To appear in *Concepts of Numerical Software*, W. Hackbusch, G. Wittum (eds.), Vieweg Verlag, Braunschweig, 1998

Design Principles of Reusable Software Components for the Numerical Solution of PDE Problems

Georg Bader and Guntram Berti
Institut für Mathematik, TU Cottbus
Universitätsplatz 3-4, D-03044 Cottbus
{bader,berti}@math.tu-cottbus.de

Summary

We first present some general considerations about software for Scientific Computing and argue in favor of a component-oriented approach. Algorithmic components are identified as rewarding but also demanding targets for reuse by means of generic programming. In the second part we examine in more detail the interaction of algorithms and grid data structures, and show how numerical procedures can be formulated in a way that is independent of concrete grid implementations. Finally, extensions of these grid-related concepts are introduced that ease the development of parallel computing software for the numerical solution of PDEs.

1 A component-oriented approach

1.1 The challenge

When we consider the development of reusable software for numerical computation, we invariably hit at a fundamental dilemma. On the one hand, we desire a framework that is general enough to allow for a very wide class of mathematical problems and different algorithmic approaches, and ideally, is open for extensions to new, unanticipated functionality. Everything, of course, should be portable to a broad range of software and hardware environments. For short, it should be as *independent* of problems and environment as possible.

On the other hand, when solving *concrete* problems, we want to profit from specific knowledge, both in terms of improved efficiency and in the possibility to integrate special *ad hoc* functionality, such as exploitation of problem symmetry or optimization for the specific hardware. That is, we want to *specialize* the software for the problem, while relying on prefabricated, well-tested parts of the library as much as possible.

Classical numerical software always emphasized the second topic, whereas many object-oriented approaches in this field tend to sacrifice efficiency for flexibility. So the fundamental challenge is to achieve a synthesis of these seemingly contradicting aims, a task we try to sketch a tentative solution for in this paper. A more detailed elaboration of these topics will be given in [4].

1.2 Component libraries

From the requirements identified above it seems evident that no monolithic concrete module or class hierarchy can satisfy them all. Rather, we propose to identify abstract *concepts* for software parts that are implemented not by one, but by many concrete components, so called *component families*, each having different trade-offs concerning simplicity, efficiency, functionality and flexibility. The resulting *component forest* is better suited to handle diverging requirements. *Generality* comes from the multitude of implementations allowed for an abstract concept (such as a geometric grid), whereas *specialization* is done by an application programmer choosing exactly the components that fit best his needs.

This results in potentially large libraries with many rather small and focused components, each implementing a specific concept. Keeping components focused makes them easier to reuse. A grid class, for example, that contains data for a specific numerical method will be harder to reuse than an implementation confined to just a geometric grid.

We aim at different components implementing the same concept to be *exchangeable*. The general situation is schematically indicated by figure 1. This requires a standardization of interfaces. The same functionality has to be represented by the same syntactic entities. Note, this does *not* imply that all components of a family have the same functionality, e. g. a structured grid may give access to the northern neighbor of a vertex, whereas an unstructured grid might allow to divide a cell into triangles. On the other hand, there are situations where structured and unstructured grids may be used interchangeably, for example for visualization, because only the *intersection* of their functionality is used.

In this example, we may see the structured and unstructured grid as refinements of the concept of a general grid. Capturing this relation into an inheritance tree and using then the abstract base as interface is in general not an ideal solution, because it results in a loss of information that may make efficient implementation impossible. We will return to this issue later.

Perhaps a more striking example is given by the notion of an equation. Although the concept itself is rather well-established in a mathematical sense, it is difficult to formulate a general interface, so equations have to be represented by a number of largely unrelated components, cf. figure 1.

The user of such a library may select components which represent the optimal compromise between efficiency, space requirements and complexity for his concrete problem at hand. So, for doing FEM analysis, he may take a lean grid component allowing only tetrahedrons if sufficient, and switch to a more complex grid with general elements, if need should arise. As a guideline, one could say that decisions about appropriate compromises should be left to the user, instead of hard coding them into the library. This in turn implies, that a library must offer sufficient freedom to choose between components with different trade-offs.

1.3 Algorithmic components

It proves useful in this context to distinguish *algorithmic* from *data-oriented* components. Modules like grids, geometric domains and matrices are typically data-oriented, whereas grid generation, visualization and numerical discretizations are algorithmic in nature.

Implementations of algorithms are shaped by the interfaces of underlying data structures, so standard interfaces of data-oriented components must be carefully designed to capture the essence of the data. The involved complexity poses a challenging problem for the development of reusable algorithmic components.

According to the standard object-oriented paradigm, data and methods are bundled together. This, however, may lead to implementations, where algorithms are *buried* in some class method, say, a sparse matrix class which embodies a particular iterative solver.

Therefore, it is nearly impossible to use them in another context – in this case, for another matrix class. Further problems occur when algorithms are developed with one specific data representation in mind, such that any arbitrary decision made in the data layout is hard-wired into the algorithm's implementation.

A first concern is to separate data and algorithms: Algorithms should be implemented as entities of their own. These algorithms are then formulated in a *generic* way, that is, independent of a concrete data layout. For this end, the common structure of the different underlying data representations is extracted in an abstract *data model* and the implementation of the algorithm is based on that model.

As the algorithms under consideration exhibit typically frequent and atomic data access, the abstractions must be carefully chosen in order not to corrupt efficiency of the resulting code. The key point here is that we must provide the compiler with sufficient information about the *concrete* types involved, so that the abstractions can be optimized away - at least in principle, depending on the quality of the compiler.

The Standard Template Library STL [8] from the **C++** standard library is a good example for the paradigm just described – and a source of inspiration for our work. Here the underlying data model is that of a linear sequence, represented by *iterators*. The algorithms of the STL, such as counting, searching and sorting, are based upon the iterator interface. In that way, iterators shield algorithms from concrete data layout.

Of course, sequence data structures have different characteristics – we can jump by a random distance in an array, but move only one-by-one in a list. Accordingly, the corresponding iterators differ in their capabilities. However, it turns out that only five different types of iterators have to be distinguished to select appropriate implementations in all circumstances. We may do a binary search on an array much faster than on a list, but this does not depend on the array but only on the iterator being a *random access iterator*.

This example illuminates an important point: Data often is not as homogeneous as suggested by the pure existence of a common abstract interface. When efficiency demands a more fine-grained distinction of data properties, we should not try to be too general. Specialized versions of the algorithm may be embedded in a so-called *poly-algorithm* that transparently branches depending on the nature of the data. For example, we could have a generic procedure that locates the grid cell containing a given point, choosing different algorithms for structured and unstructured grids, respectively. The real difficulty lies in

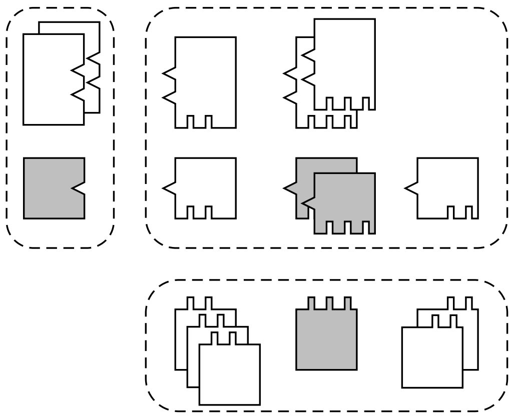

Figure 1: Three component families

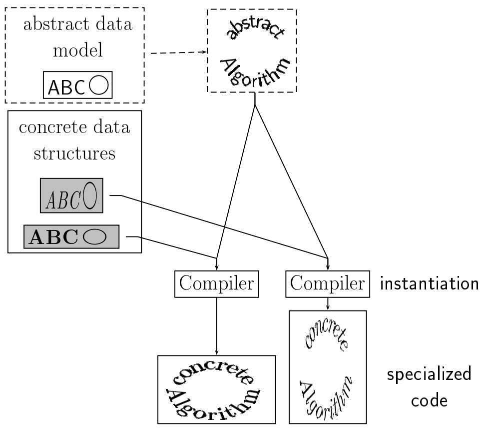

Figure 2: Instantiation of generic algorithms

identifying the abstract properties of data that have an impact on the algorithm, thus minimizing the number of versions of an algorithm that are needed.

Data structures used in the numerical solution of PDEs - such as hierarchical grids - are normally much more complex than linear sequences. Nevertheless, it seems promising to exploit the generic approach also in this context, especially as the algorithms under consideration are more difficult and hence there is greater potential gain from a generic implementation.

1.4 The pros and cons of generic programming

It should first be noticed that abstraction and efficiency are by no means irreconcilable. As an example, the STL implementation of the QUICKSORT algorithm runs on simple types up to three times faster than the corresponding Unix library routine, just because of the inlining of the comparison function.

Generic programming makes high demands on programming languages and compilers. The language must support parameterization by the *type* of arguments, like *templates* in **C++** and *generics* in Ada. We did our implementation work in **C++**, a language supporting different levels of abstraction as well as different programming paradigms – procedural, object-oriented and also some kind of functional programming via advanced use of templates.

Other approaches use a different language for the generic formulation and rely on an ad-hoc compiler that does the instantiation of the abstract implementation into specialized low-level code, see [5] and [11]. This however requires a strict separation of the two layers and make a gradual adoption of this programming style more difficult. In addition, one has to rely on that special purpose compiler.

In contrast, we use standard **C++** compilers. It must be admitted that presently template handling and efficient translation of "abstract" code still has not yet matured and constitutes a challenge [12]. Longer compile-times, incomplete debugging and tool support and potential code bloat are some of the drawbacks we have encountered.

There is also a strong need to gain more experience with the power and also the limitations of this programming style, such as which constructs help and which hinder efficient code generation.

Specialization of a generic formulation into lower level code must therefore at times be done by hand in order to achieve efficiency. However, we should notice that we are doing a machine's job here. The ultimate goal should be to let this *lowering* be done automatically by a compiler. We expect that progress in future compiler technology will help using the full potential of the generic approach. Meanwhile techniques have been developed which allow generation of Fortran-competitive object code from a high-level **C++** specification [14].

Just as today everybody relies on the compiler to generate efficient assembly code, our expectation is that the same will one day be the case for generically programmed procedures; in the meantime, we will have to live with compromises. However, we think that in the scientific context, good generic implementations will turn out to be a valuable investment. In particular, this should lead to improved maintainability, better reusability and easier construction of working software for new problems.

2 Components for grids and geometries

2.1 Overview

Grids are certainly one of the central and important data structures in software for the numerical solution of PDEs. They range in complexity from simple Cartesian grids to complicated hierarchical multiblock grid structures. This diversity and the need to trade-off complexity vs. efficiency give rise to a multitude of coexisting implementations. Many core algorithms in the PDE-context, such as numerical discretizations, visualization and grid generation, operate on grids. We present an abstract data model that hides the complexity of internal data representation by means of *iterators*, thus allowing generic implementations of many algorithms in the spirit of the preceding section.
In our design, the functionality of a grid comprehends the following:

combinatorial structure ("connectivity")	geometrical embedding ("coordinates")	data storage support ("states")

The combinatorial layer contains the entire adjacency information (sometimes called grid topology) – that is, everything that remains invariant under a geometric mapping of the grid. In other words, the underlying mathematical concept is that of an abstract *cell complex* [1]. The geometrical layer adds an embedding into a geometric space. The splitting of the concept *geometric grid* into the subconcepts *combinatorial grid* and *geometric embedding* allows for leaner components and enhances reuse by combination of different grids and geometries. So we may map the same combinatorial grid onto a bounded region in 2D-space or a part of a 2D submanifold in 3D-space. It also results in a proper localization of responsibility for these geometric decisions.

The data storage layer also is often melted into the grid data structure. The need to store, say, solution values on the vertices of a grid is very problem dependent and should not be coded into a general grid component. Therefore, a universal mechanism is needed to store *arbitrary* data on all *grid elements*, not only on vertices. To this end, we introduce the *grid function* components.

2.2 The combinatorial layer

As a cell-complex, a grid is composed of basic topological entities like vertices, edges and so on, that we call *grid elements*. The partial ordering induced by the inclusion relation on these grid elements gives rise to a *grid lattice* that contains the complete combinatorial information of the grid, see figure 3. In general, this lattice is of course present in a conceptual manner only, the actual adjacency information can normally be stored in a more compact way or is even implicitly available, as in the case of a structured grid. The term *structured* is used here rather loosely to indicate that the adjacency information can be derived, as is the case for a Cartesian grid or the regular refinement of triangles.

In order to work with a grid, we associate several types with it, in particular *grid elements* and *iterators*. The exact nature of these types depends on the particular grid type, but their generic properties are independent thereof.

We name **elements** according to their dimension and codimension, cf. figure 4. Different names do not necessarily mean different types, for a 1-dimensional grid the types

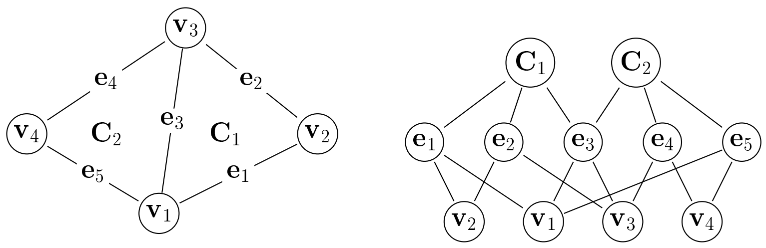

Figure 3: a simple cell-complex and its lattice

`Vertex` and `Facet` as well as `Edge` and `Cell` would coincide, whereas in a 2-dimensional grid the types `Edge` and `Facet` are the same. The difference lies in the *role* they play in an algorithm: If we want to display the wire-frame model of a grid, we do that in terms of `Vertex` and `Edge`; if, on the other hand, we formulate an algorithm for computing a numerical flux, we use `Cell` and `Facet`. In this manner, the algorithms are expressed in a dimension-independent way.

At a basic level, a grid is a container of its elements, i. e. a set of vertices, a set of edges and so on. This is captured by **sequence iterators** that perform global loops over the whole grid: `VertexIterator`, `EdgeIterator` etc. The example accumulates the total volume covered by all cells of a grid:

```
for(CellIterator c=G.FirstCell(); !c.IsDone(); ++c)
  volume += geometry.volume(c);
```

In order to access neighboring information, for example looping over all vertices of a cell, we introduce **adjacency iterators**. Such an iterator is tied to an *anchor element* (here: `Cell`) and visits all adjacent elements of its *value type* (`Vertex`). The sample code determines the mass-average of the vertices of a cell `C`:

```
for(VertexOnCellIterator v = C.FirstVertex(); !v.IsDone(); ++v)
   sum += geometry.coord(v);
average = sum / C.NumOfVertices();
```

There exists a multitude of possible adjacency iterators, especially in higher dimensions. As the complexity of grid data structures typically rises with the number of supported iterators, it is not mandatory that a grid supports them all. The user is free to decide whether he can do with a lean grid component implementing only few iterators, or if more complete functionality is needed.

Often, algorithms are operating not homogeneously on the whole grid, but do something special on a *grid subrange*, for example on the boundary. The exact nature of the

Element	dim.	codim.
`Vertex`	0	
`Edge`	1	
`Face`	2	
`Facet`		1
`Cell`		0

Figure 4: Element types

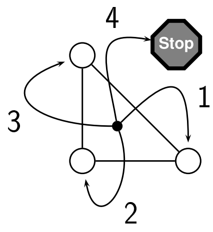

Figure 5: `VertexOnCellIterator`

subrange depends on the algorithm applied. For a simple visualization there is typically no need to treat the boundary in any special manner, whereas for higher-order numerical algorithms the boundary subrange might be rather large. Similar functionality is needed if the problem is not homogeneous, for example if there are different material properties involved, or in the case of overlapping and composite grids. Therefore grid subranges are regarded as core parts of grid components.

2.3 The grid function layer

We often need to associate data with a grid, or more precisely, with grid elements: store solution values at vertices, mark cells for refinement and so on. This is accomplished by *grid functions*. A `grid_function<E,T>` defines a mapping between grid elements of type `E` and values of type `T`, where $\mathtt{E} \in \{\mathtt{Vertex}, \ldots, \mathtt{Cell}\}$ and `T` is arbitrary:

```
grid_function<Vertex,valuetype>  state(Grid);
sparse_grid_function<Cell,bool>  to_refine(Grid);
```

There exist two different versions of grid functions : *total* or *dense* grid functions that are defined on the whole grid, and *sparse* grid functions that are set only for parts of the grid (assuming a default value for the rest). They compare to each other much like dense to sparse matrices.

The implementation of grid functions is tightly related to the underlying grid. Where possible, arrays are used as internal data structure for dense grid functions, while for sparse grid functions, hash tables are often adequate. By the way, with grid functions it is very easy to produce a sparse matrix tied to the grid:

```
typedef sparse_grid_function<Vertex,int> row;
typedef grid_function<Vertex,row> sparse_matrix;
sparse_matrix A(G,row(G,0));
/*  calculate adjacency matrix for grid G  */
for(EdgeIterator  e = G.FirstEdge(); !e.IsDone(); ++e)
  A[e.V1()][e.V2()] = 1;
```

By replacing the type of `row`, we can generate different variants of sparse matrix data structures. Of course, there is more to sparse matrix representation than what we can cover in this short example.

2.4 The geometric layer

In the context of numerical solution of PDEs, grids are typically used to approximate a physical domain. For this an *embedding* of the combinatorial grid into a geometric space is needed. This mathematical concept is implemented by *grid geometries*. The functionality of geometries may be briefly described as mapping combinatorial to geometric elements, that is, vertices to coordinates, edges to segments and so on. This can be done in many different ways. We may embed into 2- or 3-dimensional Euclidean or non-Euclidean spaces, use straight-line or curved elements and time-dependent geometries.

Additionally, a grid geometry may provide functions to calculate volumes, centers and mappings to reference elements. Grid geometries also differ in implementation, so there is freedom to choose a compromise between storing and computing of items, to allow various geometries to share the same grid or to decide to give approximate solutions (e. g. approximating the volume of a cell in the case of curvilinear embedding). Several geometries sharing one grid may be also useful when refining a grid that represents a domain with curved boundary: A curvilinear geometry can be used for refinement, and a straight-line approximation for calculation, as is depicted by figure 6.

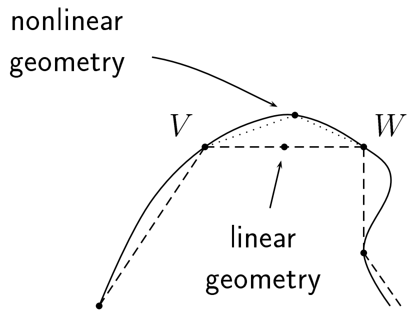

Figure 6: creation of a new vertex

2.5 Generic grid algorithms

Before presenting some concrete examples for the numerical PDE solution let us stress that there are many classes of non-numeric algorithms on grids that may profit from generic implementation, for example visualization or grid generation. A grid may be considered as a graph with grid cells acting as nodes and facets as edges. This view of a grid might serve as input for graph partitioning algorithms from a library of generic graph algorithms. Indeed, there are research efforts in this direction, see [7].

Stiffness matrix A first example is the stiffness matrix assemblation for Laplace's equation $-\Delta u = f$ with piecewise linear ansatz functions φ_v:

```
sparse_matrix A(G);
for(CellIt c = InnerRange.FirstCell(); !c.IsDone(); ++c)
  for(VertexOnCellIt v = c.FirstV(); !v.IsDone(); ++v)
    for(VertexOnCellIt w = v; !w.Isdone(); ++w)
      A[v][w] += FE.integrate(c,v,w);
```

For each cell, we loop over all pairs of vertices (`v,w`) and calculate the partial scalar product

$$\texttt{FE.integrate}(\texttt{C}, \texttt{v}, \texttt{w}) = \int_C \nabla\varphi_v \nabla\varphi_w \, dx.$$

Flux calculation A hyperbolic equation

$$\partial_t u + div_x(F(u)) = 0$$

can be discretized in space by approximating the flux over a cell boundary by

$$\int_{\partial \mathtt{C}} F(u) \cdot \mathbf{n} \, ds \quad \approx \sum_{\mathtt{f} \in \mathrm{Facets}(C)} \texttt{flux(f)}$$

The calculation of the flux sums for each cell is implemented by a facet-oriented loop, exploiting symmetry:

```
grid_function<Cell,flux_type> Flux(G,0.0);
/*  calculate fluxes for inner facets  */
for(F = InnerRange().FirstFacet(); ! F.IsDone(); ++F){
  flux_type flx = flux(f);
  Flux[f.C1()] += flx;
  Flux[f.C2()] -= flx;
}
/*  calculate fluxes for boundary facets  */
for(BC = G.FirstBCRange(); ! BC.IsDone(); ++BC)
  for(bf = BC.FirstFacet(); ! bf.IsDone(); ++bf)
    Flux[bf.Cell()] += flux[BC](bf);
```

Note that this code is completely dimension independent.

The preceding examples indicate that the presented concepts are quite powerful. We expect, however, that experience with a larger number of algorithms may lead to some refinements of grid components.

3 Distributed grids for parallel computing

3.1 Overview

In many methods for numerical PDE solution, the partition of computational work and data in a distributed application is essentially determined by a partition of the underlying grid. The corresponding algorithms typically access data locally and perform the same operation on a large number of uniform items. In this context, it seems quite natural to take a *grid-centered*, data parallel point of view towards the development of parallel code. The goal is to achieve parallelization of algorithms by shifting administrative work due to data distribution to data oriented components. As a result, many of the algorithmic components can be reused, and the underlying numerical methods remain essentially unchanged. Note, this has to be distinguished from situations where the numerical algorithm itself is substantially changed to achieve a good parallel behavior.

The architecture we bear in mind is that of a distributed memory machine, where communication is done by a message passing mechanism like MPI [9]. Although we emphasize the data parallel point of view, we do not promote automatic parallelization. The triggering of communication activity is always explicit in the application code. However, the latter is freed from low-level details by bundling the essential data-access and communication patterns into high-level primitives.

An analysis of the communication patterns reveals two major types: *Local* or *neighbor* communication with a typically high volume of data exchange, and *global* or *hierarchical* communication, for example reduce or broadcast operations, and user control.

Here we mainly concentrate on neighbor communication, where a further distinction between *static* and *dynamic* communication can be made. Static communication is understood to occur when only data *on* the grid is exchanged, the grid itself remains unaltered. There more complex dynamic case when the grid changes (e. g. when locally refined) is currently work in progress. The static case will typically account for the major communication load of a distributed application and should therefore be done with a minimal

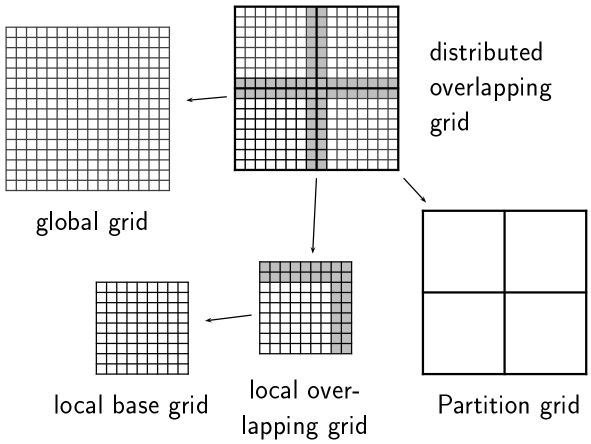

Figure 7: Parts of a DOG

overhead, exploiting that a fixed exchange pattern allows to eliminate completely transport of redundant data.

3.2 Distributed overlapping grids – *DOG*s

Generally, we take a local point of view, classifying the actions and data of one process with respect to its *neighbors*. In order to bundle communication, some kind of overlap or cache will in general be necessary. We explicitly deal with this issue by introducing *distributed overlapping grids*. The *extent* of the overlap is determined by the collection of algorithms which are executed on the grid. Below, we show how the overall *structure* of the overlap can be described in a very general and dimension-independent manner. Note that the principal mechanisms and concepts work not only for a distributed memory setting, but as well for sequential multi-block or cache-optimized blocked structures. The concepts are useful for grids with different degrees of regularity, ranging from Cartesian to completely unstructured. Obviously, some implementation details can be done more efficiently in a Cartesian setting.

The global partition of work is driven by a coarse *partition grid* whose cells correspond to the local grid parts. With a *global grid*, we mean the logical union of the local grids, an entity that does not exist physically, except in the case of the partition consisting of only one part. The implementation of both *partition grid* and *local grids* can be built on top of various, "sequential" grid data structures. These have to be augmented with some additional functionality to manage the overlap. For that purpose, the availability of grid subranges is crucial.

If we fix a local part of the global grid, we may classify the elements of the global grid with respect to this part and the given partition:

- *owned* elements for which the local part is exclusively responsible
- *shared* elements for which responsibility is distributed among two or more parts
- *foreign* elements, for which another part is responsible

Besides having read-write access to owned and shared elements, we typically need also read access to some foreign elements. These are cached locally and called *copied*, whereas

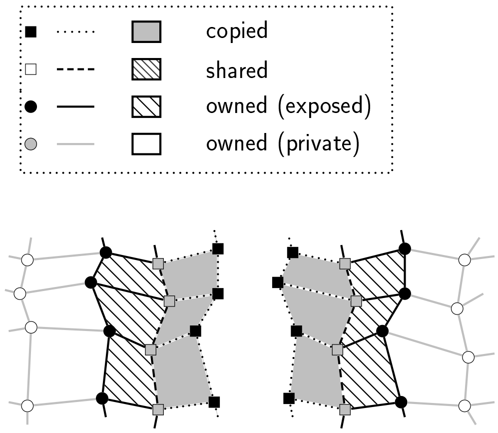

Figure 8: Two grid parts and their overlap ranges. (no shared cells)

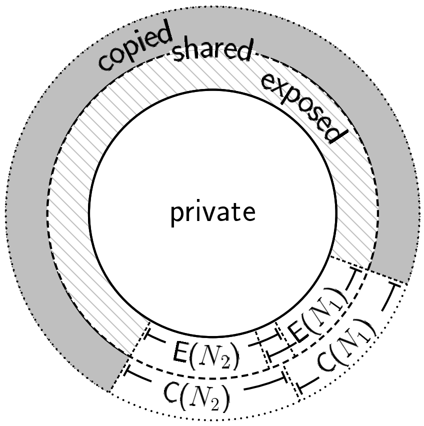

Figure 9: overlap structure

the inaccessible foreign elements are termed *remote*. Vice versa, the owned elements are divided into *exposed* ones, that are *copied* somewhere else, and *private* ones, that are known only to the local part. In sum, the local part consists of owned, shared and copied elements, whereas exposed, shared and copied elements form the *overlap region*.

We say that two elements stored in two different local parts *correspond* to each other if they represent the same element in the conceptual global grid. Two parts are neighbors of each other if they contain corresponding elements. The range of elements of a part that are copied from its neighbor N is called $C(N)$, and similarly we define $S(N)$ and $E(N)$.

Although a local element can correspond to elements in more than one neighbor, it makes sense to group corresponding ranges into pairs, such as $C(N_1)$ and $E(N_2)$ for two grid neighboring parts N_1, N_2. The key feature of such range pairs is *synchronous iteration*, that is, corresponding elements are accessed in the same order in both ranges. Synchronous iteration proves essential for the efficient implementation of static data communication. It can be supported very easily in a Cartesian setting, whereas in an unstructured distributed grid additional information must be stored.

3.3 Distributed grid functions – *DGF*s

For static communication, distributed grid functions (DGFs) are the level of abstraction at which the programmer has to deal with explicit data exchange due to the partition of the problem. We think of a DGF as being associated with a *grid-algorithm* responsible of setting the values of the grid function, executing a local function on each supporting element. The essential observation we make is that these local functions never need to be changed, whether we do parallel computation or not. That means, if we are able to factor our code into local functions and global iteration, we can restrict the changes to be made for parallelization to a thin layer of code.

A DGF in principle inherits communication ranges from the underlying distributed grid, but may restrict these if the algorithm requires less overlap. With respect to this algorithm, the state of a DGF may be classified as follows:

1. *locally consistent* or *synchronized* on a part N if all local, i. e. owned and shared, values are in the same state of computation

2. *locally inconsistent* if local values are in different states of computation

3. *globally consistent* or *synchronized* if the DGF is locally consistent everywhere and the values on corresponding elements are the same everywhere.

We generalize this definition by calling a DGF locally consistent on a range R, if all values on R are in the same state of computation. The DGF is *up-to-date* on R if in addition the values are in the most recent state of computation in that grid part.

The necessary data communication is then essentially driven by the need of achieving a globally consistent state. For this, we must first compute locally until the DGF is up-to-date on the *exported ranges*, that is, on the exposed and (perhaps) shared ranges. Then we may begin to send the data on these ranges, while continuing computation on the private range. When the receive of the copied range and the local computation is completed, we are locally up-to-date. If in every part, the DGF is locally up-to-date, we have arrived at a globally consistent state. Note that this includes the possibility to let computation and communication overlap, the so called *compute-and-send-ahead* strategy.

Given the existence of *synchronous* iterators over the sets, the implementation of data exchange is rather straightforward. We just need to send (copy) the sequence of values on $E(N)$ and $S(N)$ to neighbor N and to receive the data on $C(N)$ from N. The shared ranges need special attention, because the communication required for them may differ. We may decide to do double calculations with no data exchange or to do partial calculation and data exchange in both directions on them. The latter occurs e. g. if vectors are stored such that the values on corresponding shared vertices have to be summed up to obtain the global value, see [3].

The code using a DGF then has to issue a call to `dgf.synchronize()` after having done its calculation. If computation and communication shall be interleaved, we do

```
foreach(dgf.exported_ranges(),local_function);
dgf.begin_synchronize(); /* is up-to-date on exported */
foreach(dgf.private_ranges(), local_function);
dgf.end_synchronize();   /* get globally consistent */
```

which can be compacted into a call

```
dgf.compute_and_send_ahead(local_function);
```

As an example, we show how a finite volume flux computation with a recovery step can be parallelized using the components we have presented so far. The structure of the algorithm is described below:

```
forall vertices
   calculate average of adjacent cells' states
forall cells
  calculate recovery interpolation from vertex average
forall facets
  calculate flux from recovery
  sum up fluxes in corresponding cells
forall cells
  calculate next timestep from fluxes and old state
```

The necessary overlap is described in figures 10 and 11, where we use the notation E_V to denote exported vertices. Figure 10 shows how the new cell state depends on the previous state. We use a DGF `avg` for the vertex average with double calculations on shared and a

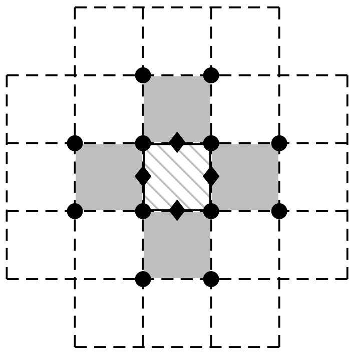

Figure 10: Domain of dependency

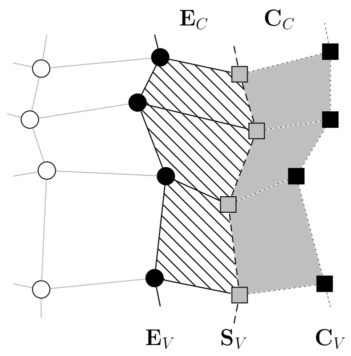

Figure 11: overlap for FV flux calc.

DGF for the state. The flux calculation on shared facets will also be doubled. This result in the following code for the vertex average:

```
for(VertexIt v= LocalRange().FirstVertex(); ! v.IsDone(); ++v){
  double volsum = 0.0;
  for(CellOnVertexIt c = v.FirstCell(); ! c.IsDone(); ++c) {
    avg[v] += State[c] * Geometry.volume(v);
    volsum += Geometry.volume(v);
  }
  avg[v] /= volsum;
}
avg.synchronize();
```

This example was taken from a solver for inviscid flows of multicomponent gas-mixtures presented in Schenk et al. [13].

4 Outlook

The approach we have presented can ease the construction of efficient solver software for PDE problems. However, the development of good component libraries is a demanding task. Components must be supplied in sufficient variations to guarantee a certain freedom of choice to the user.

The generic implementation of algorithmic components promises substantial gains, but its entire potential cannot yet be fully estimated. Reuse of the complex algorithms for numerical PDE solution still constitutes a challenge, and is subject of ongoing research.

Extracting common properties of data and abstracting them into interfaces is the key to generic programming, but its difficulty is deepened by the fact that the abstractions have to be very carefully chosen to enable automatic generation of efficient code.

Grid components are a testbed for the applicability of the generic paradigm to Scientific Computing. The development of a number of more complex data structures like composite, distributed or hierarchical grids is substantially eased by the generic combination of more basic grid components, such that much code can be shared. Here the availability of generic algorithms can substantially reduce the work involved in using more complicated data structures. This is a current subject of research.

Acknowledgment

We would like to thank our colleague Klaus-Dieter Krannich for many helpful discussions and technical support. In particular, he undertook the effort of porting the code to the SP-2, which definitely is not an easy task due to the problems mentioned earlier and a certain experimental flavor of the project.

References

[1] Alexandroff, P. and Hopf, H. *Topologie*, volume 1. Springer, 1935.

[2] Barton, J. J. and Nackman, L. R. *Scientific and Engineering C++*. Addison-Wesley, 1995.

[3] Bastian, P. *Parallele adaptive Mehrgitterverfahren*. PhD thesis, Universität Heidelberg, 1994.

[4] Berti, G. *Gitterkomponenten und generische Algorithmen im Wissenschaftlichen Rechnen*. PhD thesis, TU Cottbus, in preparation.

[5] Boyle, J. M., Harmer, T. J., and Winter, V. L. The TAMPR program transformation system: Simplifying the development of numerical software. In Arge, E., Pettersen, A. M., and Langtangen, H. P., editors, *Modern Software Tools for Scientific Computing*, pages 353–372. Birkhäuser, 1997.

[6] Jazayeri, M. Component programming - a fresh look at software components. Technical Report TUV-1841-95-01, TU Vienna, 1995.

[7] Kühl, D. and Weihe, K. Iterators and handles for nodes and edges in graphs. Technical report, Universität Konstanz, Sep. 1996.

[8] Lee, M. and Stepanov, A. A. The Standard Template Library. Technical report, Hewlett-Packard Laboratories, 1995.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. 1994.

[10] Musser, D. R. and Saini, A. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[11] Novak Jr., G. S. Software reuse by specialization of generic procedures through views. *IEEE Transactions of Software Egineering*, 23(7):401–417, 1997.

[12] Robison, A. D. The abstraction penalty for small objects in C++. In *POOMA'96: The Parallel Object-Oriented Methods and Applications Conference*, Santa Fe, New Mexico, Feb. 1996.

[13] Schenk, K., Bader, G., and Berti, G. Analysis and approximation of multicomponent gas mixtures. To appear in *Proceedings of the 3rd Summer Conference Numerical Modelling in Continuum Mechanics*, Prague, 1997.

[14] Veldhuizen, T. and Jernighan, M. E. Will C++ be faster than Fortran? In *ISCOPE'97: The International Scientific Computing in Object-Oriented Parallel Environments Conference*, Marina del Rey, Dec. 1997.