Generic Components for Grid Data Structures and Algorithms with C++

Guntram Berti
Institut für Mathematik, BTU Cottbus, Germany

Abstract

Grids are fundamental data structures for representing geometric structures or their subdivisions. We propose a strategy for decoupling algorithms working on grids from the details of grid representations, using a generic programming approach in C++. Functionality of grid data structures is captured by a small set of primitives, divided into combinatorial and geometric ones. Special attention is paid to the generic implementation of grid functions, which correspond to the notion of mappings from grid elements (e. g. vertices) to entities of a given type. Experiments indicate that the overhead of the generic formulation is low and can be completely eliminated in some cases.

1 Introduction

Representation of spatial or planar geometric structures is central to many application domains, such as computational geometry, geometric modeling, geographical information systems (GIS), and computational simulation by numerical solution of partial differential equations (PDEs). Spatial structures are typically represented by a subdivision into simpler entities like triangles or cubes, as in fig. 1. This subdivision is called grid, mesh, polyhedron, triangulation or cellular complex, depending on the application area.

Complexity of software in the domains mentioned before is often determined by the interaction of (grid) data structures and algorithms operating on them. Examples for algorithms include cell neighbor search, iso-surface extraction, rendering of geometric structures, intersections of geometric structures, grid generation and refinement, numerical discretizations like finite elements (FEM) or finite volumes (FV), or point localization in a grid.

Due to the similarity of the underlying mathematical concepts, algorithms are *in principle* independent of a particular data structure. Yet, in practice their implementations rely heavily on the details of the latter. Consequently, such an implementation can be used only with the concrete data structure it was designed for. Given the multitude of grid representations in use, this is a real problem.

It would therefore be a considerable gain in reusability if grid algorithms could be implemented in a way that is *independent* of the data representation, without compromising efficiency substantially. For the comparatively simple case of linear sequences, this has been achieved by the C++ Standard Template Library (STL). We propose an approach similar in spirit for the more involved case of grids.

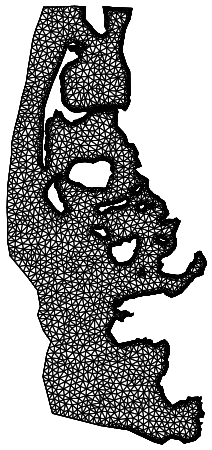

Figure 1: A grid representing the north friesian coast

The paper is organized as follows: First (section 2), we give a very short overview over the mathematical aspects of grids and the requirements of grid algorithms. In section 3, we introduce set of core functionality (a *micro-kernel*) for grid data-structures. Section 4 deals with a few selected generic components, followed by a detailed case study for grid functions in section 5. Efficiency is considered in section 6. In section 7, we discuss some difficulties that arise in generic programming. Finally, we compare our work to related efforts, and discuss some of its implications.

This work is part of the author's doctoral thesis [4], where the approach is evaluated in the context of the sequential and distributed numerical solution of PDEs.

2 Analysis of grids and algorithms

The common mathematical basis underlying all incarnations of grids can be captured by the notions of combinatorial topology (see e. g. [13]) and polytope theory [20]. Without going into the details (which can be found in [4]), we intro-

(a) A simple grid $\mathcal{G}$ ...

(b) ... and its incidence lattice

Figure 2: The combinatorial structure of a grid is given by its lattice

duce some basic terminology, necessary to understand the following.

A simple 2-dimensional grid is shown in figure 2(a). It consists of just two cells, five edges and four vertices. The core of any representation in a computer is formed by the grid's *combinatorial structure* (fig. 2(b)), namely the *incidence* relationships between its *elements* (that is, vertices, edges and cells). Two elements are incident of one is contained in the boundary of the other; for example, v_4 is incident to e_4, e_5 and c_2, and vice versa. This relationship is reflected in the *lattice* 2(b), which is, roughly spoken, the graph of incidences.

These notions generalize easily to higher dimensions, the three-dimensional case being by far the most important one. A grid $\mathcal{G}$ of dimension d consists of elements of dimension 0 (vertices – $\mathcal{G}^0$), dimension 1 (edges – $\mathcal{G}^1$), up to dimension d (cells – $\mathcal{G}^d$). The boundary of an element must be composed of lower-dimensional elements (e. g. the boundary of c_1 consists of $v_1, v_2, v_3, e_1, e_2, e_3$); and the intersection of any two elements must be another element or empty. The mathematical term for such a structure is *CW-complex*.

We call elements of dimension $d-1$ (codimension 1) *facets*, which for $d = 2$ coincides with edges. The reason for this additional name is, that many algorithms can be formulated in a dimension-independent way by using the codimension-related terms *cell* and *facet*.

Grids may differ in the pattern of incidence relationship they allow, for instance, a Cartesian grid has a very rigid checkerboard pattern, whereas unstructured grids allow for general cell types and connectivity (fig. 1).

In addition to these combinatorial properties, a grid bears also *geometric* information: it is embedded into some geometric space. The most common case is a planar straight line embedding; other possibilities are curved elements or embeddings into higher dimensional space or into a manifold, for example the surface of a sphere. Straight line embeddings (cf. figs 1 and 2(a)) can be represented by simply assigning space coordinates to vertices. General embeddings, however, are representable only in an approximate way, which is in sharp contrast to the combinatorial information.

Thus, it proves advantageous to preserve the separation of combinatorial and geometric aspects also in the software design, leading to a variety of combinations of the corresponding representations.

Finally, a mathematical concept often overlooked is that of a function on the (discrete) set of grid elements, yielding values of some arbitrary type T. Such *grid functions* are extremely important to almost every algorithm on grids. Yet they are often provided only halfheartedly or in an ad-hoc fashion, that depends on the needs of the concrete algorithms to be supported.

After clarification of the mathematical concepts, a second step consists in the analysis of representative grid algorithms. It turns out that many of them pose rather modest requirements on functionality of grid data structures, which can be captured by a small set of concepts.

A typical algorithm is the determination of the total flux into a cell across its boundary. Performed for all cells of a grid $\mathcal{G}$, this is the core of the finite volume (FV) method:

IN: a cell-based state $U : \mathcal{G}^d \mapsto \mathbb{R}^p$ (approximate PDE solution)
OUT: a cell-based flux-sum $\mathsf{flux} : \mathcal{G}^d \mapsto \mathbb{R}^p$

```
1: for all cells c ∈ G do
2:     flux(c) = 0
3:     for all facets f of c do
4:         flux(c) += numerical_flux(c, f, U)
```

What kind of functionality does this algorithm require? We must iterate over all cells of a grid, as well as over all facets of a cell, and associate states U and flux to cells. That is, U and flux are grid functions on cells. Moreover, in numerical_flux(), geometric data like cell centers, facet volumes and normals are required (not shown). We note that the algorithm above is formulated in a *dimension independent* way (except, of course, the interior of the routine numerical_flux()).

In general, the requirements of most algorithms on grid data structures can be grouped as follows:

- combinatorial functionality:
  - *sequence iteration* over all elements of a kind, e. g. over all vertices
  - *incidence iteration* over all elements of a kind incident to a given element
  - addition or deletion of grid parts
- grid function functionality: accessing data associated to grid elements
- geometric functionality: mapping of combinatoric elements to geometric entities, e. g. vertex coordinates, edge segments, and so on; calculating measures like volumes, centers, or normals.

In addition, a few algorithms have special requirements on grids, for example does the calculation of cell neighbors require that facets be uniquely determined by their vertex sets. Also, in cases like finite element methods, more information about the combinatorial structure of grid cells is required, see [4] for details.

3 A Micro-kernel for Grid Data Structures

As a consequence of the outcome of algorithm requirements analysis, we can identify a set of functional primitives, forming a *micro-kernel* for grid data structures. This micro-kernel serves two fundamental purposes: First, it separates basic (atomic) functionality from derived (composite) functionality, thus answering the question "What belongs into a grid data structure?" And second, in the context of generic programming, it serves as a *broker* layer between concrete data representations and generic components like algorithms and other data structures.

The identification of a small yet sufficient set of basic primitives is an iterative process, depending on an analysis of both typical algorithm requirements and data structure capabilities, as mentioned in the preceding section.

With respect to the universe of possible operations on grids the micro-kernel is (almost) *minimal*: No part of it can be expressed appropriately by other parts in the *general* case. With respect to the basic functionality offered by concrete grid data structures it is *maximal*: A given representation component in general implements only a *part* of it. For example, a simple triangulation data structure which only stores the vertex indices for each cell cannot provide iteration over neighbor cells. On the other hand, we are able to provide a generic implementation of neighbor iteration, using the part of the micro-kernel supported by the triangulation component. However, this will not be as efficient as a specialized implementation by extending the triangulation component.

Following the terminology of the SGI STL [18], we use the term *concept* for a set of requirements. A concrete entity (e. g. a class) satisfying a concept's requirements is called *model* of the concept. The concepts for the combinatorial requirements are listed in table 1. *Element* concepts, like Vertex, correspond directly to their mathematical counterparts mentioned above. Among others, they give access to incident elements and are used to access data stored in grid functions (see below). *Handle* concepts provide for a sort of minimal representation of grid elements. They are unique only in conjunction with a fixed grid. Their main use is the economic implementation of grid functions, grid subranges and the like.

All *iterator* concepts are refinements of the STL Forward Iterator. *Sequence iterators* allow for global loops over all elements of a grid, *incidence iterators* provide access to elements incident to a given element.

Grid functions allow access and storage of arbitrary data on grid elements. They correspond

Elements	handles	Sequence Iterators	Incidence Iterators
Vertex	Vertex Handle	Vertex Iterator	VertexOnVertex Iterator EdgeOnVertex Iterator CellOnVertex Iterator
Edge	Edge Handle	Edge Iterator	CellOnEdge Iterator ⋮
Facet	Facet Handle	Facet Iterator	VertexOnFacet Iterator ⋮
Cell	Cell Handle	Cell Iterator	VertexOnCell Iterator ⋮

Table 1: Concepts for combinatorial functionality

to the mathematical notion of functions from the discrete set of grid elements (of a fixed dimension) to value of some type T. Concepts for grid functions are shown in table 2. All grid function concepts are refinements of STL Adaptable Unary Function, with *element* and *value* type corresponding to *argument* and *result* type, resp.

We chose to make read and write operations syntactically different (operators `()` and `[]`, resp.) in order to give better control over them. For example, a partial grid function normally needs to allocate new storage when write access to a previously untouched element occurs; an effect that is clearly undesired when one just wants to read, possibly getting the default value.[1] A typical use of partial grid functions is the marking of elements during some algorithm (e. g. depth-first traversal), with marking initialized to `false` in $O(1)$ time. This allows for efficient implementations of local algorithms in sublinear time (with respect to the size of the whole grid).

If a grid function is simply passed to an algorithm, clearly the interfaces of Grid Function (or Mutable Grid Function, if it is an output variable) are sufficient. Many algorithms, however, use *temporary* grid functions internally. Therefore, it is crucial that (a) there is a uniform way to create (and destroy) grid functions, besides accessing and altering their values, and (b), that the totality of needed grid functions does *not* influence the definition of the underlying grid data structures. The latter would introduce an undue coupling from algorithms to grid data structures.

Therefore, we chose to provide the following class templates in the micro-kernel, where `E` is the element type (that is, argument type), and `T` is the value type:

```
template<class E, class T>
class grid_function;
template<class E, class T>
class partial_grid_function;
```

The class template `grid_function` is a model of Total Grid Function, and `partial_grid_function` is a model of Partial Grid Function. Whereas the value type `T` can be dealt with in a fully generic way, the dependency on the `E` parameter is more interesting, and is discussed below.

The creation of grid functions is straightforward:

```
MyGrid  G;   // create a grid
...
// associate ints ('colors') to vertices
grid_function        <MyVertex,int>
                     color (G);
// mark  edges, default: false
partial_grid_function<MyEdge  ,bool>
                     marked(G, false);
// put 2-vectors on cells, init. with (0,0).
grid_function        <MyCell  ,vec2>
                     state (G,vec2(0,0));
```

Here `MyVertex` and `MyEdge` are typedefs to models of Vertex and Edge, corresponding to the type `MyGrid`.

Grid algorithms can be formulated quite naturally using this micro-kernel. For example,

1. The same problem occurs with the `operator[]` in the STL `map` and `hash_map`, where the lack of a default value means one has to use a more complicated sequence of operations if no insertion is desired.

Concept	Feature	Member
Grid Function (G. F.)	element (arg) type	`typedef element_type (E)`
	value type	`typedef value_type (T)`
	grid type	`typedef grid_type`
	read access	`T const& operator()(E const&)` (mapping `E` $\mapsto$ `T`)
Mutable G. F.	+ write access	`T & operator[](E const&)` (see below for `()` vs. `[]`)
Container G. F.	+ creation	`grid_function()` `grid_function(grid_type const&)`
Total G. F.	+ storage on *all* elements	`grid_function(grid_type const&, T const&)`
Partial G. F.	+ storage on *some* elem.	`grid_function(grid_type const&, T const&)`
	+ default value	`set_default(T const&)`

Table 2: Concepts for grid function functionality. Refinement relationship is shown by indentation: Total and Partial G. F. are both refinements of Container G. F.

counting for each vertex the number of incident cells translates into the following code:[2]

```
// init. ncells[v] to 0
grid_function<Vertex,int> ncells(G,0);
// for all cells c of G
for(CellIterator c(G); c; ++c)
  // for all vertices of c
  for(VertexOnCellIterator vc(*c);  vc; ++vc)
     ncells[*vc]++;

for(VertexIterator v(G); v; ++v)
   cout << "ncell: " << ncells(*v) << '\n';
```

This code also shows the use of a total grid function. With a good optimizing compiler, the efficiency achieved for this generic piece of code is quite close to that of a low-level version, see below.

Modifying operations on data structures are harder to cope with. We chose to base mutating algorithms on coarse-grained mutating operations, namely grid copy, grid enlargement and grid cutting (removal of parts). These operations have proven useful in conjunction with generic implementations of distributed grids and adaptive refinement strategies, yet more work is needed to fully master mutating operations. In particular, the question of how to handle dependent data, such as grid functions, when adding or deleting parts of a grid is not entirely settled. A majority of important algorithms, however, are non-mutating (ignoring the unproblematic write-access to grid functions).

4 Generic Components

One of the purposes of having a micro-kernel is separating the truly representation-dependent issues from those which can be dealt with in a generic manner. Of course, the distinction is not sharp; we will see that even parts of the micro-kernel can be implemented generically based on other parts of the latter. However, there often are better implementations possible, which are specialized to the concrete representation and thus justify their inclusion into the micro-kernel.

This discussion suggests a classification of generic components according to their generality, or proximity to the basic kernel. On the one end ot the scale, we have components that are properly regarded as *extending* the functionality of grid data structures, for instance iterators, grid functions or grid subranges. On the other end, there are lots of (mostly algorithmic) components fulfilling very domain-specific tasks, such as numerical discretizations, grid smoothing, and so on. Somewhere in between lie data structures like hierarchical or distributed grids.

In the following, we review some generic components having been developed so far, proceeding from more general to more specific. The case of grid functions is discussed in more detail in

2. The deviation from the STL-style iteration is essentially just a matter of convenience. As our iterators must be classes by other reasons, the exclusion of pointers poses no problem here.

the next section.

Iterators Sequence as well as incidence iterators can be implemented generically in some cases. For example, sequence iterators for (non-stored) facets can be implemented using cell iterators and facet-on-cell iterators, if there is a total ordering on cells. Incidence iterators on vertices can (in 2D) be based on a so-called *switch* operation [5]. The same technique works for boundary iterators, where one can use arbitrary cell predicates to define inner and outer grid parts.

Grid subranges A grid subrange is defined by a collection of cells, and can be implemented based on cell handles. The elements of lower dimension contained in the closure of the cell set are then given by *closure iterators*, which use partial grid functions to mark visited elements.

Grid functions Grid functions have two basic parameters of variation: Element type and value type. The value type parameter poses no special problems (cf. STL). Depending on the properties of the element parameter, we can choose a generic implementation using vectors or hash tables, see the next section.

Distributed grids Applications like solution of partial differential equations needing a lot of computational power are candidates for parallel execution. The resulting management of overlapping grid parts is provided by distributed grids. Data structures for representing the overlap ranges, methods for updating distributed grid functions and algorithms for automatic generation of overlapping parts have been based generically on the kernel.

Hierarchical grids Some of the more advanced computational methods (such as the multigrid method below) are based on hierarchies of successively refined grids, with *vertical* (coarse $\leftrightarrow$ fine) relationships between them.

Multigrid algorithms Multigrid methods [10] are optimal algorithms for solving sparse linear systems 'living' on a hierarchy of grids. Grid-related operations, such as the mapping of state vectors and matrices between different grid levels (restriction and prolongation), are implemented generically.

Finite volume discretizations In addition to the basic finite volume algorithm presented before more complex higher-order methods have been implemented, which involve averaging values of cells incident to vertices.

Two prototype generic solvers for PDE problems have been based on the components just mentioned: A finite element solver for the Poisson equation using adaptive grid hierarchies and multigrid algorithms, and a finite volume solver for the incompressible Euler equations. The latter has been parallelized using components for distributed grids. We will come back to these solvers at the end of the paper.

5 A Case Study: Grid Functions

The implementation of a grid function depends crucially of the representation of the corresponding element type. If the elements of that type are numbered consecutively, for example if vertices are stored in an array (random-access sequence), it is possible to use STL vectors for the corresponding grid function. On the other hand, if no such enumeration is available (for example, if the corresponding elements are not permanently stored at all), we can resort to hash tables or balanced trees, depending on whether there can be defined a hash function or a total order on the element type.

We use *element traits* to provide the necessary information in a uniform way:

```
template<class E>
struct elem_traits {
   typedef grid_t;
   typedef handle_t;  // e.g. int
   typedef elem_t;    // == E
   typedef elem_tag;  // kind of elem.,
                      // e.g. vertex_tag
   typedef elem_iter; // sequence it. of grid_t

   typedef hasher_t;    // e.g. hash<handle_t>

   static size_type size     (grid_t const&);
   static elem_iter FirstElem(grid_t const&);
   static handle_t  handle   (elem_t const&);
};
```

For example, the traits contain a unified method of accessing the size of the underlying grid (`size(grid_t const&)`), when viewed as a container of the corresponding element type. This allows a vector-based implementation to

initialize the vector size, without knowing the kind (vertex, edge or whatever) of the element:

```
template<class E, class T>
class grid_function_vector {
  typedef elem_traits<E>  et;
  vector<T> table;  // data
public:
 grid_function_vector(grid_t const& gg)
    : g(&gg),  table(et::size(gg)) {}

  T& operator[](E const& e)
   { return table[et::handle(e)];}

  /* ... */
};
```

We provide generic implementations for both the vector and hash table case. For a concrete element type `E`, one of these implementations can be chosen by partially specializing the grid function template for `E`, leaving the value type parameterized:

```
class MyVertex { ... };
class MyEdge   { ... };

template<class T>
grid_function<MyVertex,T>
  : public grid_function_vector<MyVertex,T>
{ /*  repeat constructors */ };

template<class T>
grid_function<MyEdge,T>
  : public grid_function_hash<MyEdge,T>
{ /*  repeat constructors */ };
```

Here, it is assumed that `MyEdge` is not apt for storing associated data in a vector, for instance, it might not be stored permanently.

For partial grid functions, we provide a default generic implementation, using `grid_function_hash<>`.

An interesting problem arises here, because we want a grid function to give iteration access to the set of elements on which it is explicitly defined. That is, a grid function on vertices should provide a type `VertexIterator` and a corresponding member function `FirstVertex()`. How can this be achieved, while minimizing code duplication?

We can implement the *functionality* in a generic way (e. g. using the traits to access the iteration capability of the underlying grid for total grid functions), thus obtaining `ElementIterator` and `FirstElem()`. It remains to map these to the desired names: There must be a type `VertexIterator` as a typedef for `ElementIterator` if the element type is a vertex type, and so on. This mapping is performed by the following template:

```
template<
 class ElemIt,   // to be renamed to
                 // VertexIterator etc.
 class ElemRge,  // range (grid fct) def. ElemIt,
                 // derives from map_elem_name<>
                 // (Nackman-Barton trick)
 class elem_tag> // kind of element,
                 // to be specialized
struct map_elem_name {};

// specialization for vertex
template<class ElemIt, class ElemRge>
struct map_elem_name<ElemIt, ElemRge,
                     vertex_tag>
{
  typedef ElemIt  VertexIterator;
  VertexIterator  FirstVertex() const {
    return VertexIterator(
      static_cast<ElemRge const*>(this)->
                               FirstElem());
  }
};

// spec. for edge [not shown]
// ...
```

In order to inject the new specialized syntax provided by `map_element_name<>`, we use a sort of "Nackman-Barton trick" [2], deriving the final grid function from it:

```
template<class E, class T>
class partial_grid_function :
  // defines ElementIterator
  public grid_function_hash<E,T>,
  public map_elem_name<
           ElementIterator,
           partial_grid_function<E,T>,
           elem_traits<E>::elem_tag>
{ /* constructors */ };
```

Further complications arise if in the 2D case, we want to provide both `Edge` and `Facet` names. Building on the techniques just described, the solution is easy, if the element traits of the 2D edge type provides `edge2d_tag` instead of `edge_tag`. We simply take the union of the corresponding `map_elem_name<>` specializations:

```
template<class ElemIt, class ElemR>
struct   map_elem_name<ElemIt,ElemR, edge2d_tag> :
  public map_elem_name<ElemIt,ElemR, edge_tag>,
  public map_elem_name<ElemIt,ElemR, facet_tag> {};
```

6 Efficiency

When dealing with high-level implementations of algorithms, one generally runs the risk of losing performance with respect to a low level implementation. However, the generic approach

using C++ templates often allows good compilers to optimize out much of this so-called *abstraction penalty*. We used compilers `gcc 2.95`, KAI `KCC v3.4` and `g77 v0.5.24`[3] (with options `-O3 -fforce-addr -funroll-loops`) on Linux `2.2.14` running on a 450 MHz Pentium 86686 with 512K cache. We tested several grid sizes between 400 and 250.000 cells; the *ratios* of run times did not show large dependencies on grid size.

A first test case was the vertex-cell incidence counting algorithm shown on page 5, using a simple array-based Fortran77 data structure for triangular grids as point of reference:

```
   INTEGER ntri
   INTEGER til(1:3,1:ntri), ncells(1:nv)
   DO 20 c = 1,ntri
      DO 10 vc = 1,3
         ncells(til(vc,c)) = ncells(til(vc,c))+1
10    CONTINUE
20 CONTINUE
```

Here `ntri` is the number of triangles, and `til(v,c)` is the index of vertex number `v` of cell `c` ($1 \leq$ `v` ≤ 3). It turns out that in this case, the KAI C++ compiler (options used: `+K3`) can *completely eliminate* the overhead due to abstraction.

The algorithm involves indirect addressing, which is much more typical of unstructured grid algorithms than are plain vector loops in a BLAS style.

Another test case involving geometric primitives still shows a non-vanishing overhead, where the factor varies between 1.2 and 1.8. For instance, the following loop for summing up the facet normals of a cell has an overhead of about 1.8 if a generic grid geometry is used, and of 1.2 if the geometry is specialized to the grid type (that is, using low-level code inside the calculation of `normal()`):[4]

```
coord_type normal(0,0);
for(CellIterator c(aGrid); c; ++c)
  for(FacetOnCellIterator fc(*c); fc; ++fc)
    normal += Geom.normal(fc);
```

The only difference between the two implementations of `normal()` is, that in the high-level case coordinate values are stored in a grid function and accessed via intermediate vertices (carrying an extra pointer to a grid), whereas in the low-level case, coordinate values are stored in an array and are accessed by vertex indices obtained as in the Fortran `til` cell-vertex incidence array above.

The options used for `KCC` were `+K3 --abstract_float --abstract_pointer --restrict --inline_implicit_space_time=100`. Omitting the last option increases run time by a factor of about 3! A reason is probably the deeper nesting of function calls inside `normal()` in the high-level version.

The examples presented are in some respect a worst-case scenario for the performance of generic implementations using the micro-kernel, because no real work is performed inside the loops. The performance gain obtained by using a better data layout often outweighs the difference between high-level and low-level data access.

7 Some Recurring Difficulties

Many problems in generic programming arise from the fact that one has to deal with heterogeneity introduced by the representation of data. For example, in the generic implementation of grid functions, one has to take different action depending on whether the elements are consecutively numbered or not (a *semantic* heterogeneity). Also, the number of elements of a given type in a grid is accessed by different functions (a *syntactic* heterogeneity).

We have overcome these heterogeneities by collecting the relevant information about element types in element traits mentioned before. On a higher level, an algorithm using grid functions does not have to bother with these details, as grid functions offer a completely uniform way of dealing with data associated to grid elements.

This is an example of what can be called *homogenization*: Heterogeneous properties of entities playing the same role (grid elements and their handles) are hidden by a uniform interface (`traits::size(grid_t const&)`) or dealt with at the next higher level (grid functions), and thus do not propagate any further.

This approach crucially depends on a specialization mechanism, or still better, partial specialization. The common conception of C++

3. This is certainly not the best available F77 compiler; however, tests on a SUN Ultra (where `KCC` was not available to us) using the SUN F77 compiler showed the same ratios between F77 and `gcc` performance.

4. The ratio increased to about 1.4 for small grid sizes

templates as just a kind of macro turns out to be a misconception at this point.

The art, of course, consists in identifying a small number of key properties that allow to capture the *essential* differences of representations. For grid elements, one such key property is whether or not they are numbered (stored) consecutively.

A related problem is the adaptation of algorithms to the capabilities of data structures. An extreme case is a testing function for grid data structures: Here, we want to use exactly those concepts implemented by the data structure. A generic implementation of such a testing procedure would have to automatically adapt to the set of supported concepts. This could be achieved by breaking down the algorithm in 'atomic' pieces and put them together, based on compile-time information whether a certain feature (e. g. iterator) exists or not.

A third difficulty is the decision of what to parameterize. As many algorithms internally use a lot of data structures which could possibly affect their performance or execution, these could be made parameters as well. However, this leads to blown-up interfaces, which leave many of the decisions (and possibilities of making mistakes) to the user. We have experimented with a layering of interfaces, from minimal to maximal parameterization, to alleviate the problem. If there are many potential parameters, a more systematic approach is needed to organize them, especially sensible defaults.

8 Discussion

In this paper, we have described a micro-kernel as a basis for generic programming in the domain of grids and grid algorithms. The generic approach gains increasing attention in the field of scientific computing. To our knowledge, the work presented here is the first to successfully apply this paradigm to arbitrary grids used in this field. All libraries for numerical PDE solution that are using generic programming techniques and known to the author work on specific grid data structures, either structured grids (e. g. [6]), semi-structured (e. g. [16]) or unstructured (e. g. [9]).

Widening the focus beyond PDE solution two examples that perhaps come closest to our work in scope and problem domain are CGAL (Computational Geometry Algorithms Library, [17, 8]) and GGCL (Generic Graph Component Library, [12, 11, 19]). Also, the LEDA library [14] — which by itself does not offer generic algorithms — has been enhanced with graph iterators (GIT - Graph Iterator Extension, [15]) to support generic graph algorithms.

CGAL offers data structures and algorithms used in geometric computing, such as convex hulls and Delaunay triangulations, which correspond to the grid data structures treated here. The equivalent of incidence iterators is given by *circulators* in 2D. Grid functions are not supported as a separate entity; instead, one can parameterize some data structures by the type of data stored on the elements like vertices or faces. This has the disadvantage of coupling the data structures to the set of algorithms using them. On the whole, implementations of algorithms on grids seem to be slanted towards the particular family of *half-edge* data structures used in CGAL, thus limiting their reuse in different contexts.

GGCL implements generic graph algorithms and data structures. The equivalent of incidence iterators is modeled by a sort of 'virtual container' concept, making it possible for instance to access all edges adjacent to a given node. Instead of grid functions, one can use a decorator pattern to achieve a similar effect, but which lacks the uniformity of our approach.

Graphs can be obtained from grids in several ways, by stripping off some structure and the geometric aspects. As many algorithms on grids, for example grid partitioning, can be formulated on graphs in a more general way, using algorithms from a generic graph library like GGCL would be highly interesting. For each of the several possibilities of viewing a grid as graph, only one single adapter would have to be written. This adaptation is scheduled for the near future.

What we have described here touches only a small part of the work presented in [4]. Two complete applications for the numerical solution of partial differential equations have been developed, one using FEM and adaptive multigrid, the other using a FV approach.

Of course, the generic approach does not make the task of developing a *single* PDE solver much easier — much preliminary effort has to go into the domain analysis and development of reusable components. But once on can build on

this effort, the approach has considerable advantages. For instance, even if the underlying grid type is not intended to be changed, the micro-kernel layer effectively shields algorithmic code from changes to the low-level data representation, thus making later optimizations much easier. A typical example for such optimizations is the decision whether to store or to compute certain geometric quantities, a question which can be answered only on the basis of the concrete algorithms used.

To date, one of the most convincing proofs of the viability of the approach is given by parallel PDE solution. The distributed grid components mentioned above have been employed successfully to parallelize a generic FV solver, as well as a pre-existing Navier-Stokes solver [1]. For the latter, in order to generate a distributed version of the code, only a simple adapter mapping the functionality of the original grid data structure to the micro-kernel had to be written, and some localized changes to the numerical code had to be made. So, work that normally requires several months could be achieved within a few days.

The full potential of the generic method becomes visible when producing whole *families* of PDE solvers. At present, a rather small family is implemented, allowing a FV solver to vary — besides grid and geometry — the type of the (hyperbolic) equation to solve, and, by selecting the appropriate grid type, to choose whether code for sequential or distributed computation is generated. As grid data structures represent only a small part of the parameters of variation, there is much opportunity for future extensions. Also, a more systematic approach to implementing grid data structures is promising, using e. g. principles of generative programming [7].

Currently, the code is being prepared for public release [3].

References

[1] Georg Bader, Guntram Berti, and Klaus-Jürgen Kreul. Unstrukturierte parallele Strömungslöser auf hybriden Gittern. Final report on the SUPEA project, Technical University of Cottbus, 1999.

[2] John J. Barton and Lee R. Nackman. *Scientific and engineering C++*. Addison-Wesley, 1995.

[3] Guntram Berti. GrAL – Grid Algorithms Library. `http://www.math.tu-cottbus.de/~berti/gral`, 2000.

[4] Guntram Berti. *Generic software components for Scientific Computing*. PhD thesis, BTU Cottbus, to be published in 2000. Available under `http://www.math.tu-cottbus.de/~berti/diss`.

[5] Eric Brisson. Representing geometric structures in d dimensions: Topology and order. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 218–227, 1989.

[6] David L. Brown, Daniel J. Quinlan, and William Henshaw. Overture - object-oriented tools for solving CFD and combustion problems in complex moving geometries. `http://www.llnl.gov/CASC/Overture/`, 1999.

[7] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 2000.

[8] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *LNCS*, pages 191–202. Springer-Verlag, 1996.

[9] Oliver Gloth et al. Mouse, a finite volume library. `http://fire8.vug.uni-duisburg.de/MOUSE/`, 1999.

[10] Wolfgang Hackbusch. *Multigrid Methods and Applications*. Springer, Berlin, 1985.

[11] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *Proccedings of OOPSLA'99*, 1999.

[12] Andrew Lumsdaine, Lie-Quan Lee, and Jeremy Siek. The Generic Graph Component Library, GGCL. `http://www.lsc.nd.edu/research/ggcl/`, 1999.

[13] Albert T. Lundell and Stephen Weingram. *The Topology of CW Complexes*. Van Nostrand Reinhold, 1969.

[14] Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. The LEDA home page. `http://www.mpi-sb.mpg.de/LEDA/`, 1999.

[15] Marco Nissen and Karsten Weihe. Combining LEDA with customizable implementations of graph algorithms. Technical Report 17, Universität Konstanz, October 1996.

[16] Christoph Pflaum. EXPDE – expression templates for partial differential equations. `http://ifamus.mathematik.uni-wuerzburg.de/~expde/lib.html`.

[17] The CGAL project. The CGAL home page – Computational Geometry Algorithms Library. `http://www.cs.uu.nl/CGAL/`, 1999.

[18] SGI Standard Template Library Programmer's Guide. `http://www.sgi.com/Technology/STL`, since 1996.

[19] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. The generic graph component library. *Dr. Dobb's Journal*, 25(9):29–38, September 2000.

[20] Günter M. Ziegler. *Lectures on Polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg, 1994.