{1,C}} \big[\mathbf{u}_1|_{\Gamma_{1,C}} \cdot \mathbf{n}_1 + (\mathbf{u}_2 \circ \Psi)|_{\Gamma_{2,C}} \cdot \mathbf{n}_2\big]\theta \, ds \leq \int_{\Gamma_{1,C}} g\theta \, ds \tag{4}$$

for all θ from a suitable cone of mortar test functions defined on $\Gamma_{1,C}$ [27]. The resulting discrete obstacle problem is solved with a monotone multigrid method as described by Kornhuber et al. [13].

As the geometry of our problem we choose the distal part of a human femur being pressed onto a block-shaped foundation. The femur geometry is taken from the Visible Human data set [26] and a tetrahedral grid is generated using the AMIRA mesh generator [22]. As the grid implementation we chose `UGGrid` for its high geometry flexibility. In addition to being able to handle arbitrary grids with several element types it allows to use automatically parametrized boundaries as described in [14].

For the obstacle we choose the `SGrid` implementation, which is the prototypical implementation of a uniform hexahedral grid. The C++ methods that assemble Equations (2) and (4) take the data types of the grids as template parameters and instantiate with no problem even when two different grid implementations are used. Besides the construction of the multigrid transfer operators, which also depend on the grids via template parameters, the monotone multigrid solver is a purely algebraic algorithm and therefore independent of the grid types.

The coarse grids consist of 3787 tetrahedra for the bone and 2000 hexahedra for the obstacle. Material parameters are $E = 17$GPa, $\nu = 0.3$ for the bone and softer $E = 250$MPa, $\nu = 0.3$ for the obstacle. The latter is clamped at its base, whereas a uniform displacement of 3 mm downward is prescribed on the top section of the bone (see Fig. 6). The bone serves as the non-mortar domain. During computation the bone grid is refined twice using a Zienkiewicz-Zhu error estimator [30]. Accordingly, the obstacle grid is twice refined uniformly. The resulting grids have 104305 and 128000 elements, respectively, and the resulting linear system contains 472683 variables. The result can be seen in Fig. 7, where the left shows a close-up of the refined grid in the contact region and the right a vertical cut through the von-Mises stress field.

5 Conclusions and Future Work

We have shown that it is possible to provide an abstract template-based representation for parallel grids used in scientific computing. The basic feature of our approach is the clear separation of the underlying data structures and the algorithms via a slim grid interface.

By writing code adhering to the presented interface the programmer has the flexibility to use the same code on grids supporting different features, e. g., unstructured and structured grids. This allows the reuse of existing algorithms on new (specialized) grids. It is possible to write simulation codes although a specialized grid implementation needed for production code is not yet fully available. We have shown that it is easily possible to evaluate different (adaptive) grid implementations for a problem allowing the user to choose the most efficient solution for his or her current problem and algorithm.

By using the presented generic programming approach in C++ it is possible to get this kind of flexibility without sacrificing the run-time efficiency of the code. This allows combining the efficiency of the programmer with efficiency of the program. We showed this by comparing a well established parallel production code with a (partial) reimplementation using the same grid via our new grid interface.

The flexibility achieved by the presented approach allows coupling of existing legacy codes working on different grids. We showed that it is possible to compute coupled problems on different grids by combining different implementations.

The presented generic grid interface is far more powerful and flexible than shown with the currently available grid managers. Further grid managers for other special application scenarios, e.g. spherical grids, are easily implemented.

So far, unified interfaces exist only for the grid managers and the linear algebra. For the future it is important to design and implement a discretisation module linking the two crucial parts. This task is currently being worked on.

References

[1] ALUGrid Library. `http://www.mathematik.uni-freiburg.de/IAM/Research/alugrid/`.

[2] W. Bangerth. Using modern features of C++ for adaptive finite element methods: Dimension-independent programming in deal.II. In M. Deville and R. Owens, editors, *Proceedings of the 16th IMACS World Congress 2000, Lausanne, Switzerland, 2000*, 2000. Document Sessions/118-1.

[3] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, and C. Wieners. UG - A flexible software toolbox for solving partial differential equations. *Comp. Vis. Sci*, 1:27–40, 1997.

[4] P. Bastian, M. Droske, C. Engwer, R. Klöfkorn, T. Neubauer, M. Ohlberger, and M. Rumpf. Towards a unified framework for scientific computing. In R. Kornhuber, R. Hoppe, D. Keyes, J. Périaux, O. Pironneau, and J. Xu, editors, *Proceedings of the 15th Conference on Domain Decomposition Methods*, number 40 in LNCSE, pages 167–174. Springer-Verlag, 2004.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract framework. *in preparation*, 2007.

[6] M. Blatt and P. Bastian. The iterative solver template library. In *Proc. of the Workshop on State-of-the-Art in Scientific and Parallel Computing*, Lecture Notes in Scientific Computing. Springer-Verlag, 2006. accepted.

[7] M. Blatt and P. Bastian. On the generic parallelisation of iterative solvers for the finite element method. *Int. J. Computational Science and Engineering*, 2007. submitted.

[8] A. Burri, A. Dedner, R. Klöfkorn, and M. Ohlberger. An efficient implementation of an adaptive and parallel grid in DUNE. Technical report, Submitted to: Proceedings of The 2nd Russian-German Advanced Research Workshop on Computational Science and High Performance Computing, Stuttgart, March 14 - 16, 2005.

[9] A. Dedner, C. Rohde, B. Schupp, and M. Wesenberg. A parallel, load balanced MHD code on locally adapted, unstructured grids in 3d. *Comp. Vis. Sci*, 7:79–96, 2004.

[10] DUNE – Distributed and Unified Numerics Environment. `http://dune-project.org/`.

[11] C. Eck. *Existenz und Regularität der Lösungen für Kontaktprobleme mit Reibung.* PhD thesis, Universität Stuttgart, 1996.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[13] R. Kornhuber, R. Krause, O. Sander, P. Deuflhard, and S. Ertel. A monotone multigrid solver for two body contact problems in biomechanics. *Comp. Vis. Sci*, accepted for publication, 2006.

[14] R. Krause and O. Sander. Automatic construction of boundary parametrizations for geometric multigrid solvers. *Comp. Vis. Sci*, 9: 11–22, 2006.

[15] D. Kröner. *Numerical schemes for conservation laws.* Wiley–Teubner, Stuttgart, 1997.

[16] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 2001. ISBN 0-201-37923-6.

[17] C. Pflaum. Expression templates for partial differential equations. *Comp. Vis. Sci*, 4(1):1–8, 2001.

[18] A. Schmidt and K. Siebert. *Design of Adaptive Finite Element Software – The Finite Element Toolbox ALBERTA*. Springer, 2005.

[19] B. Schupp. *Entwicklung eines effizienten Verfahrens zur Simulation kompressibler Strömungen in 3D auf Parallelrechnern.* PhD thesis, Mathematische Fakultät, Universität Freiburg, 1999.

[20] J. Seymour. Views – a C++ standard template library extension. `http://www.zeta.org.au/~jon/STL/views/doc/views.html`, 1996.

[21] J. Sick and A. Lumsdane. A modern framework for portable high-performance numerical linear algebra. In H. Langtangen, A. Bruaset, and E. Quak, editors, *Advances in Software tools for scientific computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*, pages 1–56. Springer-Verlag, 2000.

[22] D. Stalling, M. Westerhoff, and H.-C. Hege. Amira: A highly interactive system for visual data analysis. In C. Hansen and C. Johnson, editors, *The Visualization Handbook*, chapter 38, pages 749–767. Elsevier, 2005.

[23] D. Vandevoorde and N. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley, 2003.

[24] T. Veldhuizen. Blitz++: The library that thinks it is a compiler. In H. Langtangen, A. Bruaset, and E. Quak, editors, *Advances in Software tools for scientific computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*, pages 57–87. Springer-Verlag, 2000.

[25] T. Veldhuizen. Techniques for scientific C++. Technical report, `http://extreme.indiana.edu/~tveldhui/papers/techniques/`, 1999.

[26] Visible Human Project. `http://www.nlm.nih.gov/research/visible/visible_human.html`.

[27] B. Wohlmuth and R. Krause. Monotone methods on nonmatching grids for nonlinear contact problems. *SIAM J. Sci. Comp.*, 25(1):324–347, 2003.

[28] P. Woodward and P. Colella. The numerical simulation of two-dimensional fluid flow with strong shocks. *Journal of Computational Physics*, 54:115–173, 1984.

[29] R. Young and I. MacPhedran. Internet finite element resources. Webpage at: `http://homepage.usask.ca/~ijm451/finite/fe_resources/fe_resources.html`.

[30] O. Zienkiewicz and J. Zhu. A simple error estimator and adaptive procedure for practical engineering analysis. *Int. J. Num. Meths. Eng.*, 24: 337–357, 1987.