GrAL- The Grid Algorithms Library

Guntram Berti

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, 53757 St. Augustin, Germany
berti@ccrl-nece.de

Abstract. Dedicated library support for mesh-level geometry components, central to numerical PDE solution, is scarce. We claim that the situation is due to the inadequacy of traditional design techniques for complex and variable data representations typical for meshes. As a solution, we introduce an approach based on generic programming, implemented in the C++ library GrAL, whose algorithms are able to run on any mesh representation. We present the core design of GrAL and highlight some of its generic components. Finally, we discuss some practical issues of generic libraries, in particular efficiency and usability.

1 Introduction

Software for the numerical solution of PDEs generally involves a lot of components, which can be partitioned into several layers. At the bottom level, we find container data structures and corresponding algorithms, such as sorting and searching. In this paper, we are concerned with the next higher level, namely representations for geometric structures (meshes), and algorithms operating on top of these, for instance numerical discretization such as FEM or FV algorithms, but also non-numeric algorithms such as searching cell neighbors or checking mesh quality.

Algebraic components, like matrix and vector representations and algorithms for solving linear or non-linear systems, constitute the next higher layer. Some of these components may need to access the grid representation, e. g. some preconditioners, inter-grid operators of multi-grid algorithms, or a-priori determination of matrix structure. In addition, handling boundary conditions may involve complex geometric calculations, for instance for contact problems. Besides the numerical algorithms, components for pre- and postprocessing also operate directly on grids in a multitude of ways.

All this means that the mesh layer is central to numerical PDE solution. To the author's knowledge, however, there is no dedicated mesh library available from which we can take, say, a mesh contact detection algorithm for inclusion into a PDE solver in a practically useful way.

We feel that a primary cause for this apparent lack is that traditional ways of library design cannot cope with the variability of geometric data structures, because they either introduce a tight coupling between representational and algorithmic code, or rely on a conversion (that is, physical copy) of data structures.

In this paper, we introduce the Grid Algorithms Library GrAL (available at [1]), which overcomes these difficulties by defining an abstract interface for grids. Using a generic programming approach [2] in C++, GrAL achieves a complete decoupling of algorithms and data structures.

After briefly analyzing the problems with traditional library design and discussing related work, we give an overview over the core design of GrAL, and highlight some of GrAL's generic components, which includes support for parallel PDE solution. Then, we discuss some practical aspects by which generic libraries differ from other approaches. Finally, future options are outlined.

2 Problems with Traditional Grid-related Components

If we review typical grids used for numerical simulations, we encounter a wide variety of different types: Cartesian and curvilinear mapped structured grids, multiblock and semi-structured [3], unstructured simplicial or arbitrary cell meshes, to hybrid [4] or chimera-type grids. Taking into account the virtually unlimited possibilities for representing these grids, it becomes clear that no "standardization" approach on the representation level can ever be successful. It is also clear that the chance for using algorithms implemented for one kind of grid data structure in a different context are practically zero. So, basically, traditional approaches to create reusable grid components are limited to the following:

1. Use a grid application program interface (API)
2. Use file coupling
3. Always use the same standard data structure
4. Implement the algorithm *ad hoc* for a given data structure

In the API approach, taken for instance by the VISUAL3 postprocessor [5], the grid has to be copied into the data structures predefined by the library routine, and possibly vice versa, which might not be trivial. Second, copying can be grossly inefficient if the algorithm itself is fast or operates only locally (e. g. point location). Memory might become a bottleneck, especially if the grid uses an optimized data structure with implicit connectivity. File coupling evidently has the same – but exacerbated – difficulties.

In sum, these approaches are viable only if a substantial amount of work is done on the grids, justifying both the overhead of copying and the programming work for converting the data structures. This is commonly the case for instance for mesh generation or visualization, where file coupling is the rule.

Using standard data structures works only for cases with limited representational choice, such as structured grids, see [6, 7]. A prominent example for this approach (albeit not for grids) probably is LAPACK, which prescribes a set of representations for dense matrices. Thus, in spite of its deficiencies, option 4 all too often is the only choice left, witnessed by the countless number of implementations of basic functionality like cell neighbor search in application codes.

The solution we offer here has none of these drawbacks. It uses the common underlying mathematical structure of grids to define an abstract interface for them, which is powerful enough so that a large class of algorithms can be implemented *generically* on top of this interface. This approach reverses the flow of information implicit in a classical API: In the case of an API, we have to learn the interface of the library component. In our approach, in contrast, the internals of a given user data structure are presented in a structured and uniform way to the library component.

The solution is inspired by the C++ Standard Template Library STL [8]. A similar approach is taken by the Boost Graph Library BGL [9], which applies the generic paradigm to general graphs, and achieves a comparable level in decoupling algorithms from data structures. The Computational Geometry Algorithms Library CGAL [10] focuses on general-purpose geometric data structures and algorithms. With respect to genericity, some of its algorithms are implemented in a really data-structure independent way, while many CGAL algorithm implementations tend to be less generic and to work only with CGAL's data structures. VIGRA (Vision with Generic Algorithms) [11] is a generic library for image processing, which concentrates on 2D regular image structures and achieves a high level of genericity of corresponding image processing algorithms. The boost community [12] offers a vast collection of basic generic libraries and generally pushes forward the generic programming approach.

3 The Core Design of GrAL

3.1 Requirements and Trade-offs

There are some essential requirements which our library should fulfill:

1. Complete decoupling of algorithms from data structures: Algorithms shall be usable with *any* data structure providing sufficient functionality
2. Constant reuse cost: Shall be able to use any algorithm, by creating a thin adaptation layer user data *once*
3. Efficiency: Shall maintain high performance with respect to a direct implementation

Thus, the process of creating an adequate interface is a compromise between expressiveness (we want to access the essential properties of grids), minimality (we don't want to create new interfaces for each new algorithm) and efficiency (we want algorithms to run almost as fast as direct implementations).

The software technology we chose for meeting these requirements is the generic programming approach. To cite D. Musser [13], his working definition of generic programming is

> "programming with concepts," where a concept is defined as a family of abstractions that are all related by a common set of requirements. A large part of the activity of generic programming, particularly in the

> design of generic software components, consists of concept development– identifying sets of requirements that are general enough to be met by a large family of abstractions but still restrictive enough that programs can be written that work efficiently with all members of the family.

Technically, the abstract concepts get translated into type information by the programmer. The types give implementor and compiler at each point full information on the concepts involved and thus the possibility to act accordingly. This is one of the essential differences to object-oriented programming, where this information is lost behind abstract classes.

Two examples may elucidate this salient property of generic programming, where having a unified interface neither means that we are restricted to some least common denominator, nor places unrealistic requirements on the functionality of data structures. Concerning the first point, note that *specialization* is an integral part of generic programming. For example, if we implement a generic search algorithm / data structure, we realize that it can be implemented much more efficiently for Cartesian grids. Thus, we can create a special implementation for that case, and when the generic search component is used, the compiler possesses enough information to decide which version to use. Even better (and more in the spirit of generic programming), we could develop concepts for Cartesian grids and do a partial specialization of the algorithm usable for all Cartesian grids.

Second, a concrete grid component is not required to support the complete interface – in fact, it is in general not possible. For instance, if our data structure does not know about cell neighbor relationships, we cannot support the corresponding concept of the interface. Again, generic components can specialize according to the supported subset of the interface, or the missing functionality could be added on-the-fly.

The following presentation gives a high-level view, with focus more on the intuitive grasp of the concepts then on technical details. More can be found [14, 15], and the full (more formal) concept and interface specifications are continually updated in [1]. An example of a generic algorithm is given in section 3.7.

3.2 What Is A Grid? Some Definitions

The seemingly simple question in the section header has no obvious answer, because we really have to deal with a hierarchy of concepts, not just a single one. The definition of a grid given here is a very general one[1]. More powerful functionality can be derived if the grid is a regular subdivision of a manifold (possibly with boundary). Typically, minimal requirements of algorithms induce a much finer-grained system of grid properties, to detailed to be discussed here, see [14] for detailed definitions. Here, we will indicate restrictions of algorithms on a case-by-case basis.

[1] It is more general than that originally given in [14]

We distinguish between a combinatorial grid (abstract complex) and its geometric embedding. This important distinction is preserved in the interface definition. Also, the notion of mappings defined on grids finds its equivalent in the concept of a grid function.

Definition 1 (Abstract complex) *An abstract finite complex $\mathcal{C}$ of dimension d is a set of* elements *e, together with a mapping* $\dim : \mathcal{C} \mapsto \{0, \ldots, d\} \subset \mathbb{N}$, *(*$\dim(e)$ *is called the dimension of e), and a partial order $<$ (*side-of *relation) with $e_1 < e_2 \Rightarrow \dim(e_1) < \dim(e_2)$. Elements are named according to table 1. A* isomorphism *between abstract complexes $\mathcal{C}_1, \mathcal{C}_2$ is a bijective mapping $\Phi : \mathcal{C}_1 \mapsto \mathcal{C}_2$ with $e < f \Rightarrow \Phi(e) < \Phi(f)$.*

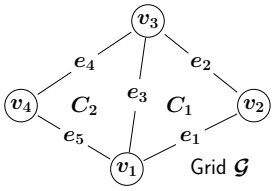

Fig. 1. A simple grid ...

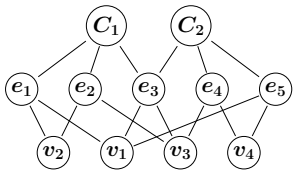

Fig. 2. ... and its incidence lattice

An abstract complex is a purely combinatorial entity, also known as partially ordered set or *poset*. In many important cases, this poset is even a *lattice* [16], cf. figures 1 and 2.

We need the notion of a geometric complex, too:

Definition 2 (Geometric realization of an abstract complex) *A geometric realization Γ of an abstract complex $\mathcal{C}$ is a Hausdorff space $\|\mathcal{C}\|$ and a mapping*

$$\Gamma : \mathcal{C} \mapsto \Gamma(\mathcal{C}) = \|\mathcal{C}\| = \bigcup_{e \in \mathcal{C}} \Gamma(e) \quad \textit{with}$$

$$e_1 < e_2 \Leftrightarrow \Gamma(e_1) \subset \partial\Gamma(e_2) \quad \textit{and} \quad \partial\Gamma(e_2) = \bigcup_{e_1 < e_2} \Gamma(e_1) \quad \forall e_1, e_2 \in \mathcal{C}$$

Again, this is a very general definition. In most practical cases, the space $\|\mathcal{C}\|$ will be subset of $\mathbb{R}^k$ or a manifold.

3.3 Combinatorial Grid Interface

The combinatorial layer is concerned only with abstract complexes, that is, with posets or lattices. It is more complex than the geometric or the grid function layer, and is itself divided into several parts, discussed below: A *poset part*,

which corresponds to the very general definition given above, and a *lattice part*, which exploits the additional structure present in regularly subdivided manifolds. Furthermore, the boundaries of cells are accessible as first-class entities by means of *cell archetypes*, which allow to exploit the redundancy of cell combinatorics typical for FEM meshes. Finally, modifying operators on grids are presented in section 3.6.

Poset Layer: Incidence Iterators The *elements* of the grid are its "atoms" and named according to their dimension or codimension, see table 1. A minimal representation of an element of a fixed grid is called *element handle*, which may be simply an integer. Due to their minimality, handles are very useful e. g. for representing grid subranges or grid isomorphisms.

At a very basic level, a grid is a set of sequences: A sequence of its vertices, of its edges, and so on. We can model this property by introducing *grid sequence iterators* (table 1), which just have the standard (STL) iterator interface.

Table 1. Combinatorial grid entities

Element	dim	codim	Sequence Iterator
Vertex	0	d	VertexIterator
Edge	1	d-1	EdgeIterator
Facet	d-1	1	FacetIterator
Cell	d	0	CellIterator

In order to access the incidence relationship, we need *incidence iterators* (table 2). These allow for example to access the sequence of all vertices of a cell (VertexOnCellIterator), see fig. 3. The number of different incidence iterators is $d(d-1)$, where d is the grid dimension.

Table 2. The full set of incidence and adjacency (A) iterators in 3D

VertexOnVertexIt (A)	VertexOnEdgeIt	VertexOnFacetIt	VertexOnCellIt
EdgeOnVertexIt		EdgeOnFacetIt	EdgeOnCellIt
FacetOnVertexIt	FacetOnEdgeIt		FacetOnCellIt
CellOnVertexIt	CellOnEdgeIt	CellOnFacetIt	CellOnCellIt (A)

A similar concept are *adjacency iterators*, which relate elements of the same dimension. We define them only for vertices and cells, because there is no "na-

tural" definition for the intermediate dimensions, and they seem to be hardly used.

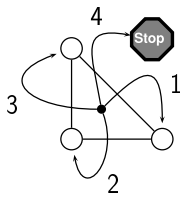

Fig. 3. Action of a VertexOnCellIterator (*Incidence iterator*)

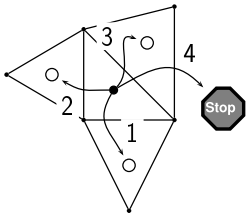

Fig. 4. Action of a CellOnCellIterator (*Adjacency iterator*)

As already mentioned, it is not required to implement all types of elements or iterators. Also, even if the kernel interface for an element type is supported, it does not need to be stored explicitly. The best example here is a Cartesian grid, where everything is given implicitly.

In spite of the apparent simplicity of this interface, it is sufficient for basing a simple finite volume solver upon it [14].

Lattice Layer: The Switch Operator Incidence iterators suffice for implementing a surprisingly large class of algorithms. However, there is no ordering relationship between different incidence iterators. In 2D, for example, vertices and edges incident to a cell might be ordered completely independently. Although a complete chain of *downward* incidence iterators (`FacetOnCell`, `EdgeOnFacet` and `VertexOnEdge` in 3D) contains in principle all combinatorial information of the mesh, we sometimes need a different, more direct way to access it.

The switch operator exploits the special combinatorial structure of regularly subdivided manifolds. The definition of switch for 2D meshes is explained by figure 5. Using switch, we can traverse a connected component of a grid's boundary (see [14] for details), or calculate the dihedral angle at a vertex (more precisely, we can traverse the corresponding edges or facets).

The switch operator is strictly local: Whereas two vertices incident to the same cell can be arbitrarily "far away" in some sense, all entities related by switch are incident.

Recursive Layer: Cell Archetypes It is very useful to access the boundary of a cell not just as a collection of grid elements of lower dimension via incidence iterators, but as a proper grid of dimension $d-1$. We call this cell boundary grid the *archetype* of the cell, because it can be seen as a "construction plan" which typically is shared by many cells of a grid.

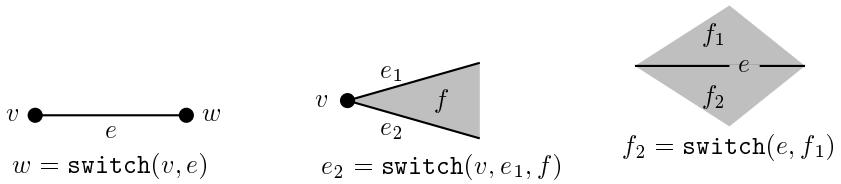

Fig. 5. The switch operation in 2D

Archetypes introduce a kind of recursion into the grid interface definition. They can be used to define the cell-based incidence iterators, which are just the sequence iterators of the cell archetype mapped to the global vertex number of the cell.

In most existing mesh data structures, archetypes are present only implicitly in some node ordering convention. Yet, to achieve for example generic copying of (especially 3D) meshes, explicit access to cell archetypes is essential to automatically find an isomorphism between two different numbering conventions, which is a common problem e. g. for hex elements.

3.4 Grid Geometries

Grid geometries represent geometric realizations (embeddings) of combinatorial grids. Thus, they map combinatorial to geometric entities: Vertices to points, edges to arcs, and so on, see figure 6. The grid geometry interface is open for additional properties or measures, for example lengths of edges, thus entailing better encapsulation of geometric decision: If edge lengths are computed in client code under the implicit assumption of linear segments, it would fail to profit from pre-calculated edge lengths, or would break for, say, isoparametric elements of higher order.

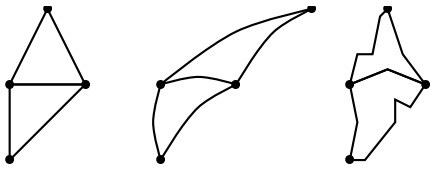

Fig. 6. Three embeddings for the mesh of fig 1

The separation from the combinatorial grid layer has a number of practical advantages. It allows to reuse the same combinatorial grid data structure with different embeddings, for example 2D domain and 3D surface grids. We can also use different geometries for the same grid simultaneously, for instance use

straight edges for FEM computation, and curved edges on the boundary for grid refinement. It is also easier to work with geometric embeddings into non-Euclidean spaces, because there are no (or much less) implicit assumptions on the geometry.

3.5 Grid Functions

Grid functions, also known as *fields* in PDE software, allow to store and access data on grid elements of any dimension. The underlying mathematical concept is simply that of a mapping from grid elements of a given dimension to some objects of some type T.

The typical grid functions used in PDE solvers are *total* ones: One data item is physically stored on each element. Equally useful are *partial* (sparse) grid functions. Their distinguishing feature is that the assignment of a default value to all element of a grid is a constant-time operation, and the size of a partial grid function is proportional to the number of elements with a value different from the default one. Many locally operating algorithms depend on this complexity guarantee of partial grid functions to achieve good runtime behavior.

For both kinds of grid functions generic implementations are provided by GrAL, see below and [17] for details on implementation techniques.

It is possible to decouple the storage of this application-dependent data completely from the combinatorial grid data structure, such that arbitrary types of data can be stored on a given grid. This approach is crucial to avoid coupling data structures to the algorithms using them – which would be even worse than the inverse coupling we overcame with the generic approach. Yet, it is often found in object-oriented grid data structures, where for instance state information is stored in vertex objects. The GrAL interface for grid functions is however general enough to cover this case, too: When a user-provided data structure has grid functions implemented this way, an adapter to GrAL can easily be written.

3.6 Mutating Primitives

A large class of grid algorithms important for PDE solutions does need only read access to the grids; for these, the interface components presented so far are sufficient.

However, in some cases we need to change the grid: Obviously, this is the case if we read the grid from some file, or copy it from another grid. Also, for refinement, coarsening or optimization, a grid has to be changed.

In search of a general solution which allows efficient implementations for a large class of data structures, we found that in virtually all cases we investigated it proved sufficient to use *coarse grained* mutating primitives (in contrast to *atomic* primitives like Euler operators [18]). We can do with just three of them: Copying grids, enlarging (gluing) grids, and cutting something off a grid. These primitives maintain a grid morphism between the source and the copy, in order to allow the transfer of additional information, such as grid functions. Mutating primitives are discussed in more detail in [14] and [15].

The copy primitive can be seen as a generalized constructor, and it can be used to implement transparent file I/O or data structure conversion, necessary for using traditional libraries using API or file coupling. To this end, a pair of input/output adapters which both have a minimal grid interface is implemented for each file format. Reading the file is achieved by copying from the input adapter, and writing is equivalent to copying to the output adapter. An example in GrAL is the output adapter for GMV [19].

A generic copy operation poses interesting challenges: For instance, in 3D grids, numbering of local vertices often differs between applications e. g. for hex cells. In order to copy from one numbering to another, we need to calculate a grid isomorphism between the two hex representation (or more precisely, their *archetypes*, see above), which is performed by a GrAL algorithm.

3.7 A simple example

In order to show the generic paradigm in action, we discuss a very simple example – averaging cell values on incident vertices, which is for example needed in some higher order finite volume algorithms. In the algorithms, $\mathcal{G}^k$ denotes the set of elements of dimension k of the grid $\mathcal{G}$.

IN: $U : \mathcal{G}^d \mapsto \mathbb{R}^k$ *(state on cells)*
OUT: avg: $\mathcal{G}^0 \mapsto \mathbb{R}^k$ *(average on vertices)*
  avg $\leftarrow$0
  for all Cells $c \in \mathcal{G}^d$ **do**
    vol $\leftarrow$volume(C)
    for all vertices v of C **do**
      avg(v) += vol * $U(C)$
  for all vertices $v \in \mathcal{G}^0$ **do**
    divide avg(v) by total volume of cells incident to v

The generic implementation of this algorithm is very similar to the pseudo-code. Compared with typical low-level implementations, the code is very readable and features an increased *intentionality*: For example, it is clearer over which grid entities is iterated, than it would be if just some integer loops were used:

```
// input data
grid_type grid;
geom_type geom;
grid_function<Cell,  vec_k> U;
// ...

// output data
grid_function<Vertex,vec_k > avg  (grid,vec_k(0));
grid_function<Vertex,double> vol_v(grid,0);

// algorithm
for(CellIterator c(grid); !c.IsDone(); ++c) {
  double vol_c = geom.volume(*c);
```

```
  for(VertexOnCellIterator vc(*c); !vc.IsDone(); ++vc) {
    avg  [*vc] += vol_c * U(*c);
    vol_v[*vc] += vol_c;
  }
for(VertexIterator v(grid); ! v.IsDone(); ++v)
  avg[*v] /= vol_v(*v);
}
```

4 Components of GrAL

GrAL is organized in largely independent modules, in order to allow the decoupled development of components from diverse fields Here, we concentrate on some rather basic kernel components from the `base` module, which have a very broad range of application and are used by more specialized parts of the library or directly in applications. An exception is the discussion of distributed grids in 4.6, which shows how a coordinated ensemble of generic data structures and algorithms can achieve encapsulation of many tasks common to parallel PDE software. A full, up-to-date catalog of components can be found on the GrAL web-site [1].

4.1 Mesh Data Structures

Several grid data structures are implemented in GrAL: Cell complexes in 2D and 3D with general cells (simple polygons and polytopes, resp.), Cartesian grids in 2D and 3D, and a simple triangulation data structure in 2D. As the focus of GrAL currently is more on algorithms than on data structures, these serve merely as examples and 'working horses' – typically, the generic components of GrAL will be instantiated for user-provided data structures.

For the mid-term future, a sort of *generative* mesh data structure (along the lines of [20]) is planned, where the user himself can configure a custom data structure containing exactly the functionality he needs. A similar idea is propagated in [21].

4.2 Generic Sequence and Incidence Iterators

As already mentioned, the generic approach implemented by GrAL has one of its greatest strengths when it comes to deal with existing, user-defined data structures. In order to ease the creation of a GrAL adaptation layer for the latter, a number of generic iterator classes are defined in GrAL.

For example, in the case of a simple mesh data structure storing only cell-vertex incidences, we can implement edges, facets (in 3D) and the corresponding iterators in a generic way. Assuming we have access to *cell archetypes* (see 3.3), we can implement the incidence iterator types `EdgeOnCellIterator` and `FacetOnCellIterator` by using the corresponding sequence iterators of the archetype. The objects representing edges and facets then simply contain an

`EdgeOnCellIterator` or `FacetOnCellIterator`, respectively. Comparison for equality of, say, two facets has to be made by comparing their vertex sets.

On top of the cell-based incidence iterators, also global sequence iterators `EdgeIterator` and `FacetIterator` are defined. Each facet (edge) is visited via all incident cells, and an internal hash table is used to decide whether the current element already has been visited. Again, the vertex set is used to evaluate a hash function on an element.

These generic classes reduce the effort for implementing or adapting a grid data structure to the GrAL interface. They represent a first step towards a more comprehensive and even less demanding (in terms of manual work required) generic adaptation framework aiming for minimizing adaptation work; a task which is currently of high priority, as such a framework will substantially lower the initial barrier of using GrAL.

It may be objected that this kind of iteration (maintaining a table of visited elements) is rather costly, compared to iteration over direct (permanent) representations of edges or facets. While a true observation, it does not render such iterators obsolete: In any case, the additional information of, say, an explicit edge table has to be built at some stage – and it is exactly here where our table-based iterators come into play! So, if we make heavy use of edge iteration in my application, we definitely should not use the generic version for it – however, we can use the latter to *initialize* our enhanced data structure, instead of building our own (and probably not more efficient) solution.

4.3 Incidence Calculation

Typically, grid data structures provide only a limited amount of incidence information; for example, a simple data structure with only cell-vertex incidence information (as mentioned before in the context of generic edge and facet iterators) does not provide cell neighbor adjacencies. While e. g. in a basic FEM implementation, there is often no use for this type of information, we need it in many other situations, for instance finite volume calculations (if the cells correspond to the control volumes). Judging by the frequency related questions occur in the corresponding Usenet forums, this task seems to be in the top ten of mesh management problems.

Cell neighbor information can be calculated from cell-vertex incidence relationships iff facet vertex sets are unique, as is typically the case in meshes for PDE solution. GrAL offers a generic algorithm based on a hash table for facets (resp. their vertex sets), which as a side effect provides access to boundary facets (namely those which do have only one incident cell). It can also used incrementally, for example if used in conjunction with a sequence of grid enlargements (see 3.6). Using a judicious formulation of the algorithm in terms of co-dimensional mesh entities, its implementation can be kept completely dimension-independent, which is a substantial advantage in terms of maintenance effort.

4.4 Grid Functions

As already mentioned, GrAL features generic implementations for both total and partial grid functions. Depending on the storage characteristics of elements, either array-based or hash-table-based implementations are selected; the latter are the default for partial grid functions. Generic grid functions are very easy and comfortable to use; essentially, all that has to be done for a new mesh data structure is to say e. g. "vertices are numbered consecutively starting from 0", by using a simple C++ `typedef`. So, the user can benefit with almost no effort from very powerful components, which in this generality are missing from most existing mesh data structures, especially for the partial grid function case.

4.5 Subranges and Closure Iterators

Often, algorithms have to be restricted to grid subsets, for instance near the boundary, or for parallel PDE solution. In many cases, these subranges are defined by enumerating their cells. In order to iterate over, say, the vertices of such a subrange, we need to perform the (topological) *closure* of its cells, a task solved by *closure iterators*. In the case of the generic edge and facet iterators mentioned before, the closure is simply taken over the whole grid. For small subranges, efficient implementation of closure iterators crucially relies on the availability of partial grid functions.

4.6 An Advanced Family of Components: Distributed Grids

Parallel PDE solution typically follows the domain partitioning paradigm: The computational domain, i. e. the mesh, is cut into parts of approximately equal size, and the algorithms operate locally on the parts. It is up to the programmer to ensure that each algorithm has access to all needed pieces of data which might reside on a different part. Typically, this is done be providing overlapping meshes, where data in the overlap is managed by other parts and must be updated regularly.

While this is a simple principle, its realization poses a considerable burden on the programmer. On the other hand, most of the logical functionality to be implemented for distributed grids is repeated over and over in different applications. The approach taken by GrAL is to encapsulate the technical details related to parallel PDE solution with distributed grids, while still allowing the programmer to work on the original underlying mesh in more or less the same way as in the sequential case.

In order to achieve this, GrAL offers generic data structures for distributed grids (including overlap ranges and distributed grid functions) which can be "wrapped around" existing mesh data structures. As the required size of the overlap typically varies between algorithms (and applications), GrAL offers a means to describe the overlap by using generalized stencils of algorithms which are meaningful also for unstructured grids. Based on this description, the overlap can be determined automatically by a GrAL algorithm. For more details, see [14], and [22] for a depiction of the stencil calculus.

5 Using Generic Libraries

Generic and traditional libraries have some marked differences, for example concerning generality, granularity, efficiency, ease-of-use, and tool support. We will discuss some of these issues.

5.1 Reuse Effort

How much work is it to reuse generic components? We assume here that a programmer provides his own data structure and wants to use selected generic algorithms. Then the answer is largely independent of the number of generic components: The abstract interface layer has to be implemented once for the given data structure (more precisely, that interface parts it supports). Then, all generic components can be used, provided that the data structure satisfies the components' requirements.

Thus, the reuse effort is constant. By providing in GrAL a wrapper around data structures defined by APIs, we can offer the same easy access to third-party traditional components. This has e. g. be done for the Metis mesh partitioning library [23]. In the traditional approach, the user would have to create ad hoc driver code for each such library in order to copy his data to and from the library's format.

However, it has to be acknowledged that creating a GrAL interface layer still involves quite a bit of work. In particular, it requires some prior familiarity with GrAL's concepts, which is an initial hurdle. So, more effort is needed along the lines of the generic iterators discussed in section 4.2 to provide adapters for the most common patterns of mesh data structures. For example, most data structures used for PDE solution are essentially based on cell-vertex incidence relationship. Here, a ready-made adapter could be provided, requiring the user to implement only a handful of routines.

5.2 Generality & Granularity

The desire for a higher level of generality has been the driving force behind the development of GrAL. In the case of grids, a similar degree of generality is simply not achievable for traditional libraries.

The fact that no copying is needed makes much smaller-grained components practical, for example, we can provide iterators traversing boundary components, or local search algorithms. Thus, the number of components exposed to the user increases, as does the flexibility of the components themselves. This creates serious challenges with respect to their documentation.

5.3 Efficiency

The overhead of generic components with respect to code specialized to concrete data structures is called *abstraction penalty*. Measuring this penalty is practical

only for small pieces of code – in more complex cases it is often just too tedious to create an equivalent ad-hoc low-level component. Such measurements are also highly dependent on the compilers and optimization options used. In some cases, similar to the example in section 3.7, the penalty can be removed completely, in others, an overhead of 50% or more can be measured, see [14]. It has to be noted that the loops presented there do essentially measure the pure overhead and thus give a sort of worst-case result.

Coming back to the design variants discussed in section 2, we have to keep in mind that this comparison is with respect to ad-hoc implementations for a given data structure. Comparing with API or file coupling approaches, the performance of generic components is in general much better, and memory bottlenecks can be avoided.

All this means that our generic grid components are usable very well for high-performance applications. First, their performance is in general quite good, even compared with direct implementations. Second, if there really is a hot spot, we still have the possibility of transparently specializing the generic version *after profiling*. In general, there tend to be only few such hot spots.

However, we have to use a good optimizing compiler; non-optimized generic code will in general run an order of magnitude slower. Fortunately, recent versions of the free `gcc` compiler seem to do the job quite well.

5.4 Ease-of-use, Documentation and Testing

A very important advantage of the generic approach is that it makes it practical to provide sufficiently small, focused and self-contained components, which can be tested separately and automatically. Such unit tests are currently being implemented successively for all GrAL components, and run on a nightly basis.

By using different types for instantiating and running generic components, it can also be checked much better than in the non-generic case that they do not depend on some arbitrary property of their arguments, which could break later. Whereas in non-generic code assumptions on the functionality of arguments are *implicit* – they just rely on what is provided – generic components can (or should) make only *explicit* assumptions on their argument types.

These assumptions (or requirements) have to be documented thoroughly; often encountered sets of requirements can be captured by definition of *concepts* (see section 3.1), a technique promoted by the SGI STL documentation [24], and also used for GrAL. Such concepts ultimately lead to a domain-specific language, which is an invaluable aid for reasoning and communicating. So, besides the runtime pre- and postconditions for the runtime arguments, documentation of generic components involves in addition the description of the compile-time type arguments.

How to enforce the requirements (constraints) imposed by a generic component is a controversial issue. C++ offers no built-in ways of doing so, unlike Eiffel [25]. However, there are means of checking constraints near the library entry points by so-called *concept checks* [26]. Such checks, accompanied with a meaningful message in case of failure, are of great help to a user and can

compensate somewhat for the additional source of errors introduced by the additional degrees of freedom. An important task for future work will be to provide a layered user interface for heavily parameterized generic components, offering a path from minimal parameterization to the most general versions.

All in all, tool support for generic programming is still inadequate, resulting in nuisances such as long compile times and incomprehensible error messages.

6 Discussion

The generic approach presented here provides for the first time universally usable mesh-based components for direct incorporation into software for PDE solution, *independently* of the underlying data structures. The effort for reusing GrAL components is restricted to creating a thin interface adaptation layer once, and thus independent of the number of components used. By writing such wrappers for traditional mesh-based libraries in GrAL, these can be used with no extra effort.

It has to be acknowledged that using generic libraries still faces difficulties, some of a more technical nature, related to insufficient tool support, others due to the raised level of abstraction. The latter point poses an initial hurdle; however, in the long run it substantially contributes to better understanding.

Efficiency of generic components is a design criterion and turns out to be quite satisfactory, although the overhead cannot be eliminated in all cases. In view of the inefficiencies of the traditional approaches (due to the necessary copying), and the fine-grained opportunities for specialization and tuning, the approach of GrAL should be seen as a step ahead also in terms of efficiency.

Future work will on the one hand concentrate on easing the use of GrAL by enhancing support for adapting user mesh data structures. Plans for further extension of GrAL include components for mesh optimization and checking, partitioning and visualization. Coupling GrAL to other generic libraries of interest for PDE solution, such as MTL [27] or BGL [9] will help to assess the viability of the generic approach: It should be possible to run graph algorithms on top of meshes in a seamless way, making it indistinguishable from using 'native' GrAL algorithms.

References

1. Berti, G.: GrAL – the Grid Algorithms Library. `http://www.math.tu-cottbus.de/~berti/gral` (2001)
2. Musser, D.R., Stepanov, A.A.: Generic programming. In Gianni, P., ed.: Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: proceedings. Number 358 in LNCS, Springer (1989) 13–25
3. Pflaum, C.: Semi-unstructured grids. Computing **67** (2001) 141–166
4. Shaw, J.A., Peace, A.J.: Simulating three-dimensional aeronautical flows on mixed block-structured/semi-structured/unstructured meshes. Int'l Journal for Numerical Methods on Fluids **39** (2002) 213–246

5. Haimes, B.: Visual3 homepage. http://raphael.mit.edu/visual3/visual3.html (1999)
6. Brown, D.L., Quinlan, D.J., Henshaw, W.: Overture - object-oriented tools for solving CFD and combustion problems in complex moving geometries. http://www.llnl.gov/CASC/Overture/ (1999)
7. Karmesin, S., et al.: POOMA: Parallel Object-Oriented Methods and Applications. http://www.acl.lanl.gov/PoomaFramework/ (1999)
8. Lee, M., Stepanov, A.A.: The standard template library. Technical report, Hewlett-Packard Laboratories (1995)
9. Siek, J., Lee, L.Q., Lumsdaine, A.: BGL – the Boost Graph Library. http://www.boost.org/libs/graph/doc/table_of_contents.html (2000)
10. The CGAL Consortium: The CGAL home page – Computational Geometry Algorithms Library. http://www.cgal.org (1999)
11. Köthe, U.: VIGRA homepage. http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/ (2000)
12. Dawes, B., et al.: The boost homepage. http://www.boost.org (2002)
13. Musser, D.: Generic programming. http://www.cs.rpi.edu/~musser/gp/ (2002)
14. Berti, G.: Generic software components for Scientific Computing. PhD thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany (2000)
15. Berti, G.: A generic toolbox for the grid craftsman. In Hackbusch, W., Langer, U., eds.: Proceedings of the 17th GAMM Seminar on Construction of Grid Generation Algorithms, Online proceedings at http://www.mis.mpg.de/conferences/gamm/2001/ (2001)
16. Ziegler, G.M.: Lectures on Polytopes. Volume 152 of Graduate Texts in Mathematics. Springer-Verlag, Heidelberg (1994)
17. Berti, G.: Generic components for grid data structures and algorithms with C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
18. Mäntylä, M.J.: Computational topology: a study of topological manipulations and interrogations in computer graphics and geometric modeling. Acta Polytech. Scand. Math. Comput. Sci. Ser. **37** (1983) 1–46
19. Ortega, F.: General mesh viewer (GMV) homepage. http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html (1996)
20. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Techniques, and Applications. Addison-Wesley (2000)
21. Remacle, J.F., Karamete, B.K., Shephard, M.S.: Algorithm oriented mesh database. In: Proceedings of 9th Int'l Meshing Roundtable. (2000) 349–359
22. Berti, G.: A calculus for stencils on arbitrary grids with applications to parallel pde solution. In Sonar, T., Thomas, I., eds.: Proceedings of the GAMM Workshop on Discrete Modelling and discrete Algorithms in Continuum Mechanics, Logos Verlag Berlin (2001) 37–46
23. Karypis, G.: METIS home page. http://www-users.cs.umn.edu/~karypis/metis/ (1999) (last visit: 11/1999).
24. Silicon Graphics Inc.: SGI Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl (since 1996)
25. Meyer, B.: Eiffel: The Language. Object-Oriented Series. Prentice Hall, New York, NY (1992)
26. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)

27. Lumsdaine, A., Siek, J.: The Matrix Template Library (MTL). `http://www.lsc.nd.edu/research/mtl/` (1999)