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 can be executed 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, there are container data structures and corresponding algorithms, such as sorting and searching. Here, we are concerned with the next higher level, namely representations for geometric structures (meshes), and associated algorithms.

On top of these geometric components operate numerical discretization such as FEM or FV algorithms. Often, we also need algebraic components, namely, matrix and vector representations implementing the vector-space operations, and algorithms for solving linear or non-linear systems, which again may access the grid data structures, e. g. preconditioners or multi-grid algorithms. 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 library dedicated to meshes available from which we can, say, take a mesh contact detection algorithm for inclusion into a PDE solver.

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 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 in C++, GrAL achieves a complete decoupling of algorithms and data structures.

After briefly analyzing the problems with traditional library design, 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 consider grids used for numerical simulation, we see a wide variety of different types: Cartesian and curvilinear mapped structured grids, multi-block and semi-structured [15], unstructured pure-simplex or general cell grids, to hybrid 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 API.
2. Use a 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, the grid has to be copied into the data structures predefined by the library routine, and possibly vice versa, which might be not 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 – worsened – 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, 8]. The most 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 to often is the only choice left.

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 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: Whereas in the case of an API, we have to learn the interface of the library component; in our approach, we present the internals of our data representation in a structured way to the library component.

The solution is inspired by the C++ Standard Template Library STL [10]. Similar approaches are BGL [16] for graphs, CGAL [7] for general computational geometry, and VIGRAL [9] for image processing.

3 Design and Core Components of GrAL

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. Shall maintain high performance with respect to 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).

Now, having a unified interface does neither mean that we are restricted to some least common denominator, nor does it place 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.

Similarly, there is no need for a concrete grid component 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 can be added on-the-fly.

3.1 The Kernel Interface – A Very Quick Tour

The following gives a bird's-eye view only. Details can be found [2, 3], and the formal interface specifications are continually updated in [1].

The definition of a grid corresponding to the interface given below is slightly more general than that originally given in [2]. Restrictions for meeting requirements of algorithms are made on a case-by-case basis. We distinguish between a combinatorial grid (abstract complex) and its geometric embedding, a distinction which 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* $\dim : \mathcal{C} \mapsto \{0, \dots, d\} \subset \mathbb{N}$, *(*$\dim(e)$ *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* morphism *between abstract complexes $\mathcal{C}_1, \mathcal{C}_2$ is a mapping $\Phi : \mathcal{C}_1 \mapsto \mathcal{C}_2$ with $e < f \Rightarrow \Phi(e) < \Phi(f)$.*

An abstract complex is a purely combinatorial entity, also known as *poset*. 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*

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

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

Combinatorial Grid Interface The combinatorial layer is concerned only with abstract complexes. 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. Handles are useful e. g. for subranges.

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. 1. The number of different incidence iterators is $d(d-1)$, where d is the grid dimension.

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

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)

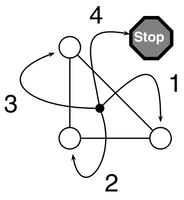

Fig. 1. Action of a VertexOnCellIterator (*Incidence iterator*)

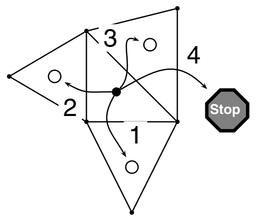

Fig. 2. Action of a CellOnCellIterator (*Adjacency iterator*)

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.

More Combinatorial Functionality: Switch Operator and Archetype Incidence iterators suffice for implementing a surprisingly large class of algorithms. However, there is no ordering relationship between different incidence iterators, for example in 2D, vertices and edges incident to a cell can be ordered independently.

If we need such relationships, we can use the *switch* operator, which allows e. g. to traverse a connected component of a grid's boundary (see [2] for details). Also, the boundary of a cell (its *archetype*) can be accessed as a grid of dimension $d-1$. The interface for switch and archetypes is still experimental.

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. 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.

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.

Grid Functions Grid functions allow to store and access data on grid elements of any dimension. 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 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 approaches to 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.

Grid functions can be *total* (dense) or *partial* (sparse), both of which have generic default implementations, see below and [4].

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 grid refinement, coarsening or optimization, it 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 [12]). 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 [2] and [3].

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 [14].

A generic copy operation also 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.

4 Components of GrAL

A full, up-to-date catalogue of components can be found on the GrAL web-site [1]. Here, we present a small selection of basic components.

Mesh Data Structures Several grid data structures are implemented in GrAL: Cell complexes with general cells in 2D and 3D, 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.

Generic Iterators In order to ease the creation of a GrAL adaptation layer for user-defined data structures, a number of generic iterator classes are defined in GrAL, for example edges and facets for 3D grids. A consistent framework aiming for minimizing adaptation work is of high priority.

Closure Iterators When using grid subranges, for instance subsets of grid cells, we often need to access all elements of a given dimension incident to them, without duplication. This is what the generic closure iterators do.

Incidence Calculation Typically, grid data structures provide only a limited amount of incident information; e. g. in a basic FEM implementation, there is often no need for cell neighbor information. In case this information is needed at a later point, it can be calculated by a generic GrAL algorithm.

Grid Functions As already mentioned, GrAL contains 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.

Distributed Grids For parallel PDE solution, grids and grid functions have to be distributed and augmented by additional information. The necessary data structures and algorithms for creating distributed grids and grid functions (using arbitrary overlaps) are generic GrAL components and can be wrapped around any sequential grid representation, see [2] or [5] for details.

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 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.

5.2 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 to tedious to create an equivalent ad-hoc low-level component. Such measurements are also highly dependent on the compilers used. In some cases, the penalty can be removed completely, in others, an overhead of 50% or more can be measured, see [2]. It has to be kept in mind that the loops presented there do essentially measure the pure overhead and thus give a sort of worst-case result.

Coming back to 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 have always 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 an optimizing compiler; non-optimized generic code will in general run an order of magnitude slower.

5.3 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*, a technique put forward by the SGI STL documentation [18], 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 [13]. However, there are means of checking constraints near the library entry points by so-called *concept checks* [17]. 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.

A serious use of GrAL typically requires the user to create an adaptation layer for his data structure. Although not difficult, this requires a prior understanding of the underlying abstractions, and blocks quick success. Thus, enhancing the support for adaptation layers is a key task.

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 creating 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. Also, coupling GrAL to other generic libraries of interest for PDE solution, such as MTL [11] or BGL [16] will help to assess the viability of the generic approach.

References

1. Berti, G.: GrAL – the Grid Algorithms Library. `http://www.math.tu-cottbus.de/~berti/gral` (2001)
2. Berti, G.: Generic software components for Scientific Computing. PhD thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany (2000)
3. 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)
4. Berti, G.: Generic components for grid data structures and algorithms with C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
5. 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
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. The CGAL Consortium: The CGAL home page – Computational Geometry Algorithms Library. `http://www.cgal.org` (1999)
8. Karmesin, S., et al.: POOMA: Parallel Object-Oriented Methods and Applications. `http://www.acl.lanl.gov/PoomaFramework/` (1999)
9. Köthe, U.: VIGRA homepage. `http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra/` (2000)
10. Lee, M., Stepanov, A.A.: The standard template library. Technical report, Hewlett-Packard Laboratories (1995)
11. Lumsdaine, A., Siek, J.: The Matrix Template Library (MTL). `http://www.lsc.nd.edu/research/mtl/` (1999)
12. 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
13. Meyer, B.: Eiffel: The Language. Object-Oriented Series. Prentice Hall, New York, NY (1992)
14. Ortega, F.: General mesh viewer (GMV) homepage. `http://www-xdiv.lanl.gov/XCM/gmv/GMVHome.html` (1996)
15. Pflaum, C.: Semi-unstructured grids. Computing **67** (2001) 141–166
16. Siek, J., Lee, L.Q., Lumsdaine, A.: BGL – the Boost Graph Library. `http://www.boost.org/libs/graph/doc/table_of_contents.html` (2000)
17. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Programming, Erfurt, Germany. (2000)
18. Silicon Graphics Inc.: SGI Standard Template Library Programmer's Guide. `http://www.sgi.com/tech/stl` (since 1996)