Generic Software Components for Scientific Computing

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Diplom-Mathematiker
Guntram Berti
geboren am 8. Dezember 1967 in Dortmund

Gutachter: Prof. Dr. Georg Bader
Gutachter: Prof. Dr. Heribert Blum
Gutachter: Prof. Dr. Robert Glück
Gutachter: Prof. Dr. Christoph Zenger

Tag der mündlichen Prüfung: 25. Mai 2000

Contents

Chapter 1

Introduction

Man muß etwas Neues machen,
um etwas Neues zu sehen.
Georg Christoph Lichtenberg

At the origin of this thesis stood the own painful experience with the difficulties of developing and reusing scientific software, even if the underlying mathematical problems solved by the individual software in our working group were closely related.

Indeed, software development *is* a serious problem in almost all computational fields, and it is certainly not easier in the branch of scientific computing dealing with the numerical solution of partial differential equations. This is the context this thesis grew out of, and it runs through the whole work as a *leitmotiv*. Yet, both the methods applied as well as the results obtained reach beyond it.

The increasing number of publications and conferences devoted to scientific computing software is a clear indication of the problem's significance. Where do these difficulties come from, and is there anything special about scientific computing?

One leading goal of software development in scientific computing is *efficiency*: The problems solved are large, often as large as current technology allows.

This pressure for efficiency has had a deep impact on the methods of software development. Typically, many decisions are 'hard-wired' into the code and thus not localized any more. Data structures and algorithms are closely interlocked. If aspects of a working software, grown over some period of time, are to be adapted to new requirements, major difficulties arise.

Obviously, starting from scratch is not an option, due to the size of complete applications — systematic software reuse is on the agenda of computer science at least since McIlroys vision [McI68], but traceable to the earliest day of computing [Gol93]. Yet, success is mixed at best in general, and scientific computing makes no exception. What makes reuse so hard?

Scientific computing is an *algorithm-dominated* field, as are Computational Geometry, Computational Graph Theory, or what is more generally grouped under the term

'Computational Science and Engineering' (CSE). The natural *unit of reuse* in these fields often is an *algorithm*, or small sets of algorithms, in combination with associated data structures. Different applications combine different selections of algorithms in a unique way to achieve their specific aims. Yet, it is very common that a given algorithm is used by a whole class of applications.

But reuse of single algorithms is admittedly hard. It is true, there are a number of successful software packages, specifically for the different types of PDE solution. Yet, conventional packages are not organized in a way that would allow single-algorithm reuse *out-of-context*: Implementations of algorithms must make certain assumptions on the way the data they 'live on' are represented. The more complex the data structures are, the less stable are these assumptions, and the lower the probability that they hold across the boundaries of the current context.

Thus, problems occur whenever algorithms are to be reused outside the narrow context they were developed for, or if this context changes: So substituting new, better data structures may be impossible; in the limit, evolution of the application may face unsurmountable difficulties and come to an end.

The first major contribution of this thesis is an in-depth analysis of the problem, and the development of a solution strategy. This solution is worked out in detail for the case of algorithms operating on computational grids. Such grids are central to numerical PDE solution, and are important in other fields, like Computer Graphics or Computational Geometry. Furthermore, the corresponding data structures are sufficiently complex and variable to virtually exclude out-of-context reuse of algorithm, *unless special measures are taken*.

The solution we present attacks the central problem, namely the commitment to specific data representations.

The main task here lies in identifying and describing the relationship between data and algorithms. Towards this aim, we move a step back from the specifics of concrete data structures and algorithms, to gain a broader perspective. The mathematical branches of topology and combinatorics are chosen to provide the conceptual framework. Their precise language allows to express the essential mathematical structures of the problem domain in a general, abstract way, independent of any representational issues.

Guided by a review of representative grid algorithms, we then arrive at an abstract description of the capabilities of data structures: A small set of *kernel concepts*, on which a large portion of relevant algorithms can be based.

In order to transfer the abstract notions found into working software, the paradigm of *generic programming* is employed. It has already proven its usefulness and power for data structures representing a comparatively simple mathematical structure, such as linear sequences and matrices. The kernel concepts give rise to a small *brokerage layer* separating data from algorithms. We show that the approach leads to efficient implementations, with performance coming near to that of direct, low-level implementation.

Thus, a effective decoupling of grid algorithms from data structures is achieved. This decoupling raises the reusability of these algorithms to a new level.

The second major contribution of this thesis tackles the problem of parallel grid computations. A key problem in high performance scientific computing is the parallelization of applications. The driving force here is the need to calculate ever larger problems, or, to minor extent, the same problems in less time. Many computational tasks are beyond reach without the use of parallel computers.

Efficient parallelization of algorithmic code is a non-trivial task. It cross-cuts both data structures and algorithms. Automatic methods are of restricted use in this area, because a satisfying efficiency is generally not obtained this way.

A promising alternative are *domain-specific* approaches, which have been the topic of many research efforts in recent years. We derive a new, general concept, founded on an abstraction for *distributed overlapping grids*, for parallelizing algorithms working on arbitrary grid types. In particular, a novel characterization of the data dependencies for algorithms on unstructured grids is developed. Based on this characterization, we present a new algorithm for automatic generation of grid overlaps. This method is of optimal complexity, namely linear in the size of the overlap, and constitutes a progress of significant practical value.

Using the generic approach, the algorithms and data structures are implemented in a way that is independent of both dimension and the underlying sequential grid representation. These components add very low overhead in terms of memory consumption and runtime (linear in the size of overlap). They drastically reduce the effort for developing new parallel applications, or parallelizing existing ones.

A vast set of components — among them those described in this thesis — have been implemented using the C++ language. Their source code is available online at `http://www.math.tu-cottbus.de/~berti/gral`.

The body of this work is organized as follows. In chapter 2, we investigate in detail the problems and conditions of software development and reuse in scientific computing, and point out the fundamental role of grid data structures for numerical PDE solution. Subsequently, the generic programming approach is introduced as a possible solution to the problems identified.

Chapter 3 is devoted to a *domain analysis* for grids and grid algorithms. First, a detailed analysis of topological and combinatorial properties of grids is undertaken, followed by a study of several representative algorithms. The resulting list of *algorithm requirements* is used as guiding principle for an investigation of grid data structure variabilities and commonalities.

The next chapter contains a development of the abstract layer mentioned above, leading to a *micro-kernel* capturing the essential properties. We then show how a number of grid components can be implemented on top of this kernel, leading to a repository of generic, highly reusable components.

Chapter 5 tackles the parallelization problem. First, a set of abstract concepts for data-parallel grid computations is introduced, including a formalization of the notion of a *stencil* for algorithms on unstructured grids. This notion encapsulates essential aspects of algorithms, and allows to formulate distribution methods independently of concrete algorithms. Then, the generic approach is used to implement a family of algorithmic and data-oriented components, representing the abstract concepts in a faithful way, that is, without sacrificing generality. These components more or less completely encapsulate any technical detail related to data distribution, and are a convincing example for the power of the generic approach.

An investigation of the practical usability of the approach is done in chapter 6. We show how the generic paradigm can be used to construct complex applications (in this case a finite volume solver for hyperbolic equations of gas dynamics), leading to a sort of *program families*, and general implementations of finite-element and multigrid algorithms. By discussing the parallelization of an existing application for solving the incompressible Navier-Stokes equations, we demonstrate the seamless integration of the generic approach with more traditional ways of software design.

Furthermore, by assessing the performance of generic programs with a set of small kernels, we show that our methods lead to efficient code that can compete with low-level implementations in C or FORTRAN.

Finally, the results obtained are discussed, and promising further research directions are indicated. There is reason to expect that along the lines of the concepts developed in this thesis, substantial progress can be made in algorithm reuse, and therefore in scientific computing software development in general.

Acknowledgment

First of all, I want to thank my advisor, Prof. Dr. Georg Bader. He was a permanent source of motivation, inspiration, and sometimes the necessary impetus for pushing things forward. During all this time, he showed a great interest in my work. His readiness for discussing questions — occasionally, during a beer in the evening — was a constant and welcome factor. In addition, he also assured a comfortable and powerful working environment, indispensable for this type of work.

My coworkers — current and former — at the chair *Numerical Mathematics and Scientific Computing* at the Technical University of Cottbus have been of great help and backing for me during the past five years.

Klaus-Dieter Krannich has been an invaluable source of knowledge on about every aspect of parallel and scientific computing, and accompanied the various stages of this work with constructive criticism. He also undertook the effort of periodically porting generic software components to other platforms, a task which was (and is) no pure joy. Having to fight with my programming habits, real-existing C++ compilers and templates all at once has been a challenge he mastered excellently.

Klaus Schenk wrote the essential algorithmic portions of a generic finite-volume solver, which is a cornerstone for proving the practical relevance of my work. He also was a thorough teacher of the analytical and algorithmic aspects related to hyperbolic conservation laws.

Ralf Deiterding took part in many critical discussions on software design issues in general, and the generic approach in particular. He helped me a lot with various technical details.

Klaus-Jürgen Kreul wrote the Navier-Stokes solver which then served as an acid test for the parallelization components.

I thank (in no particular order) Klaus-Dieter Krannich, Klaus Schenk, Ralf Deiterding, Jörg Sawollek and Carsten Berti for proof-reading the whole or parts of the document, and their helpful comments.

Things like free software, bibliographic databases and the work of other researchers, all available over the Internet, made my own work much more comfortable. With some right, I may state that without such support, it would not have been possible to do it *this* way.

Finally, I want to warmly thank Wiebke that she took upon herself a weekend relationship — the term still is a euphemism — for such a long period of time. After all, it is quite a long distance from Cottbus to Essen!

Chapter 2

An Algorithm-centered Approach to Software Development in Scientific Computing

Software is hard.
Donald E. Knuth[1]

At the beginning of the computer age, years before the terms *scientific computing* and *computer science* were coined, these two disciplines were virtually identical: The very first applications of computers were in the domain of what today is called scientific computing, namely cryptography and numerical simulation of missiles and detonation waves.

Since those days, things have changed profoundly. Mainstream computer science has forked into a multitude of branches most of which have evolved far away from its roots. Scientific computing is now regarded as a discipline in its own right, related to, but distinct from, computer science, numerical analysis and mathematical physics and engineering, and is interdisciplinary by its very nature.

At the heart of scientific computing lie *algorithms*. Here tremendous progress has been made since the first computers where built, not only in scientific computing, but also in any other algorithm-centered field of computational sciences. Development of more powerful algorithms on the one hand, broadening the scope of what is accessible to a computational treatment, and the development of more and more powerful hardware on the other hand, driven by the increasing use of computers in science and business, have always been stimulating each other.

Algorithms, however, have to be implemented to unfold their power. And it is here where computer science and scientific computing meet again — or should do so. As applications in scientific computing grow in size and employ more involved algorithms

[1] On his experience with implementing TeX, see [Knu96], p. 154

requiring more complex data structures, the reuse of building blocks — algorithms and data structures — gets more and more a necessity. Reuse in general is a difficult problem; reuse of algorithms operating on nontrivial data structures is by no means a solved problem. In this chapter, we try to identify a possible solution to this challenge.

In section 2.1, we begin by explaining in some detail which role software development plays in scientific computing and which specific problems it faces. We then show how complexity has grown over the last decades, both of the involved algorithms and data structures, how the need to consider hardware aspects contributes to complexity, and give some evidence for the claim that this development will continue for the near future.

Software reuse is an aim nearly as old as software itself. In section 2.2 we introduce the notion of *reusable components*, discuss what quality measures and tradeoffs are connected to them, how reuse of components works in scientific computing practice, and when it fails. We will see that reuse is hard to achieve when algorithms interact in involved ways with complex data structures exhibiting *high variability.* This applies in particular to grids and algorithms working on grids, which are central to numerical PDE solution.

In section 2.3, we introduce the generic programming approach as a possible solution to the problem. We use a well-known example (STL) to introduce the key ideas, and point out how they may successfully be applied to grid-based algorithms, a topic fully explored in later chapters. Finally, we review some approaches to reusable generic components that are similar in spirit, and briefly point to recent developments in the field of software engineering and programming which could be of interest for scientific computing.

2.1 The Challenges of Software Development in Scientific Computing

2.1.1 Fundamental Issues

Software for numerical simulation of physical phenomena (e.g. solution of PDEs) is not mass-fabricated. It typically is written by the *domain experts* themselves, that is, by people having a sufficient understanding of the numerical analysis involved. In the extreme, the domain expert, software designer, programmer, tester and end-user is one and the same person; in any case, these roles tend to overlap much more than in mainstream computing.

This is especially true when the purpose of the software is to validate novel algorithms and methods, or to investigate physical phenomena not yet understood. Then only a handful of people have the necessary understanding of the numerical or physical background. The point is less virulent for scientific software for engineering applications, where the physics and numerical techniques are accessible to a broader circle.

On the other hand, programmers of scientific software — experts for numerical algorithms or mathematical modeling — often lack a formal education in software engineering or computer science. This is beginning to change, as more people with computer science background move into the field, but one can still observe a somewhat conservative attitude towards achievements of software engineering and modern programming paradigms. This is partially due to the lack of interest of mainstream computer science into issues like numerical errors and *practical* efficiency, discussed below. These difficulties are likely to persist in principle, as scientific computing is an inherently interdisciplinary field, and expertise in all disciplines involved is simply out of question.

A major problem for *testing* and *validation* of numerical software is, that mistakes are often not so obvious to detect. First of all, it may already take a lot of experience to decide if an outcome is erroneous or not, and if so, whether it is a programming bug, a badly chosen parameter or an ill-suited algorithm. Furthermore, errors may be 'creeping' into the solution, becoming visible only after hours of calculation, when the data leading to the error may not longer exist. This type of *continuous* or numerical error is not so common in other branches of computing.

Moreover, a typical program run involves huge amounts of data being created and processed. This makes it difficult to detect erroneous variable values especially if they have no physical interpretation and the problem cannot be arbitrarily downsized. Approaches to attack this problem include graphical [SR97] or comparative [MA97] methods. The large sets of data being generated and processed are a problem *per se*: The data that must be held in machine memory can be a limiting factor for the size of problems one can handle; the size of data that is produced may exceed by far the amount of data one can keep permanently.

Efficiency – both in terms of memory and CPU time – is a major quality factor of numerical simulations, where it is by far more important than in mainstream computing. Typically, the numerical problem size can be scaled up at will, resulting in better approximation of small-scale features. The limiting factors are computing resources and the effective use a program makes of them. So, from a practical point of view, a very efficient application provides more functionality, because larger or more interesting problems can be tackled. On the other hand, for a typical problem setting there often is a sort of minimal approximation size, below which a calculation is not very useful, e. g. the iterations may diverge. This means a *very* inefficient (1-2 orders of magnitude) program is only capable of solving 'toy-problems'.

The notion of efficiency used in scientific computing has to be contrasted with that of computational complexity as used in computer science. The latter addresses issues of asymptotic complexity, giving typically only a growth rate like $O(n)$, and, perhaps, a constant for the leading term, measuring the number of what is considered the central operation, like comparisons in sorting algorithms. Complexity analysis gives an important characterization of *algorithms*, yet is insufficient for the purposes of scientific computing, where low level *implementation* issues may account for actual differences in performance by an order of magnitude or more. This is one factor of software complexity in scientific computing, a topic we are now ready to discuss.

2.1.2 Increasing Complexity

In the following, we discuss four main sources of complexity: First, algorithmic complexity, by invention of better algorithms and data-structures, second, complexity due to the need of obtaining efficient implementations, third, complexity stemming from more involved physics, and finally, complexity caused by increasing requirements on 'soft factors' like user interaction.

2.1.2.1 Algorithmic Complexity

A trend that could be observed during the past two decades is that algorithms tend towards increasing structural complexity, and so do data structures. Here we focus on algorithms for numerical PDE solution, and discuss in an exemplary manner the following algorithmic developments: Increased geometric flexibility of meshes, adaptivity, and hierarchical methods.

In the beginning, computation took place on structured grids, and virtually the only data structures where fields of dimension 1,2, or 3. Cartesian grids still play an important role — due to their efficiency and simplicity — but are increasingly replaced or complemented by more sophisticated structures.

There is an obvious need to represent complicated computational domains with geometric meshes, which cannot be achieved with Cartesian grids. One route taken was the development of *unstructured grids*, with explicit storage of connectivity information. These grids exhibit greatly improved *geometric flexibility*, and good localization of geometric resolution. Another path to achieve geometric flexibility led over *body-fitted* [TWM85] to *multi-block* and *overset* [Pet97] grids, also known as *Chimera* grids. This gave rise to the new research field of *mesh generation*. Overviews are found in [Fil96, Tho99]. The problem of automatic mesh generation still lacks a fully satisfactory solution, especially in 3D.

A major algorithmic milestone was the advent of *adaptivity* in numerical PDE solution [BR78]. An adaptive algorithm measures the quality of the calculated approximation by some means, and detects regions of insufficient approximation. One method to achieve the needed approximation consists in *dynamically refining* the underlying grid in these regions (*h-version*). For unstructured grids, this has led to a number of refinement algorithms and data structures supporting dynamic grid changes [Ban98, Bas94]. In the context of structured grids, the technology of (structured) *adaptive mesh refinement* (AMR or more precisely SAMR) [BO84] emerged. Here hierarchies of local Cartesian grids are built, which allows the underlying numerical kernels to be reused with only minor changes.

A second method, used especially in FEM, enriches the underlying function spaces by higher-order polynomials (*p-version*). These two methods can also be composed, leading to the so-called *h-p-version*.

Another milestone is the invention of *hierarchical methods*, especially the multigrid algorithm for the solution of systems of linear equations arising from discretizations of PDEs [Hac85]. By solving these systems recursively on increasingly coarse scales of approximation, an *optimal complexity* of $O(n)$ can be achieved, n being the number of unknowns. This is true provided the system is *sparse*, that is, the number of non-zero coefficients is $O(n)$.

Using hierarchical methods together with adaptivity further complicates the necessary data structures.

2.1.2.2 Efficiency Driven Complexity

Another major source of complexity is the need to exploit hardware characteristics for high efficiency. The techniques used for this aim can differ considerably, depending on the concrete architecture. Especially vector computers and hierarchical memory machines require a somewhat opposed handling. However, as vector computers do not play a such an important role for irregularly structured problems, the main problem in our context consists of the non-uniform access cost of machine memory found in hierarchical memory architectures. The extreme end of the scale is marked by distributed-memory machines, but also common desktop computers have at least one cache level.

The key to high performance is *data locality*: As already mentioned, scientific computing means processing large amounts of data. Therefore, the time necessary to fetch data from the place where it is stored to the place where it is used (registers) is crucial for the overall performance. The fact that todays typical computers can process data much faster than fetch it has led to hardware caches, to overcome this so-called *von Neumann bottleneck*. This results in the before-mentioned non-uniform access property. Frequent access to slower memory (cache misses, remote memory) leads to substantial performance losses, which can be reduced by applying specific programming techniques, see [Kue98, PNDN97].

Well-known techniques for enhancing data-locality are clustering of operations (e. g. loop tiling) in order to do as much work as possible at once with the data in cache, and partitioning of data into smaller parts that are processed separately. The latter approach is combined with data duplication, it is also suited for distributed memory machines.

While data locality is a problem already for single-processor, single-memory machines, it dominates in distributed computing, where performance drastically deteriorates if data-locality is not taken into account. Hardware complexity still continues to increase; today *multi-tier* machines are built which are clusters of shared-memory machines, resulting in several layers of memory hierarchies, see for example the U. S. DoE 30 TFlops Project [Wol98]. Adapting high-performance software quickly to exploit the power of these architectures will be a serious challenge.

An emerging technology is *Network Scientific Computing (NSC)*, which can actually be seen as introducing yet a new hardware layer: By linking heterogeneous computing facilities via a high-speed network, very large clusters are in principle available for scientific computations. Also, network computing opens up a new view of how software is composed: Instead of building monolithic blocks, an application may use distributed services which are bundled only at program runtime [BGMS98].

2.1.2.3 Physics Complexity

The more successful numerical simulation is, the higher the expectations and the demand for handling yet more difficult physical problems. A representative number of some most important problems (so-called *Grand Challenges*) are listed by the US Federal High Performance Computing and Communications (HPCC) program [Nat95], defining them as

> fundamental problems in science and engineering with broad economic and scientific impact, whose solutions require the application of high-performance computing.

Among these are coupled (structural and fluid mechanics) field problems [GC-99a], GAFD (geophysical and astrophysical fluid dynamics) turbulence [HPC99], oil reservoir modeling [GC-99b], ground-water transport and remediation [PIC99], and global climate modeling [MSD93b, MSD93a].

Many of the Grand Challenges require in fact simulation of *multi-physics models*. In the beginning, numerical simulations have focussed on a single physical model, e. g. structural mechanics or fluid dynamics. Now the coupling between those models comes into reach, for example modeling the flight of a complete airplane, including interactions between fluid flow and wing deformation. Besides the mathematical problems related to find sound discretizations [HW95], the problems of coupling different types of solvers, multiple grids [PHS98], and varying time-scales have to be tackled.

2.1.2.4 User Interaction Complexity

Many numerical applications traditionally ran (and run) in batch mode: Once started, they calculate for hours, days or even weeks, and at the end, the data is *post-processed* by a visualization program. One step further is on-line visualization by *co-processing* simulation data. This is interactive only in the sense that the graphical representation of the data can be controlled, but not the simulation itself. Finally, the user can be allowed to interactively *steer* the computation: The parameters of the algorithms involved can be changed during the run, allowing to experiment with different alternatives. This is of particular interest during the validation phase of a novel algorithm or if some parameters have to be adjusted [YFD97, PWJ97].

Numerical applications distributed on many computers in general produce much more data than fits into a single memory. Visualization on a single node quickly becomes a bottleneck, if not impossible. Therefore, strategies for data reduction have to be developed and implemented, which also work in a distributed fashion and very near to the place where the data is produced. This place might be the simulation code itself. Achieving this locality while at the same time maintaining a high degree of interactivity for the user is a nontrivial task [Cro96]. It also tends to closely intertwine numerical and visualization tasks.

2.1.3 The Central Role of Grids for Numerical PDE Solution

If we reconsider the algorithmic factors of software complexity in PDE solvers we have discussed, it turns out that all of these revolve around grid data structures and algorithms operating on them. There is certainly not much question about the fact that grids are *the* central data structure in this field — the large majority of algorithms in this context directly or indirectly operate on grids.

First of all, the numerical discretizations directly access the grid, whether it is finite differences (in which case one makes very special assumptions on the grid), finite volumes (see section 6.2) or finite elements (see 6.3).

A possible next step is the solution of a linear system assembled by such a discretization, for example by a finite element method. While simple iterative solvers for such system do not require any grid access, more advanced methods do, like multigrid or preconditioners that exploit problem physics.

Another source of grid-related algorithms is visualization. Here simulation data, given in a discrete fashion on the grid, is used to produce geometric and finally graphic data in a multitude of ways [PvW93].

Parallelization of PDE solvers means in essence distribution of the grid and thereby of the work in a data-parallel fashion. This requires both additional data structures managing the distribution information, and algorithms for setting up and maintaining these data structures. Both types of components closely interact with the basic sequential grid. This topic will be explored in depth in chapter 5.

The dominating role played by grids in numerical PDE solution means that controlling the complexity of the corresponding data structures and algorithms is a key to mastering software construction in this field. But use of grids is not limited to this domain: If we take a look over the narrow borders of numerical simulation, it becomes evident that grids also play an important role in other algorithm-dominated fields, like Computer Graphics [FvFH90], Computational Geometry [Ede87], Computational Topology [DEG98], and Geometric Modeling [Mor97].

Therefore, it can be expected that advances in the field of grid-related components in PDE solution also benefit other fields. Perhaps most important, it can ease the cooperation between software from different domains. An example is the increasing use of graph-algorithms in simulation programs, such as downstream ordering [Bey98]. Using such algorithms *directly* from a library for combinatorial algorithms is impossible in virtually all cases today.

The connection to Computer Graphics is quite clear from the characterization of scientific visualization above. Methods from Computational Geometry, like Delaunay triangulations, are routinely used in unstructured grid generation [She96]. This in turn relies on a representation of the computational domain coming from Geometric Modeling component, like a CAD program. In fact, the passing of geometric information from CAD to grid generators is a problem still lacking a satisfactory solution, and in practice is much too often associated with information loss.

2.2 Components for Scientific Computing

2.2.1 What Are Components?

As a result of the situation just described, technical, combinatorial and geometrical aspects and algorithms are becoming increasingly important. Numerical analysts are not (and cannot be) experts in all of these fields — one-man-efforts are less and less sufficient for creating advanced scientific computations. The same is true in other fields where numerical algorithms can be used with profit, for example in determining numerically the intersection of surfaces in CAGD.

Instead, teams have to work together, and work from third parties has to be leveraged in form of *off-the-shelf* building blocks or components for data-structures, algorithms and high-performance strategies.

Taking profit from the work of others has, of course, always been done in scientific computing— the reuse of *algorithms*, that is, *ideas*, is well-established. But this does not suffice. Given the increasing number, complexity and implementation difficulties of algorithms, also these implementations must be — and are — reused.

The term 'component' is used in quite different ways in the literature. COULANGE [Cou98] defines it as (p. 51)

> an entity, which can be used, possibly with modifications, in a new software development.

He further points out that not only code (implementations) can be reused, but also design and specifications. We contrast this definition with that given by SZYPERSKI in the preface of [Szy98]:

> Software components are *binary* units of independent production, acquisition, and deployment that interact to form a functioning system.

In this work, we use the term *component* rather loosely in the sense of Coulange, not in the more specific sense of Szyperski. In general, we have more fine-grained entities in mind which are composed at compile time, rather than binary components with "very late binding". However, we will use the term 'component' also for whole libraries or packages, if they are reused as a whole.

It is a well-known fact that software reuse is not easy, see the general literature devoted to this topic [Cou98, Szy98, Sam97, Kar95]. For scientific computing, reuse is employed rather successfully in some areas, but is limited in others. In the following sections, we will try to shed light on some reasons for this fact, concentrating on the reuse of *algorithm implementations*. The case of *grid-based algorithms* will act as point of reference.

In section 2.2.2, some criteria for judging the quality of scientific components are examined, and the role of *generality* is highlighted as a key to reuse.

We then review the major types of existing components, and analyze some reuse scenarios (section 2.2.3). As a result, the strong dependency of algorithms on specific data structures will be identified as a major obstacle to reuse in section 2.2.4. The higher the *variance* of the underlying data structures is, the higher the obstacles to reuse — an issue highly relevant for grid data structures and algorithms operating on grids.

This thesis concentrates on reuse of algorithms — small, focussed components. Yet, this small- or medium-scale reuse is not the only possible way to exploit existing work. On an application level, also *design* and *specification* of entire simulation environments are resources that can be reused. Research in this area — for the context of PDE solution — is just emerging recently [Åhl99, Nor96, LMR+97], and is in a sense complementary to this thesis.

2.2.2 Quality Criteria for Scientific Computing Components

In the following, we give an extensive list of properties measuring quality of components, geared towards the needs of scientific computing. We first focus on properties relevant to a component *user*, and then list properties that are a particular concern for a component *developer*.

One can divide these criteria coarsely into four groups: *quality-of-service, functionality, relationship to other components*, and *support*.

Beginning with the **quality-of-service** properties, which measure how well tasks are solved, we can ask for the following:

- **Correctness/Accuracy:** Does the component deliver the correct output? Which is the testing strategy? Is it numerically stable, is there an error control, and does it give reasonable approximations? Is there a loss of precision, e. g. when calculating internally which lower precision, or using ill-conditioned order of evaluation?

- **Robustness:** Measures how stable a component works for *valid* input, and how consistent the error treatment on detection of *invalid* input is. Specifically, the more checks can be performed at compile-time, and the more usage-errors are detected directly at library *entry points*, the more robust the library is. For example, a linear algebra library offering factorization algorithms for different matrix representations should make sure, that the format of the matrices passed by the user matches the one expected by the factorization algorithm.

  For numerical algorithms it is sometimes not easy to decide whether a given input is valid or not, because floating-point implementations generally fail in the *vicinity* of an invalid value, such as a singular matrix.

- **Efficiency:** This means both memory efficiency (temporary work space, copying of input data) and runtime performance (internal use of efficient algorithms, optimization of implementation). It can further be distinguished between theoretical performance (complexity) and actual performance, depending on the translation into machine code and the underlying hardware. This latter point is well-known in scientific computing, but appears to be neglected to some extent by work taking a 'pure' algorithm-theoretic perspective. An important subtopic is **Parallel Scalability:** A component for distributed computing is scalable if performance does not degrade substantially if both problem size and resources (number of processors) are scaled up.

The next group are **functionality** issues — which tasks can be handled at all?

- **Modularity and Granularity:** If a component offers bundled, high-level functionality, can parts of it be reused? For example, does a package for the solution of elliptic PDEs allow to use iterative linear equation solvers only? Are complex operations broken into meaningful partial operations? For example, a linear algebra package could provide methods to scale matrices to achieve better condition numbers.

- **Flexibility:** Means the ability of the user to control parameters that influence internal execution, for example termination criteria in iterative processes, debug level, or the exchange of algorithmic blocks, especially at runtime.

- **Generality:** Relates to the scope of *concrete* problems that can be handled: A linear algebra package that allows complex-valued matrices (or even arbitrary-valued) is more general than one allowing only single-precision entries. The degree of generality determines how often usage is possible at all: If the component can solve only a 'neighboring' problem, the user must find some workaround (problem transformation), or else he cannot use it. The greater the variability in the problem domain is (different data structures, different algorithmic choices), the more difficult it is to produce a sufficiently general component. The number of procedures (about 1000) in the Lapack library speaks for itself.

  At this place, we would like to stress that by generality we do *not* mean *breadth*, i. e. the spectrum of *different* problems that are addressed, see also the *focus* property. Instead, one and the same *high-level* mathematical problem can give rise to many different *low-level* computational problems, due to variability in data representation and implementation decisions that are irrelevant to the abstract mathematical problem but highly relevant to its concrete computer solution.

  It will turn out that lack of generality is a major problem of most existing components; the question how to provide components with sufficiently large (ideally maximal) generality is the driving force behind this work.

- **Extensibility:** Measures how easy it is to extend the functionality of the component, while keeping the user interface as intact as possible. A more exact term is **scalability** [BSST93], which is a measure how much effort has to be taken to add a new feature enhancing *generality*, e. g. adding support for complex numbers to a linear algebra library. Scalability is hard to achieve with conventional library implementation styles, which cannot deal with the *combinatorial explosion* arising from satisfying all *potential* demands.

The first two groups (quality-of-service and functionality) are somewhat *intrinsic* properties of a component; they can be answered by looking at a given component in isolation. It is also important to consider **relationships between components**.

- **Interoperability:** Takes into account how well a component can cooperate with other components. One of the major obstacles for interoperability is prescription of a fixed set of data structures in interfaces, especially in conventional procedural languages. Obstacles for cooperation can also be name conflicts (see also under *focus*), or implicit or too demanding assumptions about the context, for example the requirement that all processes are involved in a component's internal collective operations (in the case of parallel computation), see [Gro98]. Another example is a grid generator *requiring* being run interactively through a GUI.

- **Focus and Compactness:** Does the component fulfill a single, well-defined task, or is it an unordered bunch of unrelated functionality? Does it 're-invent the wheel' by implementing service functionality that could be taken from other packages, for example the unavoidable QUICKSORT and container implementations? Does the size of the package correspond to the offered functionality? A component which is not sufficiently focussed tends to be a lot bigger than necessary: Small is beautiful in this case. Also, ad-hoc written service sub-components (like linear lists) tend to be of somewhat inferior quality than their well-tested, optimized counterparts from a dedicated package. Finally, the probability of conflicts raises if many elementary data types are implemented within the component.

- **Dependencies:** Does the component depend on other, 3rd party components or the availability of tools like a particular compiler? The more dependencies there are, the less likely it is that a component is reused, because installation can entail a lot of other components to be installed first. A large number of dependencies often means portability limitations. The more specialized the dependencies are (a particular version of particular library, or a non-standard language extension requiring a non-standard compiler), the more problems they cause. Dependencies have to be set in relation to the level of functionality offered, a high-level component (e. g. parallel non-linear solver) is more likely to exhibit dependencies than a low-level component (e. g. a message-passing library).

A last aspect is somewhat outside the 'pure' software, namely the **support** properties, which depend on temporal evolution of the component and largely on the people behind it.

- **Stability:** Components are no static entities, but evolve over time. Stability concerns the frequency of changes relevant to a user, such as interface syntax, pre- and postconditions, resources consumption. Usually, it is acceptable if pre-conditions are weakened, post-conditions are strengthened, memory and time consumption decreases and interfaces get more general, leaving old user code valid.

- **Maintenance:** Nobody and no component is perfect. Maintenance means the frequency by with questions are answered, bugs are detected, documented and fixed, new algorithms are implemented, and implementations are optimized.

- **Ease-of-use:** Measures how difficult it is to effectively get the desired functionality out of the component, e. g. are there complicated calling sequences, are there lots of obscure parameters to pass, or has one to provide extra work memory to the components? This has to be set in relation to the complexity of the task the component solves: No one would use a library demanding 15 lines of code in order to call a procedure that can be implemented in 10 lines.

For a *developer*, in addition the following criteria are important:

- **Maintainability:** Is the ease with which to correct known bugs, enhance or extend the library, without breaking user code.

- **Portability:** Is the relative lack of dependency on environment parameters, such as hardware, special libraries, particular compilers, system versions and others.

- **Coding efficiency:** measures how much work is necessary to achieve a given functionality. This is a very important topic, as too low coding efficiency ultimately leads to very incomplete or poorly implemented components. It is intimately related to the scalability property, to the *internal* reuse of library components, and the use of third-party components.

Tradeoffs between criteria

Evidently, these quality measures sometimes contradict each other. Especially, efficiency seems to be in opposition to nearly every other property. Due to the great importance of efficiency in the numerical computation domain, trade-off decisions are often made in favor of efficiency.

- Efficiency $\leftrightarrow$ Robustness: Serious checking of argument validity takes additional time or memory, sometimes more than the algorithm itself. A possible solution is to offer various degrees of checking, and let the user decide how much is needed.

- Efficiency $\leftrightarrow$ Generality: The less efficiency is taken into account, the easier it is to develop general implementations. This is because for efficient programs, one typically has to take into account more concrete details about the problem. Overgeneralization tends to result in very poor performance, a gross example being sparse matrices treated like dense matrices in a matrix multiplication routine. Finding the right abstractions that allow both is fine art; it is one of the central problems we try to solve in this thesis.

- Efficiency ↔ Portability: In order to obtain high performance, one often has to commit to hardware architecture and details (e. g. cache sizes) or use special compiler switches. These aspects are not always easy to encapsulate. One may try to factor out low-level basic operations which have to be ported (optimized) for each platform, and shield the rest from these issues. An example is the `BLAS` (see p. 25), which encapsulates basic vector and matrix operations. In [WD97], the authors describe how to automatically generate optimal BLAS implementations based on runtime measurements.

- Efficiency ↔ Coding Efficiency: Highly optimized software is typically harder to make work correctly, because the connection of program statements to a high-level mathematical formulation gets obscured. Techniques like manual loop unrolling, tiling and manual optimization of complicated algebraic expressions open up a bunch of new sources for errors. Hand-optimization may therefore drastically increase the effort necessary to create correct software.

- Efficiency ↔ Maintainability: The more code has to be touched in order to correct or change a certain behavior, the more laborious and error-prone it is. Hand-tuned code typically replaces understandable, higher-level code with more obscure, lower-level and consequently more explicit (i. e. longer) pieces of code. If changes have to be made, one must detect which part of the code corresponds to the behavior under consideration, thus mentally undoing the optimization transformations made before.

  Automatic transformation of high-level code into *efficient* (relative to hand-tuned) low-level code could provide a solution to these problems, and is an active area of research [BHW97], [DQ99],[Rob96], [DHH99].

  Despite some successes, the solutions found so far seem partial only, and most require language extensions or a pre-compiler. Techniques that use built-in language features (for C++) are *expression templates* [Vel95a] and *template meta-programs* [Vel95b].

- Dependencies ↔ Compactness: Dependencies are necessarily connected to each form of reuse; the main question is the degree of *arbitrariness* in the dependency. For example, a non-linear solver generally requires a linear solver component, but it makes a big difference whether it requires *one* particular package, or if, on the other extreme, every linear solver package can be used, possibly with adaptation.

  Increased interoperability tends to alleviate this problem; however, full substitutability of equivalent components is a largely unreached ideal. The generic programming approach presented later on takes a step towards this goal.

- Ease-of-use ↔ Generality: The more general a component, the more 'loose ends' exist that are tied up only at time of actual usage. This can be a substantial burden to a user who has to understand much more parameters than he is actually willing to vary.

Offering several layers of access is one possible way to preserve full generality while keeping interfaces simple. For example, the PETSc toolbox [BGMS99] defines four levels of access: basic, intermediate, advanced and developer. Language features like default arguments can help in creating these layers.

Most of the criteria discussed above can be matched by techniques that are already well-established and used in scientific computing— which does not mean that every component performs in a satisfying manner with regard to every criterion, nor that these aims are easy to achieve.

Correctness is essentially a function of correct implementation of adequate algorithms. *Robustness* requires in general some type-checking of parameters. This is easier to achieve in a strongly-typed language with support for data abstraction, as is offered by many object-oriented languages. *Efficiency* (in isolation) has traditionally been achieved by implementing on level close to the physical data layout. *Modularity* is a question of organization, as are *focus* and *dependencies*; some languages give better support then others for modular development. Object-oriented techniques are particularly suited for achieving *flexibility*, where object state and dynamic binding give decisive advantages over procedural approaches.

The remaining topics *generality*, and, somewhat related, *scalability* and some aspects of *interoperability*, seem to be somewhat harder to achieve, in particular with the additional requirement of acceptable efficiency in mind. In the following two sections, we will argue that generality is a real challenge in fields with *high variability of data representations*, that a quite different approach is needed to cope with this problem, and that such an approach is still missing for the field of (unstructured) grid computations.

2.2.3 Review of Components for Scientific Computing

The purpose of this section is to give the reader a better feeling for the nature of concrete software components in the field. It is by no means meant to give an exhaustive overview. We pay special attention to grid-related components, and hope having gathered a somewhat representative collection.

The comparatively small number of reusable components exclusively dedicated to grids, — mostly concentrated in the fields of grid generation and partitioning — is in sharp contrast to the central role of grids, and may be seen as a sign of some fundamental difficulties in the creation of such components. This topic is further investigated in section 2.2.4.

Existing components for scientific computing can very coarsely be classified into several types:

- stateless *subroutine libraries* and object-oriented/ object-based *toolboxes*
- object-oriented *application frameworks* and complete applications

Subroutine libraries are the oldest component types, and still account for the majority of available numerical software. *Toolboxes* are similar to subroutine libraries, but contain also data structures. These two groups of components focus on groups of algorithms and/or data structures.

Application frameworks model the architecture of an application, and cover a rather broad spectrum of sub-domains, like those needed for numerical PDE solution. Complete applications solve a (class of) concrete problems.

The first group of components is typically more general purpose than components of the second group, which are normally more specialized to a specific class of applications. Also, the first group tends to support more small-scale or selective reuse, whereas in the second group, reuse often means using a whole monolithic block, replacing just a small part of it.

Obviously, the frontiers between these classes are not sharp, and mixed forms are commonly found.

Subroutine libraries The classical way of creating reusable software components has been to group sets of related routines together into libraries. Today, a large body of code – mostly written in FORTRAN– exists for a vast range of problems, including *dense* linear algebra (BLAS [LHKK79] , LAPACK [A+95]), *sparse* linear algebra (ITPACK [KRYG82]), numerical quadrature (QUADPACK [PdKÜK83]), ordinary differential equations (ODEPACK [Hin83]), and partial differential equations (ELLPACK [RB84], CLAWPACK [LeV94]).

These libraries typically offer a set of procedures, which users can call (or a well-defined sequence of them) to access the desired functionality, passing their own data-structures as arguments.

ELLPACK consists of a *problem-solving environment* (PSE) on top of a conventional subroutine library.

The BLAS is a special case in that it is a low-level *support library* that provides implementations for basic algebraic operations on vectors and matrices. The underlying idea is that the efficiency of these operations depends crucially on how well the implementation takes into account hardware characteristic like cache sizes. The BLAS layer can be specialized for a concrete hardware and shield higher-level libraries like LAPACK from these details. Therefore, one can say the BLAS encapsulates the hardware-dependent *aspect* of efficient vector arithmetic.

Among components dedicated to grid algorithms, we find grid generators like GEOMPACK [Joe91], TRIANGLE [She99] or NETGEN [Sch99], grid partitioners like METIS [Kar99] or JOSTLE [Wal99], and visualization packages like VISUAL3 [Hai99]. Virtually all of these require the user to convert his data to the specific format expected by the implementation, and/or back, if output is produced. It is needless to say that this format generally differs from case to case.

Toolboxes In contrast to pure subroutine libraries, a toolbox also provides *domain-specific data structures*, such as sparse matrices or computational meshes. Consequently, toolboxes are often written in an object-based style in C or and object-oriented style in C++, which offer better support for data abstraction than FORTRAN.

The PETSc library [BGMS99], written in ANSI C, consists of algorithms for the solution of linear and non-linear systems, and data-structures for (distributed) vectors and sparse matrices. It runs on sequential and distributed-memory machines, and has support for user-defined data-distribution. Most algorithm implementations accept user-defined data structures, together with routines working on them (e. g. Matrix-Vector multiplication).

AGM^{3D} [Bey99] (C) is a data structure for three-dimensional adaptive multigrid hierarchies, which implements algorithms for refining and coarsening such hierarchies, see [Bey98] for details. A user of this toolbox will construct his application around the given multigrid data structure.

GRAPE [GRA99] (C) is an object-based toolbox offering 2D/3D triangulation data structures with support for FEM calculation, as well as extensive visualization algorithms operating on these structures. Besides using the GRAPE data structures, it is possible to pass a procedural interface to a user-defined grid type and the simulation data [RSS96] which is then used by visualization algorithms.

POOMA [K$^+$99] (C++) offers array-like data-structures with distribution support, but no numerical algorithms.

LEDA [MNU99] (C++) implements advanced container types (such as balanced trees) and combinatorial algorithms, especially graph algorithms. Containers are parameterized by element type and implementation strategies. Algorithms, however, in general work only with the data structures of the library itself. Please note also the remark about the Graph Iterator Extension 44.

Packages and Frameworks for PDE solution Frameworks define a comprehensive model of a whole application domain. In general, they provide an overall structure for application programs. A framework user can than fill in or replace single components of the framework. In a *pure* framework, there is no support for the actual implementation of the concrete algorithmic and data-type components. Thus, frameworks can be considered as being complementary to toolboxes.

Some early packages for PDE solution include PLTMG (Piecewise Linear Triangle Multigrid Package, [Ban98], Fortran) and MGHAT (Multigrid Galerkin Hierarchical Adaptive Triangles [Mit99], Fortran), These are hardly modular programs, reusable only as a whole.

Most modern approaches use object-oriented or object-based paradigms. Diffpack [BL97] (C++) is a framework for PDE solution by the Finite Differences (FD) or Finite Elements (FE) methods. The same application domain is covered by Kaskade [REL99, REL97] (C++). Both frameworks provide abstractions for finite elements, linear operators and iterative solvers. The ug (Unstructured Grids, [B$^+$99], C) package provides support for FE and FV methods, as well as for parallelization.

Both ug and Kaskade are centered around unstructured, adaptive hierarchical grid data structures.

DAGH (Distributed Adaptive Grid Hierarchies, [PB99]) (C++) is a framework supporting algorithms like SAMR methods [BO84] on Cartesian domains. Overture

[BQH99] (C++) also supports structured grids only, but allows for curvilinear geometries and overset Chimera-type grids. It includes mesh generation, discretization and visualization sub-packages. Both packages support distributed memory parallelism.

While it is not too difficult to *extend* these frameworks with own components (which is what they are designed for), reusing *parts* of it, e. g. single algorithms, in a *different* context is not so easy. Implementations of algorithms are rarely self-contained, but depend on the environment provided by the framework. So combining the best from two or more frameworks normally is not possible.

2.2.4 Shortcomings of Scientific Computing Components

From the overview of the vast range of (grid-based) algorithms usable in scientific computing, provided in section 2.1.3, it is rather evident that a natural *unit of reuse* is a single algorithmic component — or maybe a small group of collaborating algorithms, sharing some data structures. The ideal way would be to take these components from an algorithm repository: If one needs, say, a method for grid smoothing, one takes an appropriate component from the repository.

However, actual components do not allow this so easily. For subroutine-libraries and toolboxes, which a-priori have an adequate granularity, the problem arises that data-structures do not match, and conversions are required — the drawbacks of this approach are discussed below (section 2.2.4.5).

Even more difficult the problem is for frameworks and complete application packages, where reuse of single algorithms is not intended: The usual approach is to reuse the infrastructure of the package to implement additional algorithms.

The key problem here is that the components discussed are not able to deal with the variability of the context in which they are used — in the terms introduced in an earlier section, they do not provide sufficient *generality*. Those which attempt to do so, like LAPACK, run into a combinatorial explosion of library size.

The major cause for this limitation can be expressed by stating that *decisions tend to be taken to early*. Many decisions are best based on information only available when the component is *actually used*: Which data structures to use, whether the application runs in a sequential or parallel context, whether utmost performance or enhanced debug support is desired, and so on. Yet many of these points are already fixed in the library, often leading to situations where the services offered are not quite satisfactory.

We will now explore the problem in some more detail, using examples from dense linear algebra (sec. 2.2.4.1), sparse linear algebra (sec. 2.2.4.2), and mesh computations (sec. 2.2.4.3). This sequence of examples can be regarded as an escalation in *data structure variance*, which is the core problem here. Two possible strategies for dealing with data structure variance, namely *standards* for data structures (section 2.2.4.4) and data structure *conversion* (section 2.2.4.5) are discussed, and it is shown that they cannot offer a general solution.

2.2.4.1 Reuse Case Study: Dense Linear Algebra

Say we want to do a *LU*-factorization of a dense matrix using a high-precision floating-point data type like [Bri99], for example for doing rounding error analysis. A high-quality library offering *LU* algorithms is LAPACK. However, we have no chance to do it with LAPACK, because it only supports the four basic floating point types (single/double precision reals and complex).

Would it help if LAPACK routines were parameterized over the scalar type? Such a parameterization — although not supported by FORTRAN— could easily be done by macros, and in fact this technique is used (to the author's knowledge) by the developers to generate versions for the basic scalar types. However, this would not help us much, as FORTRAN cannot support the basic arithmetic operations for the new high precision type. Therefore, in order to extend LAPACK to support this scalar type, one would have to rewrite each routine, inserting the arithmetic operations by hand: The library is not *scalable*.

Now let us take a package implemented in a language allowing operator overloading and generics, like C++, and a linear algebra package offering a *LU* decomposition parameterized by the scalar type. (This could in fact be easily obtained by just mapping the FORTRAN code to C++ and parameterizing the array arguments.) The desired functionality is obtained by instantiating the *LU* decomposition with this new type, provided it has the arithmetic operations properly overloaded.

Where is the difference? Certainly, C++ has the template facility, but here the same effect would have been possible with macros. The main reason is, however, that we were able to *hide the difference* between the new scalar type and the built-in types by providing the operations required by the *LU* algorithm.

However, we must confess that it worked *by accident* only: The generic *LU* implementation and the new scalar type fit well together. But what if the developer of the high-precision type would have used `add(x,y)` for addition and `mul(x,y)` for multiplication? It was only the fact that there is a *common language* relating to arithmetic operations, existing since several hundred years, that saved us here — the operations are called `+`, `-` , `*`, `/` and `=` , and that's that. Yet, already for exponentiation, there would be a problem.

A further remark must be made concerning the somewhat simplistic parameterization approach: This would result in a correct program, but perhaps not achieving the highest efficiency that is possible. The reason for that is, that LAPACK internally uses the BLAS routines for elementary operations. These in turn are optimized to cache sizes, using e. g. blocking. As the new types demand more memory, the block-size of the BLAS routines do not work well any more. Thus, not only the meaning of algebraic operations, but also the *size* of the new type is an important parameter.

2.2.4.2 Reuse Case Study: Sparse System Solver

Assume we have one component ($\mathcal{A}$) assembling a linear system (a sparse matrix A and a right-hand side vector b) from some FEM discretization, and another component ($\mathcal{S}$)

solving a linear system. What are the chances to use them together?

Well, it depends. In a typical case, $\mathcal{A}$ will use one data structure $D_{\mathcal{A}}$ for representing sparse matrices, and $\mathcal{S}$ will use a different one, $D_{\mathcal{S}}$. So, our best bet will be to copy the matrix from its $D_{\mathcal{A}}$ to its $D_{\mathcal{S}}$ representation, requiring us to know details of *both*. The same reasoning applies to vector types, although differences in data representation and technical difficulties typically are somewhat lower.

If we are luckier, however, $\mathcal{S}$ will allow to pass user-defined data types (matrices and vectors), along with the operations needed by the underlying algorithm. This is the approach chosen in, for example, PETSC (termed *data-structure neutral* there). In an object-oriented implementation, the algorithm would be defined on an abstract base class for sparse matrices, and one would use the bridge pattern [GHJV94] to insert the own matrix class into the class hierarchy underneath that abstract base.

The object-based PETSC gives the user the possibility to register a new matrix type by providing pointers to the used matrix/vector functions, and to create vectors from raw C arrays. In fact, the PETSC type `Mat` is just a placeholder for arbitrary matrix implementations (including non-stored matrices).

The PETSC components impose very light requirements on passed arguments: They do not have to be of the particular types accidentally chosen by the component, but only the *really used aspects* must match. In this case, it might suffice to provide routines for matrix-vector multiplication and vector addition.

A plain iterative scheme has very modest requirements on matrix and vector data types, as was indicated above. Normally, however, iterative methods require *preconditioning* to enhance convergence. Creating a preconditioner from a given linear operator greatly depends on the concrete problem to solve, and consequently, a vast number of concrete algorithms exist. To cite [Saa96], page 265:

> Finding a good preconditioner to solve a given sparse linear system is often viewed as a combination of art and science. Theoretical results are rare and some methods work surprisingly well, often despite expectations.

So we use a third component, $\mathcal{P}$, using a data structure $D_{\mathcal{P}}$, for calculating a preconditioner from our Matrix A in format $D_{\mathcal{A}}$.

Creating preconditioners typically requires more knowledge of the underlying matrix data structure, ranging from access to diagonal elements for a simple Jacobi preconditioning to a detailed knowledge of the non-zero pattern for incomplete factorization (ILU), which can also include a reuse of the structure-describing part of the matrix [Saa96]. Handling individual entries is very fine-granular functionality, and the function-pointer approach used by PETSc does not work well any more.

Therefore, if one chooses, say, ILU preconditioning, the matrix passed is supposed to perform this itself. The ILU algorithm implementations present in the PETSc toolbox cannot be reused in the case of a user supplied data structure. This is not very surprising, in view of the difficulties discussed above.

Finally, it is often advantageous to permute the order of matrix elements to achieve better convergence, less fill-in, or better cache reuse. This essentially is a graph-theoretic problem, so we would like to pass the matrix to a package $\mathcal{G}$ applying a reordering algorithm. This package expects a graph data-structure $D_{\mathcal{G}}$ as input, and outputs some permutation of the graph-nodes in form of an array.

The algorithm may be something general as the CUTHILL-MCKEE method [Saa96] (see also page 62), but it may also depend on problem-specific data, see e. g. [Bey98].

In sum, there is the following procedure:

1. Assemble the matrix $A = A_{\mathcal{A}}$ in data structure $D_{\mathcal{A}}$, with component $\mathcal{A}$
2. Convert $A_{\mathcal{A}}$ from $D_{\mathcal{A}}$ to $D_{\mathcal{G}}$, yielding $A_{\mathcal{G}}$
3. Calculate a matrix ordering from $A_{\mathcal{G}}$ with component $\mathcal{G}$
4. Apply the ordering to $A_{\mathcal{A}}$
5. Convert $A_{\mathcal{A}}$ from $D_{\mathcal{A}}$ to $D_{\mathcal{P}}$, yielding $A_{\mathcal{P}}$
6. Construct a preconditioner $P_{\mathcal{P}}$ from $A_{\mathcal{P}}$, using component $\mathcal{P}$
7. Convert the matrix $A_{\mathcal{A}}$ and the preconditioner $P_{\mathcal{P}}$ to the values $A_{\mathcal{S}}$ and $P_{\mathcal{S}}$ of types $D_{\mathcal{S}}$ and $DP_{\mathcal{S}}$ expected by the solver component $\mathcal{S}$. [2]
8. Solve the linear system, using $\mathcal{S}$.

It involves copying or conversion of the initial data structure $D_{\mathcal{A}}$ to two other representations ($D_{\mathcal{G}}$ and $D_{\mathcal{P}}$), which could require substantial work on the users side, and also requires an understanding of all three data structures involved. If we want to use additional packages, such as a sparse matrix visualization tool [BL94] for debugging or assessing the quality of the reordering, additional transformations are needed.

As a concluding remark, we note that sparse matrix-vector products are the key operation used in iterative solvers, which in turn are the key components of most PDE solvers. Efficient implementation of these operations depends both on the data structures used and the underlying hardware, just as is the case for dense matrices. The development of a *sparse* BLAS, however, has to cope with difficulties, most notably the great diversification of data structures for representing sparse matrices, which enforces a corresponding proliferation of different implementations for the basic operations — in [SL98], the authors note that the NIST sparse BLAS [RP97] contains about 10000 routines.

[2] As noted before, this might also be accomplished by *registering* the matrix-type $D_{\mathcal{A}}$ and the preconditioner-type $DP_{\mathcal{P}}$ to the solver component, e. g. by deriving from a base class provided by $\mathcal{S}$.

2.2.4.3 Reuse Case Study: Particle Tracing on a Grid

A well-known method for visualization of unsteady flow-phenomena is *particle tracing* [TGE97]: A set of particles is inserted into the domain, and the path they follow over time is displayed graphically. Numerically, this requires the integration of an ordinary differential equation (the Eulerian equation of motion) for each particle.

In the context of numerical PDE solution, the flow field is given at discrete locations on a computational mesh, for example one value per computational cell. Any algorithm for the calculation of particle paths will therefore require close interaction with the grid data structure and the representation of the flow field, cf. p. 64.

Now different scenarios for reusing existing particle tracing components are conceivable. First, we might have access to a general purpose flow visualization package, such as VISUAL3 [Hai99]. In this case, we have to pass the entire grid and flow field to that package. This is generally ok as long as the time required for copying the data is small compared to the time used for the calculation. Typically this is the case if the sampling rate is small: Data is written only every nth time step.

A small sampling rate does not hurt for *static* visualization techniques, acting on a fixed-time snapshot. For *dynamic* techniques like particle tracing, however, this may result in an unacceptable loss of accuracy. Therefore, it is tempting to update the current particle positions in every time step, a task which is computationally much cheaper than copying the complete flow field if there are relatively few particles.

Can one take such an implementation from another package offering it? Well, chances are low. In general, the grid data structures of two flow solvers will differ, and any implementation that committed to one of these will not work on the other one. Therefore, it will be necessary either to re-implement the particle tracer, or to pass the entire data every time-step to a visualization package, which is costly.

The problems discussed in the three short subsections above all have to do with incompatible data structure assumptions made by algorithmic components. Before moving to generic programming as a possible solution, we briefly discuss why two possible solution approaches — namely usage of 'standard' data structures and data structure conversion — do not offer a general solution.

2.2.4.4 Data Structure Variance in Scientific Computing

The source of all evil seems to be the variability in data structures used. Why can't one just accept a standard representation format for each type of mathematical object, such as unstructured grids or sparse matrices? At any rate, this seems to work to some extent in classical numerical libraries, e. g. for dense linear algebra.

By itself, this is a reasonable idea, but it is bound to fail in the large.

Disregarding political reasons — people (vendors, scientists) are simply not willing to commit to a common standard — the true problem is *inherent* in the diversity of the mathematical structures themselves, combined with efficiency considerations. A computational grid may be Cartesian, in which case it is given implicitly only, it may be completely unstructured, allowing only simplices or more general cell types. Furthermore, different *computational* requirements on functionality often mandate different data layouts.

The 'obvious' way of simply choosing the most general of these structures to represent grids is ruled out by efficiency considerations. Moreover, this would *not* solve all problems. Some algorithms used on grids, such as *downstream ordering* [Bey98] or *grid partitioning* [Els97] actually act on *graphs*. Therefore, they would — in the 'standard data structure' approach — be formulated on the standard *graph* representation which *cannot* match that of the grid *viewed as* graph.

The conclusion is that one has to live with a lot of different data structures. The richer the diversities of the underlying mathematical structures are, the more diversity there is.

2.2.4.5 Reuse of Algorithms and Data Structure Conversions

If we are not able to prescribe *identical* data structures across different components, we can try to *convert* between different representations in order to be able to use them.

We will distinguish between *conversion* and *copying*: By conversion we mean the more general situation that the data structures may differ substantially and may not be equivalent, like is the case for the graph/grid example above. We will use the term *copying* only if no loss of information occurs and therefore the process is *reversible*, and the work is proportional to the size of the original data.

It has already been stressed that converting data structures between different components rises several problems. The first one is the overhead it introduces, both in terms of speed and memory usage. In some circumstances it may be negligible, as in the matrix/graph conversion example, when seen in context — the work for linear system solution will probably dominate. Still, it increases time and memory requirements, which is generally not welcome.

In other circumstances, the overhead of conversion may be prohibitive:

- The data may be given implicitly, and may not even fit into memory
- Conversion may destroy asymptotic runtime behavior, if the algorithm is sublinear in the size of the data; for example a local search or the overlap generation method presented on page 133.
- It may slow down an algorithm by orders of magnitude if the problem occurs in a *nested* fashion. Put another way, it can make *composition* of algorithm implementations from different sources practically infeasible.

The next major problem is *usability*. In general, one has to know the internals of *both* data structures to convert between them, so conversion may be a non-trivial task and in fact a serious obstacle to reuse.

A third problem occurs if additional data has to be associated with the input, for example a partition number for a domain decomposition of a FEM mesh. Then the correspondences between entities of the source and the copy have to be preserved, which may be a tricky issue and further increases memory consumption. Even more difficult is the case when the data structure is changed by the algorithm, for example grid optimization or sparse matrix reordering. In general, it is completely unclear how these changes are mapped back to the original data structure.

There is, however, another aspect of the conversion issue. Apart from the circumstances where two copies are really required, there may also be performance reasons for it: Sometimes, the optimal data structure for an incremental build (*dynamic usage*) is significantly different from a data structure optimal for use by a particular algorithm (*static usage*), so that conversion may indeed pay off.

On the other side, knowledge of optimal data structures is closely related to algorithmic details. This decision should ideally be left to the algorithmic component.

Therefore, conversions and copying *are* of course useful and often necessary operations; support for them is crucial. Yet, a naive approach would need, for n different data structures, $n(n-1)$ conversion algorithms, thus failing to scale with a growing number of data structures.

In sum, implementations should strive to render conversions unnecessary. Unfortunately, current practice and technology makes them often necessary in order to reuse algorithms with incompatible assumptions on the underlying data structures.

Therefore, the central problem is: how to formulate algorithms in order to operate in a way *independent* of concrete data structures. This question is tackled in the rest of this thesis.

2.3 The Generic Programming Approach: A Step Towards Algorithm Reuse

2.3.1 The Idea of Generic Programming

The preceding section made clear the fundamental problem to be solved: On the one hand, there is a great (and inevitable) variance of data-structures found in many branches of scientific computing. On the other hand, while *algorithms* are *in principle* independent of most of these differences, concrete *implementations* typically are hampered by committing to random details. The main task therefore consists in decoupling of algorithm implementations from the concrete data representations *while preserving efficiency*. Thus, a high level of reusability could be achieved.

The solution we present here is associated with a certain shift in perspective. Traditionally, in procedural languages like FORTRAN, algorithms where the only entities that could be modeled by the language. Later, object-oriented languages allowed data-abstraction by separating data-structure interfaces from its implementation. Unfortunately, this has entailed a tendency to *bury* algorithms into classes that are mainly data-oriented. Algorithms, however, are concepts of their own right which must have a separate representation, in order to be reused independently.

The 'hiding' of algorithms is not a major problem in domains like GUI programming, where they play a secondary role. It becomes a *severe* problem when algorithms are the *fundamental abstractions* of a domain, like is the case in scientific computing.

Generic programming [MS89, MS94, Ste96] shifts the focus back to algorithms. It concentrates on their prerequisites in an abstract form, that is, it classifies data structures with respect to an algorithm or a class of algorithms. An algorithm's *domain of application* typically stretches over a vast range of data-structures. Implementations should preserve as much as possible of this generality, in order to be a faithful concrete representation of the abstract entity.

Because of this emphasis on algorithms and their implementations, generic programming is also sometimes referred to as *algorithm-oriented* [MS94]. By the separation of algorithms and data structures both concepts are lifted on an *equal* level of abstraction, as has been noted by [Wei98]. The virtues of data abstraction are by no means abandoned, but rather complemented or even raised by a corresponding abstraction for both algorithms and their relationships to data structures.

A necessary prerequisite to generic programming is a thorough *domain analysis* [PD90]:

Domain Analysis A necessary prerequisite for the construction of generic components is an investigation of the algorithms and data structures one wants to cover.

- Identify the mathematical structures that are the object of interest
- Identify the classes of algorithms for which generic implementations are sought
- Set up a taxonomy of data structures used to represent the abstract mathematical structures, corresponding to the functionality they support
- Analyze a representative number of algorithms for their essential requirements on mathematical as well as on data structures.

This step will be carried out in chapter 3.

Abstraction The next step consists in uncovering *commonalities* and *variabilities* [Cop98] in requirements and functionality of algorithms and data structures, respectively.

- Find requirements that algorithms have in common, find functionality that satisfies a large class of algorithms
- Match these requirements with the abilities of data structures; classify data structures with respect to classes of algorithms
- Represent the requirements and their corresponding functionality by an (ideally minimal) set of *concepts*

Concepts are just another term for *sets of requirements*. A central class of *core concepts* will act as a *thin broker layer* separating data structures and algorithms. The most prominent example for such a concept (in the domain of sequences) is the *iterator* concept, discussed below. Concepts for the grid domain will be identified in section 4.1.

Implementation The abstract concepts have to be mapped to concrete software entities.

- Choose suitable programming language and features, like templates (generics) or polymorphism
- Find a suitable syntax — a "common spelling" for expressing the concepts

In section 4.2, we describe how generic components can be based on the concepts developed before.

At this stage, one is not entirely free in the choice of the target language. Our choice has been C++, a decision motivated in section 2.3.5. It should be noted, however, that the results of this thesis are largely language independent.

At the beginning of this section, efficiency has been identified as an important goal of generic programming. As a consequence, the abstract concepts have to obey to two principles: First, they have to expose — instead of hiding — fundamental performance characteristics of the data structures, such as whether random access takes constant time or not. Second, the overhead introduced by using an implementation of an abstract concept instead of a low-level representation directly, has to be minimized. Ideally, it should be possible to remove it entirely in an automatic way, for instance by an optimizing compiler.

Efficiency considerations are one aspect that distinguishes the system of concepts and requirements from the notion of an abstract data type (ADT) — in contrast to a pure ADT, the implementation actually *matters* in some well-defined respect.

A second difference to classical ADTs is the *granularity* of concepts. It has already been mentioned that a central class of abstractions is that of an iterator, which essentially plays the role of a *broker* between algorithms and (sequence) data structures. This notion captures a single, yet essential aspect of data structures, but cuts in a sense a 'thin horizontal slice' out of the functionality described by usual ADTs. Concrete data structures typically play many different roles in different contexts, a situation which is generally not dealt with very well by the notion of an ADT.

On the other side, the whole set of concepts and requirements for a given domain may well include contradicting issues, which cannot be fulfilled simultaneously by any concrete component. In the context of a particular algorithm (or a class of algorithms), we actually need to satisfy only a tiny subset of those requirement. If a particular combination occurs sufficiently often, it may be useful to name it explicitly. In theory, however, there may be exponentially many combinations, and naming them is impractical.

To gain a deeper understanding of the generic programming approach, we work out in some detail a well known example.

2.3.2 An Introductory Example: The STL

The C++ Standard Template Library (STL) [LS95] has become one of the best-known examples of generic programming. The importance of the STL does not primarily lie in the functionality it provides as a container class library (though it by far exceeds that provided by others), but rather in the way of structuring the problem domain. The increased interest in generic techniques certainly has one of its sources in the availability of this library.

The STL is concerned with the domain of linear sequences and algorithms operating thereon, such as counting, sorting and partitioning.

Sequences are a truly ubiquitous concept. Besides their 'natural' home in container data structures, they are often an aspect of more complex constructs, and thus are often represented *implicitly*.

The STL identifies several core *categories* of concepts representing the fundamental abstractions of the domain:

- *sequences* of some arbitrary *data type*, most notably *containers*,
- *iterators* abstracting the notion of a position or address within a sequence,
- *algorithms* working generically on sequences, represented by pairs of iterators `[begin,end)`,
- *function objects* encapsulating operations on the opaque data types, like comparisons, predicates or algebraic operations.

Other categories often mentioned in this context are *allocators* encapsulating memory management strategies, and *adapters* changing the behavior or syntax of some entity. However, allocators are used exclusively in the implementation of containers and do not interfere with other categories. Also, adapters are somewhat orthogonal to these categories, and we contribute them to the category where the modified concepts belong. So, a function-object adapter that binds one argument of a binary operator to produce a unary operator belongs to the function-object category.

On the other hand, one could also consider the *data type* occurring for example in the basic container type parameterization as additional concept. In the context of the STL, it is perhaps just not worth bothering, because in general the only requirements on this type are that it be *assignable* (the expression `t1 = t2;` is valid), and (sometimes) *default constructible* (the expression `T t;` is valid). Everything else is parameterized in an algorithm or container implementation, in particular comparison operations.

A very simple example for a generic algorithm, counting occurrences of items satisfying a predicate `Pred`, is the following:

```
template<class Iter, class Pred>
int count(Iter begin, Iter end, Pred p) {
  int cnt = 0;
  while (begin != end) {
```

```
    if(p(*begin))
      ++cnt;
    ++begin;
  }
  return cnt;
}
```

From this piece of code, some observations can be made:

- Algorithms are not parameterized by container, but by iterators, allowing general sequences to be processed, including input streams and sequences created on-the-fly.

- Sequences are given by an *half-open* interval `[begin,end)` of iterators.

- The syntax of iterators is borrowed from C pointers; in fact, pointers *are* valid iterator types.

- In this code, the iterator must provide the following operations:

  - increment (`++begin`)
  - dereference (`*begin`)
  - equality test (`begin != end`)

  That is, the *footprint* of the algorithm consists of only three operations on iterators.

Interestingly, these operations are sufficient for a large class of algorithms — in addition, one occasionally needs to know types associated with an iterator, most notably `value_type` which is the type the iterator 'points to'. The major source of differences is the increment capability of iterators. Compared to the properties of pointers, the unit-stride forward increment used here is a rather weak requirement. A finer distinction of increment capabilities leads to five different sub-categories of iterators.

These are *random access* (allowing arbitrary jumps in constant time), *bidirectional* (allowing increment and decrement), *forward* (allowing increment only), *input* (requiring increment and dereference to alternate, no assignment to value possible), and *output* iterators (the same, but no read possible). In table 2.1 we list the categories along with some prototype models. These categories are hierarchically ordered; see figure 2.1.

All iterator operations are guaranteed to be constant time. Once again: it would be possible to 'enhance' a bidirectional iterator with random access functionality, but not with constant time random access. Such a *pseudo-random-access* iterator would not be considered as substitutable for a random-access-iterator.

It turns out that algorithms can be implemented fully generically based on this classification of iterators: If necessary, there is a *compile-time branch* on the iterator category to select the appropriate implementation, which then has optimal complexity for this concrete iterator. Optimality is to be understood with respect to the selected

category	**prototype sequence**	**prototype algorithm**
random access	array	QUICKSORT
bidirectional	doubly linked list	sequence reversal
forward	singly linked list	BUBBLE SORT; multi-pass
input	input stream	copy from input; single pass
output	output stream	copy to output; single pass

Table 2.1: The five iterator categories and some prototypical examples of where they arise, and where they are required.

random access — bidirectional — forward — input
output

Figure 2.1: The *is-a* relation of iterator categories, read from left to right.

algorithm; for example, BUBBLESORT never has optimal complexity with respect to the class of *all* sorting algorithms.

For illustration, consider BINARY SEARCH: This algorithm is implemented once for general forward iterators, using $O(\log n)$ comparisons and $O(n)$ increments, and specialized for random access iterators, using only $O(\log n)$ increments.

In fact, the code actually is *the same*, because the different behavior of forward and random access iterators relevant to BINARY SEARCH has been encapsulated in the `advance` primitive, which is $O(1)$ on random access and $O(n)$ on forward iterators. This technique generalizes to an important design principle in generic programming:

Try to handle non-uniformity at a level as low as possible.

We will encounter it time and again, because it can be seen as a means of *homogenization* enabling genericity: Higher levels (BINARY SEARCH) can be implemented in a *fully generic* way.

It should be noted at this point that the homogenization technique crucially depends on the fact that generic procedures and classes can be *specialized*. This property of a genericity mechanism marks a significant increase of power over simple macro-based approaches. It becomes even more powerful if specializations may stay partially generic (*partial specialization*), for this allows specialization hierarchies to be built.

Seen from 'below', the homogenization pattern turns into a *polyalgorithm* pattern: Based on (static) properties of the data, a particular 'algorithm' is selected, ADVANCE in our case. This approach can be used in the implementation of a copy algorithm: If the sequence is a contiguous chunk of memory of bitwise-assignable type (e. g. built-in arrays of built-in types), then the faster `memcpy()` routine is called. On a higher level, `copy()` can be called without worrying if things wouldn't be more efficient using low-level tools: Genericity without remorse. The optimization performed here goes beyond what would normally be done in a direct, non-generic implementation.

The branching requires that we be able to decide at compile-time if a type has the *bitwise-assignable* property. Therefore, a necessary precondition for the polyalgorithmic approach to work properly is a thorough analysis of the performance sub-categories of the concepts, followed by implementation techniques that make the information available at compile time. This analysis usually is the hardest part of the work.

2.3.3 Sparse matrices revisited

For sparse matrices, it would be very tempting to make an implementation fit into an ADT with a random-access subscripting with two arguments (i, j). One could then apply all sort of matrix algorithm to that abstraction. This approach, however, is practically worthless, as it neglects the fundamental performance issues — which are the very purpose sparse matrices are invented for. Some characteristic properties that could be of interest for algorithms are:

- Can all non-zero elements of a sparse matrix A be enumerated in time proportional to the number of non-zero elements $NZ(A)$? This would distinguish dense representations from sparse ones.
- Can one determine in constant time if a given position (i, j) is a non-zero?
- Can one access in constant time an entry at a given position (i, j)? If not, at least for the diagonal positions?
- Can one access all entries of a given row (column), in time proportional to the number of non-zeros in that row (column) ?

An important layer of functionality is the so-called 'sparse BLAS': basic algebraic matrix-vector operations, like matrix-vector product. On top of this level, most iterative solution algorithms can be built, see for example the Iterative Methods Library (IML++) [DLPR99] or Iterative Template Library (ITL) [LS99a]. However, up to date, there is no generic sparse BLAS building on yet lower levels, like those touched upon above; direct approaches lead to a combinatorial explosion of components (cf. p. 30). A very promising approach uses concepts from data-bases [Kot99].

As noted before, concrete data types in general are not confined to be models of a *single* ADT, rather, different ADTs may represent possible *views* on the concrete type. A sparse matrix $A = (a_{ij})$ may be viewed (among others) as

- a container of its *non-zero* entries a_{ij}, with $(i, j) \in NZ(A) = \{(i, j) \mid a_{ij} \neq 0\}$
- a container of its rows (columns)
- a directed graph with the indices $1 \leq i \leq n$ as nodes, and an edge from i to j iff $(i, j) \in NZ(A)$
- a full matrix with $a_{ij} = 0$ iff $(i, j) \notin NZ(A)$

In order to use algorithms from the corresponding domains, one has to provide adapters that implement the *viewed-as* relation. For example, if one wants to use STL algorithms on the sequence of non-zeros, one has to provide STL-style sequential iterators for this sequence.

2.3.4 Generic Programming for Grids

Generic programming has proven its usefulness for a class of mathematically simple, but very important concepts, namely linear sequences. Can it be extended to more complex problem domains?

One such candidate domain is the domain of grids and grid-based algorithms. Their importance for scientific computing, especially the numerical solution of PDEs, has already been discussed, as well as the role they play in other domains. Also, our investigation of existing software has revealed a lack of components which are sufficiently *general* to cope with the variabilities of the domain.

The drawbacks of possible solutions for the data-structure variance problem, namely, copying and standard data structures, have been described earlier. The arguments discussed there are highly relevant to the grid domain.

Typically, in numerical PDE solution, grids and associated data structures account for a high fraction of the total memory consumption of the program, which can be a limiting factor on problem size. Therefore copying leads both to memory and performance bottlenecks, and is ruled out, except for 'peripheral' operations like preprocessing (grid generation and partitioning) and postprocessing (visualization), where it is common practice. But also these steps can greatly benefit from the generic paradigm.

Also, the great variability of both mathematical grid types and data structures for their representation, to be discussed in sections 3.1 and 3.3, makes it impossible to choose a standard data structure. For example, for Cartesian grids, one may use an implicit representation. Nevertheless many algorithms do not exploit the Cartesian properties and could benefit from a generic implementation.

In addition, some important algorithms used by grid-based applications more properly operate on graphs, as already noted. Thus, the standard representations of grids and graphs would have to match, if graph algorithms are to be reused, which is impossible.

Developing generic grid components turns out to be more difficult then generic sequence components. This has several reason. On the one hand, the mathematical structure is richer. Sequences mainly differ by traversal functionality, expressed by iterator categories like forward, bidirectional or random access. Grids, in contrast, may differ in much more aspects that may influence the choice of the best-suited algorithms.

The richness of mathematical structure leads to more detailed concepts. We will identify three main subconcepts of a geometric grid, namely a combinatorial layer defining connectivity, a geometric layer defining the embedding into a geometric space, and a data association layer defining how to map grid entities to arbitrary values. The combinatorial concept itself consists, among others, of iterators, but the single iterator abstraction so powerful for sequences gives way to a whole family of iterators.

Furthermore, as has been noted above, grids may play many different roles, e. g. they may act as sequences or graphs in several ways. Therefore, some algorithms acting on grids more properly belong to another domain. If one takes the generic approach seriously, their implementations should be based on the abstractions of that *other* domain. In order to reuse these algorithms in the grid domain, one has to define a suitable mapping between the abstractions of both sides, thus formalizing the meaning of the 'viewed-as' relationship.

Finally, the semantics of *dynamic* data structures are more complicated for grids than for sequences — there are more ways to change a grid than to change a sequence, and at the same time, changes to grids have to respect more structure. A number of important algorithms most naturally act on dynamic data structures, for example grid refinement. Finding generic implementations for such algorithms is a challenge.

So far, the discussion has revolved around the generic implementation of algorithms. During our investigation of the grid domain, we will see that also data structures can benefit from a generic approach, see the components described in section 4.2. Perhaps the most convincing example can be found in chapter 5 on distributed grids, where a system of generic algorithms and data structures enhance a given sequential grid with parallelization support.

2.3.5 Language and Implementation Issues

After choosing the generic paradigm as leading principle, a first decision is the choice of a programming language. In the present case, the choice was C++. Let us explain why.

There are a number of languages supporting parametric polymorphism, like ADA *generics* [MS94], EIFFEL [Mey92], or C++*templates* [Str97, BN95]. Of all these languages, only C++ has found widespread use in scientific computing.

In this context, C++ has proven its ability to deliver efficient code. Recently, techniques have been developed [VJ97] allowing to achieve efficiency on par with FORTRAN, which continues to be the reference in this respect.

A further argument is the *multi-paradigm* nature of C++ [Cop98]: One can use object-oriented features where appropriate, classical procedural style where it matters, and, most importantly for us, *generic* programming. All these paradigms can be used in parallel in scientific computing applications, such as discussed in chapter 6.

Indeed, C++ has a very elaborate support for generic programming. The development of the STL as part of the C++ standard library paralleled the development of the C++ standard [Int98], and had a noticeable impact on the latter [SS95]. Consequently, the template mechanism of C++ is by now far more complex and powerful than in its original version: Features like *template specialization*, *partial specialization* and *partial ordering of function templates* allow to build hierarchies of genericity, and to do the sort of *homogenization* described above (p. 38). These features also allow techniques like *template meta-programming* [Vel95b], or can be used to let the C++ compiler do some sort of *partial evaluation* [Vel99b].

Member template seamlessly integrate generic and object-oriented programming. *Default template parameters* are useful for offering simplified versions of generic components, while allowing advanced usage with higher degrees of parameterization. Finally, the potential of template parameters that are themselves templates is yet to be uncovered.

In sum, these features clearly show that the C++ template mechanism goes far beyond simple macro-like substitution techniques. In a comparative richness, it is not found in any language adequate for scientific computing.

The second decision to take is to define a syntax (a *common spelling*) for the concepts identified in the analysis step (see section 4.1 and appendix 7.3). All implementations in the generic library will stick to this syntax.

Any implementation therefore (re-)introduces some elements of *arbitrariness*, which have been removed with much labour in earlier stages. The central question is how these arbitrary decisions can be *localized* and *controlled*.

The 'obvious' approach would be that of a standardized syntax for a problem domain. But this turns out to be rather impractical, as standardization processes are slow, if successful at all. The more complicated a domain is, the less probable it is that there is consensus even on the question what belongs to the domain, not to mention such details like interfaces.

Perhaps the most difficult problem is *cross-domain* syntax and semantics (the '*view-x-as-y*' relationship). It is here where the standardization approach is ultimately bound to fail, because evidently, every domain has its own language, see also the discussion in section 2.2.4.4.

In general, the best what can be hoped for is a *de-facto* standard. In the C++ community, the 'standard' set by the STL seems to approach this state for the domain of sequences.

The STL syntax generalizes predefined language constructs, such as pointers (for iterators), and function call (for function objects). It turns out that chances to agree on syntax are higher if it is 'natural'. In C++, this most often means that it uses the 'canonical' operator syntax, well supported by operator overloading. Unfortunately, this is no longer sufficient for more complex cases.

One possibility to deal with the situation is to bundle the 'parameters of arbitrariness' into special entities, for example the so-called *traits* classes known from C++ template programming [Mye95]. Via parameterization by these traits, the implementations allow users to introduce *translations* from their native spelling to that of the algorithms library.

Thus, translator traits are by no means some minor work-around, but an important concept making reuse possible in the first place. They include type definitions and query functionality required by an algorithm.

Traits also allow to pass more context information to a generic component as is usually possible, for example different comparison operations. An example for this technique is the `CGT` parameter of the neighbor search procedure documented in the appendix C.1. In most examples, such traits are not shown, to keep presentation simple.

The real test for the viability of the generic approach will be the interoperability of independently developed components. This is rather straightforward for the case of sequences. Some preliminary examples have been developed for inter-operation with the GGCL graph library (see below), but a full proof has yet to be done.

2.3.6 Related Work

Generic programming increasingly moves into the focus of attention in scientific computing and other algorithm-centered fields like Computational Geometry. Consequently, generic software components are becoming available. In the following, we discuss some examples, including components for dense and sparse linear algebra, Computational Geometry and graph algorithms.

Blitz++ [Vel99a] implements basic n-dimensional arrays along with algebraic operations on them. The declared objective is to achieve performance on par with or superior to Fortran 77. This is achieved by using advanced techniques like *template meta-programming* [Vel95b]. For cache-optimized stencil operations, the author reports significant speedup over Fortran code using a straight-forward approach. The stencil-operations use a data-locality enhancing array traversal based on Hilberts space-filling curve [VJ97]. This could in principle be implemented also in Fortran. The decisive advantage of the generic approach is that while in a conventional setting this strategy would have to be reproduced for each loop, a *single* generic implementation of this strategy is sufficient — a clear *separation of concerns* [HL95].

The Matrix Template Library MTL [LS99b] contains data structures for dense and sparse matrices, as well as generic low-level algorithms (BLAS-functionality), and a generic LU-factorization.

GMCL (Generative Matrix Computation Library [CEKN99]) defines generative matrix data structures. Parameterization possibilities include entry type, structure type (dense, banded, triangular etc.), data layout, and more. Therefore, they allow a rich collection of concrete data structures to be instantiated: The GMCL is reported to support about 1800 different instantiations, counting only built-in floating-point types for possible entry types.

Generic implementations of iterative methods for the solution of linear systems are provided by the Iterative Template Library ITL [LS99a]. It contains iteration algorithms like CG and preconditioners like ILU.

CGAL [pro99] implements generic classes and procedures for geometric computing, for example convex hulls and triangulations. The complexity of the underlying data structures is much higher than in the matrix/array libraries cited before. Most algorithms are parameterized by *traits* classes (see above) for customization of primitive operations. Incremental algorithms are typically implemented as classes, such as Delaunay triangulation and incremental convex hulls in 3D. These classes are parameterized by geometric traits classes importing the needed basic geometric operations and predicates. Geometric entities such as points are parameterized by the arithmetic type, such as exact or floating point representations of real numbers.

A very recent library is GGCL (*Generic Graph Component Library* [LSL99, LLS99]), which uses a container-based approach to graph adjacency relationships, somewhat similar to the concepts we develop in later sections. Access to data on vertices or edges is modeled using the *decorator pattern* (see [GHJV94]), which seems to be rather general, but not as comfortable as the grid function approach developed later in this work (page 75, section 4.1.4).

The LEDA (p. 26) library itself does not define generic graph algorithms. There has, however, recently been added a Graph Iterator Extension package [NW96], which implements some basic graph algorithm, for example breadth-first-traversal or DIJKSTRAs shortest-paths algorithm. These are implemented as algorithm classes allowing stepwise execution. Earlier research in this direction has been conducted by Weihe, Kühl and Nissen [KNW97] [KW96]. Access to edge-data (for example length in DIJKSTRA) is parameterized with *data-accessors* [KW97]. These play the role of data associations (grid functions), discussed on page 75.

In recent years, many interesting developments have taken place in the field of programming and software development, seeming highly relevant also for scientific computing, like *active libraries* [VG98], *generative programming* [CEG+], or *aspect oriented programming* [KLM+97]. A detailed presentation of these developments would go beyond the scope of this thesis. A good overview is found in [Cza98, CE00]. We will briefly come back to these issues in section 7.3, where we discuss possible directions for further research.

Chapter 3

The Grid Domain

Was man nicht versteht,
besitzt man nicht.
Johann Wolfgang von Goethe

In the previous chapter, we have pointed out the fundamental role of grids for many fields of computational science, like scientific computing, Computer Graphics and Computational Geometry. But what *exactly* is a grid? If we asked this question to people working in the fields just mentioned, the answer would perhaps seem clear on first sight in any single case. However, a closer look would certainly reveal that it is impossible to give *one* formal definition matching *all* uses of the term *grid* — in fact, there is a *family* of mathematical structures commonly referred to as grids, meshes, subdivisions, or complexes.

Therefore, an important step towards reusable grid components is a thorough *domain analysis*. In a first step, we present some basic mathematical properties of grids, point out their distinguishing features, and establish a sort of glossary for grid-related mathematical terms. Then, in section 3.2, some typical grid algorithms are analyzed to find out which properties of grids they require, and what functionality a grid data structure must offer in order to apply the algorithm. Finally, section 3.3 analyzes grid data structures with respect to the requirements of algorithms. Existing implementations of such data structures are examined from that perspective, too.

The results of this chapter are an essential precondition for the development of generic reusable grid components in chapter 4.

3.1 Mathematical properties of grids

Mathematical fields like topology and polytope theory have developed concepts which provide a firm basis for reasoning about grids. The following sections review some definitions and results from these disciplines. We first investigate topological foundations, then review contributions from the theory of convex polytopes, and finally discuss

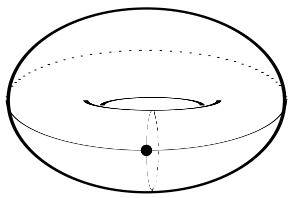


Figure 3.1: A non-regular cell decomposition of the torus with one vertex, two edges and one face

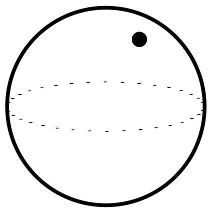


Figure 3.2: A non-regular decomposition of the 2-sphere with one vertex and one face

combinatorial properties of grids and their relationship to geometric grids.

The distinction between combinatorial and geometric aspect arises in a natural way, when one considers finite representability in a computer. This is typically easier to achieve for the combinatorial kernel, which consequently also forms the basis of almost every grid data structure.

The geometric aspects are, by their very nature, non-discrete and thus in general are not representable exactly. It turns out that in this regard the notions of topology often are "to general", while the geometries admitted by polytope theory normally are "to constrained".

3.1.1 Topology of Cellular Systems

René Thom once remarked: "Topology is precisely that mathematical discipline which allows a passage from the local to the global." One tool for passing from local to global structures are *cellular complexes*, which provide a means to represent complicated geometric structures by locally relating simple objects (cells). It is this representation that enables the computational treatment of topology-related problems in the first place.

This section defines only notions that bear a direct relationship to the topic; for the basic terms, see any introductory text, like Henle [Hen79] or Jänich [Jän90]. Comprehensive treatments of the theory of cellular system can be found in the books of Moise [Moi77], Lundell and Weingram [LW69], or Fritsch and Piccinini [FP90].

The basic building blocks are homeomorphic images of open balls in $\mathbb{R}^k$.

Definition 1 (open cell). Let X be a Hausdorff space. A set $c \subset X$ is an *open k-cell* if it is homeomorphic to the interior of the open k-dimensional ball $\mathbb{D}^k = \{x \in \mathbb{R}^k \mid \|x\| < 1\}$. The number k is unique by the *invariance of domain* theorem, and is called *dimension* of c.

A collection of cells that are fitted together in an appropriate way form larger structures, so-called *complexes*.

Definition 2 (CW-complex). A finite *CW-complex* $\mathcal{C}$ is a system of pairwise disjoint sets c and a Hausdorff topology on $\|\mathcal{C}\| := \bigcup_{c \in \mathcal{C}} c$, where the sets c are open cells in $\|\mathcal{C}\|$, and for each k-cell c there is a continuous *characteristic mapping*

$$\Phi : \overline{\mathbb{D}^k} \mapsto \overline{c}$$

which is a homeomorphism of $\mathbb{D}^k$ to c and maps $\mathbb{S}^{k-1}$ to the union of cells of $\mathcal{C}$ of lower dimension, the *sides* of c. The closure $\overline{c}$ in $\|\mathcal{C}\|$ is the union of c and all its sides. The complex $\mathcal{C}$ is said to be a *cell decomposition* of the underlying space $X = \|\mathcal{C}\|$.

If for a k-cell c, the mapping Φ extends to a homeomorphism on the whole of $\overline{\mathbb{D}^k}$, then c is *regular*; its sides are then a homeomorphic to a decomposition of $\mathbb{S}^{k-1}$. The CW-complex $\mathcal{C}$ is *regular* if all of its cells are.

Intuitively, non-regularity comes from identifying parts of the boundary of a cell, as in the torus in figure 3.1. In practice, non-regular cells arise in the coarse grid underlying a *multi-block* grid, for example so-called *C-grids* or *O-grids*.

Remark 1. *In topology literature, CW-complexes are not necessarily finite; the additional axioms postulated for them (which gave rise to the 'CW' in the name) are trivial in the finite case. To avoid confusion, we will henceforth only use the term* complex*, meaning a* finite *CW-complex.*

Definition 3. If $\mathcal{C}$ is a complex of dimension d, we call its 0-cells *vertices*, its 1-cells *edges*, its 2-cells *faces*, its $d-1$-cells *facets*, and its d-cells simply *cells*.

Whereas the terms *vertex* and *edge* are used consistently throughout the literature, this is not so for *face* and *facet*; see also section 3.1.2 on polytopes. While it is fairly common in topology to use the term *cell* for cells of *any* dimension, in numerical simulation practice, in particular *finite volume* parlance, its use is restricted to cells of maximal dimension d.

Therefore, in a context where terms like *vertex*, *facet* etc. are used, *cell* will denote d-cells exclusively, and the terms k-cell, k-*element* or just *element* will be used to mean cells of dimension k or undetermined dimension, respectively. There should be less danger of confusion with *finite elements*. See table 3.1 for an overview.

The definition of a complex suggests to see it as a subdivision of a given topological space X into cells. A complementary way to think about complexes is to construct them by gluing cells together. This leads us to define neighborhood relations:

Definition 4 (incidence). If an element f is a side of another element c, that is, $f \subset \bar{c}$, we say f and c are *incident*, and write $f < c$. If, in addition, $\dim f = \dim c - 1$, then we also write $f \prec c$.

If two elements e, f are incident, without any information about the relative dimensions, we use the general notation $e \lesseqgtr f$. The *sequence* of all k-elements $e \in \mathcal{C}^k$ incident to an element f is denoted by $\mathcal{I}_k(f)$, cf. page 89.

name	dimension	codimension	alternative names
vertex	0	d	0-cell, 0-element
edge	1	$d-1$	1-cell, 1-element
face	2	$d-2$	2-cell, 2-element
ridge	$d-2$	2	$d-2$-cell, $d-2$-element
facet	$d-1$	1	$d-1$-cell, $d-1$-element
cell	d	0	d-cell, d-element, solid, volume

Table 3.1: Overview over element terminology

Elements of the same dimension k are never incident, because the boundary of an open cell has no interior point and therefore cannot contain a homeomorphic image of $\mathbb{D}^k$.

The notion of adjacency is not used very consistently in this context. We define it for vertices and cells:

Definition 5 (adjacency). Two cells are *adjacent* if there is a facet incident to both; vertices are adjacent if they are incident to a common edge. Adjacent cells (vertices) are called *neighbors*.

Definition 6 (homogeneous dimension). An element that is not part of the boundary of another element is called a *principal* element. A complex is of *homogeneous dimension* d if every principal element is of dimension d.

Following [AH35], it is possible to define a discrete topology on the finite set of elements of a complex $\mathcal{C}$, which permits to express many combinatorial statements in a topological language.

Definition 7 (discrete topology). In the *discrete topology* on a complex $\mathcal{C}$, a set A is closed if an only if it is the union of a k-element with its sides, or the union of such sets.

It follows that the smallest open set (in the discrete topology) containing a given element e is obtained by taking its union with all incident elements of higher dimension. This set is called the *open star* $\mathrm{st}(e)$ of e. The boundary of $\mathrm{st}(e)$ is called the *link* $\mathrm{lnk}(e)$ of e. As long as the sets under consideration are unions of elements, it does not matter whether the boundary is taken with respect to the original or the discrete topology.

Definition 8 (subcomplex). A subset $\mathcal{C}' \subseteq \mathcal{C}$ of a complex $\mathcal{C}$ is called a *subcomplex* of $\mathcal{C}$ if $\mathcal{C}'$ is itself a complex. It is understood here that $\|\mathcal{C}'\|$ is endowed with the subspace topology with respect to $\|\mathcal{C}\|$.

This is equivalent to saying that for each $c \in \mathcal{C}'$, also its sides are contained in $\mathcal{C}'$; or that $\|\mathcal{C}'\|$ is closed in $\|\mathcal{C}\|$. If $R \subset \mathcal{C}$, then the closure $\overline{R}$ of R is the smallest subcomplex of $\mathcal{C}$ containing R. A special kind of subcomplexes are *skeletons*, obtained by removing elements of higher dimension.

Definition 9 (k-skeleton). By $\mathcal{C}^k$, we mean the set of k-elements of $\mathcal{C}$. The set $\mathcal{C}^{\leq k}$ consisting of all l-elements, $l \leq k$, is called *k-skeleton*.

The *k-skeleton* is a subcomplex of $\mathcal{C}$ for each k. In particular, the 1-skeleton is a graph. Note that $\mathcal{C}^k$ is in general not a subcomplex, except for $k = 0$.

An important special case of complexes are *simplicial* complexes:

Definition 10 (simplicial complex). If $\|\mathcal{C}\| \subset \mathbb{R}^n$ for some n and the k-cells of $\mathcal{C}$ are k-dimensional simplices $(0 \leq k \leq d)$, then $\mathcal{C}$ is called a *simplicial complex* and a *triangulation* of the space $\|\mathcal{C}\|$.

Simplicial complexes are always regular, because each simplex is regular. Historically, simplicial complexes have been used long before CW-complexes were introduced by Whitehead. From a topological point of view, CW-complexes are often preferred over simplicial complexes because of their greater flexibility. So it is possible to decompose a torus into a *general* complex with one 2-cell, two 1-cells and one 0-cell, see figure 3.1. Many more simplicial cells are needed to create a homeomorphic *simplicial* complex, see [Jän90]. This increased versatility is also interesting for geometric modeling purposes, especially as the only geometric constraint is that cells be homeomorphic to a disk of appropriate dimension. On the other hand, many solid modeling approaches use cells homeomorphic to a unit ball with finitely many holes. This case is not covered even by the general definition of a CW-complex.

Until now, the underlying space X of a complex is rather arbitrary. For most practical purposes, a considerably restricted class is sufficient:

Definition 11 (manifold). A *manifold* of dimension d is a Hausdorff space locally homeomorphic to $\mathbb{R}^d$; a *manifold with boundary* is locally homeomorphic to either $\mathbb{R}^d$ or the closed halfspace $\mathbb{H}^d = \{x \in \mathbb{R}^d \,|\, x_d \geq 0\}$. The boundary of M is the set of points where M is locally homeomorphic to $\mathbb{H}^d$.

Accordingly, a regular subdivision of a manifold (*with boundary*) is called a manifold-(*with-boundary-*) complex, m-complex and mwb-complex for short. Such a subdivision is evidently of homogeneous dimension. Simple examples for manifolds are the sphere $\mathbb{S}^d$, and the closed unit ball $\overline{\mathbb{D}^d}$ for manifolds with boundary. Manifolds can evidently be considered as special cases of manifolds with boundary.

An example for a non-manifold complex is shown in figure 3.3. Note that a subcomplex of a mwb-complex is not necessarily a mwb-complex, even if it is of homogeneous dimension, see figure 3.4. This complex can obviously be embedded into a mwb-complex of dimension 2, whereas the first example cannot, as the middle facet has three incident cells.

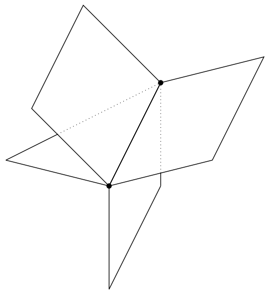


Figure 3.3: A non-manifold complex, not embeddable into a 2-manifold

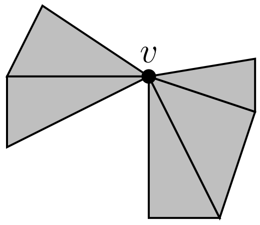


Figure 3.4: A non-manifold vertex v, embeddable into a 2-manifold

Definition 12 (boundary). A facet that is incident to only one d-cell is called *boundary facet* of a mbw-complex; other facets (with two incident d-cells) are *interior facets*. The *boundary complex* is the closure of all boundary facets.

What we have defined above is more properly called *combinatorial boundary* (MOISE [Moi77]). It is non-trivial to show that it is the same as the topological manifold boundary, and requires the famous Jordan curve theorem for the case $d = 2$. The case $d = 3$ is harder, which can be appreciated when knowing that an analogue for JORDANS theorem fails in $\mathbb{R}^3$: There exist so-called *wild spheres* that do separate $\mathbb{R}^3$ into more than 2 components, see again the book of MOISE for an example. However, the main difficulty here stems from admitting arbitrary homeomorphic images of the unit ball as cells. From a practical point of view, having the finite representability of such mappings in a computer in mind, this poses no problems. For restricted types of geometric cells (like simplices) the proof is simple.

Most of the topological spaces that occur in physical modeling are d-dimensional domains or *solids*.

Definition 13 (solid). A d-dimensional *solid* is a d-manifold with boundary embedded in $\mathbb{R}^d$. A corresponding cell decomposition is consequently called a *solid complex*.

The boundary of a compact solid is a compact $d-1$-manifold without boundary.

It might seem at first glance that only mwb-complexes are of interest for scientific computing, but more general types of complexes arise at many places. The k-skeleton of a complex is not a manifold-complex (except in special circumstances), nor are in general the closures of sets of cells. In the geometric modeling of complicated solution phenomena such as *shock fronts*, the so-called *shock tracking*, or in simulation of surface evolution under some forces [Bra92, Bra] non-manifold complexes can arise in a natural way.

3.1.2 Convex Polytopes

Among the most fundamental geometric objects are convex polytopes. They may be regarded as cellular structures, but exhibit a great deal more regularity than general complexes. For an introduction into the theory of convex polytopes, see e.g. the book of ZIEGLER [Zie94]. Here we cite just some definitions and results.

Definition 14 (convex polytope). A convex polytope $\mathcal{P}$ is the convex hull of a *finite* set of points in $\mathbb{R}^{d+1}$ for some $d \geq 0$. The dimension of $\mathcal{P}$ is the dimension of its affine hull.

Intersection of $\mathcal{P}$ with 'tangent' halfspaces gives us the *faces* of $\mathcal{P}$ (in this section, we use the term *face* as it is employed in polytope theory, it corresponds to k-elements or k-cells of topology).

Definition 15 (faces of a polytope). If, for a fixed $c \in \mathbb{R}^{d+1}$, the inequality $c^T x \leq \gamma$ is valid for all $x \in \mathcal{P}$, then $f = \mathcal{P} \cap \{x \in \mathbb{R}^{d+1} | c^T x = \gamma\}$ is a *face* of $\mathcal{P}$. With this definition, also $\mathcal{P}$ and $\emptyset$ are (*improper*) faces. All other faces are called *proper*. The dimension of a face is the dimension of its convex hull; by convention, the dimension of $\emptyset$ is set to -1. As above, the faces of dimension 0 are called vertices.

The following theorem collects useful geometric and combinatorial properties of polytopes.

Theorem 1. *Let $\mathcal{V}(\mathcal{P})$ be the vertex-set of a polytope $\mathcal{P}$. The following properties hold:*

1. *$\mathcal{P}$ is the convex hull of its vertices:*

$$\mathcal{P} = \{ \sum_{v \in V(\mathcal{P})} \lambda_v v | \sum \lambda_v = 1, \quad \lambda_v \geq 0\}$$

   For a point $x = \sum_{v \in \mathcal{V}(\mathcal{P})} \lambda_v v \in \mathcal{P}$, the numbers λ_v are in general not unique, except for simplices. They are called the barycentric coordinates *of x.*

2. *$\mathcal{P}$ is the intersection of finitely many halfspaces of $\mathbb{R}^{d+1}$. The faces of dimension d defined by these halfspaces are called* facets *of $\mathcal{P}$.*

3. *Every face f is itself a polytope, with vertex set $\mathcal{V}(f) = \mathcal{V}(\mathcal{P}) \cap f$. The proper faces of f are called* subfaces.

4. *The faces of f are the faces of $\mathcal{P}$ that are contained in f, in particular $\mathcal{V}(f) = \mathcal{V}(\mathcal{P}) \cap f$.*

5. *Intersections of faces are faces.*

6. *A face is uniquely determined by its vertex set, that is, no two faces have the same vertex set. Consequently, it is the convex hull of its vertex set.*

7. *A face is uniquely determined by the set of facets containing it. It is the intersection of the affine spaces defining these facets.*

It follows from the definition that the faces of a polytope are closed. An *open face* is a face without its subfaces. From the theorem, we see immediately that the open faces of a polytope are the elements of a complex. This complex is a solid (see page 50), and the boundary complex of a polytope is a manifold complex. Note that properties 5 - 7 are not true for general solid complexes, cf. fig. 3.5.

The geometric characterizations 1 and 2 are complementary. They correspond to the combinatorial properties 6 and 7. It should be noted that the geometric characterizations are in general not robust: For a *non-simple* vertex (at a vertex the polytope has the local structure of a simplex), disturbing the hyperplanes that define the incident facets will cause a combinatorial change. The same thing happens when moving a vertex of a *non-simplicial* facet. Therefore, if the combinatorial structure is of interest, geometric characterization in combination with inexact arithmetic is not adequate for representing polytopes, except for simplices.

3.1.3 Combinatorial Structure of Complexes

Both general cellular complexes and convex polytopes were introduced using geometric notions. We are, however, unable to represent arbitrary *geometric* complexes in a computer. In contrast, the *combinatorial structure* of finite complexes is well suited for this purpose, and is the core of any discrete representation.

Therefore, in the following two sections, we will adopt a different point of view, considering complexes primarily as combinatorial entities, and treating geometric structure as an additional one, which will have to be attached in a later step.

To see what combinatorial properties of grids can (and cannot) tell about the underlying complex, it is of interest to study them in isolation. These properties have a great impact on the internal layout of data structures, as well as on the class of algorithms applicable on a grid.

It is important to see which algorithms depend only on the combinatorial – as opposed to geometrical – properties of a complex. As a rule of thumb, using only combinatorial information results in more stable algorithm, and often it is a sign of improper reasoning to use geometric information, when an algorithm may be formulated in combinatorial terms alone.

On the closed elements of a complex $\mathcal{C}$, the inclusion $\subset$ defines a partial order $<$, the incidence relation defined on page 47. This relation contains the combinatorial structure of $\mathcal{C}$, captured by the notion of a *poset*:

Definition 16 (poset). A poset $(\mathcal{S}, <)$ is a finite set $\mathcal{S}$, together with a partial order $<$ which is antisymmetric and transitive.

Figure 3.6 uses a *Hasse diagram* to visualize the poset of a complex. Elements are drawn above all lesser elements. Any two element are connected by a line if they are comparable and there is no element 'in between'.

A poset is *bounded* if there are unique minimal and maximal elements $\hat{0}$ and $\hat{1}$. A *chain* is a totally ordered subset of a poset. A bounded poset is called *graded* if every maximal chain has the same *length*, i. e. number of elements minus 1. For $a \leq b$, the *interval* $[a, b]$ is the set of all elements in between:

$$[a, b] = \{c \in \mathcal{S} | a \leq c \leq b\}$$

If $\mathcal{S}$ is graded, the *rank* of $a \in \mathcal{S}$ is the length of a maximal chain in $[\hat{0}, a]$.

The poset of $\mathcal{C}$ can be made into a bounded one by adjoining the *improper elements* $\hat{0} = \emptyset$ and $\hat{1} = \|\mathcal{C}\|$, with dimensions -1 and $d + 1$. Than the following theorem holds:

Theorem 2. *For a regular complex $\mathcal{C}$ of homogeneous dimension d, the poset formed by the cells is graded, and the rank of an $k-1$-element is k.*

Proof. The claim is evidently true for $d = 0$, and if it holds for $d - 1$, we can argue as follows: Because of the homogeneous dimension, each maximal chain must contain a cell of dimension d, and the regularity ensures that there exists a cell c' of dimension $d - 1$ in $\bar{c}$. Now c' lies in the $d - 1$-skeleton and by induction has rank d, therefore c has rank $d + 1$. □

Definition 17 (lattice). A lattice is a bounded poset in which any two elements a, b possess a unique maximal lower bound $\min(a, b) = a \wedge b$ (the *meet*) and a unique minimal upper bound $\max(a, b) = a \vee b$ (the *join*). A *graded* lattice of rank $d + 1$ is *atomic* if each element is the join of elements of rank 1 (the *atoms*); it is *coatomic* if each element is the meet of elements of maximal rank d (the *coatoms*).

The meet of two elements c_1, c_2 corresponds to their geometric intersection $\overline{c_1} \cap \overline{c_2}$, the join to the element c of least dimension containing both of them: $\bar{c} \supset c_1 \cup c_2$. The join of two elements may be the whole grid. In general, the meet and join operations are *not* distributive.

Theorem 3. *The poset of a convex polytope is an atomic and coatomic lattice. Moreover, it has the* diamond property*: If e and g are incident faces of dimension $k - 1$ and $k + 1$, respectively, then the interval (e, g) contains exactly two faces f_1, f_2 of dimension k.*

For an arbitrary complex, the poset is in general not a lattice, see fig. 3.5. If a grid's lattice is atomic, this means that k-cells are uniquely determined by their vertex set; if the lattice is coatomic, the vertex sets of k-cells for $k < d$ can be determined from the knowledge of the vertex sets of d-cells alone. Therefore, it suffices in principle to store the vertex sets for each d-cell of a grid known to have a coatomic lattice. This observation can also be useful when reading a grid from some format that is only partially known, for example, combinatorial cubes given by indices of their vertex-set, but without a rule how these relate to the facets of a cube.

Solid complexes with non-empty boundary are not coatomic, because boundary facets cannot be obtained by intersection of cells. Under certain circumstances, for

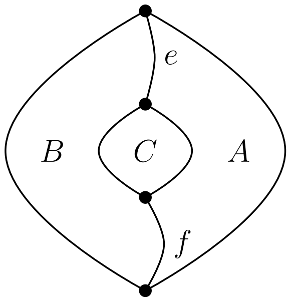


Figure 3.5: Facets e and f have no unique join (A or B possible)

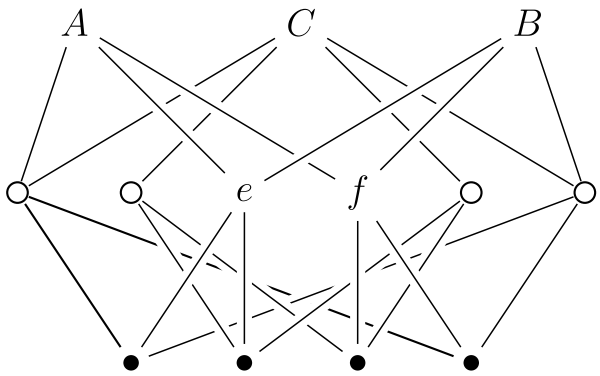


Figure 3.6: Hasse-diagram of the lefthand cell-complex

example if the complex is the subcomplex of a convex polytope, this can be remedied by adding artificial cells for each boundary facet (or giving the vertex sets of boundary facets as additional input to an algorithm).

Definition 18 (Abstract complex). An *abstract* (or *combinatorial*) *complex* $\mathcal{A}$ of dimension d is a finite bounded poset (not necessarily graded) whose elements $e \in P \setminus \{\hat{0}, \hat{1}\}$ are labeled with a dimension $\dim(e) \in [0, d]$, such that $e < f$ implies $\dim(e) < \dim(f)$.

If an abstract complex $\mathcal{A}$ is graded, then the dimension of an element obviously must coincide with its rank minus one. For a complex $\mathcal{C}$, we write $\mathcal{A}(\mathcal{C})$ for the corresponding abstract complex.

Definition 19 (Combinatorial morphism). A combinatorial morphism between abstract complexes $\mathcal{A}_1$, $\mathcal{A}_2$ is a mapping $\Phi : \mathcal{A}_1 \mapsto \mathcal{A}_2$ that respects incidences: $e < f \implies \Phi(e) < \Phi(f)$. If Φ is injective, it is called *monomorphism*; if it is bijective, it is called *isomorphism*.

If two geometric complexes $\mathcal{C}_1, \mathcal{C}_2$ are homeomorphic, and the elements of $\mathcal{C}_1$ are mapped to the elements of $\mathcal{C}_2$, then $\mathcal{A}_1$ and $\mathcal{A}_2$ are obviously isomorphic.

In general, the converse is not true: If two abstract complexes are isomorphic, this does not imply that they are homeomorphic, for example, the two-dimensional torus, the Klein bottle and the projective space $\mathbb{RP}^2$ can all three be represented by the same abstract complex consisting of one cell, two edges and one vertex. However, one can show that two simplicial complexes are homeomorphic if and only if their abstract complexes are isomorphic.

Definition 20 (dual poset). We say that two posets $\mathcal{P}$ and $\mathcal{Q}$ are *dual* if they have the same elements with the reversed order relation.

Thus, the process of obtaining the dual to a poset $\mathcal{P}$ can be thought of as turning its Hasse diagram upside down. For a graded poset of rank d, this means that elements with rank k correspond to elements with rank $d-k$ in the dual.

Two complexes are called dual if their posets are dual. In the case of a manifold-complex, it is possible to construct a *geometric realization* (see below) with is a subdivision (see [AH35]) of the same manifold. For manifolds with boundary, this is not possible — one would have to add an artificial cell for each boundary component. Therefore, one typically modifies the dual poset on the boundary, for example by adding the dual poset of the boundary (which *is* a manifold complex) to the dual poset. Geometrically, this is equivalent to intersecting the dual of the 'enlarged' complex (including the artificial outer cells) with the original complex, see also [Ale98].

The Euler-Poincaré Formula
A fundamental topological property of a space X is the *Euler characteristic* $\chi(X) \in \mathbb{Z}$. With aid of a cell decomposition $\mathcal{C}$ of X, it can be computed by the *Euler-Poincaré Formula*

$$\chi(X) = \sum_{c \in \mathcal{C}} (-1)^{\dim c} = \sum_{k=0}^{d} (-1)^k |\mathcal{C}^k| \tag{3.1}$$

where $|\mathcal{C}^k|$ is the number of k-cell of $\mathcal{C}$. In spite of the apparent dependency on the decomposition $\mathcal{C}$, the Euler characteristic is a topological invariant of the underlying space X.

For the 2-sphere, we have $\chi(\mathbb{S}^2) = 2$, in which case equation 3.1 reduces to the classical Euler formula

$$V - E + F = 2$$

where V, E and F are the numbers of vertices, edges, and faces, respectively.

The Euler characteristic of a two-dimensional solid $\Omega \subset \mathbb{R}^2$ is the number of components minus the number of holes. For three-dimensional solids, the Euler characteristic equals number of components - number of tunnels + number of holes. For convex polytopes of any dimension, the Euler characteristic is equal to 1.

3.1.4 Geometric Realizations of Abstract Complexes

We arrived at a finite, representable structure by "forgetting" geometric properties of complexes. In practice, however, we generally want to work with geometric complexes. It is therefore necessary to answer the question how to associate geometric information to an abstract complex in a consistent and constructive manner.

The separation of combinatorial from geometric properties is not only of theoretical interest, it also leads to a cleaner *separation of concerns* in the corresponding data structures, see section 3.3.1.3.

Definition 21 (geometric realization). A *geometric realization* Γ (or *geometry* for short) of an abstract complex $\mathcal{A}$ is a complex $\mathcal{C}$ with an isomorphic poset. We write $\|\mathcal{A}\|_\Gamma := \|\mathcal{C}\|$ for the image of $\mathcal{A}$ under Γ.

This definition answers the question about consistency of geometric and combinatorial structure; yet it gives no clue of how to describe such a realization in a constructive way representable in a computer.

In principle, it would be sufficient to give the characteristic mapping for each element. However, these give no relationship between the geometry of an element and that of its sides. We are therefore lead to the somewhat more restricted notion of element *archetypes*, which contain also combinatorial information, and introduce a standard enumeration of their sides. This allows to treat also some types of non-regular cells.

Definition 22 (archetype of an element). Let c be a k-element of a combinatorial complex $\mathcal{A}$. An *archetype* $\mathfrak{a}_c$ of c is a regular complex of dimension k with exactly one k-element a_c and satisfying the following properties:

1. It is $\mathfrak{a}_c = \overline{a}_c$, and $\mathfrak{a}_c \subset \mathbb{R}^k$

2. There is a continuous characteristic mapping

$$\Phi_c : \mathfrak{a}_c \mapsto \overline{c}$$

   that is a homeomorphism of a_c on c, and which induces a *surjective* combinatorial morphism $\mathcal{A}(\mathfrak{a}_c) \mapsto \mathcal{A}(\bar{c})$ (also named Φ_c).

3. Φ_c maps each side of a_c onto a side of c *of the same dimension*. Moreover, if $\bar{c}$ is regular, then Φ_c is an isomorphism.

4. For each dimension $j < k$, there is a fixed order on the j-sides of $\mathfrak{a}_c$.

5. For each side f of c, there is a fixed archetype $\mathfrak{a}_f$.

The mapping Φ may send several sides of a_c to the same side of c, thus having these sides "glued together". The archetype allows to distinguish those sides. However, as sides with the same pre-image under Φ_c must be of the same dimension, the definition excludes cells with sides "squeezed together" resulting in dimension loss. Thus a cell like in the torus decomposition in figure 3.1 is allowed, one as in figure 3.2 is not.

Archetypes allow for economic storage of repetitive incidence patterns, especially in cell-oriented data structures, see section 3.3. Often, they are used implicitely, see section 3.2.1.

A geometric realization of an abstract complex $\mathcal{A}$ can now be defined by listing archetypes for each d-cell of $\mathcal{A}$. These archetypes must be consistent, that is, for each element e incident to two cells c_1, c_2, we must have

$$\Phi_{c_1}(e) = \Phi_{c_2}(e) \tag{3.2}$$

Sometimes, we need more, for example when defining finite element spaces over cells. In this case, we can require that each archetype is a convex polytope. This defines barycentric coordinates on each d-cell. For the elements of lower dimension, we may strengthen the condition 3.2 by requiring the following: For each element e incident to two cells c, c', there exists a linear mapping

$$L = L_e = L_{e,c,c'} : \mathbb{R}^d \mapsto \mathbb{R}^d$$

such that the *continuity relation*

$$\Phi_c = \Phi_{c'} \circ L \quad \text{on} \quad \mathfrak{a}_c \tag{3.3}$$

holds. By affine invariance of barycentric coordinates, these are uniquely defined also on elements with dimensions $k < d$.

In many practical approaches, a geometric embedding is given by simply fixing the geometric locations of vertices. This, however, is unambiguous only if all cells are combinatorial simplices. Then we can define a linear embedding by simply giving the values of the Φ_c on the vertices of $\mathcal{A}$. In other cases, for example hexaeder cell, one implicitly assumes trilinear embeddings, which degenerate to linear ones for planar and parallel sides. In general, however, it certainly is a good idea to make this mapping explicit.

In the case of simplices, the mappings L can be defined in a straightforward manner. In other cases, there need exist no linear mappings between faces of $\mathfrak{a}_c$ and $\mathfrak{a}'_c$, for example, two planar quadrangles are not necessary affinely equivalent to each other. In this case, one would have to replace the linear map L_f with a more general identification mapping.

However, for most practical purposes, dimension is bounded by 3, and available archetypes are limited to just a few standard ones, where this poses no problem. Note that the linear mapping is required only between archetypes, not between the geometric cells themselves.

3.1.5 Some Special Grids and Geometries

For use in numerical algorithms like the finite element method, the general complexes defined above are to general. For modeling complex geometries, convex polytopes are to restrictive. In general, different context require varying degrees of flexibility and generality, both of combinatorial and geometrical aspects.

Definition 23 (grid). From now on, we reserve the term *grid* for a complex of homogeneous dimension. Consequently, a *subgrid* of a grid is a subcomplex which is also of homogeneous dimension.

Starting with the most simple and restricted possibility, a *Cartesian grid* is a grid whose combinatorial structure is equivalent to the grid whose cells are

$$\{[i_1, i_1 + 1] \times \ldots \times [i_d, i_d + 1] | 0 \leq i_j \leq m_j\}$$

A *pseudo-Cartesian* grid is a subgrid (of the same dimension) of a Cartesian grid.

A *multiblock grid* of dimension d consists of a general complex (coarse grid), whose cells are d-dimensional cubes, each subdivided by d-dimensional Cartesian grids (blocks). Blocks incident to the same cube side are isomorphic on that side. The coarse grid of a multiblock grid is a typical example for the use of non-regular grids: A very common cell type is a so-called *O-grid*, which is an annular part of a polar coordinate system, that is, a quadrangle with two opposite sides identified.

Any grid not belonging to one of the three types mentioned above is called *unstructured grid*. A special case are *simplicial grids* which have cells combinatorially equivalent to simplices.

A computational domain almost always is a manifold with boundary. A manifold-with-boundary complex with cells that are *combinatorially* equivalent to convex polytopes allows the kind of geometric realization described above. If, in addition, the incidence-poset is a lattice, we call such a grid a *polytopal grid*.

The lattice-property of a polytopal grid guarantees that intersections of the vertex-sets of elements always uniquely define a lower-dimensional element, which is an important precondition to some algorithms, for example algorithm CELL NEIGHBOR SEARCH, p. 61. Typically, the types of element polytopes are restricted to just a few, like triangles and quadrangles in 2D, and tetrahedrons, cubes and prisms in 3D. The reason is that one has to define some function space on the grid. Typically, the elements of this function space are piece-wise defined polynomials of low degree and small support, which are in general required to be at least continuous over facets. This is not a trivial task for arbitrary polytopes.

A *linear* or *straight-line geometry* (of dimension n) is a geometric realization where each k-element is mapped to subset of a k-dimensional affine subspace of $\mathbb{R}^n$. In particular, edges join vertices always by straight line segments.

A *mapped geometry* is defined by combining a *global* mapping to a simpler embedding, for example a linear geometry.

A *Cartesian geometry* is a geometry for a Cartesian grid, which is a uniform scaling and translation of the integer grid; thus the geometric cells are axis-parallel squares.

3.2 Grid Algorithm Requirements

We have stressed time and again the importance of algorithms for scientific computing, and that reuse of algorithm implementations is a key to successful mastering software complexity in this field — like any other algorithm-centered domain as well.

As pointed out before, separation of data representation issues from algorithmic aspects is an essential step towards reusable algorithmic components. Separation is the more important (and the more difficult), the more complex the underlying mathematical structures are.

It is now time to analyze algorithms operating on grids to see how they interact with grids and what kind of functionality is required from grid data structures. Needless to say, there is no hope of analyzing every algorithms in the field; however, a carefully selected group of representative algorithms will provide us with helpful insight, in particular, the analysis will shed light on many commonalities of requirements.

3.2.1 Persistent Storage of Grids

One of the most basic operations needed is to output a grid on some persistent storage medium. A multitude of formats exist for this purpose. For the sake of example, we use the format described in table 3.2 for a general two-dimensional grid, mixing combinatorial and geometric information.

Data			Description
n_v			Number of vertices
n_c			Number of cells
x_1	y_1	$[z_1]$	coordinates of vertex 1
$\vdots$			
x_{n_v}	y_{n_v}	$[z_{n_v}]$	coordinates of vertex n_v
n_1	$v_{1,1} \dots v_{1,n_1}$		vertices of cell 1
$\vdots$			
n_{n_c}	$v_{n_c,1}$	$v_{n_c,n_{n_c}}$	vertices of cell n_c

Table 3.2: A simple file format for general 2-dimensional grids

An algorithm for writing a file in this format from a grid is given by algorithm table 3.1. We can readily detect the requirements of this algorithm:

Algorithm 3.1 Write a 2D complex to a file representation

```
print number of vertices of 𝒢
print number of cells of 𝒢
for all vertices V ∈ 𝒢^0 do
    print coordinates of V
for all cells C ∈ 𝒢^d do
    print number of vertices of C
    for all vertices V_C incident to C do
        print the number of V
```

1. Query a grid about the number of cells and vertices.

2. Iterate over all vertices and all cells
3. Get the number of vertices of a cell; iterate over vertices incident to a cell.
4. Read-access to the coordinates of a vertex
5. Access to a number for each vertex, such that vertices are consecutively numbered, starting from 1.

Concerning the consecutive enumeration, this can be produced in the course of the algorithm as follows:

$n \leftarrow 1$
for all vertices $V \in \mathcal{G}^0$ **do**
    print coordinates of V
    associate n with V
    $n \leftarrow n + 1$

Then we have a new requirement, namely, to associate temporary (integer) data with a vertex.

Of course, this algorithm neglects any sophisticated type of geometry, as the file format can only represent vertex coordinates, but no curved edges or similar things. Also, there is a subtle implicit assumption about the sequences of cell vertices which comes into play when trying to reconstruct a grid from such a representation, namely, that consecutive vertices belong to the same edge. In fact, this is an assumption about cell archetypes. There may not be much choice in two dimensions, but it is a real problem in three dimension, where the archetypes are no longer 'natural' and have to be fixed somewhere explicitly.

3.2.2 Cell Neighbor Search

Many algorithms need access to their *neighbor* cells. This information often has to be created from more basic data, such as provided by the file format described in table 3.2. In this case, the vertex sequence of cells is known, such that consecutive vertices are the vertex set of a facet, i. e. an edge in two dimensions. A file format for three-dimensional grids would need to provide additional information about the vertex set of facets.

Starting from this situation, cell neighbors can be calculated by algorithm 3.2, yielding the following requirements:

1. The vertex set of each facet is unique (for example, the grid is polytopal, see p. 58)
2. there are at most 2 cells per facet
3. iteration over all cells of a grid
4. iteration over all facets of a cell, without knowing cell neighbors

---

Algorithm 3.2 Cell Neighbor Search: Find cell neighbors from facet vertex sets

---

IN: A grid $\mathcal{G}$

OUT: A mapping $\mathcal{I} : \mathcal{G}^d \times \mathbb{N} \mapsto \mathcal{G}^d$ which maps each cell to its sequence of neighbors, such that the nth facet of a cell c is incident to the neighbor in $\mathcal{I}(c, n)$.

$\mathcal{N} : \mathbb{P}\mathcal{G}^0 \mapsto \mathcal{G}^d \times \mathbb{N}$ is a mapping from vertex sets to (cell,local side) pairs.
$\mathcal{N} \leftarrow \emptyset$
for all cells $C \in \mathcal{G}^d$ **do**
    for all facets $F \prec C$ **do**
        $f \leftarrow$ local number of F in C
        if $F^0 \notin \mathrm{dom}\,\mathcal{N}$ **then**
            $\mathcal{N}(F^0) \leftarrow (C, f)$ *(Store C and local facet number)*
        else *(facet already found from the other neighbor D.)*
            $(D, f_D) \leftarrow \mathcal{N}(F^0)$
            $\mathcal{N}(F^0) \leftarrow \emptyset$
            $\begin{array}{lcl} \mathcal{I}(C,f_C) & \leftarrow & D \\ \mathcal{I}(D,f_D) & \leftarrow & C \end{array}$
        end if
(Facets still in $\mathcal{N}$ are boundary facets.)

---

5. access to the vertex set of a facet, (iteration over the vertices of a facet will suffice)
6. comparison of vertex sets for equality, this implies: vertices must be equality comparable
7. in order to guarantee an efficient implementation of the map $\mathcal{N}$, such vertex sets must be either be totally ordered or endowed with a hash function, which essentially means the same requirement for vertices.

It would also be advantageous (in favor of memory efficiency) to have a sort of minimal representation for cells and vertices, given a fixed underlying grid $\mathcal{G}$.

3.2.3 Bandwidth-reducing Ordering of Grid Elements

Often, grid elements correspond to the unknowns of linear systems of equations. For solution algorithms based on elimination strategies, it is often advantageous to reduce the *bandwidth* of the matrix, that is, the maximal difference in the numbering of two unknowns coupled by a linear equation, see [Saa96].

This problem is best formulated in terms of graphs, where nodes correspond to unknowns, and an edge between two nodes means that the corresponding unknowns are connected by a non-zero entry in the matrix. The task is then to number nodes such that the maximal difference of adjacent node numbers is minimized.

A well-known heuristic for band-width minimization is the Cuthill-McKee algorithm, which essentially is a breadth-first traversal which sorts each layer in order of increasing node degree.

Algorithm 3.3 CUTHILL-MCKEE ORDERING: Bandwidth minimization

IN: a graph $\mathcal{G}$ and a starting node $v_0 \in \mathcal{G}$
OUT: a numbering $N : V(\mathcal{G}) \mapsto \mathbb{N}$ *($V(\mathcal{G})$ is the vertex set of $\mathcal{G}$)*

$n \leftarrow 1$ *(current vertex number)*
$N(v_0) = n$
$L_0 \leftarrow \{v_0\}$ *(Level 0 consists of v_0)*
Mark$(v) = 0 \; \forall v \in \mathcal{G}$
Mark$(v_0) = 1$ *(mark v_0 as visited)*
$k \leftarrow 0$ *(current level is 0)*
while $L_k \neq \emptyset$ **do** *($L_k = \emptyset$: no new vertices found, stop.)*
    $L_{k+1} \leftarrow \emptyset$
    for all $v \in L_k$ **do**
        if Mark$(v) = 0$ **then**
            Mark$(v) = 1$
            Add v to L_{k+1}
            $n \leftarrow n + 1$, $N(v) \leftarrow n$
        end if
    sort L_{k+1} in order of increasing degree
    $k \leftarrow k + 1$
end while

Conforming to the spirit of generic programming, this algorithm should be implemented in terms of graphs, rather than of grids. It is evident how this increases reuse possibilities: If we have a grid with unknowns living on, say, cells, and two unknowns connected by an equation if and only if their corresponding cells are neighbors then this grid can be mapped to such a graph quite easily, by viewing cells as nodes and facets as edges. But this is by no means the only possible way a graph can arise from grids: For example, if unknowns live on vertices, a different graph results.

Thus, it is very important to find the 'natural' mathematical structure for an algorithm to operate on — a description based on grids would have to distinguish all these cases, whereas a graph-based one does not.

Now, in order to analyze CUTHILL-MCKEE ORDERING in terms of grid functionality, we have to fix one interpretation. For simplicity, we choose the cell-facet case, afterwards pointing out what changes when taking a different case.

1. Iteration over the neighbors of cells

2. Query on the number of neighbors of a cell

3. Association of integer (N) and boolean (Mark) data to cells

A special remark applies to the mapping Mark: It would be advantageous if the assignment Mark$(v) = 0 \, \forall v \in \mathcal{G}$ could be done in constant time, especially if the algorithm is executed only on a part of the grid (graph).

If we store unknowns on vertices, and for example define the linear system such that vertices belonging to a common cell are coupled by an equation (typical for FEM Matrices), then we would have to replace 'cell' with vertex, and neighbor-iteration with a somewhat more complicated iteration over all cell incident to a vertex, and all vertices incident to cell; in fact, what is needed here is iteration over the *link* of a vertex (see page 48).

Things get worse when unknowns are coupled across wider distances than in this example. For typical numerical applications, this coupling can be described by so-called *stencils*, a topic examined in section 5.5.4.

The mapping of grids to graphs can be done in a generic fashion. Currently, a preliminary mapping of the cell-neighbor-graph to the GGCL libary (p. 44) is implemented. Thus, there has to be only *one* software component for each type of graph arising from grids, and one implementing the actual algorithm.

3.2.4 Graphical Output

Graphical output of grids certainly is an important tool, for example, it can give visual control over *grid quality*. Actually, the task of producing visual output consists of two sub-tasks: First, generation of geometrical entities from the grid data structures, and second, *rendering* of these entities on some graphical device, such as pixel graphics or postscript file. The rendering step also includes things such as clipping, illumination and camera positioning (for three-dimensional objects).

Here, we are only concerned with the first step, relying on a generic `draw` method of some unspecified graphics device which works for *simple* objects like lines and polygons.

The possibilities of graphical output are manifold, we confine ourselves to give some typical examples. A simple line graphic ('wire frame') can be produced as follows:

for all edges $e \in \mathcal{G}^1$ **do**
  GraphicsDevice.draw(segment(e))

A somewhat more advanced visualization method, especially for 3D grids, is a shrink-view of cells: Each cell is shrinked by some factor towards its center, offering better insight into the connectivity:

IN: shrink-factor $\alpha < 1$
for all cells $c \in \mathcal{G}^d$ **do**
  $p_c \leftarrow$ center of c
  for all facets f of c **do**
    let P be the polygon defined by the vertices v of f, with vertex coordinates $x'(v) = p_c + \alpha(x(v) - p_c)$.
    GraphicsDevice.draw(P)

The requirements of these tiny algorithms are the following:

1. iteration over all edges or over all cells

2. iteration over all facets of a cell

3. mapping of an edge to a geometric segment
4. mapping of a facet to a geometric polygon
5. write access to vertex coordinates of polygons
6. calculation of cell centers

Note that in general, the segments belonging to edges could be curved. In this case, additional work would have to be performed to transform them into simpler entities. The same holds for polygons.

However, it is wise to assign responsibility for these transformations to the rendering engine GraphicsDevice, because here the knowledge about the representable primitives is located. Also, resolution issues can be handled in a more meaningful manner; for example the question, into how many line segments a curve segment has to be broken.

3.2.5 Particle Tracing

The technique of particle tracing, already discussed in section 2.2.4.3, is an important tool for visualizing instationary flow. Basically, it is the integration of a time-dependent vector field, given on a grid. We sketch the integration of a single particle for a given time interval, e. g. a time step of a numerical simulation.

Algorithm 3.4 particle tracing

IN: A velocity vector field F on $\mathcal{G}$
IN: a time interval $[t_0, t_1]$
IN: A pair (x, c) of a particle position $x = x(t_0)$ in a cell $c \in \mathcal{G}$
OUT: an updated pair $(x' = x(t_1), c')$

$t = t_0$
$\Delta t = \min(\Delta_0 t, t_1 - t_0)$
while $t < t_1$ **do**
  while $x \in c$ **do**
    $x \leftarrow$ integrate$(x, t, t + \Delta t, F_c)$
    update Δt (corresponding to error estimation)
    $t \leftarrow t + \Delta t$
  end while
  $c \leftarrow$ locate$(x, c, \mathcal{G})$
  if $c \notin \mathcal{G}$ **then**
    STOP
  end if
end while

Here we use some numerical ODE solver method integrate which works on the *local* field F_c which we assume smooth enough, and offering evaluation in constant time at any point $x \in c$. For instance, it could be a constant or linear function. Furthermore, some method locate for locating a point x in a grid is necessary. Here, we know a good starting cell as a hint, and hence can use a local search method:

IN: a point x, a starting cell c_0, and a grid G
OUT: an updated cell c with $x \in c$ if there is such a cell in the vicinity of c_0, an invalid cell denoting 'out of grid' otherwise

```
for all cells c that share a vertex with c0 do
  if x ∈ c then   (found)
    return c
  end if
  (Not found)
return invalid cell
```

Here we search only the nearest vicinity of a cell c_0. The assumption is that increments will be small compared to cell diameters, which may or may not be true. In general, for a local search, one could perform a breadth-first-traversal of the grid, starting from c_0, until a cell containing x has been found. The problem here is the stopping criterion. If a cell lies on or near the boundary, one has to decide whether the point has left the grid via the boundary. For doing so, one could also include directional information into the search, which could guide the breadth-first traversal. For example, one could take always that neighbor which is in the appropriate direction.
The requirements of the simple version shown are the following:

1. store floating point data on cells (or vertices)
2. test for point inclusion in cell
3. iteration over cell neighbors, or over all cells sharing a vertex with a given cell.

For the advanced search, we need in addition:

1. iterate over facets of a cell
2. intersect rays with facets
3. test if a facet is on the boundary of the grid

The iteration over a restricted set of cells like those sharing a vertex with a given cell will often require marking already visited cells. In order to be efficient, this marking must absolutely be linear in the number of marked cells, that is, the *unmarking* of all cells at the beginning must be $O(1)$.

A different approach is to 'fake' a globally defined function from the discrete grid data, providing evaluation from arbitrary space positions. In this case, however, cell location has to take place in the global function, which ultimately leads to the same problems with incremental search. Global search is in general to expensive, as it does not exploit the fact that successive evaluations are typically close in space.

3.2.6 Grid Refinement

The refinement of computational grid in response to some error indicator is a key component of adaptive numerical methods for partial differential equations. The algorithms are among the more complicated ones, especially in three dimensions.

Two popular schemes are the bisection-based approach and the so-called *red-green* refinement [Ban98, Bey98], based on a number of refinement templates. The earliest and simplest bisection algorithm for 2D is *longest-edge* bisection, as introduced by RIVARA [Riv84]. Its basic outline is as follows:

IN: a grid $\mathcal{G}$
IN: a refinement indicator on cells $R \subset \mathcal{G}^d$
OUT: a refined grid $\mathcal{G}'$, where all cells $c \in R$ have been bisected at least once
  for all cells $c \in R$ **do**
    split c at longest edge e_l
    remove c from R
    if the neighbor n incident to e_l is not refined **then**
      add n to R
    end if

In this algorithm, it is critical to satisfy the following requirements:

1. The grid is a triangulation
2. one can combinatorially split the vertex set of a cell into two sets that belong to the two descendents
3. access the edges of a cell
4. access the cells incident to an edge
5. access the vertices of an edge

Evidently, the distinguishing requirement here is the ability to split a cell — the rest just demands some fairly 'standard' incidence accesses. If we choose to build the refined grid in a data structure separate from the original one, we can reduce the requirements on the latter, using a data structure optimized for the algorithm to represent intermediary results.

3.2.7 Finite Volume Flux Calculation

A simple scheme for flux-balance methods basically has the following form (cf. page 152):

IN: a cell-based state $U : \mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: a cell-based flux-sum flux : $\mathcal{G}^d \mapsto \mathbb{R}^p$
1: **for all** cells $c \in \mathcal{G}$ **do**
2:    flux$(c) \leftarrow 0$

for all facets f of c **do**
  if f is on the boundary **then**
    add boundaryflux(c, f) to flux(c)
  else
    $n \leftarrow$ neighbor of c corr. to f
    add interiorflux(c, n) to flux(c)
  end if

Here boundaryflux and interiorflux are two routines calculating numerical fluxes over boundary and interior facets, respectively. Both use the state U. In general, if we have a flux F over an interior facet from cell c into cell n, then the flux from n into c is $-F$. Therefore, we can test if the flux already has been calculated, and use the result. One way to do this is to introduce an arbitrary order on the cells, and to calculate only the flux from the lesser into the greater cell, adding it with reversed sign to the flux of the greater cell.

Beyond this basic algorithms, there are methods to achieve higher order approximations. One example is the so-called *recovery* strategy, which constructs piecewise linear functions from constant cell values. A first step is volume-averaged interpolation of these values to vertices:

IN: a cell-based state $U_c : \mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: a vertex-based state $U_v : \mathcal{G}^0 \mapsto \mathbb{R}^p$
for all Vertices v of $\mathcal{G}$ **do**
  $U_v \leftarrow 0$
  $\Sigma_{\mathsf{vol}} \leftarrow 0$ *(sum of volumes of incident cells)*
  for all Cells c incident to v **do**
    add volume$(c) \cdot U_c(c)$ to $U_v(v)$
    add volume(c) to Σ_{vol}
  $U_v(v) \leftarrow U_v(v)/\Sigma_{\mathsf{vol}}$

A different way to do this is a loop over the cells instead of the vertices:

IN: a cell-based state $U_c : \mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: a vertex-based state $U_v : \mathcal{G}^0 \mapsto \mathbb{R}^p$
for all Vertices v of $\mathcal{G}$ **do**
  $U_v \leftarrow 0$
  $\Sigma_{\mathsf{vol}}(v) \leftarrow 0$
for all Cells c of $\mathcal{G}$ **do**
  for all vertices v of c **do**
    add volume$(c) \cdot U_c(c)$ to U_v
    add volume(c) to $\Sigma_{\mathsf{vol}}(v)$
for all Vertices v of $\mathcal{G}$ **do**
  $U_v(v) \leftarrow U_v(v)/\Sigma_{\mathsf{vol}}(v)$

Collecting the requirements of all algorithmic alternatives, we have the following:

1. iteration over all cells and all vertices of a grid

2. iteration over all vertices incident to a cell
3. iteration over all cells incident to a vertex
4. association of vector-values to vertices and cells
5. temporary association of scalar values to vertices ($\Sigma_{\mathsf{vol}}(v)$)
6. access to cell volumes
7. access to cell and facet centers of gravity, facet volumes and oriented normals (in interiorflux())

3.2.8 Finite Element Matrix Assembly

The construction of a stiffness matrix is a key component of the finite element method. The details essentially depend on the type of finite elements involved, that is, on the underlying function spaces. See section 6.3.1 for details.

The simplest case certainly are vertex-centered linear elements:

IN: a grid $\mathcal{G}$
OUT: a sparse matrix $A : \mathcal{G}^0 \times \mathcal{G}^0 \mapsto \mathbb{R}$

$A(v, w) \leftarrow 0 \quad \forall v, w \in \mathcal{G}^0$
for all Cells $c \in \mathcal{G}$ **do**
  for all Vertices v of c **do**
    for all Vertices w of c **do**
      add $a(c, \phi(v), \psi(w))$ to $A(v, w)$

In the special case of the Poisson equation (6.18), p. 161, $a(c, \phi(v), \psi(w))$ evaluates the integral

$$a(\phi_v, \psi_w) = \int_c \nabla\phi_v \nabla\psi_w \, dx$$

In general, this routine will use *local coordinates* of the vertices to evaluate a more complicated integral numerically.

In the general case, there will be ansatz function associated to *nodes* which can be located in any element type. One has to replace the double loop over vertices by a double loop over all nodes, which requires iteration over all element types incident to a cell. Certainly access to local coordinates will be required, too.

To sum up requirements:

1. iteration over cells of a grid
2. iteration over all element types incident to a cell
3. local coordinate systems for cells
4. assignment of unique numbers to nodes, that is to elements in the first place.

3.2.9 Summary of Requirements

We can group the requirements into a combinatorial (see table 3.3), a geometric (see table 3.4) and a data association (see table 3.5) section. To sum up the tables, the following set of combinatorial functionality is constantly recurring:

1. iteration over all elements (vertices, edges, cells) of a grid or a subrange of it, and queries about the number of elements,

2. iteration over all elements incident to some *anchor* element, e. g. all vertices incident to a cell, and queries to the number of such elements,

3. consecutive enumeration of elements,

4. subsets of elements, requiring minimal representation of elements belonging to one and the same grid,

5. random access to elements in the *closure* of a given anchor element, with respect to a combinatorial archetype of the anchor, for example the vertex sets of the two halfs of a bisected simplex, or (FEM) the local coordinates of nodes associated to the sides of a cell,

6. the association of many different types of data to elements, sometimes to all elements of a given type, sometimes only to a small part of them.

The geometric part of the functionality also has some recurring parts:

1. mapping of combinatorial entities to geometric entities, for example vertices to space coordinates, edges to curve segments, or faces to polygons,

2. calculation of geometric measures of the geometric images of elements, e. g. volumes, centers of gravity and facet normals.

In principle, most geometric measures could be calculated based on the geometric entities, but it seems wise not to explicitly require a conversion into the latter in order to calculate geometric measures. Moreover, some, as oriented (outward) normals are defined more uniquely based on combinatorial entities, in this case a facet with one of its two incident cells selected.

Depending on which type of functionality they use, we can classify algorithms as combinatorial (neighbor search, Cuthill-McKee), geometrical (graphics, refinement, flux, stiffness-matrix), or function-oriented (particle tracing). The latter class of algorithms assumes a (piece-wise) continuous function to be defined on the entire grid.

Algorithm	Grid	Element	global iteration	local iteration
File output	no. of vertices no. of cells	cell, vertex	cells, vertices	vertices/cell
Neighbor search		cell vertex hashable / comparable	cells	facets/cell vertices/cell
CuthillMcKee		cell: no. of neighbors		cells/cell
Graphics wireframe		edge	edges	
Graphics shrink		cell, facet, vertex	cells	facets/cell vertices/cell
Particle tracing	test if facet is on boundary	cells [facets]		cells/cell [facets/cell] [vertices/cell]
Refinement: Bisection		cells, (sets of), edges, vertices		edges/cell vertices/cell cells/edges vertices/edge
FV flux		cell, vertex	vertices, cells	vertices/cell [cells/vertex]
FEM stiffness matrix		cell, vertex [elements supporting Dofs]	cells	vertices/cell [elements/cell]

Table 3.3: Combinatorial requirements of algorithms

Algorithm	**vertex**	**edge**	**facet**	**cell**
File output	coordinates (R/O)			
Neighbor search				
CuthillMcKee				
Graphics wireframe		segment		
Graphics shrink	coordinates		polygons (write access to vertex coordinates)	solids center
Particle tracing			intersection with ray	point inclusion test
Refinement: Bisection		length		
FV flux			volume center normal	volume center
FEM stiffness matrix				local coordinates

Table 3.4: Geometric requirements of algorithms

Algorithm	**vertex**	**edge**	**facet**	**cell**
File output	[integer]			
Neighbor search				per facet: cell/facet pair
CuthillMcKee (cell-facet graph)				integer, bool
Graphics wireframe				
Graphics shrink				
Particle tracing	[vector]			[vector]
Refinement: Bisection		vertex handle		bool
FV flux	vector, scalar		[vector]	vector
FEM stiffness matrix	vector	[vector, nodes]	[vector,nodes]	[vector,nodes]

Table 3.5: Data association requirements of algorithms

3.3 Grid Data Structures

Having investigated a representative selection of grid algorithms, it is time to complement the study of mathematical properties of grids by a study of computational aspects of grid representations. This study will be guided by the findings of the grid algorithm domain analysis.

We first discuss fundamental variabilities in the implementation of grid data structures, and then compare existing approaches in the light of our analysis so far.

3.3.1 Variabilities of Grid Data Structures

In our previous analysis, it turned out that the requirements of grid algorithms can be divided into three large groups: combinatorial functionality (iteration), data association / storage on the grid, and geometric primitives, which we discuss in turn.

3.3.1.1 Combinatorial Variabilities

Underlying mathematical model. In section 3.1 we have seen that even the underlying mathematical structures vary considerably. For Cartesian grid, the combinatorial structure can be represented in an fully implicit manner. Furthermore, there is variation in whether only special combinatorial cell types (such as simplices) are to be represented, general convex polytopes or even more general types. These latter occur in Geometric Modeling, where cells with holes are commonly used, which are not even homeomorphic to unit balls. Moreover, often even unstructured grids contain some structure, for example in regions that have been regularly refined several times. This can be exploited in a clever way to save some storage, but restricting the admissible cell types to simplices [OR97].

Another common distinction is the question whether manifold-with-boundary grids can be represented or not — the latter is often the case in computational geometry, when grids represent surfaces of solids (B-rep).

The large majority of data structures can represent only grids of a fixed dimension; however, there are results on representing grids of arbitrary dimension, see [Bri89]. Most work in Computational Geometry is centered around two-dimensional grid.

Functionality. The operations on a grid representation can be separated into different groups:

- *Element access:* Which element types are defined? Can one access all elements sequentially? How much information is needed to uniquely determine an element of a given grid?
- *Incidence-queries:* Which questions of the type "all vertices of a cell" can be answered? See section 4.1.3 for a systematization of a certain class of incidence queries.

- *Global-property-queries:* Does the data-structure provide information about topological properties such as number of connected components, number of k-cells or boundary components?
- *Consistency maintenance:* Often, a data structure can represent a wider class of mathematical objects than it is supposed to hold, for example, a data type for storage of subdivided manifolds with boundary may contain a configuration as in figure 3.4 (p. 50). Are there operations to detect this kind of degeneracy?
- *Modification:* This encompasses global grid construction on the one hand, and incremental dynamical updates like deletion and addition of cells or *Euler operations* (see [Män83]), on the other hand. How is consistency ensured?

Efficiency. This is of primary concern for all contexts where the size of the grid dominates the computation time. Typically, there is a trade-off between the three criteria listed below.

- *Query efficiency:* How costly are incidence queries? This is not meant only in asymptotic terms, but also in the context of a concrete implementation on concrete hardware. Shewchuk [She96] reports a difference by a factor of 2 when replacing the grid data structure. However, the accumulated cost of queries is not a function of the grid alone, but also depends on the query structure of the algorithms using it, see also the benchmark result in section 6.5.
- *Storage efficiency:* How much memory is required for storing a grid? This depends to a good deal on the mathematical model, e.g. whether arbitrary cells are admitted or not, or if there is some regular or hierarchical structure in the grid, in which case all or parts of the incidence information may be deduced, see for example [OR97]. The supported query and modification functionality also influences required storage size. For simplicial grid, there exist various compression methods, see [Ros98].
- *Modification efficiency:* How much cost incremental or accumulated changes to the data structure? Cheap modifications are often contrary to query efficiency. In cases where there first is a building phase of the grid and afterwards only queries are performed, it may be better to copy the whole grid into another representation.

Implementation Complexity. Inherent complexity of data structures often hinders the production of correct implementations substantially, causing otherwise inferior solutions to be preferred. A further point is how well incremental development is supported, that is, whether functionality can easily be added at a later time.

The central question about grid data structures is, whether there exists a natural 'maximal' set of functionality (a *micro-kernel*), beyond which functionality can be implemented in a *generic* way. We will start to answer this question in section 4.2.

3.3.1.2 Data Association Variabilities

In a mathematical context, or when abstractly describing an algorithm, we often speak in a rather innocent manner of things like “Let f be a function on the edges of $\mathcal{G}$” or “mark all vertices incident to c as visited”. Because the mathematical concept of a function hides computational aspects of how these functions perform their mapping, it considerably eases reasoning about grids.

The underlying mathematical concept here is that of a function To make things more precise, we call *grid function* a function

$$f : \mathcal{G}^k \mapsto \mathcal{T}$$

mapping k-elements (the *element type* of f) of some grid $\mathcal{G}$ to objects of some type $\mathcal{T}$ (the *value type* of f). There are two different possibilities for f: Either, it is *total*, that is, defined on every element of $\mathcal{G}^k$, or it is *partial*, that is, defined only on a subset of $\mathcal{G}^k$. We will call such a function a *total* or *partial grid function*, respectively.

The *domain* of f, $\operatorname{dom} f$, is the set of elements where f is defined. Consequently, $\operatorname{dom} f = \mathcal{G}^k$ for a total grid function, and $\operatorname{dom} f \subset \mathcal{G}^k$ for a partial grid function.

Additional computational aspects must be taken into account:

- Is the mapping calculated or explicitly stored?
- How efficient is the access to a value?
- If stored, is there a possibility to change the value on a given element?
- If stored, how can one access the value of an element?
- If stored, will the function have to be defined *explicitly* for all k-elements of a given grid, or will there be only a small set of elements where it differs from a default value?
- Is there a possibility to create temporary grid functions? How many can coexist at any one time? Are there any restrictions on the value type $\mathcal{T}$?
- Are the preconditions for evaluation to be checked at runtime?
- What is the semantics of accessing the value of an element outside the domain of a partial grid function? Is it an error, is an invalid value returned, or can the user define a value to be returned?
- *dynamic behavior:* Does the grid function remain valid if the underlying grid changes? Is the underlying grid expected to be changed frequently?

The mathematical distinction between total and partial functions deserves some special attention. It is different from the question whether for a stored grid function all values have to be defined explicitly, but these aspects are nonetheless closely intertwined.

For a stored total grid function, the expected default behavior is relatively clear: It can be accessed at any element in $\mathcal{G}^k$, but the value is undefined if not explicitly set. For partial grid functions however, it is often convenient to define a default value for elements for which no *explicit* value has been defined.

To see this, consider the following common situation: In the high-level description of an algorithms, one finds a statement like "mark all vertices as non-visited". If the algorithm in question is sub-linear in the size of the grid, for example a local search, it is crucial that the action of "marking as non-visited" is $O(1)$. Therefore, what we want here is a grid function

$$f : \mathcal{G}^0 \mapsto \{\texttt{true}, \texttt{false}\}$$

which has *total* function semantics for read access, but *partial* function semantics for write access.

It turns out that this behavior is a very commonly desired one, and it is rather easy in practice to obtain *partial* function behavior for read access if desired, e. g. by setting a unique, invalid default value. Moreover, an implementation of such a partial grid function in general has a means to determine whether an element belongs to the domain or not, which can be used by a client.

The distinction between totally and partially defined grid functions seems to be a fundamental one; typically, it can be unambiguously determined whether a given function is total or partial. Therefore, it is crucial to provide implementations for both of these *total* and *partial definition* traits; giving rise to *total* and *partial* grid functions.

Other important differences between partial and total functions include the space-runtime tradeoff (partial functions use less space but are typically somewhat slower) and extra functionality: In a clever implementation, it should be possible to get the *domain* of a partial grid function f, that is, the set of elements where f is *explicitly* defined.

3.3.1.3 Geometric Variabilities

In section 3.1.4 we have seen how combinatorial and geometrical aspects of mathematical grids cleanly separate. Earlier in this chapter, we noticed that some algorithms did not use any geometric information at all.

It is both straightforward and useful to keep this separation also in the field of software components: *Straightforward*, because the inherent combinatorial structure is not touched upon by a geometric embedding. *Useful*, because this separation makes for focused components, enhanced reuse and clean classification of algorithms, depending on whether they use geometric information or not.

There are the following possibilities for mathematical embeddings:

- The type of the topological space X: It may be $\mathbb{R}^n$ for some $n \geq d$ (where

$n = d$ is the most common case), a manifold embedded in $\mathbb{R}^n$, or something more complicated.

- If $X = \mathbb{R}^n$: The mapping of archetypes into $\mathbb{R}^n$ may be chosen very differently. In the easiest case, it may be linear, mapping each entity into affine subspaces of $\mathbb{R}^n$, but also arbitrarily curved embeddings are possible.

The computational aspects are especially rich for grid-geometries:

- *Functionality:* What type of geometric information is available? See table 3.6 for a (restricted) overview.
- *Efficiency trade-offs:* How is the trade-off between calculation and storage for these primitives managed?
- *Exactness:* Does the component perform exact computations of primitives as opposed to numerical approximations? This is particularly important for nonlinear embeddings.
- *Arithmetic:* Does the geometry use exact or floating-point arithmetic?
- *Dynamic behavior:* What happens if the underlying grid changes?
- *Mutability:* Is is possible to incrementally change the geometry?
- *Exclusivity:* Can different geometries be used in parallel for one and the same grid?

The question of used arithmetic is of particular importance for the class of applicable algorithms: Many methods of Computational Geometry assume exact predicates, for example if a point lies on, above or beneath a hyperplane, and may fail otherwise. On the other hand, numerical methods tend to be less sensitive for the imprecision inherent to floating-point arithmetics. Here performance issues are paramount. Decisions based upon arithmetic results are made less frequently and with less urge for consistency than in geometric computing. Consequently, different geometry components may be required when using both types of algorithms on a grid.

Adjusting the trade-off between storage and calculation offers an almost continuous spectrum of possibilities. They range from Cartesian geometries, where everything (including vertex coordinates) can be calculated on the fly or is constant (volumes), to nonlinear embeddings, where all required quantities might be computed in advance. Here fine-tuning for a given application may prove worthwhile without being overly complex.

The set of operations that may be offered by grid-geometry components is vast. First of all, a geometry can define types that represent the geometric counterparts of the combinatorial elements: Some type of n-dimensional vector for vertices, some kind of segment or arc for edges and so on, together with mappings from combinatorial to

operation				restriction
geom_0	: vertex	$\mapsto$	point	
geom_1	: edge	$\mapsto$	segment	
geom_2	: face	$\mapsto$	polygon	
geom_{d-1}	: facet	$\mapsto$	hyper-polygon	
geom_d	: cell	$\mapsto$	polyhedron	
volume_1	: edge	$\mapsto$	$\mathbb{R}$	X is measurable
volume_2	: face	$\mapsto$	$\mathbb{R}$	X is measurable
volume_{d-1}	: facet	$\mapsto$	$\mathbb{R}$	X is measurable
volume_d	: cell	$\mapsto$	$\mathbb{R}$	X is measurable
center_1	: edge	$\mapsto$	$\mathbb{R}^n$	geom. is linear
center_2	: face	$\mapsto$	$\mathbb{R}^n$	"
center_{d-1}	: facet	$\mapsto$	$\mathbb{R}^n$	"
center_d	: cell	$\mapsto$	$\mathbb{R}^n$	"
normal	: (cell, facet)	$\mapsto$	$\mathbb{R}^n$	also $n = d$

Table 3.6: Some basic primitives for grid geometries

geometric entities. Further, there are *measuring* operators: The lengths of edges, area of faces and volumes of cells. (For sake of simplicity, we identify in this section the combinatorial elements with their geometric images.) Next come center points, normals and more. These, however, do not make sense any more in each possible case. What is the normal of a facet of a 2D-grid embedded in $\mathbb{R}^3$? Or the center of a cell, if this embedding is not linear? Table 3.6 gives an overview, along with possible restrictions.

3.3.2 Review of Grid Data Structures

Due to the diverging needs, different fields have developed different types of data structures. In *Computational Geometry*, most work has focussed on the representation of subdivided 2-dimensional manifolds, sometimes with, sometimes without boundary.

One of the best-known is the Quad-Edge data structure by Guibas and Stolfi [GS85]. It represents the subdivision of a manifold and its dual in a symmetric way. As a consequence, vertices and facets are interchangeable and cannot be distinguished syntactically. Cells (faces) may be arbitrary simple polygons that need not be regular, that is, a vertex may occur several times on the boundary of a face.

Other common representations for 2-manifolds include the Winged-Edge and the Half-Edge data-structure. For an overview and discussion, see [Ket97]. The incidence-queries offered by these implementations rely on the ordering of edges around faces and vertices and give access to the next and previous edges at both vertices. Laszlo and

DOBKIN introduce in [DL89] a 3D-variant of this approach, where the ordering of faces and cells around edges is exploited. BRISSON generalizes these order relations in [Bri89] to arbitrary dimensions, using the diamond-property of the element-lattice, see section 4.1.3.3. He proposes to store the necessary information in a graph of so-called *cell-tuples*. A cubic cell would then be represented by $6 \cdot 4 \cdot 2 = 48$ nodes, each connected to 4 other nodes, making a total of $48 \cdot 4 = 192$ links, and a tetrahedron would still require 96 links.

A common feature of all these approaches is the easy access to directly incident elements, but queries that produce, say, all vertices of a given cell are not so obvious, especially in higher dimensions. The latter access pattern, however, is much more typical in numerical simulation algorithms than in computational geometry.

In *Geometric Modeling* [Mor97], one of the most employed data-structures are *Boundary Representations*, or *B-rep*s for short. If the solid to be modeled contains inner boundaries, that is, consists of different regions to be distinguished, the corresponding B-rep represents a non-manifold grid, because edges may be incident to more than two faces.

In *numerical simulation*, a grid typically is a subdivision of a solid domain, that is, a manifold with boundary. Typically, cell types are restricted to one or a few fundamental convex polytopes, like triangles or quadrangles in 2D and tetrahedrons, prisms or cubes in 3D. For FEM-algorithms, there have to be defined function spaces of polynomials of low degree and local support over the grid, which severely constrains the possible cell shapes ([Wie97]). Other approaches, such as the Finite-Volume-Method (FVM), are less restrictive on the type of cells.

Numerical algorithms operating on grids often access data in a cell-based fashion, requiring for example all vertices or facets of each cell. Therefore, corresponding grid data structures normally are *cell-oriented*, too. They mostly consist of lists of cells, each with references to its vertices and cell-neighbors. While these data-structures are adapted to numerical algorithms, it is interesting that they may also be used successfully in Computational Geometry: SHEWCHUK reports in [She96] a 2-fold speedup in an implementation for Delaunay triangulations after switching from a QUAD-EDGE-like to a cell-based data structure.

For Cartesian grids, finally, incidence information is fully implicit. Grid elements are represented by d-dimensional integer indices, and incidence queries reduce to simple index manipulation.

The support of modifying operations is another marked difference between grid representations in Computational Geometry and Numerical Simulation. The former typically offer some variants of Euler operators, such as attaching, splitting and deleting cells, whereas the latter in general only allow refinement and derefinement operations. Refinement replaces a cell with a fixed pattern of smaller cells, by so-called *refinement rules*. These typically contain *closure rules* ensuring matching subdivisions in adjacent cells. Some well-established refinement procedures are *red-green-refinement* [Ban98] and bisection-based approaches [Joe95].

Again, function spaces defined on the grid are a main source of constraints here. Also, refinement is often used in combination with hierarchical techniques such as multigrid methods which make a proper nesting of grid levels necessary.

Grid generation software lies somewhere in between Numerical Simulation and Computational Geometry. It often uses other types of data structures than the numerical algorithms.

Grid functions There are different ways of dealing with grid functions. In numerical analysis, one routinely employs d-dimensional arrays for Cartesian grids.

If unstructured grids are used, the needed data is usually stored in the elements themselves. This is also true for non-permanently needed data, such as temporary flags. The burden of keeping track of which data is used and which is not then is attributed to the algorithm implementor. Also, in the design of the grid data structure, it has to be anticipated which data is going to be used, thus introducing a strong coupling between data structures and algorithms: The data structures have to know which algorithms are to be executed on them!

This 'data-in-element' approach has however the advantage of making dynamic grids somewhat easier to handle, because the lifetime of data is coupled to that of the underlying element. It has the disadvantage of requiring the permanent storage of all element types, even if data is going to be stored only temporarily on them. Worse, it often entails the permanent storage of temporary data, possibly becoming a serious performance bottleneck.

Attempts to remedy this situation include usage of untyped pointers, being under the responsibility of the client, and introduction of multi-purpose fields. The latter approach is for example taken in the Stanford Graph Base ([Knu93]). All of these techniques make it more difficult for a client to use the data structures correctly. Moreover, there is no support for *partial* grid function — if a default value is to be set, *every* element has to be accessed.

Grid geometries These are almost always only *implicitly* represented, giving access to vertex coordinates, and assuming a linear embedding. This makes it hard to use different geometries, because it is difficult to track down every use of these implicit assumptions: The decision which geometric embedding to use is not localized, but smeared across all components using geometric information.

Grid geometries are often fused with their underlying grids, preventing the simultaneous use of different geometries.

3.3.3 Discussion

In a previous section, we have studied and classified requirements of typical grid algorithms, leading to a first coarse division into combinatorial, geometric and data association aspects. Then, we investigated the variances and trade-offs of grid data structures, and finally reviewed existing implementations of such structures in the light of our findings.

Notwithstanding the apparent diversity of these data structures, it is clear that they all represent strongly overlapping classes of mathematical grids, as studied in section 3.1. Therefore, algorithms operating on such a mathematical grid should in principle be applicable to a large portion of these data structures.

Most of these data structures can actually deliver *in principle* the necessary functionality, with some restrictions regarding grid functions, in particular partial grid functions. In other case, it is clear how the functionality could be added, possibly by adding auxiliary data structures.

However, the actual manner in which this functionality is delivered varies considerably. Often, data abstraction is not well developed. In particular, grid functions and geometries often fail to be recognized as abstractions *separate* from combinatorial grids.

Frequently, one can observe that data structures stemming from a certain domain, for example FEM simulation or Computational Geometry, are geared towards the needs of typical algorithms in that domain: Operations heavily used by these algorithms are supported in a comfortable and efficient manner, yet others have no or only weak support, albeit it would be possible to do better.

Even within a single domain, however, there is seldom such a thing as a 'best' data structure. The decision which to choose depends on the set of algorithms (*algorithmic profile*), as well as on an actual or representative set of data — the same data structure may perform well for small or regular data sets, and bad for large or irregular ones.

Once again, then, it proves necessary to separate representational issues from algorithmic ones, a topic in favor of which has been argued at length in section 2.3. A common platform of functionality is therefore desirable and possible, as an outcome of the overlap pointed out in the preceding sections. This will be the task of the next chapter.

Chapter 4

Components for Grids

Alles Festlegen verarmt.
Christian Morgenstern

We have argued for both the potential *gain* and the principal *feasibility* of a common framework for the functionality of grid data structures. Now the time has come to set down the details of such a common kernel. On the way, we will decide which set of functionality belongs to the *innermost* layer — a *micro-kernel* that will have special implementations for each concrete grid type; and, in contrast, what can be implemented generically in an *outer* layer. The micro-kernel will closely correspond to the actual grid concept in the mathematical sense, and anything else will be cleanly separated. This approach will lead to *scalable* grid software (as defined on p. 20): The implementation effort for the micro-kernel functionality remains fixed, everything beyond that can be implemented generically.

This chapter is thus divided into two major sections: A first one describing the special parts belonging to the micro-kernel, and a second one, describing *generic components* built on top of this micro-kernel. Here special emphasis is on components which are general purpose enough to serve as basis for many higher-level algorithms. Some of these components can even be used to implement micro-kernel components, thus reducing the labour of creating new kernel-compliant implementations.

4.1 A Micro-kernel for Grid Data Structures

4.1.1 What Belongs into a Micro-kernel?

One point has to be clear right from the start: *One cannot prescribe a common set of functionality that has to be supported by all grid data structures.* If this set would be large enough to be of any use, some concrete implementations would be excluded without necessity. On the other hand, the intersection of functionality over all possible and useful grid representations is essentially empty.

In fact, the generic approach works the other way around: Starting from algorithms, one derives sets of requirements for these. This has been done in section 3.2 in an exemplary manner. Common patterns of requirements can be formalized to yield *concepts*. A concrete type is a *model* of a concept if it fulfills the requirements bundled in the concept. Concrete types are often models of several concepts at once.

A type can be used as a parameter of a generic component, if and only if it is a model of all requirements the component imposes on that parameter. For example, in order to apply the CELL NEIGHBOR SEARCH algorithm (p. 61) on a grid type, the latter must allow iteration over cells, as well as over facets incident to a cell and vertices incident to a facet. Expressed with the concepts[1] we are going to present, the grid type must be a model of Cell Grid Range (→ p. 210), its cell type must be a model of Facet Grid Range (→ p. 210), and its facet type must be a model of Vertex Grid Range (→ p. 210).

There is a considerable number of such concepts necessary to describe grid functionality adequately. Semi-formal definitions of some of these concepts are given in the appendix 7.3. They would disturb the flow of presentation here, because they must inevitably introduce some elements of arbitrariness. The description of the concepts is based on C++ syntax and follows the example of the SGI STL documentation [sgi96]. The grid concepts can be used to document generic components, see e. g. the generic implementation of CELL NEIGHBOR SEARCH in the appendix (C.1).

The *micro-kernel* is defined as a *collection of concepts*; a concrete grid type typically implements a subset of them. For example, there is a concept for iteration over edges incident to a vertex, yet many grid data structures do not support this type of query and therefore leave it out.

Now, *which* concepts belong to this kernel? A first aim when developing a *micro*-kernel which merits its name is evidently to keep it minimal or at least small. The more grid functionality can be provided by generic components building *on top* of the kernel, the better.

Beneath a certain level, however, it turns out that too many specializations are necessary to exploit specific knowledge on the underlying data representation. An obvious 'lower bound' for a micro-kernel is the ability to reproduce a grid in another representation — to implement a copy mechanism. On top of that, other components *could* be defined generically. As an example, consider a grid offering iteration over cells, as well as facets and vertices of cells. From this information, cell neighbor information can be generated by CELL NEIGHBOR SEARCH if needed, at least if the poset of the grid is a lattice. However, if the grid already stores this information, or even can derive it (Cartesian grid), this approach is clearly wasteful. Therefore, iteration over cell neighbors should be considered a kernel concept.

On the other hand, if we look at structures like subranges of grids (sec. 4.2.1), it seems rather improbable that much can be gained by providing special implementations for each grid type. Even in the case of Cartesian grids, *general* subranges cannot be

[1] We write concepts in Sans serif font

implemented more efficiently than with generic components shown below. *Cartesian* subranges of course play a special role here.

We are now ready to discuss the micro-kernel in detail. The coarse dissection of grid functionality into combinatorial, geometric and data association aspects, observed in the analysis of algorithms, is also our fundamental classification principle for the micro-kernel concepts.

Certainly, the most complex part is the *combinatorial* one. Here we distinguish *grid elements*, directly corresponding to the k-elements of section 3.1, *element handles* which provide a minimal representation of grid elements, and *sequence iterators* which allow to view a grid as sequence of its elements, all discussed in section 4.1.2. The incidence information contained in the element poset (see page 52) is encapsulated by *incidence iterators* (section 4.1.3).

The *data association* aspect is captured by *grid functions* (4.1.4), as introduced above. They give rise to a set of carefully layered concepts.

Geometric embeddings are represented by dedicated *grid geometry* components (4.1.5), which introduce additional concepts for geometric entities corresponding to the combinatorial grid elements.

Mutating combinatorial operations play a much lesser role then could be expected. In practice, we can often do with primitives doing coarse-grained operations, such as copying a grid, enlarging a grid and removing pieces of a grid. This is discussed in 4.1.6.

Some *concrete* grid types (models of some micro-kernel concepts) are discussed in section 4.1.7.

If one compares the list of requirements on page 69, with the concepts we will introduce, one can state that all requirement are represented by appropriate concepts, except those related to detailed knowledge of the *archetype*, such as local coordinates or simplex bisection. This is still a topic for future work, but it seems probable that the notion of an archetype is the right abstraction.

For illustrating purposes, we sometimes show pieces of program code using C++ syntax. Nevertheless, it should be clear that the concepts introduced are programming language independent.

4.1.2 Grids, Elements and Handles

What remains if we forget all about the incidence structure of a grid $\mathcal{G}$? There will be nothing more left than the bare sets of grid elements, that is, the sets $\mathcal{G}^0$, $\mathcal{G}^1$, ... , $\mathcal{G}^d$.

In order to distinguish between elements of different dimension, types for different dimension k, $0 \leq k \leq d$ are required to be different: There are concepts (Grid) Vertex (→ p. 203), Edge and so on, formalizing the notion of k-dimensional elements. Note that some elements, like Facet and Cell (→ p. 205) are named according to *co-dimension*, instead of dimension. This allows to formulate many algorithms in a *dimension-independent*

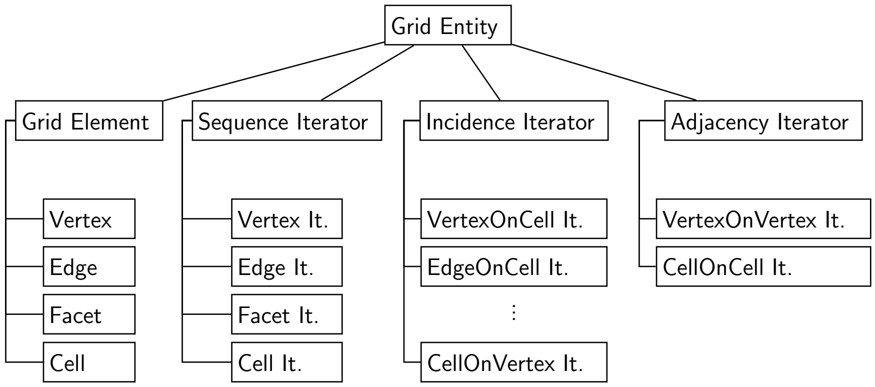


Figure 4.1: Overview on some concepts

manner, for example the basic flux calculation example on page 66, which is naturally expressed in terms of co-dimension.

Instances of elements must be *equality comparable*, and this comparison has to distinguish different elements, even from different grids. Furthermore, given such an element, it must be possible to deduce which grid the element belongs to. This grid we call the *anchor* of the element. In the context of incidence iteration (section 4.1.3.1), we will encounter anchors that are not grids, but other elements.

The element concepts are complemented by Element Handles (→ p. 205). Handles are a sort of index for elements and must also be equality comparable. However, the comparison is required to distinguish elements only within the same grid. Handles are not required to know their underlying grid; they are thought of as a *minimal* representation with regard to storage. Thus, they allow efficient implementation of element sets and grid morphisms (see p. 54). In the simplest cases, handles are just integral or pointer types.

There has to be a bijection between elements and pairs of grids and handles: An element must be able to get its handle, a grid has to produce an element from a handle.

As noted at the beginning of this section, a grid can be seen as a collection of up to $d + 1$ sequences of the different element types. This leads to the definition of Element Grid Range concepts: For instance, a grid type is a model of Vertex Grid Range (→ p. 210), if it provides a vertex type, itself model of Grid Vertex (→ p. 203), a corresponding handle type, and, most importantly, a possibility to iterate over its vertices by means of a Grid Vertex Iterator (→ p. 206). We require a Vertex Grid Range to 'know' the number of vertices it contains.

The family of such iterators is called (Grid) Sequence Iterators (→ p. 205). A grid type is in general model of several different Element Grid Range concepts, but it is not required that it be a model of *every* possible element range. For example, we will see below (sec. 4.1.7.1) examples for grids defining an interface for simple serial grid representations, which are models of Vertex Grid Range and Cell Grid Range, but not of Edge Grid Range nor Facet Grid Range; more precisely, they are models of Cell-Vertex Input Grid Range (→ p. 212), a concept that simply bundles some common set of requirements.

This certainly restricts their usability; however, as their major purpose is to encapsulate the details of the serial representation, they are used mainly to create a copy of this representation in a more powerful data structure. Thus, it does not pretend to be more than it actually is, it only gives access in a unified manner to the information readily available in the representation.

Up to now, we did not talk about a Grid concept. This is in part due to the fact that not much can be prescribed for grids in general, and the most important aspects are captured by the Element Grid Range concepts. The difference between grids and grid ranges is rather subtle. It has to do with incidence iteration, and will be discussed in section 4.2.1. The corresponding Grid (→ p. 211) concept introduces (almost) no additional functionality, it just adds constraints.

The definitions presented so far define the bottom layer of a grid micro-kernel, where it cannot really be spoken of a grid. However, even at this stage, there are algorithms — sequence algorithms rather than grid algorithms — that do not require more functionality than is present. First of all, it is possible to count the number of elements of every dimension, and to deduce the Euler characteristic (see page 55) as a global property of $\mathcal{G}$. If a *grid geometry* (section 4.1.5) Γ is available, we may calculate the extent of the geometric embedding $\|\mathcal{G}\|_\Gamma$, find vertices with extremal coordinates and perform similar tasks. However, depending on the implementation of the geometry, most geometrical functions require a minimal set of incidence information.

An implementation for calculating the bounding box of a grid (of type `Grid`) and a grid geometry (of type `Geom`) could look like the following:

```
Grid G;
Geom geo(G);
// ...
Geom::coord_type
   ll( infinity(), infinity()),  // lower left corner
   ur(-infinity(),-infinity());  // upper right corner
for(Grid::VertexIterator v = G.FirstVertex(), v != G.EndVertex(); ++v) {
  Geom::coord_type pv(geo.coord(*v));
  ll.x() = min(pv.x(), ll.x());
  ll.y() = min(pv.y(), ll.y());
  ur.x() = max(pv.x(), ur.x());
  ur.y() = max(pv.y(), ur.y());
}
```

Looking at the requirements of this little algorithm, we see that `Grid` must be a model of Vertex Grid Range, which implies the existence of the type `Grid::VertexIterator` which is a model of Grid Vertex Iterator. The type `Geom` must be a model of Vertex Grid Geometry, that is, must export a type `Geom::coord_type` and have a method `Geom::coord(Grid::Vertex)` returning the coordinates of a vertex.

The algorithms works fine as long as the geometry is a *linear* one (see page 58), for curved cells it would not be correct.

One could also consider using a generic algorithm for bounding boxes, operating on an arbitrary sequence of points. This would require to pass an iterator adapter to this algorithms, which makes the grid 'look like' a set of points. Because a grid is not an arbitrary set of points, however, specializations are possible. For example, if the dimension of the geometric space equals the combinatorial dimension of the grid, it suffices to examine only vertices on the boundary.

It should be highlighted that the requirements mentioned above do *not* mean that each grid element must be stored *permanently*. Besides the already mentioned cases of (semi-)regular grids, which enable (partially) implicit representation, permanent storage is not mandatory even for unstructured grids. We can manage with storing only cells and vertices, representing elements like edge and facets implicitly, and at the same time assuring fully transparent access to them. This is for example done in the `Triang2D` grid type (B.1).

The possibility of representing elements implicitly has implications for element equality comparison. The equality of two objects of an element type does not imply that they contain the same data, rather, it means that they denote the same 'mathematical' element. For elements not stored permanently, we cannot rely on *object identity* (a unique memory reference) to ensure the correct semantics of equality test.

If, for example, vertices are stored, they may be uniquely identified by their number or memory location. On the other hand, we may choose not to store facets (edges in 2D) permanently, but to create them 'on the fly', represented by sets of vertices or by pairs of cells and local facet number. The (cell,local facet) pair is not a *unique* representation: For each inner facet, there are two different possibilities. Thus, to distinguish facets in this case, we need to compare a unique representation. The ordered pair of vertices is unique, but it is *unambiguous* only if each element is determined by its vertex set. This is the case if the grid poset is an atomic lattice (see p. 53). Only in this case, the implicit representation is justified.

4.1.3 Incidence Queries

Access to incidence information is the central part of grid data structure functionality. We can identify two different types of incident queries, one that asks for *all* elements of a specific dimension incident (or adjacent) to a given element, and another that gives access to a neighborhood by local operations, ignoring element type.

Although the operations of the second type may be used to emulate the first-mentioned ones, it is in general not straightforward and also not so efficient to do so. Because accessing *all* elements of a *given* type is a common recurring pattern in many algorithms, we chose this as a basis for incidence-based functionality, presenting the local operators as an alternative and complement (section 4.1.3.3).

4.1.3.1 Incidence Sequences and Iterators

Incidence sequences build on the enumeration of a cell's sides provided by its archetype. For non-regular elements, there is the problem of repetitions: Elements that are different in an archetype may be mapped to the same element in the grid. It seems more natural to count these elements several times, because this allows a consistent treatment of regular and non-regular elements. Also, variants which count such an element only once can be more easily derived from the 'multiple count' approach (by a simple marking) than vice versa.

Definition 24 (downward incidence sequence, anchor). A *j-k-downward incidence sequence* $\mathcal{I}_{j\to k}$ for $j > k$ is a mapping

$$\begin{aligned} \mathcal{I}_{j\to k} : \mathcal{G}^j &\mapsto \operatorname{seq} \mathcal{G}^k \\ e &\mapsto (\Phi(\mathfrak{a}_1^k), \ldots, \Phi(\mathfrak{a}_{n_k}^k)), \quad \Phi = \Phi_e, \mathfrak{a} = \mathfrak{a}_e \quad \text{archetype of } e \end{aligned}$$

which for an j-element e produces the sequence of incident k-elements (the term $\operatorname{seq} \mathcal{G}^k$ means the set of finite sequences over $\mathcal{G}^k$). The element e is the *anchor* of the sequence $\mathcal{I}_{j\to k}$. Given an element $e \in \mathcal{G}^i$, we often abbreviate $\mathcal{I}_{j\to k}(e)$ to $\mathcal{I}_k(e)$.

In the case of 'upward' incidences, for example cells incident to vertices, we have in general no archetype for the lower-dimensional anchor. We can, however, use a local view of the neighborhood to define incidence sequences. There is an archetype for each incident higher-dimensional element, which may contain the anchor several times, and should accordingly be present in the incidence sequence of the anchor the same number of times:

Definition 25 (upward incidence sequence). A *j-k-upward incidence sequence* $\mathcal{I}_{j\to k}$ for $j < k$ is a mapping

$$\begin{aligned} \mathcal{I}_{j\to k} : \mathcal{G}^j &\mapsto \operatorname{seq} \mathcal{G}^k \\ e &\mapsto (f_1, \ldots, f_n) \end{aligned}$$

where each element $f \in \mathcal{G}^k$ with $f > e$ appears as many times in $\mathcal{I}_{j\to k}(e)$ as e appears in $\mathcal{I}_{k\to j}(f)$.

An example for repetitions is the torus of fig. 3.1 on page 46, where the only cell occurs four times in $\mathcal{I}_2(v)$.

These definitions lead to algorithmically simple solutions that work also for non-regular grids: For example, we can obtain the upward incidence sequences $\mathcal{I}_{k\to j}$ from the knowledge of the sequences $\mathcal{I}_{j\to k}$ by a global loop over j-elements, appending the current j-element to the sequences of all incident k-elements.

Just like ordinary sequence lead to the iterator concept in the STL framework, incidence sequences lead to Incidence Iterator (→ p. 206) concepts, for example VertexOnCell Iterator (→ p. 207) (fig. 4.2) or FacetOnCell Iterator (→ p. 207), to name two of the most heavily used ones.

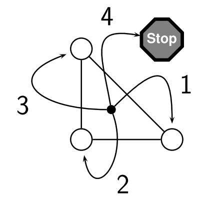


Figure 4.2: Action of a VertexOnCell Iterator (*Incidence iterator*)

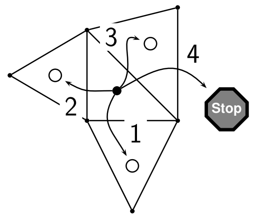


Figure 4.3: Action of a CellOnCell Iterator (*Adjacency iterator*)

A simple example demonstrating the use of incidence iterators is a loop calculating, for each vertex, the number of incident cells, and storing the result in a grid function.

```
Grid G;
grid_function<Grid::Vertex,int>  nc(G,0);
for(Grid::CellIterator c(G); ! c.IsDone(); ++c)
  for(Grid::VertexOnCellIterator vc(*c); ! vc.IsDone(); ++vc)
    nc[*vc]++;
```

In this case, `Grid` must be a model of both Vertex Grid Range and Cell Grid Range, and `Grid::Cell` must be a model of Vertex Grid Range, which implies the existence of a type `Grid::VertexOnCellIterator`. Grid functions are introduced below, in this case, the template `grid_function<Grid::Vertex,int>` is required to be a model of Total Grid Function (→ p. 215).

4.1.3.2 Adjacency Iterators

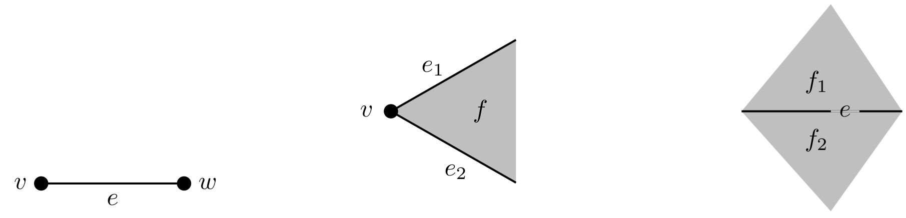
A *k-k-adjacency sequence* ($k = 0$ or $k = d$) is much the same as an incidence sequence, only that it contains elements *adjacent* (p. 48) to the anchor. Again, duplicates are allowed, which can exist even for regular anchor elements, for example the boundary of an open cylinder formed by two logical squares. If the grid poset is a lattice, no duplicates can occur.

For a d-d-adjacency sequence, we require the ordering of adjacent cells to be the same as in the corresponding facet sequence, but it may contain fewer elements, if facets are on the boundary and hence do not correspond to an adjacent cell.

The corresponding iterator concept is Adjacency Iterator, an important example being CellOnCell Iterator (→ p. 207), see figure 4.3.

4.1.3.3 An Alternative: The switch Operator

The *switch*-operators introduced by Brisson [Bri89] provide a unifying and dimension-independent approach to similar operations for 2D and 3D grids, see for example Guibas and Stolfi [GS85] or Dobkin and Laszlo [DL89]. They rely on the grid poset having the *diamond-property* (see page 53) and are defined as follows:



(a) $w = \texttt{switch}(v, e)$ (b) $e_2 = \texttt{switch}(v, e_1, f)$ (c) $f_2 = \texttt{switch}(e, f_1)$

Figure 4.4: The `switch` operations in two dimensions

Definition 26 (switch operation). If $e_{k-1} \prec e_k \prec e_{k+1}$, then the interval (e_{k-1}, e_{k+1}) contains two elements, say e_k and $e_k^{\Diamond}$. The `switch` operator is then defined by

$$\texttt{switch}(e_{k-1}, e_k, e_{k+1}) = e_k^{\Diamond} \qquad \text{for} \quad 0 \leq k \leq d$$

Here dimension is denoted by subscripts. If $k = 0$, then e_{-1} is understood to be the improper element $\hat{0} = \emptyset$, and the `switch`-operation flips the vertices of the edge e_1. Likewise, e_{d+1} is $\hat{1} = \mathcal{G}$, and $\texttt{switch}(e_{d-1}, e_d, e_{d+1})$ returns the other cell adjacent to the facet e_{d-1}. Improper elements are generally suppressed in the argument list of `switch`.

A manifold-with-boundary grid does not have the diamond property for boundary facets. In this case, we can agree to define one (or more) artificial 'outer' cells, identified by a special value. In general, it is an error to do anything with an outer cell, except testing it for validity. There can, of course, be designed data structures which make it possible to work with such a 'cell'. However, as these cells typically contain much more facets than interior cells, their implementation would differ considerably from that of other cells. In fact, it would contain the whole boundary surface grid. In practice, it seems more convenient to deal explicitly with the boundary, if the corresponding functionality is needed. Anyhow, many algorithms (especially numerical discretizations) have to take special action here.

Most algorithms do not use the *switch* operator. An exception are boundary iterators, section 4.2.2.4.

4.1.4 Grid Functions

We have seen that an essential part of algorithm requirements is the access to data associated to grid elements. If we look a little bit closer, we can see that the actual requirements can be described by a hierarchy of related concepts.

The most basic concept for grid functions is that of an Grid Element Function (→ p. 212): It simply maps elements of a given type to some other type. This is a refinement of the STL Adaptable Unary Function concept. An example would be a simple adapter which returns the number of vertices of a cell.

The next refinement is the Grid Function (→ p. 213) concept: We bind the grid function to a specific grid. This makes it possible to iterate over the domain and the range of a grid function. If values can be changed, we speak of a Mutable Grid Function (→ p. 214).

The concepts discussed so far contain no provisions to *create* grid functions, for example for storing temporary values specific to an algorithm. The Container Grid Function (→ p. 214) concept adds this functionality.

We have pointed out earlier (page 76) the fundamental difference between total and partial grid functions. Correspondingly, we refine Container Grid Function into Total Grid Function (→ p. 215) and Partial Grid Function (→ p. 216). A simple example is found on page 90 above, generic implementations are discussed in section 4.2.3.

4.1.5 Grid Geometries

A grid geometry is the software equivalent to the mathematical concept of a geometric realization of a combinatorial complex. Many of the mathematical as well as the computational variabilities have been analyzed in section 3.1.4. There are many decisions to be taken here; a grid geometry is the natural place to *localize* them.

The basic concept is that of a Vertex Grid Geometry. It simply provides a mapping from the vertices of a grid $\mathcal{G}$ into some topological space X. Typically, this space is $\mathbb{R}^n$ for some $n \geq d = \dim(\mathcal{G})$. The dimension of X is called *exterior* dimension the geometric embedding. The Vertex Grid Geometry concept provides an associated type that is a computational representation of a point of X, for example a n-dimensional vector if $X = \mathbb{R}^n$, see the example on page 87.

Although access to vertex coordinates is the only geometric functionality commonly supported by existing data structures, it is hardly sufficient for even the most simple algorithms. The solution of this apparent paradox is, of course, that many of the decisions mentioned before 3.3.1.3 are *implicit* in the client code. For example, if we simply take the distance of the two endpoints, in order to calculate the length of an edge, we make first the assumption that edges are linear segments, and second, depending on the details of distance calculation, we often assume a vector space structure of X. Albeit reasonable in the most important practical cases, these assumptions to not hold in general: Just think of a grid used to model the surface of the earth, where edges are geodesics. Moreover, it could be that edge lengths are used so often that it pays off storing them permanently. Client algorithms should not know about this.

It is clear from section 3.2 that most algorithms use slightly more geometric information than just vertex coordinates. The next step is the mapping of all combinatorial elements to geometric objects: Edges to segments, faces to polygons, and cells to d-dimensional polytopes. In general, these objects can be arbitrarily curved and need only be homeomorphic images of some 'linear' archetype.

If in addition a geometry supports volume measurement of these entities, we speak of a Volume Grid Geometry. With 'volume' we mean here the k-dimensional measure of a k-dimensional entity, that is, length of edges, area of faces, etc.

Beyond this, many different pieces of geometric information are needed by algorithms. Among the more frequent ones are centers (of inertia) and normals to facets.

But here already difficulties arise: What is the center of a curved segment? Is it the center of inertia, which in general lies outside? Or is it the center with respect to same parameterization, for example arclength? Answers will typically depend on the context — for example, whether the 'center' is needed for bisection (grid refinement) or numerical integration. It seems therefore wise to use more specific names for these entities, depending on the context. Unfortunately, this approach considerably rises the number of functions to be defined for geometries.

For linear geometries, on the other hand, all these possible interpretations lead to the same value, and can be deduced directly from the functionality defined by the Volume Grid Geometry concept. Therefore, this common case can be handled essentially by a generic component, discussed later (page 105).

4.1.6 Grid Creation, Copy and Modification

It is a common observation that mutating operations on data structures tend to be more difficult than non-mutating ones. This is evidently also the case for grids.

Modifying operations often depend deeply on the internal representation. Also, the amount of 'dynamic change' that a data-structure can support in an *efficient* manner varies considerably, from atomic operations (such as *Euler operators*) over coarse-granular replacements of grid parts, to a very restricted kind of dynamic behavior in the case of Cartesian grids.

Such elementary modifying operations clearly belong to the non-generic, grid-specific part. It turns out, that the apparent need for atomic operations, for example in grid refinement, can be circumvented. In many important cases, we can do with very few coarse-grained mutating primitives, which in addition have the advantage of offering a higher level of functionality. This makes it possible to treat grids with different dynamic capabilities uniformly.

More concretely, we identify three basic mutating operations: Grid *copy*, grid *enlargement*, and grid *cutting*. Grid enlargement means to 'glue' another grid to an existing grid, using a set of vertices or facets that are to be identified. Grid cutting means the removal of grid parts.

A problem common to all modifying operations is maintenance of consistency in dependent components. This concerns in particular grid functions and geometries. For example, if a grid is enlarged by some part, and there is a grid function bound to this grid, it may be undefined on the new part of the resulting grid.

This is a problem especially when grid functions are physically decoupled from the grid data structure, cf. page 80. Loose coupling with the underlying grid has its merits, but seems to make consistency maintenance harder.

A key for ensuring consistency in such cases is the use of *associative copies*, which create a *grid isomorphism* (→ p. 54) between the source and destination copies. By means of this morphism, information related to the source grid can be mapped to the target grid.

It would of course be more satisfactory to handle consistency maintenance in a more 'automatic' manner. A possible way would be to use a 'notifier-observer' pattern [GHJV94], where a grid plays the role of a notifier, and the dependent components are observers reacting to the grid-changed event. Still, there remains the question how to find the corresponding 'source component' associated with the source grid, at least in the case of grid copy or enlargement. This is subject to further work.

The copy/enlarge/cut functionality has in general to be implemented for each grid type separately; internally, however, it can use generic algorithms for performing standard tasks, for example, calculation cell neighbors, by an implementation of the algorithm 3.2 on page 61. Also, the 'source' part is not modified; it can be made a parameter. Therefore, we call these components *semi-generic*.

These procedures are not really concepts in the sense this term is used here. However, they serve as basic building blocks for many mutating algorithms, like Construct Overlap on page 134, and therefore play a role analogous to the other non-mutating non-generic components like grid iterators. See the appendix C.3 for a description of the copy operation.

The number of tasks that can be performed with these primitives is surprisingly large. Grid *copy* can be used to implement generalized constructors: Besides ordinary copy constructors and assignment operators, it is also essential for implementing grid conversions.

An important application of the copy primitive is the abstraction from specific file formats: For each such format, a *grid adapter* (see section 4.1.7.1) can be defined, which then can be used to construct a grid from this format, or to write a grid to this format.

Thus, grid types do not have to know about file formats, it suffices to provide semi-generic grid copy. The same technique can of course be used for other (serialized or not) representations of grids, for example via message-passing buffers. We routinely use it to create sample grids from standard templates, like Cartesian grids or Neumann triangulations.

Grid *enlargement* and *cutting* can be used for hybrid grid generation and grid refinement. In the first case, parts of a grid are generated separately by different algorithms and then glued together. In the second, a part of a grid is replaced with a refined version, thus encapsulating the refinement logic in a separate component.

A further important use of these operations occurs in distributed grids components, discussed in chapter 5.

4.1.7 Examples of Concrete Grid Kernel Components

To get a feeling of how things really work, it is often useful to have a look at concrete implementations. The following major grid kernels have been developed:

- `Triang2D`, a very simple data structure for two-dimensional triangulations, is discussed in the Appendix (B.1).

- `Complex2D` is a data structure for general two-dimensional complexes, whose cells are simple polygons.

- `RegGrid2D` is an example for an *implicit* representation, namely a Cartesian grid. Grid elements are represented by two integer coordinates.

Besides these types, there are a number of other components which are models of the GridRange concept, or more precisely, of Cell-Vertex Input Grid Range (→ p. 212), such as file format adapters, Neumann triangulations masks for Cartesian grids, and so forth, some of which are discussed below.

4.1.7.1 Serialized-grid Adapters

Often, grids are given in a *serialized* form: A 'flat' representation in a file, in a message-passing buffer, or the like.

The term *serialization* refers to the mapping of a (non-linear) in-core representation of some data structure onto a linear sequence of items; accordingly, *de-serialization* means the inverse process.

Especially for file representations, there are a multitude of different formats in use. If the (semi-) generic copy procedures (see section 4.1.6) are used to construct a given grid representation from such a format, one must provide a sort of *mask* for the flat file which offers some basic grid functionality described above. Then, for n grid data structures with their corresponding `CopyGrid` operations and m grid file formats, we are able to produce $n \cdot m$ $\texttt{ReadGrid}_{kj}$ procedures that read grid type G_k from file format F_j. This is true at least in principle, unless the information encoded and the information required differ so much that a different copy mechanism is needed.

To make things more concrete, we consider a file format for two dimensional triangular grids, where first the number of vertices and cells in the grid is given, then the coordinates of vertices and finally the vertex sets of cells, see table 3.2 on page 59.

The file in figure 4.5(a) contains the grid in figure 4.5(b). From the file contents, we can readily produce the following information: Number of vertices and cells, iteration of vertices and cells, and iteration over vertices incident to cells.

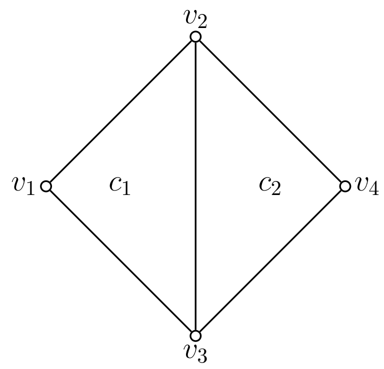
We can also access coordinates of vertices. This geometrical information is not needed to build the combinatorial structure, but it can be used to initialize a linear geometry. In order to make this information available when it is needed, we can simply fuse the concepts of (combinatorial) grid and geometry into one single software component. The resulting class `grid_file_xyz` is a model of Cell-Vertex Input Grid Range (→ p. 212); the implementation is documented in the appendix (figure B.3, B.2).

Reading a grid from a file is then as simple as

```
4
2
-1.0 0.0
0.0 -1.0
0.0 1.0
1.0 0.0

1 2 3
4 2 3
```

(a) a simple file representation ...



(b) ... of a tiny grid

Figure 4.5: Example for a serialized representation of a grid in a file

```
grid_file_xyz Gfile("xyz.grid");
grid_type     G;
CopyGrid(G,Gfile);
```

If we also implement the inverse copy procedure from arbitrary grids to `grid_file_xyz`, then we can consider the adapter class for a specific file format as a *definition* of this format in terms of the programming language: In this way, the functions

```
CopyGrid(a_grid        &, grid_file_xyz const&);  // read from xyz format
CopyGrid(grid_file_xyz &, a_grid        const&);  // write to  xyz format
```

are the only interfaces to the `xyz` file format. That is, they contain the only code concerned with the details of this format, no matter how many different grid implementations exist.

In much the same way, wrappers for grids encoded in message-passing buffers can be created that shield the mechanics of the serialization from the grid data structures using them.

4.1.7.2 Implicit Grids

In many cases, grids are not encoded explicitly, but rather given in an implicit way. Examples are Cartesian grids, tensor-product meshes, dual grids or standard triangulations, for example the Neumann-triangulation of a Cartesian grid. They have in common a certain *light-weight property*: Incidence relationships are calculated in a cheap way (*implicit grids*), or deduced from an underlying data structure (*grid adapter*).

Probably the most heavily used of these implicit representations are Cartesian grids. Here, only the geometry might possibly be stored explicitly; in the case of a Cartesian geometry, even this is not necessary. Often, algorithms can be formulated to run more efficiently when exploiting regular structure, especially in the field of numerical simulation. Therefore, a full implementation of a Cartesian grid will contain not only a general grid interface, but also give access to the Cartesian structure. In this manner, generic algorithms for general grids can be used, while not compromising efficiency where it matters.

Grid masks implement the notion of a grid derived in some well defined way from another one, for example a tensor-product grid or a dual grid (see p. 54).

4.2 Generic Grid Components

This section presents a selection of components which are based generically on the micro-kernel just developed. This means that any grid type conforming to the micro-kernel can use these components without any further labour.

Special attention is paid to components which are general enough to serve itself as a basis for more complex components, like algorithms, and can thus be seen as extending the functionality of the basic grid.

So, much space is devoted to grid sub-ranges (section 4.2.1) and iterators (section 4.2.2). These behave exactly like the specialized 'native' versions provided by a concrete grid like `Triang2D` (→ p. 217), thus allowing seamless *nesting* of generic components.

Kernel concepts like container grid functions (section 4.2.3) and grid geometries (section 4.2.4) can be implemented generically for the most common cases. These generic versions reduce programming effort when developing new compliant grid kernels, or adapting existing data structures to the kernel.

In general, we discuss generic algorithms where they arise, for example chapters 5 and 6. In section 4.2.5, we refer to the exact locations of such discussions — still more than for other components, a severe selection had to be made here.

4.2.1 Element-ranges and Subgrids

One of most common generalizations of an algorithm is to let it operate on a sub-structure — a sub-sequence, a sub-graph, a sub-grid. The ability to restrict the working region of an algorithm is important for example in distributed computing or for adaptive visualization, if one wants to focus on a small part of a grid.

In a mathematical setting, this possibility is so simple and obvious that we hardly ever even bother to mention it. In a computational context, on the contrary, some work remains to be done before one can say "Let $\mathcal{S}'$ be a subgrid of $\mathcal{G}$ defined by ... ". First, there must be software entities representing the notion of sub-structure in an efficient way — unless we want to copy. And second, implementations of the algorithms in question must be able to operate on these data components. Put the other way around, the components have to behave like the 'real thing' in all essential aspects.

We distinguish two main types of grid substructures, according to the layered description of grids developed in sections 4.1.2 to 4.1.3. On the one hand, there are *element* *subranges*, which are simply subsequences of the grids element ranges. On the other hand, we have *grid subranges* and *subgrids*. These are proper subcomplexes in the sense defined in definition 8, but have subtle differences in the semantics of some incidence iterators, to be discussed now.

If we assume subgrids and subranges to be of homogeneous dimension $d' \leq d$, then the closure of their set of d'-elements determines which elements of lower dimension are contained, by virtue of the subcomplex property. An algorithmic determination of these elements leads to *closure iterators*, discussed in the next section.

At first glance, we would expect incidence iterators associated with a grid substructure $\mathcal{R}$ to be restricted to elements of $\mathcal{R}$. However, this is not the behavior we need in many cases: For example, in parallel execution of a numerical discretization algorithm for PDE solution, algorithms are restricted to a local work range, but access information in a certain 'halo' around this range. The actual extent of this halo is precisely determined by the stencil of the algorithm, which in turn can be described by sequences of incidence queries. See chapter 5 for a detailed account on this problem.

We therefore define a grid sub*range* to be a subcomplex of a grid, with the incidence relationship *inherited* from the underlying base grid: Any upward incidence sequence (see p. 89) of an element in $\mathcal{R}$ is identical to that in $\mathcal{G}$. In contrast, an incidence sequence in a sub*grid* $\mathcal{S}$ of $\mathcal{G}$ contains only elements of $\mathcal{S}$. We use the general term grid substructure or part to denote either of two possibilities.

Note that for downward incidence sequences, there is no problem, as the corresponding element are contained in the grid substructure anyhow. Therefore, subranges can in principle reuse all incidence iterators of their base grids, whereas subgrids can reuse only the downward incidence iterators.

There are many ways to characterize a grid part:

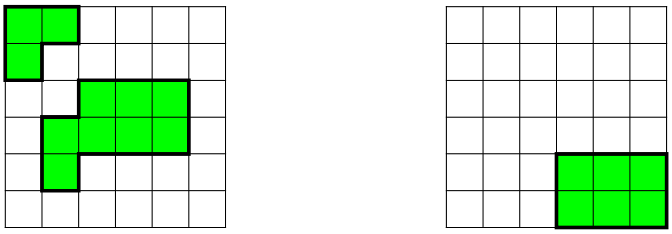
- Plain enumeration of elements (of the highest dimension),
- combinatorial characterization (skeletons, boundary grid or some neighborhood of an element),
- geometrical predicates ($f(v) > 0$ for all vertices of a cell c),

or combinations of these.

Generic implementations exist (not presented here) for enumerated subranges and for boundary grids, where the grid underlying the boundary may itself be a grid part.

The latter possibility shows clearly one of the strengths of the generic approach: Components can be nested at will. In this case, a simple application is the economic visualization of grid partitionings by only drawing the boundary facets of each partition.

If the underlying grid has special structure (for example, if it is Cartesian), we would like to have also grid parts respecting this structure, that is, belonging to the same class of grids and exhibiting the same special functionality. In these circumstances, we can complement the generic versions with specialized grid part components, which coexist with the generic ones, see figures 4.6(a) and 4.6(b).



(a) general subgrid of a Cartesian grid, by a generic component

(b) Cartesian subgrid, by a specialized component

Figure 4.6: Coexistence of generic and specialized subgrid components

Implementation issues Enumerated element ranges can be implemented using a container of element handles and a reference to the underlying grid.

Both (enumerated) sub-ranges and sub-grids build on element-ranges. Here some freedom is left whether to explicitly store sequences of lower-dimensional elements, or to rely on closure iterators for sequential access. Note that closure iterators, too, in general require storage $O(|\mathcal{S}^k|)$ for a grid part $\mathcal{S}$, because they need to keep track of already visited elements. A useful strategy could be a sort of *lazy evaluation* approach: lower dimensional elements are stored explicitly, but only when iteration is actually requested.

4.2.2 Generic Iterators

We have seen that iterators encapsulate a good deal of the core grid functionality. Therefore, it can hardly be expected that *all* iterators can be provided by generic templates. However, depending on the functionality mix offered by a basic grid implementation, some iterators can be derived generically. The following sections show detailed examples for such cases. The iterator templates described here can reduce effort for a creating a grid compliant to the micro-kernel.

4.2.2.1 Sequence Iterators

If k-elements are explicitly stored for every dimension $0 \leq k \leq d$, then evidently sequence iterators correspond directly to iterators over the underlying containers; there is no need for generic ones in this case. If, on the other hand, some element types are lacking a direct representation, there is some potential for genericity.

We assume here a cell-based grid, and show how to derive generic implementations for facet-iterators and edge-iterators. In two dimensions, these evidently coincide, and we may choose one and the same generic version.

Facets may be represented by a cell(-handle) and a local number in the archetype of that cell. Iteration over facets in a manifold-with-boundary grid can exploit the fact that each facet is incident to at most two cells (only to one if it is on the boundary). Giving the 'outer' cell a handle that is strictly less than any other cell handle, we can choose from the two possible representations of a facet that one with greater cell handle. Iteration over facets means iterating over all facets of all cells, skipping those whose 'opposite' representation is greater. This approach assumes that the opposite representation is readily available, for example, if cell-cell-adjacencies are provided. As a cell's facets are in a one-to-one correspondence with its neighbors (except for boundary facets), it is understood here that a local facet number gives immediate access to the corresponding neighbor in the case of cell-cell-adjacencies. A generic component implementing these ideas is `FacetIterator` ($\rightarrow$ p. 226).

If cell-neighbor information is not available, we can resort to explicit marking of facets: A grid function

$$\texttt{visited} : \mathcal{F}_{\mathcal{G}} \mapsto \{0, 1\}$$

helps to keep track of already seen facets (*visit-and-mark technique*). Note that in this case, grid functions on facets cannot use a representation by two incident cells to distinguish facets. A different unique representation suitable here is the vertex set of facets. This is what is done in the Cell Neighbor Search algorithm, presented on page 61, whose very aim is the determination of cell neighbors.

Edges, instead of being determined by two cells, can be uniquely represented by vertices. If edge-on-vertex- and vertex-iterators are available, edge iterators can be implemented just like facet-iterators, comparing vertices instead of cells. Else, again an explicit marking strategy is used. Comparing edges is done via their vertex sets, which is easier than for facets because there are always exactly two vertices.

It should be noted that these iterators rely essentially on vertex-set uniqueness, which is satisfied if the grid's poset is an atomic lattice.

4.2.2.2 Incidence Iterators

We can use the `switch`-operator (sec. 4.1.3.3) to implement some types of incidence iterators. In 2D, we observe that edges and cells are circularly ordered around each vertex. Given a triple $v \prec e \prec c$, all edges and cells incident to the vertex v can be obtained by alternating edge- and cell-switches, see figure 4.7. Taken the symmetric of v with respect to e, we get all adjacent vertices.

Therefore, in order to add vertex-centered incidence iterators to a two-dimensional grid, a mapping

$$\begin{array}{rcl} \texttt{germs} : \mathcal{V}(\mathcal{G}) & \mapsto & \mathcal{E}(\mathcal{G}) \times \mathcal{C}(\mathcal{G}) \\ v & \mapsto & (e, c) \qquad \text{with} \quad v \prec e \prec c \end{array}$$

has to be provided, giving for each vertex v a *germ* (e, c), needed to start the iteration. A suitable data structure is a grid function mapping vertices to (edge,cell) handle pairs:

```
grid_function<Vertex, pair<edge_handle, cell_handle> >   germs;
```

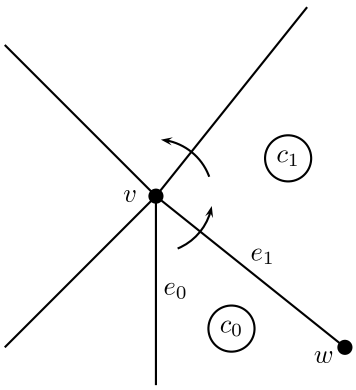


Figure 4.7: $e_1 = \texttt{switch}(v, e_0, c_0), c_1 = \texttt{switch}(e_1, c_0), w = \texttt{switch}(v, e_1)$

For three dimensions, the situation is slightly more complicated. For edges, there is a circular order of incident facets and cells, and for facets, there is one for edges and vertices. So a technique similar to the 2D case can be used. In contrast, the sets of elements incident to vertices exhibit no natural order. A possible approach is to use breadth-first traversal of the *flags* (see below) incident to a vertex.

4.2.2.3 Closure Iterators

When a subgrid is defined by a set $C \subset \mathcal{G}^d$ of cells, we must find a way to sequentially access all elements of lower dimension in the closure $\overline{C}$. We present simple algorithms to determine the sequences of facets, edges and vertices in the subcomplex $\mathcal{G}_C = \overline{C} \subset \mathcal{G}$.

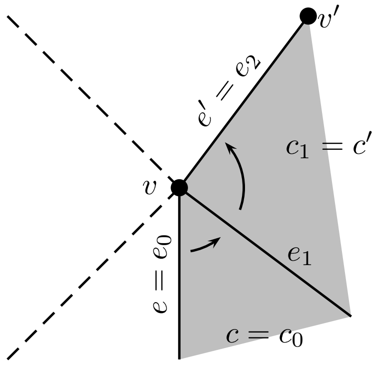
FacetIterator There are two possibilities, depending on the availability of an efficient test for inclusion in the set of cells C:

1. There exist a predicate `outside` $: \mathcal{G}^d \mapsto \{0, 1\}$ with complexity $O(1)$. Then it is possible to use the generic facet iterator of section 4.2.2.1. This requires no extra storage and visits each facet at most twice.

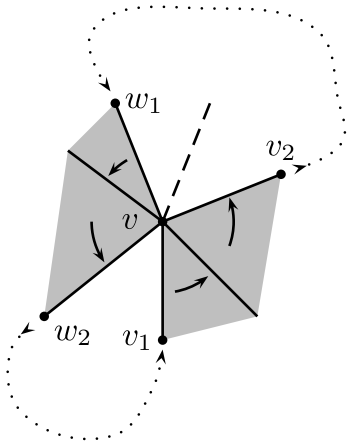
2. We use the *visit-and-mark technique.* This uses storage $O(|\mathcal{G}_C^{d-1}|)$, each facet it visited at most twice.

EdgeIterator The strategy of choice is visit-and-mark, by using a nested loop over all cells of C and all edges of each cell. Storage used is $O(|\mathcal{G}_C^1|)$, and each edge e is at most visited as often as there are cells in C incident to e. This number is not bounded; however, one can show that for constrained types of cells, the *average* number of cells per edge is bounded, see [Loe76].

VertexIterator Here virtually the same approach as for edge-iterators is used, and the same remarks apply.



(a) finding the next boundary flag:$(v, e, c) \mapsto (v', e', c')$



(b) Vertex v has two wedges,circular order is $\ldots v_1, v, v_2, \ldots w_1, v, w_2, \ldots$

Figure 4.8: How NEXT-BOUNDARY-FLAG-2D works

4.2.2.4 Boundary Iterators

Recall that the boundary of a grid (part) has a purely combinatorial definition: It is the closure of the set of facets having exactly one incident cell *in the grid part*. For maximal generality, we assume here a predicate

$$\texttt{inside} : \mathcal{G}^d \mapsto \{0, 1\}$$

is given on the cells of $\mathcal{G}$, defining a cell-set $C = \{c \in \mathcal{G}^d | \texttt{inside}(c) = 1\}$. We aim for the boundary of $\mathcal{G}_C = \overline{C}$. This task separates into two parts: A local iteration over a component of the boundary, and a global iteration over all components.

Boundary Component Iterator

A local iterator over a component of the boundary uses the local `switch` operations. The local state of the iterator is described by a so-called *flag*, that is, a tuple (v, e, c) in two dimensions, or $(e_0, \ldots, e_d)$ in d dimensions, where $e_0 \prec e_1 \prec \ldots \prec e_d$. For each component, the boundary of a 2-dimensional grid is circularly ordered. If the grid is locally homeomorphic to a halfspace in each boundary vertex, every boundary component is homeomorphic to $\mathbb{S}^1$. Given an initial boundary flag (v, e, c), where $v, e \in \partial\mathcal{G}$ and $c \in \mathcal{G}$, the algorithm produces the next boundary flag by successive application of `switch` operations.

Algorithm NEXT-BOUNDARY-FLAG-2D forms the increment operation of a boundary component iterator. The output of this iterator defines a circular order on a boundary component. In case the environment of a boundary vertex v consists of n wedges, v will occur exactly n times in the sequence, see figure 4.8(b).

Algorithm 4.1 NEXT-BOUNDARY-FLAG-2D: find next boundary flag of 2D grid

IN: (v, e, c) *(A boundary flag)*
OUT: (v', e', c') *(The next boundary flag)*
Require: $\texttt{inside}(c) = 1$
Ensure: $v \prec e' \wedge e' \neq e$
Ensure: $\texttt{inside}(c') = 1$

$e \leftarrow \texttt{switch}_{(v,c)}(e)$
while $\texttt{inside}(\texttt{switch}_e(c))$ **do**
  $c \leftarrow \texttt{switch}_e(c)$
  $e \leftarrow \texttt{switch}_{(v,c)}(e)$
end while
$v \leftarrow \texttt{switch}_e(v)$
output (v, e, c)

The algorithm uses only constant space. The number of operations is dominated by the number s_e of `switch` operations on edges, and s_e is equal to the number of cells incident to v inside the grid. This number is in principle unbounded, even if the *average* number of cells incident to a vertex is bounded when the average number of vertices per cell is bounded, what is the case for grids used in numerical PDE solution.

So, in the worst case, the work to be done for a complete scan of a boundary component may be proportional to the number of cells in the corresponding grid component; this occurs, however, only in pathological cases.

Anyway, in order to determine the boundary of an arbitrary grid (given by the `inside`-predicate), it is in general necessary to scan all of its cells: If we take just one cell c out of the interior of any grid, there is a new boundary component we would fail to detect if none of c's neighbors would be checked. These considerations lead to the global part of boundary iteration.

Global boundary grid iterator In order to ensure that each part of the boundary is visited, we have to provide a facet (germ) for each boundary component. If there are n components, we have to provide an iteration germ for each. Now, if we know there is only one component, it suffices to find an arbitrary boundary facet. In the general case, however, it is necessary to scan all cells of the grid to determine all boundary components. Once this has been done, the boundary can be stored compactly by keeping just one germ for each component. Extracting germs from a sequence of boundary facets can be done with a boundary component iterator and an explicit mark of visited facets.

4.2.3 Container Grid Functions

Container grid functions can be implemented almost completely in a generic fashion, if one agrees to store the data outside the elements, as discussed on page 80. Genericity is rather evident for the image type `T`, but is the case also for the domain type `E`, which is an element type of the grid type under consideration.

All we need to know about the element type is, in fact, how it can be used to access a memory location. Here essentially two possibilities occur: Either, the corresponding handles are consecutively numbered, for example integers from 0 to $n-1$, where $n = |\mathcal{G}^k|$ is the number of elements. In this case we may use arrays for the (total) grid functions. If this is not the case, we can rely on either hashing or sorting of element handles to map elements to memory locations. We discuss only the case of hashing, because it is in practice more efficient than using a balanced tree.

In the case of partial grid functions, by default, always hash tables are used. Please note that there always remains the possibility to select a particular implementation for a concrete element type, if the generic version is not considered adequate. The *partial specialization* feature of C++ offers a comfortable way of fixing just one of two generic parameters.

The two basic classes defined are

```
template<class E, class T>
class grid_function_vector;

template<class E, class T>
class grid_function_hash;
```

From the hash version, we derive classes for total and partial grid functions, which have slightly different interfaces:

```
template<class E, class T>
class partial_grid_function_hash
 : public grid_function_hash<E,T>;

template<class E, class T>
class total_grid_function_hash
 : public grid_function_hash<E,T>;
```

For the vector version this is not necessary, as it can be used only for total grid functions.

If we want to use these generic versions for a concrete grid class, we just have to derive once again from one of these three classes, while fixing the element parameter:

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

template<class T>
class partial_grid_function<MyVertex,T>
  : public partial_grid_function_hash<MyVertex,T> {
  // repeat constructors
};
```

In order to deduce the necessary information about elements from the type `E`, we use a traits-technique: A class template called `element_traits` is specialized for each element type. It contains, among others, the type of the hash function to be used.

In these generic versions, only the most basic parameters of variation, as listed in section 3.3.1.2 are taken into account, namely the types `E` (element) and `T` (image). A more comprehensive approach could also build in other parameters, such as argument checking, and implementation choices.

4.2.4 Grid Geometries

Grid geometries are another type of micro-kernel component which does in general not need much information about the underlying combinatorial grid.

In the simplest case of a linear geometry (→ p. 58), access to vertex coordinates is all what is needed to derive the rest of geometric functionality. In the case of a 'pure' combinatorial grid, even this type of information can be added generically, by using a grid function mapping vertices to some type representing coordinates.

A shortened example looks like the following:

```
template<class GRID, class POINT>
class linear_geometry_2d {
  // some typedefs ...

  grid_function<Vertex, POINT> XY;
public:
  // coordinates of vertex
  POINT const& coord(Vertex const& V) const { return XY(V);}

  // outward pointing normal, with length = length(*fc)
  POINT outer_area_normal(FacetOnCellIterator const& fc) const {
    typedef point_traits<POINT> pt;
    return  POINT(pt::y(XY[fc.V2()]) - pt::y(XY[fc.V1()]),
                  pt::x(XY[fc.V1()]) - pt::x(XY[fc.V2()]));
  }
};
```

4.2.5 Generic Algorithms

Grid-based algorithms form a very heterogeneous and complex field — much broader than the topics discussed in the previous sections. Therefore, presentation of algorithms and their generic implementations occurs in the context the algorithms arise from.

In the sequel, we restrict ourselves to hint to these locations, and additionally mention some examples that have not found their place elsewhere.

Combinatorial algorithms are perhaps the most broadly usable ones, like Cell Neighbor Search (introduced on page 61, implementation in appendix C.1), or next-boundary-flag-2d, used as a basis for boundary iterators in 2D.

Distributed grids are discussed at some length in chapter 5; some of the algorithms introduced there are Incidence Hull (see page 133), Distributed Hull (p. 137) and Construct Overlap (p. 134). The major unit of reuse in this context turns out not to be a *single* algorithm or data structure, but a complex *system* of coordinated algorithmic and data oriented components, cf. section 5.7.

Visualization methods are an important ingredient of numerical applications. This field has not been discussed here in detail. Generic implementations have been created for simple xy plots, color plots and isolines over two dimensional grids, as well as a first version of a particle tracing method (see also p. 64).

In this context, there pop up many more variabilities than just the ones related to grid / grid function data layout:

- the type of the 'rendering engine' producing the graphics
- the type of geometric output, for example color maps, isoline density, or particle paths rendering
- algorithmic parameters, like the numerical ODE solution method or the cell localization strategy for particle tracing

In particular for the case of particle tracing, a detailed analysis of the variabilities seems an interesting and promising task.

Finally, **numerical** algorithms, which were one of the starting points of this thesis, are discussed in chapter 6. In section 6.2, a generic finite volume solver is presented, which is parameterized – besides grid and geometry – over hyperbolic equation and flux calculation method. Section 6.3 is devoted to the solution of elliptic problems, involving a simple finite element discretization and components for multigrid methods, including hierarchical grids.

Chapter 5

Concepts and Components for Parallel PDE solution with Distributed Grids

Ce qui est simple, n'est pas vrai,
ce qui ne l'est pas, est inutilisable.
What is simple, is not true,
and that what isn't, is not usable.
Paul Valéry, *Mauvaises pensées.*

5.1 Introduction

Parallel computing is an issue getting more and more important for large-scale numerical simulations. It is true that (sequential) computing power per unit cost increases from year to year, but the demand for doing larger and faster calculations rises as well. Parallel computers perform the same computation in a fraction of sequential execution time, thus making the solution of a given problem faster, and allow to increase the size of problems, thus overcoming the limiting factor of memory capacity when solving large-scale problems.

In last few years, parallel hardware has matured and become more common. In particular, as interconnections get faster and cheaper, high-performance clusters that rival dedicated parallel architectures can be built from affordable components. Thus, powerful computing resources become broadly available, and have to be exploited by parallel software.

However, usage of parallel computing power does not come for free. Parallel architectures are complicated, and still exhibit a lot more diversity than sequential hardware does. For the latter, the simple RAM (von Neumann) view of a computer is a very successful, reasonable unifying abstraction (in spite of some restrictions relating to memory hierarchies, see p. 15).

There is no such common computing model for parallel machines. Some models make it easier to reason about parallel programs, but hide significant performance aspects, in particular data locality issues. More realistic abstractions for parallel computers tend to be a closer model of the complicated architectures of parallel computers. Thus, additional dependency on the structure of the underlying hardware is inevitablely introduced into the software, making it less portable.

The more involved semantics of concurrent processes open a new source of programming complexity, and thus of errors. A fine balance between ease of programming, especially concerning consistency-maintenance of distributed data, and computational efficiency has to be established. In general, it depends on the concrete application which compromises can be made. The need to distribute computational work evenly, the duplication of data, and the communication needed for maintaining consistency introduces additional overhead. If this overhead is not handled with care, the benefits of parallel computing in terms of execution time may be partially lost.

These difficulties make the production of parallel software a highly demanding task. The 'software crisis' which haunts software development for sequential hardware aggravates for parallel computing.

No *general* panacea is in sight. In contrast, *domain-specific* approaches have the potential to considerably ease the creation of parallel programs in given context.

In this chapter we propose an approach *specific* to the *domain of grid-based applications*. Our *leitmotiv* is the distributed execution of PDE solution algorithms; however, the concepts we are going to describe are neither bound to any *physical* type of distribution, nor are they restricted to numerical algorithms. It will become quite clear how the *logical* aspects of grid distribution correlate with the *structural* properties of grid-based algorithms.

The outline of this chapter is the following: First, a brief look on the basics of parallel machines and programming models is taken (section 5.2). Next, we introduce the paradigm of *geometric data partitioning* which underlies any concrete parallelization for the class of problems under consideration (section 5.3), and give a short overview which types of software tools are available to help implementing this paradigm (section 5.4).

The second part of this chapter is devoted to a detailed presentation of the concepts and components we have developed to support parallel grid-based computations. The central concept of *distributed overlapping grids* is introduced in section 5.5. In particular, we formally define stencils of algorithms on unstructured grids (s. 5.5.4) allowing a general specification and practical determination of appropriate grid overlaps. The algorithms needed to set up data structures for distributed grids are described in section 5.6. We conclude in section 5.7 by discussing some aspects of a generic implementation of these concepts, which allows usage on top of *arbitrary* grid data structures. This implementation of non-trivial algorithms and data structures gives strong evidence for the viability of the generic approach.

5.2 Machines and Models for Parallel Computing

The aim of this section is not to give a full-fledged introduction into parallel computing, but rather to provide as much information as necessary to estimate how and where our approach fits into the field. A very broad and comprehensive treatment can be found in [Zom96], an introduction more leaned towards scientific computing is [FWM94].

On the level of parallel hardware, one distinguishes between machines with processors working strictly synchronized (SIMD – single instruction stream, multiple data stream) and those working asynchronously (MIMD machines).

Examples for SIMD machines are processor arrays like the MP-I and Connection Machine CM-2, vector computers and systolic arrays, but on a smaller scale also the MMX technology of modern PC processors falls into this category.

MIMD machines can be distinguished further depending on whether they provide global address space (shared memory, symmetric multiprocessors – SMP) or not (distributed memory, multicomputers). This distinction is sharp only in a logical sense (programmers view), because there exist hybrid designs and a global address space is sometimes provided on top of physically distributed memory with the aid of complicated hardware or system software, so-called *distributed shared memory* (DSM). Examples for distributed memory architectures are networks of workstations (NoWs), the IBM SP-2 or the Thinking Machines CM-5. Recent trends in parallel hardware favor *multi-tier* machines (see [BF99] and the references cited there): clusters of SMPs connected via high-speed networks. Thus, instead of getting easier, parallel architectures get more complex, and so do the local hierarchical memory systems of sequential machines. In sum, one can end up with several layers of memory hierarchies: two or three local caches, a machine-wide memory of a SMP, the virtual memory of a MIMD machine with high-speed interconnections, and possibly a number of such clusters connected via WANs. Getting optimal performance out of such an architecture is a highly demanding task.

An abstract machine model fitted to SIMD architectures is the PRAM model, which assumes that each processor has access to each memory location at unit cost. This makes the design of efficient[1] algorithms simpler; however, in the case of MIMD, the assumption of uniform memory access is not realistic. There are several extensions of the PRAM model, however, all of these contain the assumption of uniform memory access.

The typical programming approach for SIMD architectures is the *data parallel* programming paradigm, which assumes very regular data like vectors and matrices. Note that in this thesis, the term ‘data parallel’ is used in a more general sense, relating to more coarse-grained operations than single instructions are.

The distinguishing feature of MIMD computers is the *non-uniform* memory access characteristic, with its sharpest incarnation in distributed memory machines, where remote memory cannot be accessed directly, but only by using special *message-passing* primitives, today often provided by standardized libraries like MPI [MPI] or PVM [PVM].

[1] with respect to the PRAM model

But also shared-memory machines have non-uniform memory access.

A modern abstract model for such machines is provided by the *bulk synchronous parallelism* model (BSP) [Val90], which incorporates measures for the computation / communication efficiency ratio and network capacity.

Programming styles appropriate for MIMD computers are the SPMD (single program – multiple data) or *data partitioning* approach, and the MPMD (or *task-parallel*) style. The distinction is not sharp, as different branches in a single program could be executed in different processes, in the extreme yielding a similar effect as MPMD.

It is easier to work with a SPMD approach, because only one program has to be maintained, and normally, only a single flow of control has to be considered. For many situation, especially in scientific computing, SPMD has proven appropriate.

The main programming effort concentrates therefore on *data distribution*. If there is no shared memory, access to remote data must be done by explicitly invoking *message-passing* mechanisms. This complicates programming considerably, but gives strong control over the interprocess communication, and thus more opportunity for fine-tuning performance. On the other hand, if the non-uniformity of data access for SMP machines is taken into account, the need for efficient parallel execution tends to impose a logical program structure similar to the distributed memory case.

For parallel PDE solution, a variant of the data-partitioning approach, termed *geometric partitioning* or *domain decomposition*, has proven its usefulness.

5.3 Geometric Partitioning for Parallel PDE Solution

What are the key properties of computations related to the numerical solution of partial differential equations? Their most outstanding feature in the parallelization context certainly is, that they are grid-based: The important algorithms work on grids, and the distribution process will essentially be driven by a distribution of the underlying grid.

Some of the characteristic features of grid-based algorithms are the following:

- There is a natural notion of *locality* in the underlying grid data structures.
- Algorithms typically work locally on the data.
- Algorithms typically have a predictable pattern of data access,
- Data is either somehow associated with grid elements, or it is global (e. g. tolerances, time-steps)
- The amount of work performed is roughly proportional to the size of the underlying data structures

An adequate parallelization strategy for these types of applications is therefore *geometric partitioning* or *domain decomposition*[2].

[2]This is meant in a general sense, not in the sense of the Schwarz domain decomposition method.

Algorithms are considered immutable by this data-parallel approach: The same method is applied by all owners of the corresponding data (*owner-computes rule*); the results are identical to the sequential case. Obviously, not every sequential algorithm gives rise to a useful parallel version of this type; it is the task of the application programmer to supply algorithms suited to this approach, or to decide whether the altered behavior is still acceptable.

In some cases, the resulting algorithms, albeit not yielding results identical to the sequential case, are still useful. Important examples are given by iterative methods for linear equations, or more exactly, a *single* iteration of such a method. These give rise to a block-version after parallelization. By choosing appropriate overlaps, convergence can often be ensured, but this is highly dependent on the method and the concrete problem.

For example, the Jacobi iteration parallelizes without problems, whereas for the Gauss-Seidel method, the convergence behavior of the parallel algorithm strongly depends on the concrete linear system. The situation gets worse for methods with a strongly sequential nature, like ILU. In this case, the concepts we are going to present would probably have to be refined.

Thus, the *algorithmic* decisions cannot be taken by a general software, but must remain in the responsibility of the algorithm designer, in this case, numerical analyst. However, software can support ways to *express* these decisions in high-level, concise manner.

A central task of distributed grids is assurance of sufficient *locality* of data with respect to the underlying algorithms: All data elements an algorithm uses must be cached locally. This is true in particular for distributed memory, but in a milder form, this principle also applies to shared memory settings. To achieve this, an appropriate amount of duplication (*overlap*) has to be provided by the grid data structures. The concrete shape of this overlap depends on the employed algorithms, and may be optimized with respect to the performance characteristics of the underlying parallel computer.

We note in passing, that also decomposition techniques termed 'non-overlapping' involve data duplication (and therefore overlap) in the sense we use this term here, namely entities of lower dimension on the boundary of the parts.

Duplication of data makes it necessary to maintain its *consistency*. Therefore data associated to grid elements in the overlap between the must be exchanged between the processes concerned.

Accepting grid distribution as the leading principle of geometric partitioning, applied to parallel PDE solution, two basic scenarios can be distinguished:

- *static distribution:* The distribution of the underlying grid remains fixed
- *dynamic distribution:* The distribution of the grid varies with time.

Evidently, the static distribution is more basic, because it is a special case of dynamic distribution. For algorithms without spatial adaptivity, it is often possible to achieve parallelization using static distribution only. In the static case, only data *on* the grid is exchanged between grid parts. This data belongs to the responsibility of grid functions, (see sections 3.3.1.2 and 4.1.4) and therefore distributed grid functions (section 5.5.5) are the main entities that support communication in the static case.

The general domain decomposition strategy described above leads to a number of concrete problems that have to be solved for the static case:

- How can the load be balanced evenly? How is the grid initially partitioned into parts of approximately equal size?
- How is an initial grid distribution generated?
- How is the exact extent of the overlap determined?
- How is consistency of data maintained in detail?
- How is the overhead of overlap and communication minimized?
- How is the data transport done? How are temporal dependencies recognized and handled?

The load balancing problem leads to a graph partitioning problem: Given a graph (represented by a grid, in this case), determine a partitioning of its nodes, such that the parts are (approximately) equal and the size of the boundary between graph partitions is (nearly) minimized.

The optimal solution to this problem is known to be NP-hard. Consequently, there exist a large number of algorithms for approximate solutions, see [Els97] for an overview. Also, software packages exist for graph partitioning, for example Metis [Kar99] or Jostle [Wal99].

In this thesis, we therefore treat grid partitioning as a preprocessing step. A generic interface to the Metis library has been developed, which encapsulates the details of the data structure conversions needed. In any case, grid partitioning algorithms are candidates for generic implementations, as this would remove the copying overhead, which leads to possible memory or runtime bottlenecks.

Starting from section 5.5, we develop general concepts for tackling the remaining problems, centered around the concept of *distributed overlapping grids*. These concepts give rise to data structures and associated algorithms hiding the details of solving these tasks, and providing a very high level of description for grid distribution properties and data consistency maintenance.

The dynamic case poses additional problems:

- How is dynamic load imbalance handled?
- How can the grid be repartitioned incrementally?
- How can grid chunks be *migrated* to other processes, including overlap data structures and data on the grid?

This case is not yet fully incorporated in the components implemented thus far. We point out, however, how dynamic distribution fits into the present framework.

As in the static case, the incremental repartitioning problem can be solved by existing components like ParMetis [Kar99], but generic components would be even more useful than in the static case, because the overhead associated with copying grid data structures tends to aggravate for repeated execution. Grid migration and distributed overlap generation are dealt with in sections 5.6.5 and 5.6.3.

5.4 Software Support for Parallel PDE Solution

Software tools supporting parallel programming in general and parallel PDE solution in particular form a very heterogeneous landscape. The available tools fall broadly into two classes: Those that do not use any information on the particular nature of the problem at hand (*general-purpose tools*), and those that exploit to some extent knowledge on the (geometric) data partitioning to be performed (*domain-specific tools*).

We first give a short overview on general-purpose tools, which can be further classified coarsely into libraries, parallel languages, and parallelizing compilers.

General *libraries* are typically constrained to deliver low-level support only, such as message-passing libraries like MPI and PVM. They offer very little conceptual support for performing the task of data distribution. A main advantage is their good portability, flexibility and performance, which makes them candidates for serving as a basis of higher-level tools.

Languages for parallel programming are proliferating. In particular, parallel dialects for widespread languages (like C, C++ and Fortran) are in use. For data-parallel, structured applications, High-Performance Fortran (HPF) probably is the best-known example. A tool for compiling HPF programs into efficient message passing Fortran code is Adaptor ([BZ94, BHKF99]). An overview on 16 parallel C++ dialects is given in [WL96].

Abstractions like *distributed objects* or global pointers, offered by some object-oriented approaches like Concert/ICC++ ([C$^+$, GC98]), are attractive from a programming point of view, but data distribution on top of these entities is much too fine-granular and thus too expensive for scientific computing.

Parallelizing compilers often deliver results which are clearly suboptimal with regard to performance, cf. [Zom96], chapter 30. In particular for applications involving *irregular* data structures, such as unstructured grids, the missing knowledge about data distribution is difficult to deduce automatically. Directives can often be used to guide the compiler; in the extreme, however, this may lead to an amount of problem-specific work that puts the utilization of an automatic tool in question.

All three approaches have in common that their strength — generality — is also their weakness *with respect to parallel PDE solution*, namely missing knowledge over the structure of the computational problem. Put bluntly, libraries leave to much work of explicitly stating this knowledge at a low conceptual level, whereas parallelizing compilers do not allow enough such knowledge to be expressed in order to achieve sufficient performance.

Parallel languages are somewhere in the middle between these extremes. Obtaining good results still requires a substantial amount of programming work consisting in formulating explicitly knowledge about the problem, albeit at a higher level than possible with libraries. Using a parallel programming language typically means developing new programs; existing code is difficult to reuse.

This discussion has been intentionally short; for a more extensive overview on general purpose tools for parallel programming, see e. g. [Zom96], chapters 29 and 30.

In contrast to the general-purpose tools discussed so far, *domain-specific* approaches can exploit knowledge on the problem class. In the case of parallel PDE solution, or more generally, grid-based computations, the basic structural aspects of the problems are the same or at least similar for many concrete cases. Therefore, substantial gain can be expected from encapsulating them once and for all. The domain-specific approach involves a *trade-off* between the generality with respect to the class of supported data models (ranging from general graphs over unstructured to Cartesian grids), and the amount of specific knowledge and automation they can offer.

Some approaches, like PETSc [BGMS97] mentioned in chapter 2, or Blocksolve [JP95], set up on top of the *algebraic* level: They offer a set of distributed algebraic data structures (matrices and vectors), and corresponding parallel solution algorithms. However, the work to organize the distributed construction of these matrices (e. g. FEM stiffness matrices, sec. 6.3) is left to the user. We are interested here in the grid level *beneath* this algebraic level.

One of the more general approaches is the DDD library of Birken [Bir98], using a *graph-based* model to define and establish data consistency. It augments user data structures with distribution information, and is therefore independent of specific data structures. However, the additional structure available in *grid-based* computations cannot be exploited directly; the task of specifying and producing the correct overlap largely remains to the user. In [Bir98], Birken proposes a strategy to solve this problem, but it is not clear whether this has yet led to a practical solution.

Many approaches exist that are exclusively devoted to grid-based applications, for example, Birken (ibid.) gives an overview on about 20 of these tools.

A large part of them concentrate on regular, *block-structured grids*, like KeLP ([Fin97], [FBK98]) or SAMRAI [HK98]. In this case, the local access-patterns of algorithms are very regular, and the determination of grid overlap is rather straightforward. These packages often allow adaptive algorithms with local refinement, resulting in hierarchical structures of Cartesian patches with rather complex communication behavior.

Algorithms are typically formulated on top of the data structures provided by the library, here on rectangular patches. Because of the simplicity of such individual patches, this is not a serious restriction; the approach is well suited for incorporating existing solver software written for Cartesian grids.

Unstructured grids are supported, among others, by the GRIDS package [GHR+93], DIME [FWM94], and SUMAA3D [FJP98].

GRIDS is targeted towards FORTRAN applications. Programs are specified using scripts coordinating 'local' ordinary routines. These scripts are translated into compilable code by a special preprocessor, and allow to specify data dependencies for data on grid elements, which corresponds to the stencils we introduce below (section 5.5.4).

The SUMAA3D project has the aim of providing a platform for parallel grid computations. It is intended to offer parallel mesh generation, partitioning, and refinement. These algorithms are an integral part of the system, and thus intimately related to the underlying data representation.

Virtually all of these approaches build on top of their own data structures; parallelizing *existing* applications (built on different data structures) is therefore difficult.

Similar in spirit to these approaches for unstructured grids are the *distributed overlapping grid* concepts which are developed in the following sections. They provide a means by which application programmers can work on a level of abstraction very close to the *mental* concept of a distributed unstructured grid. In detail, our approach has the following features:

- By using the generic programming approach, it is completely independent of the underlying grid data structure, and can thus easily be used with existing programs (see section 6.4).
- no extra-lingual tools (like preprocessors) are needed
- overlap is characterized by very compact stencils (a handful of numbers)
- overlap is calculated *at runtime* from the stencils, different overlaps for different algorithms are in principle possible
- overhead is proportional to the size of the overlap (both memory and speed)
- all concepts are completely dimension independent

It thus combines the advantages found in single tools mentioned above, with the additional benefit of a very lean, yet general and efficient treatment of general overlaps.

5.5 Distributed Overlapping Grids – Concepts

5.5.1 Introduction

As has been pointed out in section 5.3, the parallel solution of PDEs — or grid-based computations in general — is essentially driven by grid distribution.

The purpose of the present section is to develop concepts materializing the essential ingredients of grid distribution — grid overlap, data access patterns of algorithms, and consistency maintenance of duplicated data.

Together, these concepts allow to specify the parallelization of grid-based computations in an abstract, yet precise manner. In a later section (5.7) we show how these concepts give rise to generic software components encapsulating the details, thus allowing to program on an equivalent level of abstraction, as far as parallelization is concerned.

The concepts presented in the following sections are completely *independent* of the grid dimension. All dimension-dependent aspects are encapsulated in the grid micro-kernel, and are not relevant neither to the abstract concepts, nor to data structures, nor to the algorithms developed in this and the following section.

Concrete tests have so far only be done for two-dimensional structures; however, the extension to three dimensions seems straightforward and should mainly concern grid data structures themselves. This extension is scheduled for the near future.

This section is organized as follows: In 5.5.2, the detailed structure of *distributed overlapping grids* (DOG) is discussed, leading to the detailed concepts of overlap ranges, and local, distributed and global grids.

The important notion of a *quotient grid*, induced by a partitioning, is introduced in section 5.5.3.

In section 5.5.4, we show how access patterns of algorithms on unstructured grids can be described by *stencils*, for which a very compact notation is developed. Subsequently, some basic results for stencil operations are proven.

Distributed grid functions, introduced in section 5.5.5, are the concept that ties together distributed overlapping grids and grid-based algorithms, and are the "working horse" of static parallel grid computations. Here we also give an exact meaning to the term 'data-parallel algorithm', and discuss the related consistency of distributed grid functions.

5.5.2 Distributed Grids and Overlap Structures

When discussing distributed grids, we will prefer the term 'part' for what is often called 'process' or even 'processor', because it much more closely expresses the logical structure we are interested in; whether there is one process associated to every part or not is rather secondary. In particular, the ideas developed in this and the following sections are independent of the actual *physical* distribution: They apply as well to the case that all parts are in the same physical memory, a situation we refer to as *composite grid*.

There are two opposite views on a distributed structure: On the one hand, the *global* one, where one sees all parts at once, without giving preference to any one, and second, the *local* one, where one 'sits down' on one part and concentrates on the interaction with the neighbors.

Each of them has its merits, and so one or the other will be adopted whenever appropriate. We will see in section 5.7, how these different views can be traced down to individual software components, which stand for a local part or the global whole, respectively.

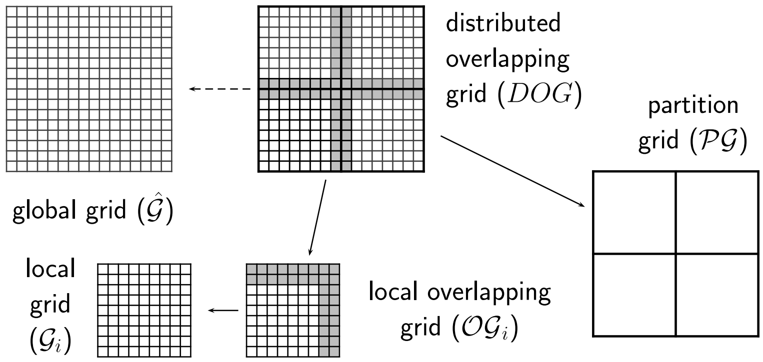


Figure 5.1: The relationship of various concepts for distributed grids. Grid quotients are introduced in section 5.5.3. The global grid $\widehat{\mathcal{G}}$ typically does not exist physically, but is useful as mental concept.

There are also two complementary ways to think of a distributed overlapping grid: On the one hand, one can consider a global grid $\widehat{\mathcal{G}}$, which is partitioned into overlapping parts $\mathcal{G}_i$. On the other hand, one can regard the $\mathcal{G}_i$ as primary entities, which are just 'standard' sequential grids plus some additional structure describing the overlap (*local overlapping grid*). A suitable identification between overlap parts on different $\mathcal{G}_i$ then acts as 'glue' permitting to *derive* the global grid $\widehat{\mathcal{G}}$ from the set of the parts (*distributed overlapping grid*).

This second view is more closely related to the actual situation of distributed grids, where a global grid does not exist physically. The general relationship of these entities is depicted by figure 5.1.

More formally, the connection between disjoint local overlapping grids is established by appropriate mappings identifying grid ranges:

Definition 27 (overlap structure). Let $\mathcal{G}_i$ be a set of grids, $1 \leq i \leq N$. An *overlap structure* on $(\mathcal{G}_i)_{1\leq i \leq N}$ is a system of grid isomorphisms Φ_{ij} ($\rightarrow$ p. 54) on *bilateral overlaps* $\mathcal{O}_{ij}$

$$\Phi_{ij} : \mathcal{O}_{ij} \subset \mathcal{G}_i \mapsto \mathcal{O}_{ji} \subset \mathcal{G}_j$$

satisfying

$$\Phi_{ij}^{-1} = \Phi_{ji}, \qquad \Phi_{ij}(\mathcal{O}_{ij}) = \mathcal{O}_{ji} \qquad \text{(symmetry)} \tag{5.1}$$

and

$$\Phi_{ij} \circ \Phi_{jk} = \Phi_{ik} \qquad \text{on} \quad \mathcal{O}_{ij} \cap \Phi_{ji}(\mathcal{O}_{jk}) \qquad \text{(transitivity)} \tag{5.2}$$

for $1 \leq i, j, k \leq N$. Formally, we set

$$\mathcal{O}_{ii} = \emptyset \quad 1 \leq i \leq N$$

Note that typically the matrix (Φ_{ij}) is sparse: Most bilateral overlaps $\mathcal{O}_{ij}$ are empty.

Definition 28 (correspondence relation). The *correspondence relation* $\sim$ between grid elements $e^i \in \mathcal{G}_i, e^j \in \mathcal{G}_j$ of different parts $\mathcal{G}_i, \mathcal{G}_j$ is defined as

$$e^i \sim e^j \Leftrightarrow \Phi_{ij}(e^i) = e^j$$

This is obviously an equivalence relation, whose classes $[e]$ are denoted by $\widehat{e}$. We define the global grid $\widehat{\mathcal{G}}$ by

$$\widehat{\mathcal{G}} := \left\{ \widehat{e} \,\middle|\, e \in \bigcup_{i=1}^{n} \mathcal{G}_i \right\}$$

Two elements in $\widehat{\mathcal{G}}$ are incident if there exist incident representants in a part $\mathcal{G}_i$:

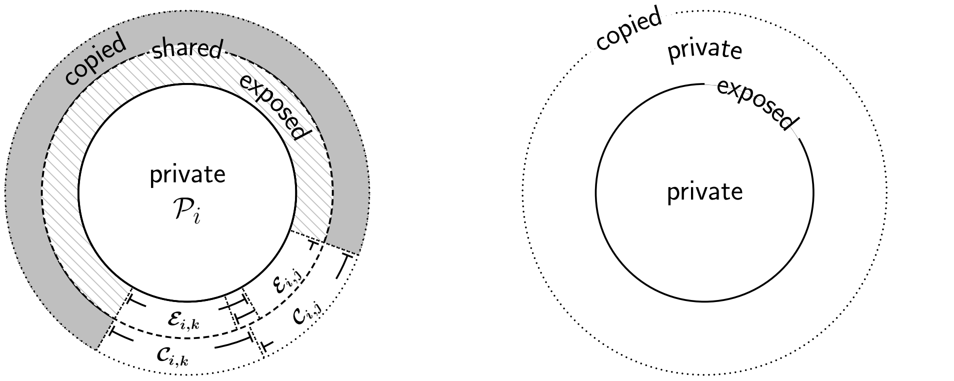
$$\widehat{e} < \widehat{f} \quad \Leftrightarrow \quad \exists\, i : \, e_i < f_i \quad \text{in} \quad \mathcal{G}_i, \quad e_i \in \widehat{e},\, f_i \in \widehat{f}.$$

The incidence relation on the equivalence classes $\widehat{e}$ is well defined, because a different pair of representatives e_j, f_j corresponds to e_i, f_i via the grid isomorphism Φ_{ji}, which preserve incidence. As the $\mathcal{G}_i$ are closed, it is not possible that an incidence $e < f$ in the global grid is 'cut' by assigning e, f to different parts without duplication.

Every grid entity on distributed grids has to be consistent with the overlap structure. For example, if there are geometries Γ_i on $\mathcal{G}_i$, we require

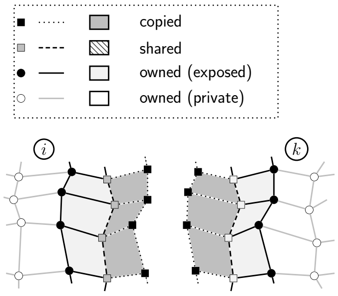
$$\Gamma_j = \Gamma_i \circ \Phi_{ji} \quad \text{on} \quad \mathcal{O}_{ji}$$

which allows to define a global geometry $\widehat{\Gamma}$ in a straightforward way.

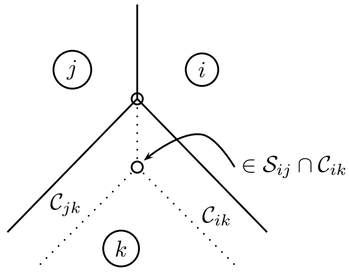


(a) generic overlap configuration

(b) overlap for Schwarz domain decomposition



(c) overlap example



(d) Copied ranges may contain shared elements

Figure 5.2: Possible overlap configurations

The next step is the definition of *ownership* relations on the overlap ranges $\mathcal{O}_{ij}$. We distinguish between *exposed* ranges $\mathcal{E}_{ij}$, that are owned by part i, *shared* ranges $\mathcal{S}_{ij}$, belonging to both part i and part j, and *copied* ranges $\mathcal{C}_{ij}$, that belong to part j. The following symmetry relations must hold between these ranges:

$$\begin{aligned}
\mathcal{E}_{ij} &= \Phi_{ji}(\mathcal{C}_{ji}) \\
\mathcal{S}_{ij} &= \Phi_{ji}(\mathcal{S}_{ji}) \\
\mathcal{C}_{ij} &= \Phi_{ji}(\mathcal{E}_{ji}) \\
\mathcal{O}_{ij} &= \mathcal{E}_{ij} \cup \mathcal{S}_{ij} \cup \mathcal{C}_{ij}
\end{aligned}$$

and $\mathcal{E}_{ij}, \mathcal{S}_{ij}, \mathcal{C}_{ij}$ are pairwise disjoint. See figure 5.2(a) for the general picture. For the classical Schwarz domain decomposition technique, there are no shared ranges, and the overlaps are disconnected (fig. 5.2(b)).

These bilateral ranges are useful primarily for data exchange between different parts. For deciding where calculation has to take place, and where not, so-called *total ranges* are needed, which are roughly the unions of the bilateral ranges:

$$\mathcal{O}_i = \bigcup_{j=1}^{n} \mathcal{O}_{ij} \qquad\qquad \mathcal{S}_i = \bigcup_{j=1}^{n} \mathcal{S}_{ij} \setminus \mathcal{C}_i \quad \textbf{s}\text{hared}$$

$$\mathcal{C}_i = \bigcup_{j=1}^{n} \mathcal{C}_{ij} \quad \textbf{c}\text{opied} \qquad\qquad \mathcal{E}_i = \bigcup_{j=1}^{n} \mathcal{E}_{ij} \setminus (\mathcal{S}_i \cup \mathcal{C}_i) \quad \textbf{e}\text{xported}$$

Their meaning is the following: On exposed elements in $\mathcal{E}_i$, the local part is exclusively responsible for performing any calculation. On the shared range, several parts will in general contribute to the calculation, or calculations are done redundantly on each part. Finally, on copied elements, no local calculations are performed.

The reason for the omission of copied elements in $\mathcal{S}_i$ and the shared ones in $\mathcal{E}_i$ is the *principle of least work*: If an element is copied from somewhere, we need not do any calculation on it. If an element is shared with other parts, we do only our part of the calculation (unless redundant calculation is chosen anyway). Note that $\mathcal{S}_i \cap \mathcal{C}_{ik} \neq \emptyset$ is indeed possible, as is $\mathcal{S}_{ij} \cap \mathcal{E}_i \neq \emptyset$, see figure 5.2(d).

In addition, we define *derived ranges*

$\mathcal{P}_i = \mathcal{G}_i \setminus \mathcal{O}_i$	**p**rivate	$\mathcal{L}_i = \mathcal{P}_i \cup \mathcal{E}_i \cup \mathcal{S}_i$	**l**ocal
$\mathcal{X}_i = \mathcal{S}_i \cup \mathcal{E}_i$	e**x**ported	$\mathcal{I}_i = \mathcal{S}_i \cup \mathcal{C}_i$	**i**mported
$\mathcal{W}_i = \mathcal{P}_i \cup \mathcal{E}_i$	o**w**ned		

Private elements $\in \mathcal{P}_i$ are those that cannot be seen by any other part. Conversely, elements of other parts that cannot be seen from part i are termed *remote* elements for part i. Exported elements are those other parts might be interested in, and imported elements are elements that the local part is interested in but does not exclusively own. Owned elements are those not shared with any other part (exclusive ownership), while local elements include also shared ones. Sometimes it is necessary to have a unique owner for *every* element. This can be achieved by formally attributing each shared element to the least of its owners, assuming a total order on the parts. This element sets will be called *formally owned*. For later reference, we summarize the inclusion relationship, starting from the private range:

$$\mathcal{P}_i \subset \mathcal{W}_i \subset \mathcal{F}_i \subset \mathcal{L}_i \subset \mathcal{G}_i \tag{5.3}$$

5.5.3 The Quotient Grid

The notion of a *quotient* that imposes a grid-like structure on a partitioning of a given grid is very useful for fixing ideas when dealing with associated subranges of the underlying grid. For example, for the distributed generation of overlap structures (see section 5.6.3), the quotient grid structure is important. Depending on the structure of the quotient, sometimes optimization of communication processes are possible.

Definition 29 (partitioning). A (cell-based) partitioning P of a grid $\mathcal{G}$ is a surjective mapping

$$P : \mathcal{G}^d \mapsto \{1, \dots, N\}$$

The partitions $P_i := \overline{P^{-1}(i)}$ are subgrids of $\mathcal{G}$.

The partitions can be seen as cells of an induced *quotient grid*. The elements of lower dimension will be shown to consist of intersections of the P_i. However, not every nonempty intersection of the P_i defines an element of the quotient.

The dimension of an arbitrary subset of a grid $\mathcal{G}$ is the maximal dimension of its elements. The *quotient relation* on k-dimensional elements of $\mathcal{G}$ is defined by

$$c_1 \sim_d c_2 \iff P(c_1) = P(c_2) \tag{5.4}$$

for cells of $\mathcal{G}$, and recursively for two k-dimensional elements,

$$e_1 \sim_k e_2 \iff [\mathcal{I}_n(e_1)] = [\mathcal{I}_n(e_2)], \quad k+1 \le n \le d \tag{5.5}$$

that is, if e_1, e_2 are incident to equivalent higher-dimensional elements. It is easy to see that $\sim_k$ is an equivalence relation.

Definition 30 (grid quotient). The *quotient* $\mathfrak{G} = \mathcal{G}/_P$ of a grid $\mathcal{G}$ and a partitioning P is the graded poset (→ p. 52) with the following (closed) elements:

1. P_i, the *cells* of $\mathfrak{G}$
2. For defining the closed elements $\mathfrak{e}$ of dimension $k = d-1, \dots, 0$ we consider the equivalence classes with respect to $\sim_k$. Here only elements e of $\mathcal{G}^k$ are considered which are contained in more than one quotient element of dimension $k+1$. The k-elements of $\mathfrak{G}$ are defined as the closure of the classes of such elements:

$$\mathfrak{G}^k = \overline{\left\{ [e]_k \,\middle|\, e \in \bigcup_{\mathfrak{e}_1, \mathfrak{e}_2 \in \mathfrak{G}^{k+1}} \overline{\mathfrak{e}_1} \cap \overline{\mathfrak{e}_2} \right\}} \tag{5.6}$$

The order relation is given by

$$\mathfrak{e} < \mathfrak{f} \iff \exists e \in \mathfrak{e}\ \exists f \in \mathfrak{f} \quad \text{such that} \quad e < f \tag{5.7}$$

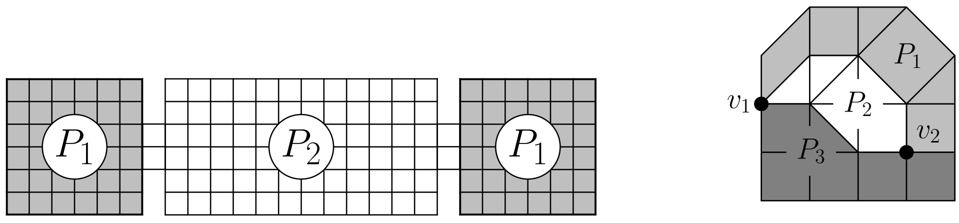
Thus, the poset is indeed graded, because the poset of the original grid is.

The open elements of $\mathfrak{G}$ are simply the closed elements, with all incident lower-dimensional elements removed. The closure of an open element $\mathrm{int}(\mathfrak{e})$ of $\mathfrak{G}^k$ has in principle two different interpretations: Either, the closure as a subset of $\mathcal{G}^k$, or, the closure with respect to (5.7). However, it is easy to see that both possibilities lead to the same subset of $\mathcal{G}$, namely, to the closed element $\mathfrak{e}$.

In general, a quotient element $\mathfrak{e}$, seen as subcomplex of $\mathcal{G}$, may be disconnected or have holes and thus fail to be homeomorphic to a disk. See figure 5.3 for examples. Therefore, we call $\mathfrak{G} = \mathcal{G}/_P$ a *generalized* grid. There is, evidently, always a subdivision of $\mathfrak{G}$ that is a *regular* grid (see 47); one could take just the original grid $\mathcal{G}$.

However, one often is more interested in the combinatorial properties of the underlying quotient poset than in its geometric meaning. The facets of $\mathcal{G}/_P$ are exactly the bilateral shared ranges $\mathcal{S}_{ij}$, and elements of lower dimension give information about elements shared between several partitions, which can be important for communication scheduling.

If the quotient grid is Cartesian — which may be the case even for unstructured local grids — then data exchange (and overlap) can be restricted to direct neighbors, see figure 5.4. In this case the bilateral ranges $\mathcal{E}_{ij}$ and $\mathcal{C}_{ij}$ have to be enlarged by the corner pieces left out by deleting the bilateral overlaps of parts that do not share a facet of the quotient.



(a) An optimal partitioning may result in disconnected partitions

(b) The set $\{v_1, v_2\}$ is a vertex of the quotient

Figure 5.3: Quotients are grids only in a very general sense

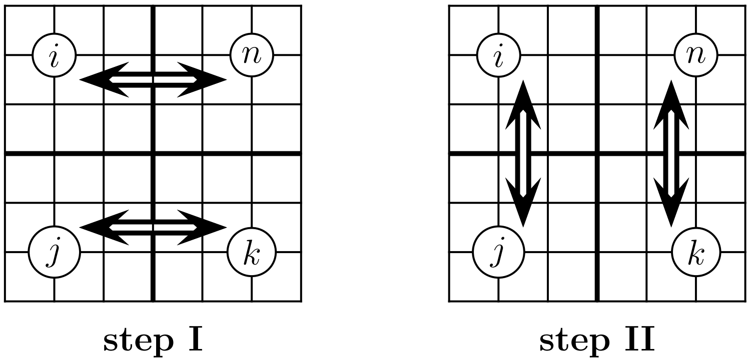


Figure 5.4: Communication pattern for Cartesian quotient grid saving 'diagonal' communication

5.5.4 Stencils of Algorithms

A central task for a general purpose overlapping grid component is the automatic determination of correct overlap structures with respect to the data access patterns of algorithms, which typically involve a certain neighborhood of an element.

This neighborhood (the *domain of dependency*) induced by an algorithm is called its *stencil*.

How can a stencil on an unstructured grid be described? The central idea is introduced best by an example. Assume we have a simple flux calculation algorithm f:

```
for all Cells c ∈ 𝒢 do
  fluxsum(c) ←0
  for all Cells n adjacent to c do
    fluxsum(c) += flux(U(c),U(n))
```

The abstract combinatorial structure of this algorithm can be expressed as going from cells to incident facets to incident cells:

$$C \mapsto F \mapsto C$$

This structure may be expressed more tersely by listing the corresponding dimensions in an *incidence sequence* $(d, d-1, d)$ or CFC (for *cell-facet-cell*),

visualized by figure 5.5(a). The meaning of this stencil with regard to grid distribution is: Whenever f is to be calculated for a cell c, all its neighbors n (sharing a facet with c) must be accessible in the local grid $\mathcal{G}_i$, and the grid function U must be *globally consistent* (→ p. 129) at this time.

More generally, a grid algorithm f is said to *operate locally*, if it depends only on the *state* S (see section 5.5.5) restricted to a local neighborhood of a grid element:

$$f(e, S) = f(e, S_{|\mathcal{N}(e)}) \qquad \text{where} \quad e \in \mathcal{N}(e) \subset \mathcal{G}_i$$

In the example above, $S = U$, and $\mathcal{N}(c)$ is the set of neighbor cells shown in fig. 5.5(a). For each element e, there is a minimal grid neighborhood such that this equation is valid. The *stencil* of f is the function V_f

$$\begin{array}{rcl} V_f : \mathcal{G}^k & \mapsto & \{\mathcal{N} \subset \mathcal{G}\} \\ V_f(e) & = & \mathcal{N}(e) \end{array}$$

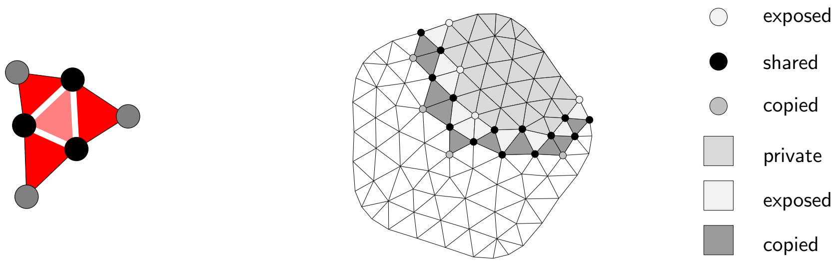
that determines the minimal neighborhood.

Normally, the stencil can be described in rather simple terms, by using incidence sequences, as in the flux example above. For a given set of initial elements, say, cells on the boundary of a grid partition, applying this incidence sequence to all these initial cells then determines all elements outside the partition that are possibly needed by the algorithm, see dark triangles in figure 5.5(b).

Definition 31 (incidence sequence, layers, hull). An *incidence sequence* for a d-dimensional grid is a sequence I of numbers $(a_0, \ldots, a_n)$ with $0 \leq a_i \leq d$. The meaning is, that from an initial element of dimension a_0, all incident elements of dimension a_1 are visited, from these all incident element of dimension a_2, and so on. More formally, let $\mathcal{K} \subset \mathcal{G}$ be an initial set (*germ*) of elements of dimension i. The *incidence layer* $\mathcal{L}_{(i,j)}(\mathcal{K})$ is defined by

$$\mathcal{L}_{(i,j)}(\mathcal{K}) := \bigcup_{e \in \mathcal{K}^i} \mathcal{I}_j(e) = \bigcup_{e \in \mathcal{K}^i} \{f \in \mathcal{G}^j \mid f \lesseqgtr e\} \tag{5.8}$$

Let $I = (a_0, a_1, \ldots, a_n)$ be an incidence sequence over a grid $\mathcal{G}$. The *hull* $\mathcal{H} = \mathcal{H}_I(\mathcal{K})$ and the layers $\mathcal{L}_I^{(k)}(\mathcal{K})$ generated by I and a germ $\mathcal{K} \subset G^{a_0}$ are then defined recursively



(a) stencil of flux algorithm: *CFC*

(b) Corresponding hulls. Dark gray triangles are in the hull outside the partition, and light gray triangles in the inside.

Figure 5.5: Stencil and corresponding hull for a simple algorithm

by

$$\mathcal{L}_I^{(0)}(\mathcal{K}) := \mathcal{K} \tag{5.9}$$

$$\mathcal{L}_I^{(k)}(\mathcal{K}) := \mathcal{L}_{(a_{k-1},a_k)}(\mathcal{L}_I^{(k-1)}(\mathcal{K})) \setminus \bigcup_{j=0}^{k-1}(\mathcal{L}_I^{(k-1)}(\mathcal{K})) \tag{5.10}$$

$$\mathcal{H}_I(\mathcal{K}) := \bigcup_{k=0}^{n} \mathcal{L}_I^{(k)}(\mathcal{K}) \tag{5.11}$$

For abbreviation, we also define partial hulls

$$\mathcal{H}_I^k(\mathcal{K}) := \mathcal{H}_{(a_0,\dots,a_k)}(\mathcal{K}) = \bigcup_{j=0}^{k} \mathcal{L}_I^{(j)}(\mathcal{K}) \tag{5.12}$$

Obviously, one has

$$\mathcal{H}_I(A \cup B) = \mathcal{H}_I(A) \cup \mathcal{H}_I(B) \tag{5.13}$$

$$\mathcal{H}_I(A \cap B) \subset \mathcal{H}_I(A) \cap \mathcal{H}_I(B) \tag{5.14}$$

because for a fixed stencil, $\mathcal{H}_I$ is a mapping from the power set of $\mathcal{G}^{a_0}$ to the power set of $\mathcal{G}$.

The stencil of figure 5.5(a) is a representative of the more restricted class of *cell-based stencils*. These are stencils of the form $(d, d_1, d, d_2, \dots, d_n, d)$, where $0 \le d_i \le d-1$. For a cell-based stencil I, we set $I_{[k]} := (d, d_k, d)$, and $I_{[k,l]} := (d, d_k, d, d_{k+1}, \dots, d_l, d)$.

The *length* of a stencil $I = (d, d_1, d, d_2, \dots, d_n, d)$ is defined as $|I| = n$. If two stencils $I_1 = (d, d_1, \dots, d_k, d)$ and $I_2 = (d, d_{k+1}, \dots, d_n, d)$ are given, the *composed* stencil is $I := (I_1, I_2) := (d, d_1, \dots, d_n, d)$. Conversely, we refer to I_1, I_2 as a *splitting* of I. Sometimes, stencils are written in the shorter form CVC (for $(d, 0, d)$ or *cell-vertex-cell*), CFC for $(d, d-1, d)$ and so on.

For sake of shortness, we set in the case of cell-based stencils

$$\mathcal{L}_{d_k}(A) := \mathcal{L}^{(2)}_{(d,d_k,d)}(A) = \mathcal{L}_{(d,d_k)}(\mathcal{L}_{(d_k,d)}(A))$$

because we are generally only interested in layers containing cells.

From now on, only cell-based stencils will be used. Similar considerations can be made for vertex-based stencils.

It seems intuitively clear, that hulls can be calculated 'by parts', that is, if $I = (I_1, I_2)$, then $\mathcal{H}_I(\mathcal{K}) = \mathcal{H}_{I_2}(\mathcal{H}_{I_1}(\mathcal{K}))$. This is *not* true for arbitrary grids, however; see fig. 5.6(b). We prove this for cell-based stencils on manifold-with-boundary (mwb-) grids. This result will be needed in the proof of a monotonicity property for stencils (theorem 5).

Theorem 4. *If $\mathcal{G}$ is a manifold-with-boundary grid, than the hull of a cell-based stencil I is not changed if in each layer, the entire partial hull $\mathcal{H}^{k-1}$ instead of $\mathcal{L}^{k-1}$ is taken as germ. That is, if $I = (d, d_0, d, \dots, d_k, d)$, then*

$$\mathcal{L}_{d_k}(\mathcal{L}_I^{k-1}(A)) \setminus \mathcal{H}_I^{k-1}(A) = \mathcal{L}_{d_k}(\mathcal{H}_I^{k-1}(A)) \setminus \mathcal{H}_I^{k-1}(A) \tag{5.15}$$

for any germ $A \in \mathcal{G}$. It follows that for $I = (I_1, I_2)$

$$\mathcal{H}_I(\mathcal{K}) = \mathcal{H}_{I_2}(\mathcal{H}_{I_1}(\mathcal{K})) \tag{5.16}$$

Proof. For brevity, we omit the fixed stencil I. We must prove that a cell reached in step k from $\mathcal{H}^{k-1}$ is also reached over a cell of the previous layer $\mathcal{L}^{k-1} \subset \mathcal{H}^{k-1}$.

First, let $e \in \overline{\mathcal{H}^{k-1}}$ be a non-cell element, and $\mathrm{st}(e)$ the *star* of e (→ p. 48). If there is a cell c in $\mathrm{st}(e)$ not contained in $\mathcal{H}^{k-1}$, there must be a cell c_b in $\mathrm{st}(e) \cap \mathcal{H}^{k-1}$ with a facet f incident to a cell in $\mathrm{st}(e) \setminus \mathcal{H}^{k-1}$.

This is because $S(e)$ is homeomorphic to a ball or a halfspace of dimension d, and hence an open, connected set. There is a cell $c' \subset \mathcal{H}^{k-1}$ incident to e, a path from c' to c in the interior of $\mathrm{st}(e)$ must leave $\mathcal{H}^{k-1}$ somewhere, and can be slightly distorted to do this through a facet.

The cell c_b must belong to the last layer $\mathcal{L}^{(k-1)}$, because if it belongs to a previous layer $\mathcal{L}^{(k-j)}, j > 1$, the cell on the other side of the facet would have been reached in layer $\mathcal{L}^{(k-j+1)}$, regardless of d_{k-j+1}, cf. (5.17).

Now, let c be a cell first visited in layer k:

$$c \in \mathcal{L}_{d_k}(\mathcal{H}^{k-1}) \setminus \mathcal{H}^{k-1}.$$

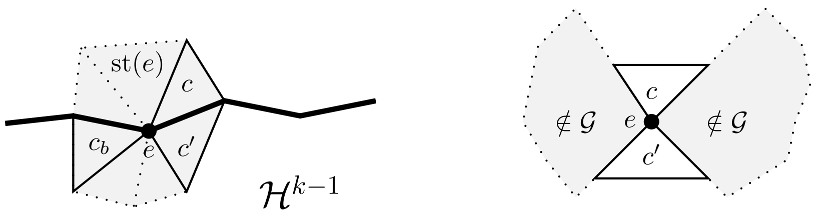
That is, there exists $e \in \mathcal{H}^{k-1}$ with $\dim e = d_k$ and a cell $c' \in \mathcal{H}^{k-1}$ with $c \cap c' \supset e$, that is, c and c' are contained in $\mathrm{st}(e)$ (see fig 5.6(a)). By the considerations made before, there must be at least one cell $c_b \in \mathrm{st}(e) \cap \mathcal{L}^{k-1}$. Then also $c_b \cap c \supset e$, and therefore

$$c \in \mathcal{L}_{d_k}(\mathcal{L}^{k-1}) \setminus \mathcal{H}^{k-1}.$$

Now, for proving (5.16), we note that we have just proved the special case $I = (J, d, d_k, d)$, that is,

$$\mathcal{H}_{(d,d_k,d)}(\mathcal{H}_J(A)) = \mathcal{H}_I(A)$$

The general case follows by induction over $|I|$. $\square$



(a) Cell c_b must be in the outermost layer $\mathcal{L}^{k-1}$, and c is reached from c_b via e

(b) Here $\mathrm{st}(e)$ is not connected. If the stencil is $(2, 1, 2, 0, 2)$, and the germ $\mathcal{K} = c' = \mathcal{L}^0$ is taken, cell c is *not* reached from a cell of Layer $\mathcal{L}^1$, but from $\mathcal{H}^1 \ni c'$

Figure 5.6: Illustrations for the proof of theorem on incrementatl hull calculation

It is clear that

$$d_1 \geq d_2 \Rightarrow \mathcal{H}_{(d,d_1,d)}(A) \subseteq \mathcal{H}_{(d,d_2,d)}(A) \tag{5.17}$$

because $c \in \mathcal{H}_{(d,d_1,d)}(A)$ means there exists $c' \in A$ with $\dim(c \cap c') \geq d_1 \geq d_2$ and thus $c \in \mathcal{H}_{(d,d_2,d)}(A)$. A similar relationship can be expected for substencils. This motivates the definition of a partial order on cell-based stencils:

Definition 32 (partial order on stencils). A stencil $I = (d, d_1, d, \ldots, d_n, d)$ *dominates* a stencil $J = (d, c_1, d, \ldots, c_k, d)$, in symbols $I \geq J$, if there exists $1 \leq i_1 < \ldots < i_k \leq n$ such that

$$c_r \geq d_{i_r} \quad \text{for} \quad 1 \leq r \leq k$$

Intuitively, we expect the dominating stencil to generate a larger hull. The following theorem shows that this is indeed true. This result can be used to decide in the case of different stencils whether some of them need not be considered due to dominance, see for example section 6.2.3.3 for different stencils in the context of finite volume methods.

Theorem 5 (Hull monotonicity of cell-based stencils). *Let*

$$I = (d, d_1, d, d_2, \dots , d_n, d)$$

be a cell-based stencil, and $\mathcal{G}$ a mwb-grid. Then

(i) For all cell sets $A, B \subset \mathcal{G}^d$

$$A \subset B \Rightarrow \mathcal{H}_I(A) \subset \mathcal{H}_I(B) \tag{5.18}$$

(germ monotonicity)

(ii) If J is another cell-based stencil, then

$$J \leq I \Rightarrow \mathcal{H}_J(A) \subseteq \mathcal{H}_I(A) \tag{5.19}$$

(stencil monotonicity)

Proof. We first prove (5.18) for the case $|I| = 1$, that is, $I = (d, d_k, d)$: If $c \in \mathcal{H}_I(B)$, then $c \in B$ or there is $c' \in \overline{B}$ such that both c and c' are incident to an element e of dimension d_k. Because $A \supset B$, the same is true with B replaced by A, and this is equivalent to $c \in \mathcal{H}_I(B)$.

For general I, we can use induction over the length of I. Using a splitting $I = (I_1, I_2)$, we have

$$\mathcal{H}_I(B) = \mathcal{H}_{I_1}(\mathcal{H}_{I_2}(B)) \subset \mathcal{H}_{I_1}(\mathcal{H}_{I_2}(A)) = \mathcal{H}_I(A)$$

To show (5.19), we use induction over the length of J. For the empty stencil, (5.19) is trivial. Now let $|J| = 1$ and k be such that $J \leq I_k$. Then

$$\begin{aligned}
\mathcal{H}_I(B) &= \mathcal{H}_{I_{[k+1,n]}}(\mathcal{H}_{I_{[k]}}(\mathcal{H}_{I_{[0,k-1]}}(B))) && \text{by theorem 4}\\
&\supseteq \mathcal{H}_{I_{[k]}}(\mathcal{H}_{I_{[0,k-1]}}(B)) && \text{by (5.17)}\\
&\supseteq \mathcal{H}_J(\mathcal{H}_{I_{[0,k-1]}}(B))\\
&\supseteq \mathcal{H}_J(B)
\end{aligned}$$

A general J with $|J| > 1$ can be split into (J_1, J_2) with $|J_1| < |J|, |J_2| < |J|$. Using a corresponding splitting $I = (I_1, I_2)$ with $J_1 \leq I_1, J_2 \leq I_2$ (which must exist by the definition of stencil dominance), we have

$$\begin{aligned}
\mathcal{H}_J(B) &= \mathcal{H}_{J_1}(\mathcal{H}_{J_2}(B)) && \text{by theorem 4}\\
&\subseteq \mathcal{H}_{I_1}(\mathcal{H}_{I_2}(B)) && \text{by induction and (5.18)}\\
&= \mathcal{H}_I(B) && \text{by theorem 4}
\end{aligned}$$

□

5.5.5 Distributed Grid Functions

For a *static* communication pattern, only data *on* the grid is exchanged. Hence, in this case (distributed) grid functions are precisely the layer of abstraction at which a programmer has to deal with distribution aspects.

We assume program state S being composed of grid functions F_A and numbers λ_a (global parameters, such as time step, tolerances etc.), plus grid and geometry (the latter considered immutable):

$$S = (F_A, F_B, \ldots, \lambda_a, \lambda_b, \ldots, \mathcal{G}, \Gamma)$$

It is not particularly useful to consider grid functions in isolation here. For a proper definition of the basic concepts, also the algorithm f that calculates the values of a grid function F, the grid range W where the calculation happens, and the grid range R where the grid function values are used by other algorithms have to be considered. Together, we call these a *calculation tuple*:

Definition 33 (calculation tuple). A *calculation tuple* is a tuple (F, f, W, R, S_f), with $S_f \subset S$, and

$$F : \mathcal{G}^k \mapsto \mathcal{T}$$

is a grid function from k-elements to a set $\mathcal{T}$,

$$W \subseteq \mathcal{G}^k$$

is a *work* (or *write*) *range*,

$$R \subseteq \mathcal{G}^k$$

is a *read range*, and

$$f : \mathcal{G}^k \times S_f \mapsto \mathcal{T}$$

is a *grid algorithm*.

Definition 34 (data-parallel algorithm). If S_f does not depend on F, that is, $F \notin S_f$, we say that f is *data-parallel*: The value of f on an element $e \in W$ does not depend on the order of evaluation of f on W.

Typical examples of local data-parallel algorithms are discretizations of PDEs. They encapsulate many of the specific details of the application, and can remain unchanged regardless of the global distribution context.

So far, only local grid functions have been considered. A *distributed* grid function (DGF) is simply a vector $(F_i)_{1 \le i \le N}$ of local grid functions, each referring to a distributed grid instead of a local grid. (We omitted the associated calculation tuple for clarity.) The fact that the algorithm f has no subscript is a consequence of the SPMD paradigm: It is the same for every part.

Now we want to define consistency for distributed grid functions. Therefore, we assume an iterative structure of the program:

for $n = 0, 1, \ldots$ **do** *(time-steps or iterations)*
$\vdots$
for all $e \in W$ **do** *(do not* use $F^{(n)}$*)*
$F^{(n)}(e) \leftarrow f(e, S^{(n)})$
$\vdots$ *(may use* $F^{(n)}$*)*

Here $W^{(n)} = W$ is assumed to be static. At step n, a grid function tuple (F, f, W, R) is *consistent* on a range $L \subset W$, if

$$F^{(n)}(e) = f(e, S^{(n)}) \quad \forall e \in L$$

A tuple is *locally consistent* if it is consistent on W. A distributed grid function $(F_i)_{1 \le i \le N}$ is (globally) consistent (or synchronized) if each (F_i) is locally consistent and any two local grid functions coincide on their common support:

$$F_i = F_j \circ \Phi_{ij} \quad \text{on} \quad \mathcal{O}_{ij}$$

A globally consistent distributed grid function allows the definition of a global grid function $\widehat{F}$ by

$$\widehat{F}(\widehat{e}) = F_i(e_i), e_i \in \widehat{e} \quad \forall \widehat{e} \in \widehat{\mathcal{G}}^k$$

According to Tanenbaum [Tan95], a consistency model is a sort of treatise between the system layer (DOG/DGF) and the application. In our case, the DGF layer promises: "If you call `synchronize()` on a distributed grid function, the values on corresponding elements will be the same afterwards".

On the other hand, it is in the responsibility of the application program to assign the right read/write ranges to a DGF, to use the right modus for shared ranges, and to *use* a DGF only between a synchronization and a subsequent local re-computation. Else, there will not be a meaningful state of the global grid function. But see below (p. 131) for an example where a controlled version of inconsistent state is useful.

Thus, the decisions a programmer has to take with respect to distributed grid functions are

1. determine ranges W and R for each grid function
2. decide, if and when a distributed grid function has to be synchronized

Item 1 requires the stencil of the associated algorithm f, and possibly a choice whether on shared ranges should occur partial or complete calculations on each node.

The write-range W_i coincides in general with the local range $\mathcal{W}_i = \mathcal{P}_i \cup \mathcal{E}_i \cup \mathcal{S}_i$, whereas the read range R_i might ly somewhere between W_i and $W_i \cup \mathcal{C}_i$. Thus, the predefined overlap ranges of the underlying grid may often be used without any modification. Some more examples are found on page 158 ff.

How the overlap can be calculated from function stencils is the subject of section 5.6.

The decision concerning the moment of synchronization normally is straightforward, simply make it globally consistent when it is locally consistent:

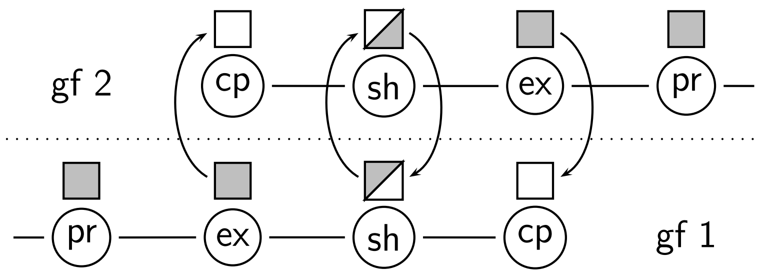


Figure 5.7: Flow of information during synchronization of *dgf* s with partial computation on shared ranges.

for all $e \in W$ **do**
  $F^{(n)}(e) \leftarrow f(e, S^{(n)})$
`synchronize`(F, W, R)

It is also possible to overlap computation and communication at this stage. The program would have to be changed to:

for all $e \in$ exported ranges **do**
  $F(n)(e) \leftarrow f(e, S^{(n)})$
`begin_synchronize`(F_i, W, R)
for all $e \in$ remaining ranges **do**
  $F^{(n)}(e) \leftarrow f(e, S^{(n)})$
`end_synchronize`(F, W, R)

This is known as *compute-and-send-ahead* strategy.

In addition, often *global reduction* operations are needed. Calling the result of such an operation $\widehat{\lambda}$, and the reduction operator $\oplus$ (associative, commutative), this can be formally written as

$$\widehat{\lambda} \leftarrow \bigoplus_{e \in \widehat{\mathcal{G}^k}} f(e, S)$$

Without further knowledge about the operator $\oplus$, we can calculate $\widehat{\lambda}$ by first *locally* calculating values λ_i on the *formally owned* elements $e \in \mathcal{F}_i^k$, and then applying a *global* reduction to the λ_i. The implementation of this latter reduction will depend on the physical distribution.

5.6 Distributed Overlapping Grids — Algorithms

The definitions of overlap structure and stencil-induced hulls in the preceding section rise the question, how these entities can be determined algorithmically. In this section, we present methods for performing these and other tasks related to distributed grids.

In section 5.6.1, issues related to synchronization of distributed grid functions are discussed. Section 5.6.2 describes a general algorithm INCIDENCE HULL for calculating hulls generated by arbitrary stencils. The construction of the entire overlap structure (algorithm CONSTRUCT OVERLAP, page 134) is the theme of section 5.6.3. Here also algorithm DISTRIBUTED HULL (p. 137) is presented, which calculates overlaps in the case of physically distributed grids, starting from a situation where only shared ranges exist. Determination of quotient grids is investigated in section 5.6.4 (algorithm QUOTIENT GRID, p. 138). Finally, section 5.6.5 discusses some aspects of dynamic grids, which are not yet fully worked out.

5.6.1 Static Communication

For static synchronization, only values of grid functions have to be exchanged. For this, one has to know the ranges where data has to be copied from or to, and the correspondence given by the Φ_{ij}. For a practical implementation, the notion of *synchronous iteration* turns out to be a key concept. By this, we mean that corresponding ranges are traversed in corresponding order: If we assume that corresponding overlap ranges $\mathcal{E}_{ij} \sim \mathcal{C}_{ji}$ are sequences, then

$$e_{ij}^n \sim c_{ji}^n, \quad 1 \le n \le N = |\mathcal{E}_{ij}|$$

In this case, the mappings Φ_{ij} are given *implicitly* by synchronous ordering of corresponding ranges.

In the simplest case, all what has to be done under the hood of a synchronization operation is copying N homogeneous data items from one location to another. How this exactly happens, depends on the context: In a *composite grid* residing in one global address space, just use ordinary copy. In a *distributed-memory* context, use some kind of message passing, as provided for example by the MPI or PVM libraries.

This simple case occurs if values on the shared range are completely computed on each local part. It also assumes that the data has a value semantic, that is, does not reference other data. This is certainly the case for numerical data.

The situation is slightly more complicated for the *partial computation* case: Here data on the shared range has to be combined by a reduction operation, typically a summation, cf. fig. 5.7. This introduces some complexity, because it must be ensured that each value is used exactly once, making it sometimes necessary to use temporary storage. Surprisingly, this problem is easier to solve in a distributed-memory context using a message-passing library, which already supplies the necessary buffering.

An important example for partial computation is Finite-Element stiffness matrix assembly, see [Bas94] or [Haa97]. There are no copied cells, and on each vertex only the matrix entries from the local cells are calculated. Here values on shared vertices have to be added to arrive at a global consistent state. In the case of matrices, one often works with data that is only locally consistent. Multiplication with vertex-based vectors result in likewise globally inconsistent data, which may be made consistent by a synchronization operation. The question of *when* to synchronize these data structures is intimately related to the algebraic meaning associated to them and cannot be delegated to the grid function level. However, the details of *how* to do this can be completely hidden.

Additional complexity arises, if some additional structure of the partition grid is to be used for optimization of communication, as discussed on page 121.

There are many more possibilities for the action triggered by a synchronization operation. So, it is obviously possible to include *periodic* boundaries in this context, see also page 133. Also, it can be used to transfer data between different types of grids *hybrid grids*, for example Cartesian and unstructured grids. Special *interface boundary conditions* could also be handled at this stage.

The technique of *overset* or *Chimera grids* (see e. g. [Pet97]) uses grids that overlap geometrically, but not combinatorially, that is, are not overlapping grids in the sense defined here. Nonetheless, the corresponding grid transfer operations can be hidden under the hood of distributed grid functions.

In principle, it is even possible to combine all these possibilities in a single application, thus achieving a very high degree of flexibility.

5.6.2 Determination of Hulls Generated by a Stencil

The algorithm Incidence Hull, presented in table 5.1, calculates a hull $\mathcal{H}$ in expected time $O(|\mathcal{H}|)$. This time bound crucially depends on the availability of partial grid functions to mark all grid elements as non-visited in constant time in a preprocessing step.

The algorithm works for general stencils, not just cell-based stencils. A fundamental precondition however is the availability of incidence iterators for each allowed sub-stencil of the form (a_k, a_{k+1}). For example, if $(0, d)$ is an allowed sub-stencil, there must be a CellOnVertex Iterator (→ p. 206) type defined for the underlying grid, if (d, f) is allowed, the grid has to provide iteration over facets incident to a cell.

For estimating the complexity of algorithm Incidence Hull, we make the assumption that all types of incidence relationships are bounded by a constant C independent of the grid size, that is,

$$|\mathcal{I}_k(e)| \leq C \quad 0 \leq k \leq d \quad \forall\, e \in \mathcal{G} \tag{5.20}$$

This is certainly a valid assumption for real-world grids; for example, in two dimensions, a value of $C \leq 10$ will do for the large majority of grids used in a PDE computation.

We note that only elements contained in the output are visited at all, due to the use of a partial grid function in line 1. The potentially expensive part of algorithm Incidence Hull is the loop in lines 7-11, and here the case that elements tested in line 8 more then once.

Condition (5.20) now implies that each element f_e in line 7 can be accessed at most C times, that is, the test in line 8 is performed at most C times for any element in the output, and never for any other element, that is, only $C \cdot |\mathcal{H}_I|$ tests are necessary.

Algorithm 5.1 INCIDENCE HULL: find hull generated by a stencil

IN: stencil $I = (a_0, \dots, a_n)$
IN: germ $\mathcal{K}$
OUT: levels $\mathcal{L}_I^{(0)}(\mathcal{K}), \dots, \mathcal{L}_I^{(n)}(\mathcal{K})$

visited ←false *(by default, partial grid function!)*
$\mathcal{L}_I^{(0)} = \mathcal{K}$
for all $e \in \mathcal{L}_I^{(0)}$ **do**
  visited(e) ←true
for $k = 1, \dots, n$ **do**
  for all $e \in \mathcal{L}_I^{(k-1)}$ **do**
    for all $f_e \in \mathcal{I}_{a_k}(e)$ **do** *(incidence iteration)*
      if not visited(f_e) **then**
        visited(f_e) ←true
        $\mathcal{L}_I^{(k)} \leftarrow \{f_e\} \cup \mathcal{L}_I^{(k)}$
      end if

5.6.3 Construction of Overlapping Grids

The problem of constructing overlapping grids is the following: Given a global grid, a partition of this grid and a stencil, determine the local grids $\mathcal{G}_i$ and the overlap ranges described in the previous sections. The actual construction of overlap structures and distributed grids differs slightly, depending on the configuration: There are differences depending on whether the whole global grid is available, whether the underlying partition is cell- or vertex-based, or whether the resulting grid is physically distributed or resides in one global memory.

For sake of simplicity, we first concentrate on the case that a global grid is given in local memory, and a composite grid is to be constructed locally. Further, we assume that the partition is cell-based. Under these circumstances, the basic algorithm 5.2 CONSTRUCT OVERLAP determines the necessary data.

For other configurations, CONSTRUCT OVERLAP has to be adapted. In the case of physically distributed grids, the construction could take place in parallel on each process, and in the final stage, only the local part $(\mathcal{G}_i, \mathcal{O}_i)$ gets constructed.

This approach does not scale to large numbers of processes, because the global grid might be to large to fit into a single memory. Below, we present algorithm 5.3 allowing to compute overlaps in a distributed setting. It starts from the local parts P_i and shared ranges $\mathcal{S}_{ij}$, and is the nucleus of a distributed construction of overlaps.

At periodic boundaries, the global grid first has to be enlarged by the appropriate overlap. Then new partitions are created in the enlarged parts, which point to the 'master' partitions in the original grid from which they were copied. This serves to distinguish the added grid parts from the original grid.

Then, the basic algorithm may be used, where the indices i, j refer to the enlarged set of partitions. Finally, when the grid parts $(\mathcal{G}_i, \mathcal{O}_i)$ are constructed, new partitions

---

Algorithm 5.2 Construct Overlap: construct overlapping grid from cell-based partition and stencil

---

IN: : $\mathcal{G}$ *(global grid)*
IN: : P *(cell-based partition of $\mathcal{G}$ into N parts)*
IN: : I *(stencil)*
OUT: : $(\mathcal{G}_i, \mathcal{O}_i), 1 \leq i \leq N$ *(local overlapping grids)*

(STEP I: Determine local overlap)
for all parts P_i **do**
    $\mathcal{S}_i \leftarrow \partial P_i$ *(shared range is boundary of partition)*
for all parts P_i **do**
    for j with $P_i \cap P_j \neq \emptyset$ **do**
        $\mathcal{S}_{ij} \leftarrow \mathcal{S}_i \cap P_j$
        $\mathcal{S}_{ji} \leftarrow \mathcal{S}_{ij}$ *(ensure synchronous sequences)*
for all parts P_i **do**
    $k \leftarrow I_0$
    if $k = d$ **then** *(stencil starts with cells)*
        $I_0 \leftarrow d - 1$
        $\mathcal{K} \leftarrow \mathcal{S}_i^{d-1}$
    else
        $\mathcal{K} \leftarrow \mathcal{S}_i^{k}$
    end if
    $\mathcal{C}_i \leftarrow \mathcal{H}_I(\mathcal{K})$ on $\mathcal{G} \setminus P_i$ *(copied range: outside partition P_i)*
    $\mathcal{E}_i \leftarrow \mathcal{H}_I(\mathcal{K})$ on P_i *(exposed range: inside partition P_i)*
for all parts P_i **do**
    for j with $\mathcal{C}_j \neq \emptyset$ **do**
        $\mathcal{C}_{ij} \leftarrow \mathcal{C}_i \cap P_j$
        $\mathcal{E}_{ij} \leftarrow \mathcal{C}_j \cap P_i$ *(ensure synchronous sequences)*
(STEP II: create local overlapping grids $\mathcal{G}_i$)
for all parts P_i **do**
    $\mathcal{G}_i \leftarrow P_i \cup \mathcal{C}_i$ (obtaining morphism Ψ_i)
    $\mathcal{O}_i \leftarrow (\mathcal{E}_i, \mathcal{S}_i, \mathcal{C}_i, \mathcal{E}_{ij}, \mathcal{S}_{ij}, \mathcal{C}_{ij}) \mod \Psi_i$

---

have to be re-identified with their masters. In this way, also overlap ranges of the form $\mathcal{O}_{ii}$ are possible. A slightly modified version of algorithm 5.2 can be used to incorporate periodic boundaries, containing the original version as a special case.

In a truly distributed setting, there will first take place a distributed grid generation. For this task, an initial subdivision of the computational domain has to be given. We can assume without loss of generality that such a subdivision is given by an initial (coarse) grid, which is known to *all* parts. Parallel grid generation is therefore equivalent to the parallel 'refinement' of some coarse grid, possibly consisting of just one cell per part. (The term 'refinement' has to be understood here in a broader sense than usual.) The coarse grid represents a common basis for the identification of grid entities from which a 'bootstrapping' approach can be used.

There will in general be involved a first step of surface grid generation, were lower dimensional grids are established on the coarse grid elements shared by several parts, that is, edges in 2D, and edges followed by facets, in 3D.

Only if the grid generation process is regular in the sense that the result is equal and easy to match on each part, this first step can be omitted. Such a situation would occur for example if grids with Cartesian structure are generated on each part, especially if the coarse grid itself is (part of) a Cartesian grid. More generally, a refinement in the usual FEM-sense following a regular pattern allows direct identification of independently created grid elements, because the parent elements are already identified.

Parallel grid generation is not an easy task, especially if load balancing is taken into account, see e. e. [SFdC+97].

After grid generation, the shared ranges $\mathcal{S}_i$ and $\mathcal{S}_{ij}$ can be assumed to be available (if grid generators do not allow to identify input boundary elements like vertices in their output, one could resort to devices like geometric matching of entities).

The total ranges $\mathcal{E}_i$ may then be determined by the hull-algorithm. For the bilateral ranges $\mathcal{C}_{ij}$ we adopt an iterative procedure, based on the following observation:

Theorem 6. *In the global grid $\widehat{\mathcal{G}}$, for any $1 \leq k \leq N$, the copied range $\widehat{\mathcal{C}_{ik}}$ needed in P_i is contained in the union of the copied ranges of direct neighbors P_j of P_i (in the quotient $\mathcal{G}/_P$ (→ p. 121)), that is,*

$$\widehat{\mathcal{C}}_{ik} \subset \bigcup_{j \in A_i} \widehat{\mathcal{C}}_{jk} \qquad \textit{with} \quad A_i = \{j | P_i \textit{adjacent to} P_j\} \tag{5.21}$$

Proof. (We omit the $\widehat{.}$ sign for indicating that we are operating on the global grid.) For a cell c, $c \in \mathcal{C}_{ik}$ implies

$$\begin{aligned} c &\in \mathcal{H}_I(\mathcal{S}_i) \cap \mathcal{G}_k \\ &= \mathcal{H}_I(\bigcup_{j \in A_i} \mathcal{S}_{ij}) \cap \mathcal{G}_k = \bigcup_{j \in A_i} \mathcal{H}_I(\underbrace{\mathcal{S}_{ij}}_{\subset \mathcal{S}_j}) \cap \mathcal{G}_k \\ &\subset \bigcup_{j \in A_i} \mathcal{H}_I(\mathcal{S}_j) \cap \mathcal{G}_k = \bigcup_{j \in A_i} \mathcal{C}_{jk} \end{aligned}$$

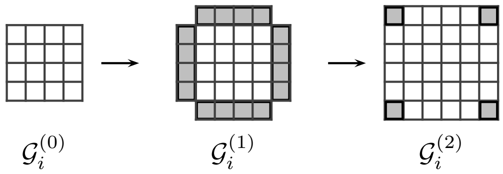


Figure 5.8: Two steps in constructing the overlap of grid $\mathcal{G}_i$ by algorithm Distributed Hull. Stencil is $I = (2, 0, 2)$.

□

The inclusion is, of course, interesting in particular if P_i and P_k are *not* direct neighbors. One can, however, construct example where P_i and P_k *are* direct neighbors, but a cell in $\mathcal{C}_{ik}$ can only be reached over a third part.

Theorem 6 leads us to the following iterative procedure for obtaining the local overlap ranges (cf. figure 5.8): We set $\mathcal{G}_i^{(0)} = P_i$, and recursively define the next grid $\mathcal{G}_i^{(k+1)}$ to be the hull taken in all direct neighbors in the current grid:

$$\mathcal{G}_i^{(k+1)} = \mathcal{G}_i^{(k)} \bigcup_{j \in N} \mathcal{H}^j(\mathcal{S}_{ji})$$

Here $\mathcal{H}^j$ is the hull operation restricted to the current grid $\mathcal{G}_j^{(k)}$ (the fixed stencil I has been omitted).

We now prove that algorithm 5.3 really calculates the overlap ranges in a finite number of steps.

Theorem 7. *Algorithm 5.3 obtains the copied ranges $\mathcal{C}_{ik}$ for a stencil $I = (d, d_1, \ldots, d_r, d)$ in at most*

$$\sum_{j=1}^{r} \max_{\mathfrak{e} \,\in\, \mathfrak{G}^{d_j}} \operatorname{diam}(\operatorname{st}(\mathfrak{e}))$$

steps. Here $\mathfrak{G}$ is the quotient grid, and with $\operatorname{diam}(\operatorname{st}(\mathfrak{e}))$ *we mean the maximal distance of two cells in* $\operatorname{st}(\mathfrak{e})$*, when passing to another cell is allowed only through a facet.*

Proof. We proceed by induction over $|I|$. If $|I| = 1$, let $I = (d, d_1, d)$. Let $c_i \in \mathcal{G}_i^{(0)} = P_i$ and $c_k \in \mathcal{G}_k^{(0)} = P_k$ such that $\widehat{c_k} \in \mathcal{H}_I(\widehat{c})$ in the global grid. We must show that c_k gets copied into $\mathcal{G}_i$ in a finite number $n = n(I)$ of steps.

There exists $\widehat{e}$ such that $\widehat{c_i} > \widehat{e}$ and $\widehat{c_k} > \widehat{e}$. Let $\mathfrak{e} = [\widehat{e}]$ be the quotient element corresponding to $\widehat{e}$. In the quotient grid, there is a path of direct neighbors leading from $[P_i]$ to $[P_k]$ in the star $\operatorname{st}(\mathfrak{e})$. Let this path be $(P_i = P^{(0)}, \ldots, P^{(n)} = P_k)$.

The cell c_k will be copied along this path, going from P_k to P_i. This is because in every part $P^{(j)}$, there is a representant of $\widehat{e}$, which is therefore shared by all these cells. Therefore, c_k is copied to $P^{(n-1)}$ in the first step, and to $P^{(0)} = P_i$ in the nth step.

Here $n = n(I)$ is bounded above by the maximal distance of two parts P_i, P_k, incident to the same element $\mathfrak{e}$ of dimension $\geq d_1 = \dim \widehat{e}$, that is, $\mathrm{diam}(\mathrm{st}(\mathfrak{e}))$.

The induction step is now easy. If $|I| > 1$, let $I = (I_1, I_2)$ with $|I_1|, |I_2| < |I|$. The hull $\mathcal{H}_I(\widehat{c_k})$, containing $\widehat{c_i}$, also contains an intermediate cell $\widehat{c_m} \in P_m$, such that

$$\widehat{c_i} \overset{I_1}{\longmapsto} \widehat{c_m} \overset{I_2}{\longmapsto} \widehat{c_k}$$

that is, $\widehat{c_m} \in \mathcal{H}_{I_1}(\widehat{c_i})$, and $\widehat{c_k} \in \mathcal{H}_{I_2}(\widehat{c_m})$. Thus, c_k is copied to $\mathcal{G}_m$ in $n(I_2)$ steps, and further to $\mathcal{G}_i$ in $n(I_1)$ steps, thus proving also the formula for $n(I)$.

If no new elements found in a step for any part, that is, $\mathcal{G}_i^{(k+1)} = \mathcal{G}_i^{(k)} \forall i$, then none will be found in further steps, and the algorithm terminates. $\square$

Algorithm 5.3 DISTRIBUTED HULL: calculate stencil hull on distributed grid

IN: $\mathcal{G}_i^{(0)} = P_i, 1 \leq i \leq n$
IN: I *(stencil)*
IN: $\mathcal{S}_{ij}, 1 \leq i, j \leq n$ *(shared ranges)*
OUT: $\mathcal{G}_i, 1 \leq i \leq n$ *(completed distributed grids)*
OUT: $\mathcal{C}_{ij}, \mathcal{E}_{ij}$ *(synchr. copied/exposed ranges)*

```
k ← 0
repeat
  for all parts P_i do   (parallel)
    for all direct neighbors P_j of P_i do
      calculate hull H_ij of stencil with germ S_ij
      transfer all new elements of H_ij to P_j,
      including owner and id
    for all direct neighbors j of i do
      get new elements of H_ji from P_j
      for all e ∈ H_ji do
        if e is still unknown then
          l ← owner of e
          C_il ← {e} ∪ C_il
        end if
    G_i^(k+1) ← G_i^(k) ∪ {new elements}
    k ← k + 1
until no new elements for any i have been added
```

5.6.4 Determination of the Quotient Grid

Algorithm 5.4 determines the set constituting the elements of $\mathcal{G}/_P$. To prove that the algorithm is correct, we show that the definition of $\sim_k$ in 5.5 is equivalent to just considering incident element of the next higher dimension:

$$e_1 \sim_k e_2 \iff [\mathcal{I}_{k+1}(e_1)] = [\mathcal{I}_{k+1}(e_2)] \tag{5.22}$$

Proof. We need to show that $[\mathcal{I}_{k+1}(e_1)] = [\mathcal{I}_{k+1}(e_2)]$ implies $[\mathcal{I}_n(e_1)] = [\mathcal{I}_n(e_2)]$ for higher dimensions $k+1 \leq n \leq d$. This is trivially true for $k = d-1$, and it follows for all k by induction over $l = d-k$:

Let $e_1, e_2 \in \mathcal{G}^k$ be such that $[\mathcal{I}_{k+1}(e_1)] = [\mathcal{I}_{k+1}(e_2)]$, and let $e_1^n > e_1$ an element of dimension $n \geq k+2$. Because the poset of $\mathcal{G}$ is graded, there exists e_1^{k+1} 'in between' : $e_1^n > e_1^{k+1} > e_1$. By 5.22, there is $e_2^{k+1} > e_2$ with $e_1^{k+1} \sim_{k+1} e_2^{k+1}$, where by induction, it does not matter whether definition 5.22 or 5.5 is taken for $\sim_{k+1}$. Therefore, there is $e_2^n > e_2^{k+1}$ with $e_1^n \sim_n e_2^n$. This shows that $[\mathcal{I}_n(e_1)] = [\mathcal{I}_n(e_2)]$. □

In case the quotient has the combinatorial structure of a Cartesian grid, then 'communication across the diagonal' can be saved by clever ordering of bilateral communication. This is shown in figure 5.4.

Algorithm 5.4 QUOTIENT GRID: determine elements of quotient grid

---

IN: $\mathcal{G}$ *(grid)*
IN: P *(partitioning of $\mathcal{G}$ in N parts P_i)*
OUT: element set of $\mathfrak{G} = \mathcal{G}/_P$

1: **for** $i = 1, \dots, N$ **do** *(determine cells of $\mathfrak{G}$)*
2:   $\mathfrak{e}_i^d \leftarrow P_i$
3: **for** $k = d-1, \dots, 0$ **do** *(determine k-elements of $\mathfrak{G}$)*
4:   **for all** $\mathfrak{e}^{k+1} \in \mathfrak{G}^{k+1}$ **do**
5:    *($k+1$-elements are already determined and numbered $1, \dots, N_{k+1}$)*
6:     **for all** $e^{k+1} \in \mathfrak{e}^{k+1}$ **do**
7:       **for all** $e^k \prec e^{k+1}$ with $\dim(e^k) = k$ **do**
8:        *(M maps elements e_k to set of quotient-elements $\mathfrak{e}_{k+1}$ with $e_k \in \mathfrak{e}_{k+1}$)*
9:         $M(e^k) \leftarrow M(e^k) \cup \{\mathfrak{e}^{k+1}\}$
10:   remove singletons from M
11:   **for all** $A \in M$ **do**
12:     **for all** $e^k \in \bigcup_{\mathfrak{e}^{k+1} \in A} \mathfrak{e}^{k+1}$ **do**
13:      *(build sets of k-elements forming the quotient k-elements)*
14:       $E(A) \leftarrow E(A) \cup \{e^k\}$
15:   sort domain of E according to lexicographic order on the sets $M(e^k)$
16:   **for** $l = 1, \dots, |\operatorname{dom}(E)|$ **do** *(number elements of $\mathfrak{G}^k$)*
17:     $\mathfrak{e}_l^k = E_l$

In line 4 of algorithm 5.4, we can easily optimize for the case $k = d-1$ by considering only those facets e, which are on the boundary of a partition P_i. Interior facets are removed in line 10 anyhow. For this purpose, a boundary iterator as presented in section 4.2.2.4 can be used. In line 7, *closure iterators* (section 4.2.2.3) are useful to access all elements of lower dimension of an element set.

5.6.5 Dynamic Grid Migration

Adaptive algorithms produce locally modified grids. As a result, work load may become inbalanced; parts of local grids must be migrated between processes.

As mentioned before (section 5.3), the components needed to support dynamic grid migration have not yet been fully developed. However, on the one hand, the necessary actions can be almost completely hidden from the application program, and thus the advantage of our approach – minimal invasiveness – will not be compromised.

On the other hand, as we are going to point out, much of the work can be accomplished — or at least be eased considerably — by 'standard' components we have introduced earlier.

More concretely, we need components solving the following tasks in order to support dynamic grid migration:

1. determining which chunks of grids have to be transferred to other parts
2. transporting grid chunks to other parts
3. transferring grid functions defined on migrated chunks
4. cutting grid chunks out of an existing grid
5. gluing a grid chunk to another grid.
6. regenerating the additional data structures associated with distributed grids, in particular overlap structures.

As has been mentioned before, the first task can be solved by an existing component like ParMetis [Kar99].

Transporting a grid means serialization/de-serialization in general, a problem largely solved by grid adapters for serialized representations (→ p. 95), and semi-generic copy operations (→ p. 93). The tasks of cutting and gluing grid chunks have been treated in section 4.1.6.

Transferring grid functions defined on grid parts is somewhat similar to the case of static synchronization, but one has, in addition, to keep track of the relationship between local grid chunks and their remote copies. As all copy/glue operations can maintain grid isomorphisms between source and copy, this problem can be solved by combining the isomorphisms involved.

Finally, we have presented an algorithm for distributed generation of overlap in section 5.6.3.

So, *in principle*, most of the necessary ingredients for dynamic grids are available. However, some additional work will be necessary to fit them together.

5.7 Distributed Overlapping Grids — Generic Components

Most of the concepts presented in the preceding sections map smoothly onto software components. The practical implementation confirms that the functionality of the grid micro-kernel is indeed a sufficient basis for supporting the distributed grid concepts. Thus, the aim of providing a generic implementation could be achieved; the components can be used *with any grid type* (base grid) implementing the interface. At appropriate places, we will point out requirements of particular components on the base grid.

Leaving internal data representation unchanged is of particular importance when parallelizing given codes, e. g. large numerical simulations, that operate directly on this concrete representation. In general, the concrete grid type used in an existing application needs to be adapted to the grid micro-kernel in order to provide the required interface. This can be achieved with the help of *additional* code alone; in particular, without changing internal representation details. In section 6.4, we present a detailed case study.

It has been pointed out at the beginning of section 5.5.2 that the concepts developed largely operate on an 'abstract distribution' level: The physical nature of distribution does not matter. In an analogous manner, these issues can be abstracted from in most concrete components; details are factored out in a data transport layer, see figure 5.9 and section 5.7.2.

In particular, essential features of distributed grids and grid functions can be factored out into common bases *overlapping grid* and *overlapping grid function*, see also figure 5.10. Specializations to concrete physical distributions are **MPDistributedGrid** (for distributed memory and message passing) and **CompositeGrid** (for single memory). The latter one is quite a useful component, it may be used for testing algorithms in a sequential setting, but can also serve to build multi-block grid structures. Analogous specializations exist for overlapping grid functions.

At times, substantial gains can be made by using specialized components for particular situations, notably Cartesian grids. Possibilities for doing so will be pointed out at the appropriate locations.

5.7.1 Data-centered Components

5.7.1.1 Overlap Representation

The grid ranges introduced in section 5.5.2 form a logical sequence from which it is often necessary to select intervals, for example, to operate on *exported* ranges composed of shared and exposed ranges. In principle, one could use standard grid ranges presented in section 4.2.1, but it is more economic to create a new set of components for this purpose.

LayeredRanges This component is parameterized over the element type. The whole

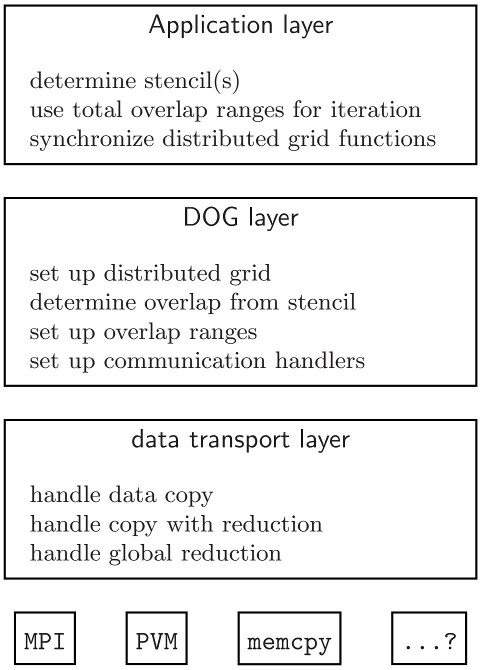


Figure 5.9: Layers and their responsibilities in an application parallelized using the *distributed overlapping grids* concepts

range formed by four basic ranges (*private*, *exposed*, *shared*, *copied*) is represented by a sequence of element handles, and intervals (basic as well as derived ranges, p. 120) can be obtained by indicating start and end positions.

Overlap An overlap is parameterized over a fine base grid and a coarse quotient grid. It is tied to a cell P of the quotient and provides the following:

- A reference to the quotient poset (or at least to local neighborhood of P)
- A reference to the local base grid
- For each element type, a layered range over the local base grid (the total ranges)
- For each element type, a mapping of quotient cells to layered ranges (bilateral ranges)

The mappings defining the bilateral ranges can easily be implemented by partial grid functions over the quotient.

Strictly speaking, the reference to a quotient is not necessary in all circumstance, however, if special optimizations are done, depending on properties of the quotient (e. g. Cartesian, see page 121), this component *is* needed.

5.7.1.2 Quotient Grid Representations

In principle, we could use a sufficiently general grid component for the quotient, which is done until now in the concrete implementation.

However, for general partitionings, such a component would have to allow disconnected elements and elements with holes, and thus be too general for most other contexts.

Therefore, it is useful to provide a component **QuotientPoset** implementing the mathematical notion of a graded poset. This component could provide mappings to underlying entities of the fine grid, by using grid functions from quotient elements to subranges of the fine grid.

In some cases, a more specialized component for representing the quotient will be useful, for example, if the quotient is a (pseudo-) Cartesian grid. Then, also a specialized version of **QuotientGenerator** (page 145) would be necessary.

5.7.1.3 Distributed Grids and Grid functions

These components fall into three layers: *overlapping*, containing overlap structures but with a 'local semantics', *distributed*, standing for the ensemble of parts, and *global*, allowing to operate on the logical global grid, for example for collecting data on a single machine. See also figure 5.10 for an overview.

layer	grid components	grid function components
overlapping	**OverlappingGrid** (OG)	**OverlappingGridFunction** (OGF)
distributed	**MpDistributedGrid** (MpDG) **CompositeGrid** (CDG)	**MpDistributedGridFunction** (MpDGF) **CompositeGridFunction** (CDG)
global	**MpGlobalGrid** (MpGG) **CompositeGlobalGrid** (CGG)	**MpGlobalGridFunction** (MpGGF) **CompositeGlobalGridFunction** (CGGF)

5.7.1.3.1 Overlapping grids / grid functions

OverlappingGrid (OG) This component, like an overlap, is parameterized over a fine grid type (or base grid) and a grid quotient type. It consists of a fine grid and an overlap. For use by application algorithms, it grants access to the total and bilateral ranges (→ p. 119). The incidence iterators of the local range type are in general identical to that of the base grid.

Overlapping Gridfunction (OGF) Overlapping grid functions relate to overlapping grids like ordinary grid functions to sequential grid. An OGF — parameterized over types of elements and value — contains a local grid function and a reference to an overlapping grid. Moreover, it may contain its own ranges, which must be subsets of the corresponding ranges of its overlapping grid.

5.7.1.3.2 Distributed grids / grid functions *Distributed grid* is not a concrete component, but a collective term for a set of components. It is here where the physical nature of distribution comes into play.

To each type of distributed grid corresponds a distributed grid function (DGF). DGFs are accompanied by global reduction operations, like sum, minimum and maximum.

MpDistributed Grid (MpDG) A *message-passing distributed grid* is physically distributed, and all access to remote data is done over a message passing interface like MPI. There is a MpDistributed Grid for each cell of the quotient, each living in its own process. In addition to an overlapping grid, a MpDistributed Grid contains a grid quotient (or a sufficient local portion of it). The difference to an overlapping grid is that the former only stands for the local part, whereas a MpDistributed Grid represents the *whole* grid distributed over the processes.

On initialization, some work has to be done to map the given grid quotient topology on a process topology. With MPI, this can be achieved by a call to `MPI_Graph_create()`.

Composite Grid (CG) This component, in contrast to a physically distributed grid, contains all overlapping grids, e. g. by using a grid function mapping quotient cells to overlapping grids, and, of course, the quotient itself.

MpDistributed Gridfunction (MpDGF) This class contains an overlapping grid function and lists of *communication handlers*, that manage the actual data transport. Their main additional functionality consists in methods
`begin_synchronize()` and `end_synchronize()`
to achieve a globally consistent state. Also, they can be parameterized (in addition to element and value) by a reduction operator specifying the semantics on shared ranges. By default, no data exchange is necessary there.

Global reduction over a MpDGF is logically equivalent reduction over the associated *global grid function*. In the case of MPI, it can be performed by first calculating on the *formally owned* (→ p. 120) part of the local grid function and then using the reduction offered by the data transfer layer, which will encapsulate a `MPI_Reduce()` routine. Reduction is generally only meaningful if the MpDGF is in a globally consistent state.

Composite Gridfunction (CGF) A composite grid function contains a reference to a composite grid and a mapping from the coarse cells to overlapping grid functions over the corresponding parts (local grid functions). As MpDGFs, they provide methods `begin_synchronize()` and `end_synchronize()`.

Global reduction over a CGF is done by looping over all local grid functions, performing the reduction on the *formally owned* elements, and applying the reduction to the vector of results of each part.

5.7.1.3.3 Global grids / grid functions Global grids are abstractions representing a global view on the distributed grid, as defined on page 118. They are practical for collecting information on a master, or, more generally, for concentrating (and redistributing) grid parts on fewer processes.

Global grid functions are relate to distributed grid functions the same way global grids relate to distributed grids. Normally, we consider global grid function as immutable.

MpGlobal Grid (MpGG) A MpGlobal Grid concentrates information from each part of a MpDistributed Grid. A use of this grid requires the transfer of the distributed grid's remote parts to the process where the MpGlobal Grid lives, as well as the transport of identification information on the bilateral boundaries of grid parts. The underlying grid data-structure may be the same as those used for the local grids.

Composite Global Grid (CGG) In the case of a composite global grid, there is not need to copy information for a composite global grid. Instead, the grid functionality can be implemented by *two-stage iterators*, plus some additional book-keeping data-structures.

MpGlobal Gridfunction (MpGGF) MpGGFs collect information from the MpDistributed grid functions. They contain a reference to a MpGlobal Grid, and a grid function over the underlying grid of the former. The use of a MpGGF makes the transport of data from each MpDGF necessary, plus possibly a reduction on shared elements.

Composite Global Gridfunction (CGGF) These components contain a reference to a Composite Global Grid. Like this, there is no need to copy information.

5.7.2 The Data Transfer Layer

The actual mechanics of data transport is one of the things that depend on the underlying hardware. Encapsulating these details shields components like distributed grid functions from committing to a particular distribution situation.

Also, this layer hides decisions like whether to use a reduction operation on a range (for example addition on shared element, see page 131), or even if some interpolation algorithm has to be used (for overset grids).

MPITransferHandler Data is transported via calls to MPI's non-blocking send and receive operations. It is parameterized over the sequence type of the local range, and an optional reduction operation.

A possible extension is a mapping on the receiver side, allowing to pass data with reference semantics.

InCoreTransferHandler In a global address space, data is transported by a simple copy operation. Parameterization is by the types of source and destination ranges.

InCoreBufferedTransferHandler Data is transported by a simple copy operation, via a buffer. This is important for reduction (on shared ranges), because data must first be copied before it may be changed by reduction with data from other sources. The type of the reduction is an additional, optional parameter.

MpGlobalReduction Global reduction of one data item per grid part is performed via message passing. The component is parameterized over the type of the data and the reduction operation.

5.7.3 Algorithmic Components

Quotient generator The procedure `CoarseGridFromPartition` takes as input parameters a partitioned grid, and constructs a grid quotient from this information. For the two-dimensional case, a simpler algorithm than that found in table 5.4 is implemented.

In case a special type of quotient is to be obtained (e. g. Cartesian), a special implementations has to be used.

Hull generator The procedure `IncidenceLayers` takes a grid G, a predicate I distinguishing an inside and an outside part of G, a set F of facets having one inside and one outside cell each, and an incidence-stencil S. It outputs a layered cell-range C which is the 'inside semi-hull' generated by S from germ F:

$$C = \mathcal{H}_S^{G^i}(C_F^o)$$

where $G^I = \overline{\{c \in G^d | I(c)\}}$ is the 'inside' part of G, and

$$C_F^o = \{c \in G^d \,|\, \exists f \in F : c \succ f \wedge \neg I(c)\}$$

is the set of outside cells incident to a facet from F.

This component builds on a number of simpler components: Grid ranges that work by masking out a part of the grid, partial grid functions for marking visited elements, and an incidence iterator for each pair (k, l) found in the stencil S.

This last dependency poses an interesting programming problem, because ideally, given a grid type, the implementation of INCIDENCE HULL should use exactly these iterators (and be able to process the corresponding sequences). This could be achieved by adding compile-time flags to grid types indicating the presence or absence of each possible iterator type, and building the algorithm correspondingly. At present, it is simply assumed that CellOnCell, VertexOnCell and CellOnVertex incidence iterators are implemented (see appendix A.2.3 ff.), which is sufficient for most FV/FEM algorithms.

Overlap generator The procedure `ConstructOverlap` takes a partitioned grid (G, P), the corresponding quotient $\mathfrak{G}$, and a stencil S. It outputs a mapping from cells of $\mathfrak{G}$ to overlaps as described above. Internally, it works as follows:

1. determine total shared ranges (internal boundaries of partitioned grid)
2. determine bilateral shared ranges (synchronous!)
3. determine total exposed and copied ranges (with `IncidenceLayers`)
4. determine bilateral exposed and copied ranges (synchronous!)

A enhanced version `ConstructOverlapPeriodic` allows in addition to handle periodic boundaries, given by a map identifying boundary vertices.

Composite grid generator For constructing a composite grid CG, the procedure `ConstructComposite` takes a partitioned grid (G, P), and a stencil S. Besides CG, it constructs mappings between the local and the global grid vertices and cells. The algorithm used is the following:

1. construct overlap O_g tied to the global grid G, using `ConstructOverlap`
2. construct local grids from (G, P) with morphisms $\Phi_{l,g}$ mapping parts of G to local grids.
3. copy the overlap O_g to the local overlaps O_l, using the morphism $\Phi_{l,g}$.

This presentation had to be terse. A more detailed description of the components will be published in a technical memo. From an application programmers point of view, however, many of these components are not *directly* relevant.

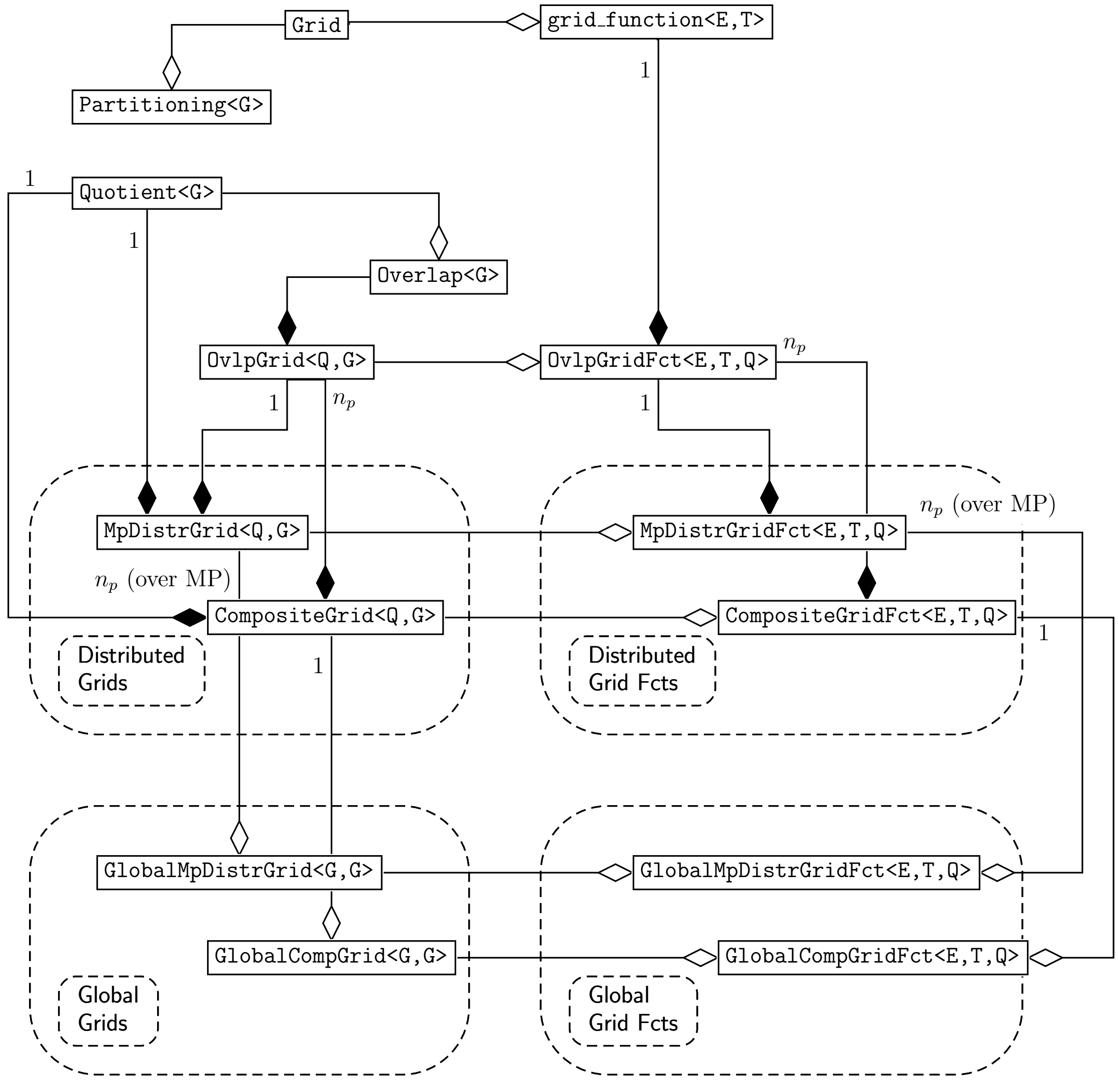


Figure 5.10: The interrelationships of data-centered distributed grid components in UML-like notation

Chapter 6

Practical Experience with the Generic Approach

Leicht beieinander wohnen die Gedanken
Doch hart im Raume stoßen sich die Sachen.
Friedrich Schiller, *Wallensteins Tod*

6.1 Introduction

The purpose of this chapter is to provide case studies evaluating the utility of the generic approach presented in this work — how do generic components help building real applications, and how well do they perform in terms of efficiency?

For doing so, we study several concrete application examples in the context of numerical PDE solution, which has been our point of departure, and has guided thought throughout this thesis.

In order to show the breadth and soundness of the approach, we treat quite different classes of problems. Therefore, one can observe a substantial diversification on the side of numerical methods used. It is here where the essential differences between applications for solving different types of PDEs arise.

In section 6.2, we introduce a class of problems related to physical flow phenomena, and described by hyperbolic equations. A large family of algorithms used to solve such problems are *finite volume* (FV) methods.

We show how a generic FV solver is constructed from individual algorithmic parts, giving rise to the nucleus of a *program family*. Individual programs may differ, among others, in the exact equations solved, the algorithms used for spatial discretization, the execution context (sequential/distributed), and the data structures used (grid, geometries). This is an example how generic programming works on a large scale.

Elliptic problems, on the other hand, give rise to a different family of algorithms, namely *finite element* methods (FEM), which are introduced in section 6.3. We discuss issues related to the generic implementation of components for *multigrid* methods, a class of algorithms particularly suited to solve the large linear systems that arise in the FEM context. In particular, the topic of generic data structures for hierarchical grids is treated.

Reuse of generic components does *not* mean that the whole application has to be designed and written with the generic paradigm in mind. Rather, the components can be used by existing software in an *incremental* way. An example illustrating this possibility is presented in section 6.4. Here, a Navier-Stokes solver is parallelized *a-posteriori* by using the distributed grid components described in chapter 5. The ease of obtaining parallel applications is a convincing argument for the advantages of the generic approach.

Grid components, and in particular the set of basic kernel components we have presented earlier, are only a part of what is actually needed in a numerical simulation. Evidently, generic components could not be developed for *all* aspects.

Therefore, it is difficult to obtain quantitative measures of the degree of reuse obtained by generic components, by giving percentages of reused code.

It has become evident, however, that components serving the needs of different applications, like grid-related data structures or visualization routines, can indeed be reused 'off the shelf', which was the aim formulated in chapter 2.

Such a reuse requires that the underlying grid type conforms to the micro-kernel. which was the case for the newly developed applications discussed in sections 6.2 and 6.3. For existing applications with a different grid data structure, as in section 6.4, an adaptation has to be developed, which is a rather standard task of strictly limited scope. In the case of parallelization using distributed grid components, this comparatively little effort definitely pays off.

For numerical grid-based components, generic components have been developed, some of which have been already reused successfully in *different* contexts.

A quite different topic is the *efficiency* obtainable with generic components. For evaluating this aspect, a number of small computing kernels was set up and used in section 6.5 to compare diverse generic and non-generic implementations. Although the results have to be taken with a pinch of salt, they clearly indicate that modern compiler technology approaches a point where the overhead introduced by layers of abstraction can be eliminated.

6.2 A Finite Volume Solver for the Euler Equations

6.2.1 The Mathematical Problem

An inviscid, compressible, instationary flow in a domain $D \subset \mathbb{R}^2$ is described by the time-dependent Euler equations

$$\boldsymbol{u}_t + f(\boldsymbol{u})_x + g(\boldsymbol{u})_y = 0 \quad \text{in} \quad D \tag{6.1}$$

with the state vector $\boldsymbol{u} : D \times \mathbb{R}^+ \mapsto \mathbb{R}^4$ in *conservative variables*

$$\boldsymbol{u} = (\rho, \rho u, \rho v, \rho E)^T$$

where ρ is the density, $(u, v)^T$ the velocity vector and E the total energy; and the *flux functions*

$$\begin{aligned} f(\boldsymbol{u}) &= (\rho u, \rho u^2 + p, \rho uv, u(\rho E + p)) \\ g(\boldsymbol{u}) &= (\rho v, \rho uv, \rho v^2 + p, v(\rho E + p)) \end{aligned}$$

Here the pressure $p = p(\boldsymbol{u})$ is linked to the state $\boldsymbol{u}$ by a thermodynamic *equation of state*, for example

$$p(\boldsymbol{u}) = (\gamma - 1)\left(\rho E - \frac{1}{2}\frac{(\rho u)^2 + (\rho v)^2}{\rho}\right) \tag{6.2}$$

for an *ideal, polytropic gas*. Equation (6.1) has to be complemented by an initial condition

$$\boldsymbol{u}(x, y, 0) = \boldsymbol{u}_0 \tag{6.3}$$

Also, boundary conditions are needed on ∂D. This is a difficult topic, because the exact nature of these conditions depends on the concrete physics of the flow. The three most important types of boundary conditions are the following: *Inflow* boundary conditions, where (part of) the flow variables are prescribed, *outflow* boundary conditions, where either nothing or a part of the variables are prescribed, and *solid wall* boundary conditions, where the flow is required to be tangential to the boundary.

The Euler equations are an example of a *conservation law*: They express the conservation of mass, momentum and energy. This is better seen when considering the integral formulation, obtained by integrating over an arbitrary volume Ω:

$$\int_\Omega \boldsymbol{u}_t \, dV + \int_\Omega f(\boldsymbol{u})_x + g(\boldsymbol{u})_y \, dV \tag{6.4}$$

$$= \frac{\partial}{\partial t}\int_\Omega \boldsymbol{u} \, dV + \int_{\partial\Omega} f(\boldsymbol{u}) n_x + g(\boldsymbol{u}) n_y \, ds \tag{6.5}$$

where $\vec{n} = (n_x, n_y)^T$ is the outward normal of Ω. The integral $\int_\Omega \boldsymbol{u}_t \, dV$ represents the temporal change of the conserved quantity $\boldsymbol{u}$, and $\int_{\partial\Omega} f(\boldsymbol{u}) n_x + g(\boldsymbol{u}) n_y \, ds$ is the *flux* of $\boldsymbol{u}$ across the boundary of Ω. Conservation of $\boldsymbol{u}$ over time is expressed by the fact that the two integrals sum up to zero.

Formulation (6.5) not only gives insight into the physical origin of these equations, but also leads to an important class of numerical methods for solving them, namely the *Finite Volume Methods* (FVM), which are discussed in section 6.2.2.

The Euler equations are *hyperbolic*: The matrices

$$A(\boldsymbol{u}, \vec{n}) = f(\boldsymbol{u})_{\boldsymbol{u}} n_x + g(\boldsymbol{u})_{\boldsymbol{u}} n_y$$

are diagonalizable with real eigenvalues for every state $\boldsymbol{u} \in \mathbb{R}^4$ and every direction $\vec{n} \in \mathbb{R}^2$. Also, the fluxes f and g are 1-homogeneous, that is

$$f(\boldsymbol{u}) = f_{\boldsymbol{u}} \cdot \boldsymbol{u} \qquad \text{and} \qquad g(\boldsymbol{u}) = g_{\boldsymbol{u}} \cdot \boldsymbol{u}$$

This is important, because some numerical methods exploit exactly this property, or, put the other way around, can be formulated *generically* for equations with this property.

A characteristic feature of hyperbolic equations is that they may have discontinuous solutions. This phenomenon occurs for example in the sudden bang of a supersonic aircraft.

Also, they exhibit a finite speed of the flow of information, which is best understood by considering the scalar *advection equation* with $f(u) = au$:

$$\begin{aligned} u_t + au_x &= 0 \qquad a \in \mathbb{R} && (6.6) \\ u(x,0) &= u_0(x) && (6.7) \end{aligned}$$

which has the analytical solution

$$u(x,t) = u_0(x - ta)$$

This means that the profile u_0 is advected with speed a. As a linear hyperbolic equation, the advection equation is also homogeneous. The same is true for linear hyperbolic systems, which in two dimensions take the form

$$\boldsymbol{u}_t + A\boldsymbol{u}_x + B\boldsymbol{u}_y = 0$$

Other examples for hyperbolic conservation laws (though not homogeneous) are the scalar Burgers equation or the shallow water equations.

6.2.2 Finite Volume Algorithms

6.2.2.1 The Finite Volume Idea

From the formulation (6.5), one can derive methods for numerically solving the Euler equations, as well as other hyperbolic conservation laws.

First, one has to chose *control volumes* $\Omega_i, 1 \leq i \leq N$ that cover the geometric domain under consideration. This is done by choosing an appropriate grid, whose cells act as control volumes. Another possible approach is to use the dual of a given grid for the control volumes, such that there is a volume corresponding to each vertex of the original grid.

Second, it is necessary to approximate the integrals of (6.5). The unknown solution $\boldsymbol{u}$ is represented by cell-wise constant values $\boldsymbol{u}^i$ approximating the integral means of the true solution. Integral means are a *conservative* approximation to a function. Then we can approximate the first integral by

$$\frac{\partial}{\partial t}\int_{\Omega_i} \boldsymbol{u}\, dV \approx |\Omega_i| \frac{\partial}{\partial t}\boldsymbol{u}_i \quad 1 \le i \le N \tag{6.8}$$

(Here $|\Omega_i|$ is the volume of Ω_i.) This leads to the *semi-discretized* system

$$\begin{aligned} \frac{\partial}{\partial t}\boldsymbol{u}_i &= -\frac{1}{|\Omega_i|}\int_{\partial\Omega_i} f(\boldsymbol{u})n_x + g(\boldsymbol{u})n_y\, ds & (6.9)\\ &=: -\frac{1}{|\Omega_i|}\mathcal{F}^i(\boldsymbol{u}) \quad 1 \le i \le N & (6.10)\end{aligned}$$

which is a system of N ordinary differential equations in t. We defer the decision of how to discretize $\boldsymbol{u}$ in *time*.

6.2.2.2 First Order Upwind Discretization by Flux Vector Splitting

The second integral of 6.5 is the more complicated part. As the approximate solutions are represented in a cell-wise constant manner, the integration of the flux functions over the cell boundaries is not well defined. There are several possibilities of dealing with this situation, one of which is the *upwind* flux-vector-splitting (FVS) method.

The motivation for upwind methods is the observation that the physical flow of information is directed: A disturbance somewhere in the field will influence locations that are *downstream*, not locations that are upstream.

This is easily seen from the exact solution $u(t, x) = u_0(x - ta)$ of the scalar advection equation (6.6). If $a > 0$, the initial profile u_0 is transported to the right, and if $a < 0$, to the left. A disturbance of u_0 at x_0 will therefore only affect 'downstream' locations $x > x_0$ (if $a > 0$), or, put the other way around, a location x is affected only by information on the left ('upstream') side.

We can try to mimic this flow of information in our numerical method, thus guiding the decision of how to define the fluxes on the cell boundaries. In one dimension, the control volume Ω_i is the interval $[x_{i-1/2}, x_{i+1/2}]$, see figure 6.1. If we try to evaluate the integral

$$\int_{\Omega_i} f(\boldsymbol{u})_x\, dx = \int_{x_{i-1/2}}^{x_{i+1/2}} f(\boldsymbol{u})_x\, dx$$

we see, as already noted, that f is not defined in the points $x_{i-1/2}, x_{i+1/2}$ due to the jumps of u caused by the use of integral means, see figure 6.1. Therefore, one replaces f at the cell borders $x_{i\pm 1/2}$ with a numerical flux function $F(u_l, u_r)$ depending on both the left cell state $u_l = u_{i-1}$ and the right cell state $u_r = u_i$.

For the one-dimensional, scalar, linear case, it is then straightforward what to do: At a cell border, we set $F(u_l, u_r)$ to the left value u_l if $a > 0$, and to the right value u_r otherwise. Formally, this can be written as $F(u_l, u_r) = a^+u_l + a^-u_r =: f^+(u_l) + f^-(u_r)$, where $a^+ = \max(a, 0), a^- = \min(a, 0)$. In other words, we *split* the flux vector f into a positive and a negative part:

$$f(u) = f^+(u) + f^-(u)$$

The total numerical flux out of a cell $C = \Omega_i$ is then given by

$$\int_{\Omega_i} f(u)_x \, dx \approx F(u_i, u_{i+1}) - F(u_{i-1}, u_i)$$

We can regard this quantity as the sum of the outward directed fluxes over all facets of the one-dimensional cell $C = [x_{i-1/2}, x_{i+1/2}]$. We will see below how this formulation easily generalizes to higher dimensions. On the boundary, one can use either artificial neighbors (*ghost cells*), or define a flux directly on the facet.

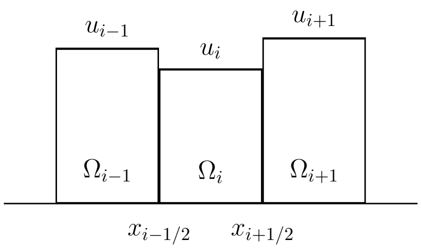


Figure 6.1: One-dimensional cell averages

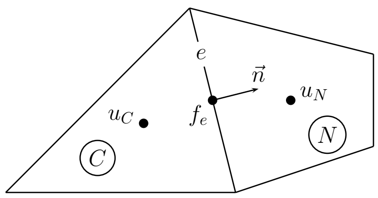


Figure 6.2: 2D flux F_e from cell C to cell N over facet e

We can extend the approach to *linear systems* if they are diagonalizable: In this case, the equation is given by

$$\boldsymbol{u}_t + A\boldsymbol{u}_x = 0 \tag{6.11}$$

with a diagonalizable $p \times p$ matrix $A = K\Lambda K^{-1}$, where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$. The diagonal matrix Λ can be decomposed in a positive part

$$\Lambda^+ = \mathrm{diag}(\lambda_1^+, \ldots, \lambda_n^+)$$

and a negative part Λ^-. Correspondingly, we define $A^+ = K\Lambda^+K^{-1}$ and $A^- = K\Lambda^-K^{-1}$, and the upwind flux

$$F(u_l, u_r) = A^+u_l + A^-u_r \tag{6.12}$$

In the case of a *nonlinear* but homogeneous system, we have $f(\boldsymbol{u}) = A(\boldsymbol{u})\boldsymbol{u}$, with $A(\boldsymbol{u})$ the Jacobian $f_u(\boldsymbol{u})$, and we can use the splitting $f^\pm(u) = A^\pm(u)u$. For an evaluation of a numerical flux, one has to decide where to evaluate the derivative $A(\boldsymbol{u})$. A possible choice, due to Steger and Warming (see [Hir90]), is

$$F(u_l, u_r) = A^+(u_l)u_l + A^-(u_r)u_r \tag{6.13}$$

The discussion so far applied to the case of *one* spatial dimension. For higher dimension, we assume that the cells are given by polytopes, such that the integral over the boundary of the cell is given by the sum of the integrals over its (planar) facets e:

$$\int_{\partial\Omega_i} f(\boldsymbol{u})n_x + g(\boldsymbol{u})n_y \, ds \tag{6.14}$$

$$= \sum_{e \prec \Omega_i} \int_e f(\boldsymbol{u})n_x + g(\boldsymbol{u})n_y \, ds \tag{6.15}$$

$$\approx \sum_{e \prec \Omega_i} |e| \big(f_{\boldsymbol{u}}(\boldsymbol{u})n_x + g_{\boldsymbol{u}}(\boldsymbol{u})n_y \big) \boldsymbol{u} \tag{6.16}$$

Here $\vec{n} = \vec{n}(e) = (n_x, n_y)$ denotes the (constant) outward normal of facet e, as above. Taking $A_e(\boldsymbol{u}) = f_{\boldsymbol{u}}(\boldsymbol{u})n_x + g_{\boldsymbol{u}}(\boldsymbol{u})n_y$, we are back in an one-dimensional situation, with the facet normal flux $f_e(\boldsymbol{u}) = A_e(\boldsymbol{u})\boldsymbol{u}$, see also figure 6.2. To sum up, the resulting numerical flux is

$$F_e(u_C, u_N) = A_e^+(u_C)u_C + A_e^-(u_N)u_N \tag{6.17}$$

where u_C is the state in $\Omega_i = C$, and u_N the state in the neighbor cell incident to e, see again fig. 6.2.

For more details on this and related methods, see e. g. [Tor97] or [Hir90]. Another method for obtaining fluxes across cell boundaries is Godunov's method: Because the state on both sides of the cell is constant, it is in principle possible to calculate the exact solution along the facets, and to derive the fluxes from that, see for example [Tor97] or [LeV92].

6.2.2.3 Second Order Method by Recovery and Limiting

A possibility to obtain higher order methods in space is to apply some averaging (*recovery*) methods to the cell states U before calculating the fluxes. To avoid oscillations near discontinuities, the process has to be suppressed in presence of high gradients. The decision when this has to occur is taken by a so-called *limiter*.

One typically starts by calculating a volume-averaged mean on vertices, collecting the states of all incident cells. This procedure is described more precisely by algorithm 6.2 (distributed case).

The next step is to determine an approximating function from these values, typically a polynomial. In the simplest case, this is just a linear or bilinear map for each cell. Depending on the value of the limiter functions, we then take either the value of the recovery functions or the original values for u_C, u_N above. Further details on this topic can be found in [Son93] or [Wie94].

6.2.2.4 Time Discretization

So far, we have the semi-discretized system

$$\frac{\partial}{\partial t}\boldsymbol{u}_i = -\frac{1}{|\Omega_i|}\mathcal{F}^i(\boldsymbol{u}) \qquad 1 \le i \le N$$

where $\mathcal{F}^i(\boldsymbol{u})$ is the numerical flux sum of cell Ω_i. This is a system of N ordinary differential equations, and hence we can in principle use any of the many methods for solving such systems. The simplest choice is the explicit Euler method, which leads to

$$\boldsymbol{u}_i^{n+1} = \boldsymbol{u}_i^n - \frac{\Delta t}{|\Omega_i|}\mathcal{F}^i(\boldsymbol{u}^n) \qquad 1 \le i \le N$$

For stability reason, there are constraints on the admissible time step Δt. The $CFL-$ *condition* states that the domain of dependency of the numerical method includes the physical domain of dependency of the equation: Information in the numerical scheme must include information influencing the physical flow. For the scalar advection equation, this results in

$$|a|\Delta t \le \Delta x$$

Other reasonable choices are explicit Runge-Kutta schemes of moderate order.

Implicit schemes avoid the restriction on the time step, but are more difficult to implement, because a non-linear system of equations has to be solved in this case. Implicit methods become attractive when either one is interested in the steady state solution, or the problem is so *stiff* that explicit methods demand very small time steps.

The overall algorithm, including time discretization, is given by algorithm 6.1.

Algorithm 6.1 Global 1st order FV algorithm

```
U^0 ← initial values
for n = 0, 1, 2 ... do   (time steps)
   for all Cells C ∈ 𝒢 do
      for all Facets f ≺ C do
         flux(C) ← flux(C) + numerical flux(f; U^n)
   calculate Δt   (obey CFL condition)
   for all Cells C ∈ 𝒢 do   (example: explicit Euler time discretization)
      U^{n+1}(C) ← U^n(C) + Δt/vol(C) flux(C)
```

6.2.3 A Program Family for Solution of Hyperbolic Equations

The central component of any software for solving PDEs of the type described above is a finite volume solver. Its task is to manage a decomposition of the geometric domain into finite volumes by means of a grid, control the cooperation of the single algorithmic pieces, and administrate the data structures shared by several such components.

We divide the components into two main groups: First, those describing the mathematical problem, and second, those implementing the numerical solution of it. Furthermore, the full application also needs other types of components, briefly discussed in 6.2.3.4.

The first group contains entities representing equations, boundary and initial conditions and geometric domains. The second one contains, among others, grid and grid geometry representations, spatial and temporal discretization algorithms, and simulation states.

In order to leverage the power of generic programming, most of these subcomponents are made parameters of the solver component. The actual program then often takes the form of a 'type equation' in the sense used by Batory [Bat95], where all these 'loose ends' are tied together. the result is a whole *family of programs* (a term coined by Parnas [Par76]). Some parameters of variation are discussed below; some are still implicit in the actual code.

6.2.3.1 Components Representing the Mathematical Problem

Generally speaking, these components contain information corresponding to an analytical, mathematical description of the problem. Very commonly, such information is not made explicit, but implicitly present in the code, thus violating the principle of 'locality of responsibility'.

For example, a class representing a hyperbolic equation contains the dimension of the vector of state (four in the case of the two-dimensional Euler equations), methods to convert between the different physical quantities, and derivatives of the flux functions. In case that a FVS algorithm is to be applied, the equation component can be specialized to additionally provide a corresponding splitting (6.12). A closer look also reveals that the entities termed *Euler equations* in reality is a *family* of equations, depending on material (gas) parameters and thermodynami equations of state, see e. g. [Tor97].

An equation component can be used not only for numerical discretizations, but also for visualization purposes, where one has to extract different scalar or vector-valued physical quantities from the state vector u, see below (page 160). It can therefore shield the visualization system from committing to a particular equation.

Concrete components have been implemented for linear advection, Euler equations (for an *ideal* gas), and a two-component gas flux, which is an enlarged system of Euler equations, see [SBB97].

Components for representing geometric domains are not simple ones — the whole branch of Geometric Modeling is concerned with this topic.

However, not much information about the domain is used *directly* in a numerical solver. Most information is used in an indirect way, namely via a generated grid representing the domain, and boundary conditions attached to the sides of the domain. These must be passed to the discrete grid entities approximating the analytical domain.

The ‘grid generation problem’ is the passage from a ‘analytical’ domain to a corresponding grid representation. This is a non-trivial process for which no satisfying standard procedure exists. Therefore, we treat it here as an entity: An abstraction for a ‘gridded domain’ encapsulatesthe way in which a grid is generated from the domain, and at the same time maintains a close relation between domain and grid. This relation is necessary, because in general changes to a grid (e. g. refinement) have to be consistent with the domain (boundary vertices must be on domain boundary, etc.).

An advantage of this encapsulation is the possibility to use special generators for specialized domains, for example logical squares. A slight disadvantage is that it breaks the separation between problem level components (geometric domain) and solution level components (grid).

6.2.3.2 Components for the Numerical Solution

We now come to the heart of a PDE solver application — components for grid and grid geometry, temporal and spatial discretizations, and possibly (linear or nonlinear) equation solvers.

Grids and geometries already have been discussed at length earlier. For the discretization component, it may be of importance whether the cell types are general or restricted, for example to triangles. Concerning grid geometries, an important optimization consists in calculating the often-used quantities, such as cell volumes, once and for all. If the numerical components consistently use grid geometries to access these quantities, these decisions will remain completely localized and transparent.

The most important solution components for FV solvers are the spatial discretizations. In general, these discretizations depend on equations (mathematical and computational properties), and on grids (mostly computational properties). In the case of the FVS method discussed above, the equations must possess the homogeneity property. For such equations, we can implement generic FVS components, with the following requirements on equations:

- Definition of the types of state ($\boldsymbol{u}$), geometric coordinates (x), and derivatives.
- Evaluation of $A_n^{\pm}(\boldsymbol{u})$ for given geometric direction (facet normal) n and state $\boldsymbol{u}$
- Flux transformation for various boundary conditions

6.2.3.3 Parallelization

A version of the solver running in parallel is obtained by applying the concepts and components presented in chapter 5. The main labour consists in identifying the global loops of the algorithms, to determine the stencils of the corresponding local functions, and to decide which grid functions have to be distributed. In the case of distributed grid functions, the synchronization operation have to be called in the right places.

In the case of finite volume algorithms with explicit time integration, all of this is rather simple, and affects just a few lines of code, namely the calculation of vertex

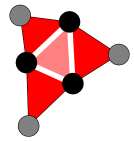


Figure 6.3: Stencil CFC from flux calculation

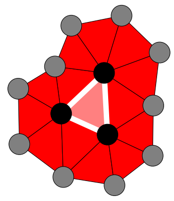


Figure 6.4: Stencil CVC from vertex averaging

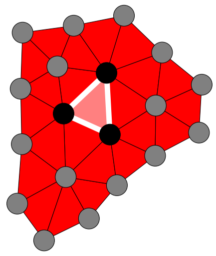


Figure 6.5: Chained stencil $CFCVC$

averages (see 6.2), the calculation of recovery functions per cell, the time integration involving the flux calculation 6.1, and the calculation of the time step size, which involves a global maximum. The only differences to the sequential algorithms are the loops over the local ranges instead of the entire grid, and the synchronization operation afterwards.

The stencil of the vertex-average operation is given by figure 6.4, the stencil for the flux calculation by figure 6.3. By (5.17) — or simply inspection — we see that the stencil resulting from vertex averages dominates the one from flux calculation. Therefore, we can safely use the former for the global overlap determination, cf. theorem 5 on page 127.

This applies to the case that we make grid functions for vertex averages, recovery function coefficients and cell states all three distributed. If we want only the minimum number of distributed grid functions, we make just the cell states distributed. In this case, we must append the stencils for the other grid functions: For the fluxes of a cell, we must access the recovery functions of each neighbor, for this, the vertex averages of each vertex of that neighbor, and therefore, each cell incident to each such vertex. This leads to the stencil of figure 6.5. A synchronization is then only necessary after the time integration, at the expense of increased local work.

It is rather obvious that the sequential case is just a special case of the distributed one. Therefore, one is lead to use the same generic solver class for both cases, controlling via a type parameter which version is to be generated. The advantages are the same as with any parameterization: Only one version has to be maintained and debugged, the decision whether the code is distributed or sequential concentrates on a single line.

6.2.3.4 User Interaction

As mentioned before, there are additional aspects that come into play when one wants to obtain numerical simulations that are practically *usable*. Here we mention in particular *data visualization* (also known as postprocessing), and runtime *parameter control*.

These aspects may be peripheral from a purely mathematical-algorithmic point of

Algorithm 6.2 Distributed vertex averaging

IN: a cell-based state $U_c : \mathcal{G}^d \mapsto \mathbb{R}^p$
OUT: a vertex-based state $U_v : \mathcal{G}^0 \mapsto \mathbb{R}^p$

```
for all Vertices v of the local range R ⊂ G do
    U_v ← 0
    Σ_vol ← 0   (sum of volumes of incident cells)
    for all Cells c incident to v do
        add volume(c) · U_c(c) to U_v(v)
        add volume(c) to Σ_vol
    U_v(v) ← U_v(v)/Σ_vol
synchronize U_v
```

view, but they are decisive for the programs appearance to the 'outside world' and its usability. Because they are indeed somewhat orthogonal from the algorithmic kernels, we would like to separate them cleanly from the latter. Done properly, this leaves open the possibility to use various visualization systems interchangeably or even in parallel, or to use different techniques of parameter control in different contexts (e. g. batch or interactive runs). The advantages of such a flexibility for practical work can hardly be overestimated.

Achieving this decoupling, however, is not so trivial. Most parameters we want to control in practice belong to the algorithmic domain, and the data to be visualized is exactly the data produced by the algorithms, so these items have to be *exposed* somehow.

We do not treat these topics in detail here. Parameter control is done using object-oriented techniques, and is described in a separate technical paper [Ber98]. The techniques presented there have proven useful also for a framework based on binary components realized in the SUPEA project (*Simulation of Processes in Energy- and Propulsion Systems,* [BBK99, BK98]).

We will now make some brief remarks on visualization. A successful decoupling of postprocessing from numerical simulation crucially depends on a clean exposition of the simulations data to the visualization component. The latter one must have access to the simulation data (state vector) as well as to the underlying grid. Luckily, we already are in possession of a standard terminology regarding these things — the grid micro-kernel developed in this work.

We may therefore simply say that the simulation is a components offering a (sequence of) geometric grids and a (sequence of) functions on this grids. The visualization component takes this sequence of grid functions and produces graphical output from them. Normally, the data produced by the numerical algorithm is not what one wants to see. For example, in the case of the two-dimensional Euler equations in conservative variables, one has a state vector of four variables

$$\boldsymbol{u} = (\rho, \rho u, \rho v, \rho E)$$

from which one has to calculate the interesting quantities like pressure from equation (6.2). The set of such functions is encapsulated in a natural way in an equation object, as mentioned before. It shields a concrete visualization component from committing to a particular type of equation. Such a decoupling has been used to develop a generic wrapper for the `Visual3` library [Hai91], allowing to use this very capable postprocessing tool with minimum effort. The library requires the user to provide a set of functions, all of which can be implemented generically on grids and equations.

6.3 A Prototype Finite Element Solver for Elliptic Problems

Physical problems governed by *elliptic* equations exhibit a behavior that is totally different from those dominated by hyperbolic equations. This difference is passed on to the numerical methods.

6.3.1 The FEM approach

The prototype elliptic problem is the *Poisson equation*:

$$\begin{aligned} -\Delta u &= f \quad \text{on} \quad \Omega && (6.18)\\ u &= 0 \quad \text{on} \quad \Gamma = \partial\Omega && (6.19)\end{aligned}$$

This form of the equation is called the *classical* one, and a solution u a *classical solution*.

By multiplying (6.18) with a function v that vanishes on the boundary $\partial\Omega$, and applying integration by parts, one arrives at the *weak* or *variational* formulation: Find $u \in H_0^1(\Omega)$ such that

$$a(u,v) := \int_\Omega \nabla u \nabla v \,\mathrm{d}\Omega = \int_\Omega f v \,\mathrm{d}\Omega \qquad \forall\, v \in H_0^1(\Omega) \tag{6.20}$$

Here $H_0^1(\Omega)$ is the space of functions vanishing on $\partial\Omega$ and whose *generalized* first partial derivatives are square integrable over Ω.

Formulation (6.20) directly leads to the *Finite Element Method*. The space $H_0^1(\Omega)$ is replaced by a finite dimensional space $V \subset H_0^1(\Omega)$, and u as well as v are varied only within this subspace (Galerkin approach).

Typically, the space V is constructed by using a grid decomposing the domain Ω into cells $c_i, 1 \leq i \leq N$, and taking piecewise polynomials ϕ_j defined over the cells.

The simplest of such spaces consists of piecewise linear functions over cells, which must hence be simplexes. For each vertex v_j, the *hat basis function* ϕ_j takes the value 1 on v_j, zero on all other vertices, and is linear on each cell. An essential feature of this basis is the small support of the ϕ_j, which is equal to the star $\mathrm{st}(v_j)$ (see p. 48).

In general, given a basis $(\phi_j)_{j=1}^M$ of V, one can write $u = \sum u_j\phi_j$, $v = \phi_k$ and evaluate

$$\sum_{j=1}^{M} \int_\Omega u_j \nabla\phi_j \nabla\phi_k \,\mathrm{d}\Omega = \int_\Omega f\phi_k \,\mathrm{d}\Omega \quad 1 \le k \le M \tag{6.21}$$

Defining the *stiffness matrix* A by setting

$$a_{kj} = \int_\Omega \nabla\phi_j \nabla\phi_k \,\mathrm{d}\Omega \tag{6.22}$$

leads to the linear system

$$Au = f \tag{6.23}$$

for the u_i, where $f_k = \int_\Omega f\phi_k \,\mathrm{d}\Omega$. Due to the fact the ϕ_j have a small support, the matrix A is *sparse*: Most integrals in (6.22) vanish.

An algorithm solving (6.20) using the finite element approach consists therefore of two essential parts:

1. Assemble the stiffness matrix A by evaluating the integrals in (6.22)

2. Solve the system (6.23)

Solving such large systems of linear equations (which may have up to one million unknowns in 3D) is the computationally most demanding task of the algorithm. Efficient methods are therefore required. Multigrid methods are particularly suited here, because they have optimal complexity for solving sparse linear systems.

A sketch of the matrix assembly procedure for *linear* finite elements can be found on page 68. For more complex elements, involving polynomial of higher degrees, this process is considerably more involved, and it will be necessary to explicitly use the local coordinate information associated with the archetype of a cell (see page 56).

This exposition has been forcedly terse; for more details on theory and numerical methods, see e. g. [GR94].

6.3.2 A Multigrid Algorithm

The invention of multigrid algorithms during the 1960s and 1970s ([Fed64, Bra73], see also the monograph of Hackbusch [Hac85]) is one of the most important contributions to the solution of systems of linear equations since the times of Gauss. It allows to solve sparse linear systems of the type (6.23) with n unknowns in $O(n)$ time, which is clearly optimal.

The fundamental idea is that an unknown function is approximated by superposition of functions on different length scales. Classical iterative methods like the Jacobi or Gauss-Seidel iteration are only effective for reducing the error on a fixed length scale.

The multigrid approach combines iteration on different scales, and thus achieves a fast global reduction of the error.

Suppose that in the formulation of the FEM approach above, we are given a *hierarchy* of grids $\mathcal{G}_0, \ldots, \mathcal{G}_l$, instead of a *single* grid $\mathcal{G}$. If these $\mathcal{G}_k$ are properly nested, they give rise to a hierarchy of function spaces $V_0 \subset V_1 \cdots \subset V_l$. On each grid, a linear system $A_k u_k = f_k$ arises, where the system on the finest grid is the one we want to solve ultimately.

Starting with some approximation u_l on the finest grid, some *smoothing* steps are applied to u_l (*pre-smoothing*).

Then, the *defect* $d_l = A_l u_l - f_l$ is *restricted* to the next coarser level, yielding d_{l-1}. This simply means that the function $d_l \in V_l$ is approximated by a function $d_{l-1} \in V_{l-1}$.

There, the *defect equation* $A_{l-1} v_{l-1} = d_{l-1}$ is solved for v_{l-1} by recursive application of the multigrid method, or by some other means if the coarsest level is reached.

The coarse grid correction v_{l-1} is *prolongated* back to the fine grid, yielding v_l, and u_l is updated accordingly: $u_l = u_l - v_l$. In our case, as $V_l \supset V_{l-1}$, the function v_{l-1} is already in V_l, and just has to be expressed in the finer basis.

Finally, we apply some steps of *post-smoothing* to u_l.

The algorithm is outlined in algorithm table 6.3. It contains various parameters needing some explanation: The mappings S_1 and S_2 are *smoothers*, that is, iterative schemes for the solution of the system $A_k x = f_k$. For example, one may choose the Jacobi or Gauss-Seidel iteration here. These smoothers are applied ν_1 and ν_2 times, respectively. Typical values are $\nu_i = 1, 2$. Depending on the nature of the problem, the choice of an adequate smoother may have great impact on the convergence speed.

The number γ controls the number of recursive applications of the multigrid method. Typical values are $\gamma = 1$ (V-cycle) and $\gamma = 2$ (W-cycle).

Finally, the mappings R_l (restriction) and P_l (prolongation) are grid transfer operations between grid levels l and $l-1$. The restriction R_l maps a function on a finer grid to one on a coarser grid, and P_l goes into the inverse direction.

Often, they are related by $R_l = P_l^T$. In the so-called *Galerkin approach*, these mappings are used to define linear systems on coarser levels, by setting $A_{l-1} = R_l A_l P_l$.

The other standard way, described above, of obtaining linear systems on coarser levels is to apply the discretization on the coarser levels. See [Hac85], p. 68 for a discussion of the relative merits of these approaches.

6.3.3 Components for Multigrid Methods

A reusable implementation of the multigrid method has to distinguish several layers. First, the general method as described in algorithm 6.3 lends itself to a generic procedure, where most of the pieces mentioned above are parameterized: The types of u, A as well as the smoothers, prolongation and restriction operators can be parameters.

Second, one has to provide implementations for the needed data types and inter-grid operators (restriction/prolongation), and the corresponding data structures relating to

Algorithm 6.3 Multiplicative multi-grid algorithm

IN: a hierarchy of linear systems $A_0, \dots, A_l$
IN: an initial guess u_l to $A_l x = f_l$
OUT: an improved approximation u_l

$u_l \leftarrow S_1^{\nu_1}(u_l, f_l)$ *(pre-smoothing)*
$d_l \leftarrow A_l u_l - f_l$
$d_{l-1} \leftarrow R_l d_l$ *(restriction)*
for γ times **do** *(solve defect equation on coarser level)*
  if $l - 1 > 0$ **then**
    Apply the multigrid algorithm to solve $A_{l-1} v_{l-1} = d_{l-1}$
  else
    solve $A_{l-1} v_{l-1} = d_{l-1}$ directly
  end if
$v_l \leftarrow P_l v_{l-1}$ *(prolongation)*
$u_l \leftarrow u_l - v_l$ *(coarse grid correction)*
$u_l \leftarrow S_2^{\nu_2}(u_l, f_l)$ *(post-smoothing)*

grid hierarchies.

The first part is relatively straightforward, one can more or less directly translate algorithm 6.3 into a programming language. It is however convenient to implement such a component as a class, in order to let the runtime-parameters be set by a transparent mechanism like that mentioned above (page 160).

The second task requires to think about which type of information a grid hierarchy should contain. As usual, this depends on the algorithms to be performed on the grid hierarchy. If prolongation and restriction operations on vertex-based elements are to be performed, we certainly need to know which vertices correspond to each other in two grids in adjacent levels.

The most simple possibility for prolongation is just using the inclusion of the function spaces $V_{l-1} \subset V_l$, thus only a new representation in the finer basis has to be calculated. In the case of linear finite elements with *red-green* refinement, where new vertices of level l only occur on edges of level $l-1$, this simply means that the coefficient on a vertex $v \in \mathcal{G}_l$ is the average of the coefficients of its *parents*. Here the parent of a vertex corresponding to an edge on a lower level simply are the two vertices incident to that edge. For a vertex already present at the lower level, we formally set the two parents equal to the lower level vertex. The corresponding procedure is given by algorithm 6.4.

For the restriction, one can take the adjoint of P_l, leading to algorithm 6.5. Another choice is *injection*, where one simply sets the values of coarse grid vertices to those of there corresponding fine grid vertices. Values on new vertices (corresponding to coarse edges) are not considered.

In general, one will need to know local coordinates of vertices in the finer grid with respect to cells in the next coarser grid they are contained in. All this information can be represented with standard grid functions. Therefore, whatever the concrete information

Algorithm 6.4 Prolongation by inclusion

IN: inter-grid mappings $\mathsf{parent}_1, \mathsf{parent}_2 : \mathcal{G}_l^0 \mapsto \mathcal{G}_{l-1}^0$ *(vertex parents)*
IN: state u_{l-1} on vertices of $\mathcal{G}_{l-1}$
OUT: prolongated state u_l on vertices of $\mathcal{G}_l$
  for all vertices $v \in \mathcal{G}_l$ **do**
    $u_l(v) = 0.5(u_{l-1}(\mathsf{parent}_1(v)) + u_{l-1}(\mathsf{parent}_2(v))$

Algorithm 6.5 Restriction by transpose of prolongation

IN: inter-grid mappings $\mathsf{parent}_1, \mathsf{parent}_2 : \mathcal{G}_l^0 \mapsto \mathcal{G}_{l-1}^0$ *(vertex parents)*
IN: state u_l on vertices of $\mathcal{G}_l$
OUT: restricted state u_{l-1} on vertices of $\mathcal{G}_{l-1}$
  for all vertices $v \in \mathcal{G}_{l-1}$ **do**
    $u_{l-1}(v) \leftarrow 0$
  for all vertices $v \in \mathcal{G}_l$ **do**
    $u_{l-1}(\mathsf{parent}_1(v)) += 0.5(u_l(v))$
    $u_{l-1}(\mathsf{parent}_2(v)) += 0.5(u_l(v))$

needed is, the representation can be kept *independent* of the underlying grid type.

One needs, in addition, *algorithmic* components for the generation of grid hierarchies. This task can be divided into three subtasks, namely

- selection of cells to be refined (or coarsened), on each level
- refinement of marked cells by mapping each cell to a set of refined cells, that is, a small grid template representing the refinement
- updating the grid hierarchy with respect to the refinement relations defined by the refiner

This decomposition into three subtasks helps in decoupling the numerical aspects (discretization and error estimation), the refinement strategy (choices include bisection and template-based red-green refinement), and the management of the grid data structure, encapsulating the choice which data is to be stored.

As to the refinement component, a (preliminary) implementation for regular red-green refinement (no coarsening) has been developed. The refined grid is an example of an implicitly defined grid: Iterators are based on the corresponding coarse grid, using predefined templates to define the fine grid iterators.

By using the coarse-grained mutating primitives discussed in section 4.1.6, a given grid can be altered by replacing cells with their refined version. Alternatively, a new level may be created.

The components developed for the multigrid method have been successfully used to build a prototype 'proof-of-concept' solver for the Poisson equation using linear finite elements. A generic hierarchical grid data structure on top of the micro-kernel has been developed, containing the necessary information for implementing the prolongation and restriction operators according to algorithms 6.4 and 6.5.

This application has not yet been parallelized. Parallelization of multigrid (or generally, hierarchical) methods is a complex task, because in addition to the horizontal neighborhood relations, discussed in chapter 5, also 'vertical' relations between grid hierarchies have to be considered, see for example [Bas94]. Different methods for selecting the overlap on coarser grids have been proposed, e. g. [Mit97].

6.4 Parallelization of an Existing Navier-Stokes Solver

6.4.1 The Need for Dealing with Existing Code

If one develops and advocates novel concepts for programming, one often is in a permanent conflict. One the one hand, it strikes the eye how existing software could be (re-) developed in a cleaner way using the new approach, yielding better and more powerful results.

Seen from a pragmatic point of view, however, one generally cannot expect all pieces of a complex application to be developed under this single approach. First, one has to exploit existing, well-proven software which one cannot afford to simply throw away. Second, even if new software is developed, there are people involved with diverging skills and background. It cannot be expected that they all quickly enough adopt the essentials of the new approach, which sometimes requires a profound rethinking of old habits.

Thus, it is important for the practical usefulness of new ways of software development to coexist with other, typically more traditional methodologies. One topic where this is particularly true is the development of parallel programs. Very often, we face the situation that a given code has been developed for sequential architectures, and only afterwards the need arises to parallelize it. This is an *acid test* for a practically useful parallelization methodology: Is it possible to separate the essential coding from the parallelization?

In the concrete example we are going to describe, a parallel solver for the incompressible Navier-Stokes equations had to be developed during the SUPEA research project (see above). The colleague in charge of this task was not an expert for parallel processing, and was supposed not to be involved into the actual parallelization effort. For this and several other reason, it was decided to first develop a sequential version of the program, and doing the parallelization in a separate step.

Furthermore, the concepts described in chapter 4 were not yet fully worked out in detail. Therefore, development of the sequential code was done completely independent of the ideas described in this work.

6.4.2 Mathematical Problem and Sequential Algorithm

The equation to be solved were the Navier-Stokes equations for incompressible, viscous flow

$$\begin{aligned} u_x + v_y &= 0 &(6.24)\\ \rho u_t + \rho u u_x + \rho v u_y &= -p_x + \mu \Delta u &(6.25)\\ \rho v_t + \rho u v_x + \rho v v_y &= -p_y + \mu \Delta v &(6.26) \end{aligned}$$

Here p is the pressure, (u, v) is the velocity vector, Δ is the Laplace operator, ρ is the constant density, and μ is the constant molecular viscosity coefficient.

The numerical method used is the SIMPLE *pressure correction method* [FP99]. Here, one alternately calculates velocity field and pressure, until both (6.24) and (6.25/6.26) are satisfied.

The concrete algorithm stores the velocity field on grid vertices and pressure values on cells. Derivatives are approximated by contour integrals around vertices and cells. The resulting systems of linear equations for velocity and pressure correction are solved by a Gauss-Seidel iteration.

6.4.3 The Parallelization

When developing a parallel version of the code, a two-stage approach had to be taken: First, the grid-related data structures had to be wrapped by adapter classes to conform to the syntax of the grid micro-kernel. Based on this adapter, the generic components for distributed grids could now be used, see figure 6.6.

In the second step, the actual parallelization had to be performed. This required the identification of the global loops and correspondingly the grid functions that had to be made distributed ones, much like described above (6.2.3.3). The act of doing this required of course close cooperation with the code author.

The effort necessary for this parallelization was much lower than for a low-level parallelization, for example directly using communication primitives. The first part, writing the adapter to the micro-kernel, contains about 1–1.5 KLOC[1], which may sound much, but is in its majority 'boiler-plate' code, additionally bloated by the existence of many small classes. In fact, it took only 2-3 days to develop and verify this part. The availability of a standardized testing strategy for such adaptations was certainly helpful here.

The second part of the parallelization effort affected only a few dozens lines. After the global loops and reductions have been identified, it is straightforward to insert the necessary synchronization operations.

These global operations may sometimes be hidden, especially in an object-oriented language like C++. For example, a step of an iterative solver is such a global loop, after which a synchronization can take place. Often, the norm of the *residual* vector is then

[1] 1000 lines of code

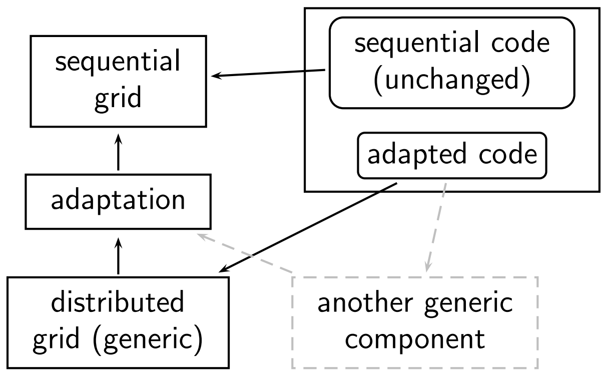


Figure 6.6: *Knows-Of*-Relationship between the various code parts in an *a-posteriori* parallelization

used to decide on termination of the iteration. But the calculation of the norm is, of course, itself a global operation (reduction). Thus, a global reduction has to take place to calculate the norm of the *global* residual, else iterations on different processes will terminate independently, leading to erroneous behavior.

Due to the relative novelty of the components and some problems with the sequential code itself, it is not so easy to give a reasonable estimate for the net work needed for this second stage, but it was certainly less than a week, which is still very little, compared to other approaches. For example, Birken speaks of a few month using his DDD library [Bir98], and also notes that ad-hoc approaches need considerably more time, for example, there have been whole projects devoted to the parallelization of a single Navier-Stokes code [LMR+97].

To be fair: Their code was much larger than ours, 100.000 vs. about 6.000 LOC (which is a crude measure anyhow). But the principal advantages of our approach are evident nevertheless: It encapsulates a large part of the effort necessary to achieve such a parallelization.

It is also true that our Navier-Stokes application does not use adaptive features and load re-balancing. Those actions can however be almost completely hidden from the parallel application, as discussed in section 5.6.5, and should thus further reduce parallelization effort with respect to the complexity of the sequential program.

Given a sufficient knowledge of the sequential algorithm, we think that — after the adaptation to the micro-kernel — a parallelization within a few days may be realistic! This, of course, depends very much on the number of algorithms involved, and on the structuring and readability of the code.

6.5 Efficiency Measurements

6.5.1 Some General Considerations

A major quality measure for software in scientific computing is efficiency, as pointed out in the first chapter. Therefore, any approach, whatever its merits otherwise are, has ultimately to prove that it does not prevent software from being efficient.

Having said this, it is perhaps useful to qualify the statement in its absoluteness. First, not every application is equally performance critical — it depends on the typical size and quantities of problems that are to be solved with the software. Often, *coding* efficiency is a more serious bottleneck than *code* efficiency. And second, even in performance-critical cases, typically only a few parts of the whole program really influence the overall efficiency.

So, efficiency requirements are best seen as a continuum, ranging from the quest for utmost performance in some heavily used kernels (notorious are the BLAS linear algebra kernels) to rather moderate requirements on more peripheral components. Still, as a rule of thumb, for any algorithmic components operating on a substantial fraction of the application's data, gross performance sins will hurt.

One now well-established approach for marrying high performance and modern (mostly object-oriented) techniques of software engineering is to identify the performance critical kernels and to implement them using conventional, low level techniques, like Fortran subroutines, see for example [DPW93].

Our approach however concentrates on high-level, generic implementations of algorithms, reusable in a broad variety of contexts. Therefore, low-level kernels cannot be the primary option, although there is no obstacle of doing so should it really be necessary. In general, one pays a price for using low-level kernels, because they restrict the generality of the application on a global scale, families of programs like those described in section 6.2 cannot be implemented in this generality.

It is therefore necessary to evaluate the generic approach under the aspect of its efficiency. The quantitative results we have obtained are, of course, to be judged with care. Generally, benchmarks are highly dependent on the environment (hardware, compiler) and on the concrete coding of the benchmark kernels, which rarely are directly related to overall performance of a real application.

Nevertheless, one can read off some general observations and tendencies from the data. A. Stepanov has coined the term *abstraction penalty*, which is the performance loss associated with formulating some algorithm on a higher level of abstraction, and having it translated to a lower level by an automatic tool, like a compiler. It will be shown that in some non-trivial cases this penalty can be eliminated.

Finally, one should be aware of the fact that a higher level of abstraction can effectively lead to more efficient software than a correspondingly lower level. This simply has to do with practical limitations of doing certain optimizations by hand. For example, in [VJ97] Veldhuizen describes how to optimize a stencil operation for data locality which would be prohibitive in terms of coding effort to repeat every time it is needed. Likewise, strategies like *compute-and-send-ahead* for overlapping computation and communication (see page 130) can be encapsulated in high-level loop templates.

6.5.2 Benchmark A: Calculating Vertex-Cell Incidences

This benchmark consists of a small loop solving the following problem: Given a grid with cell-vertex incidences (that is, having a VertexOnCell Iterator (→ p. 207) if it conforms to the micro-kernel) one wants to compute, for each vertex, the number of cells incident to it. This is solved by the following snippet of code:

```
grid_function<Vertex,int> NumCells(Grid,0);
for(CellIterator c(Grid); ! c.IsDone(); ++c)
  for(VertexOnCellIterator vc(*c);  !vc.IsDone(); ++vc) {
     NumCells[*vc]++;
}
```

The benchmark involves indirect addressing, as can be seen in the C and FORTRAN versions. This type of access is typical for unstructured grid computations, it probably limits the speed advantage of FORTRAN code with respect to its equivalents in C/C++ with is observed with simple kernels as in the BLAS.

Eight versions of this algorithm have been measured: Two versions using the `Complex2D` grid type (one with iterators, one with direct access to the data), three versions using the simpler `Triang2D` type (with iterators, unrolled loop and a `foreach` loop template), two implementations with plain C arrays (one unrolled) and a simple F77 implementation. The source code of the benchmark loops can be found in the appendix D.

Two C++ compilers were used to compile this benchmark, namely KAI `KCC v3.4` [Kuc99], and GNU `g++ egcs-1.1.2` [The00] on Linux `2.2.14`. For each, 5-6 different optimization options have been tested. The Fortran compiler used was `g77 v0.5.24` based on `egcs-1.1.2`, the options used were `-O3 -fforce-addr -funroll-loops`.

Some of the most instructive results are presented in figures 6.7 to 6.9. Several conclusion can be drawn from them:

- Without optimization, the performance is quite catastrophic, as could be expected.

- In general, the KAI compiler does a very good optimizing job, both compared to g++ and FORTRAN. In fact, the overhead of using iterators in the `Triang2D` grid is *completely* eliminated! This gives strong evidence that compiler technology has matured enough to support the generic programming style.

- Often, the data layout is more important than the high/low level distinction. Both versions of `Complex2D` clearly fall behind, due to a more fragmented memory layout (see the appendix for the data structures used).

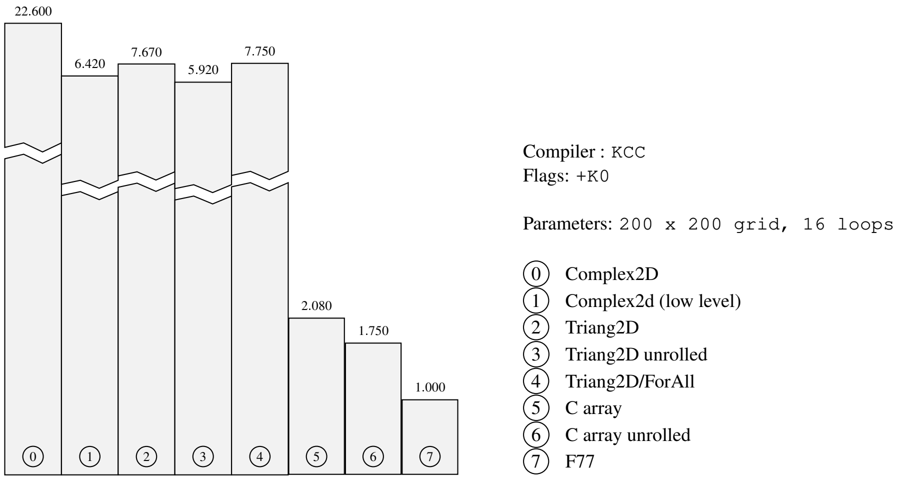


Figure 6.7: Result of the vertex-cell incidence benchmark (benchmark A) for the KAI compiler *without* optimization. Times are normalized against the `F77` version.

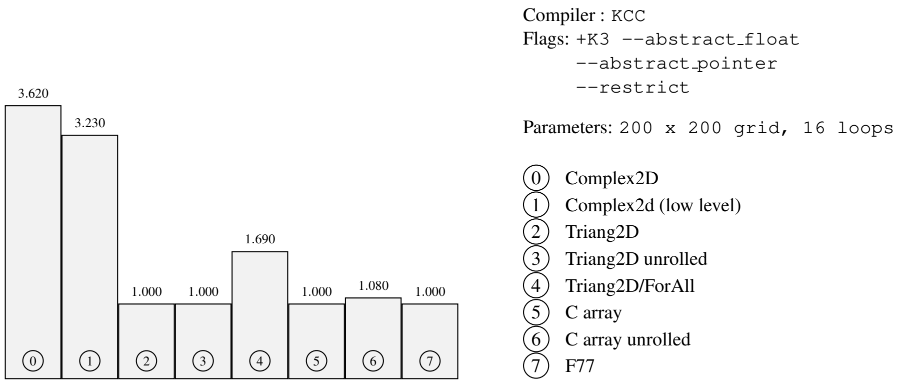


Figure 6.8: Result of benchmark A for the KAI compiler *with* optimization.

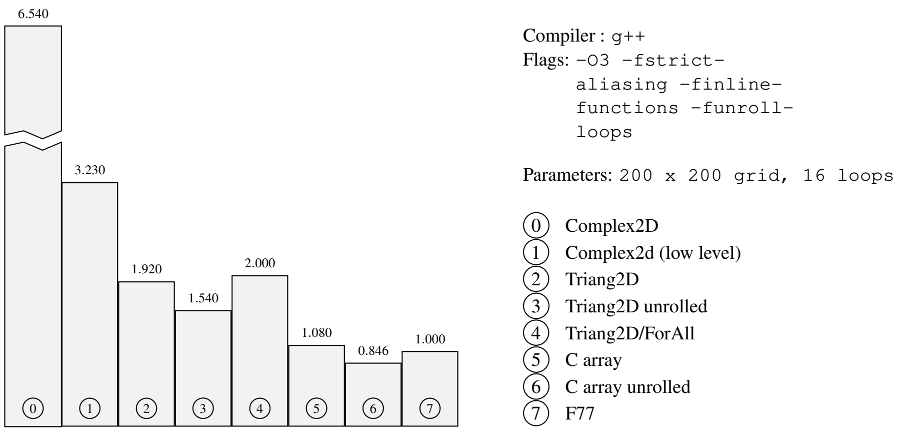
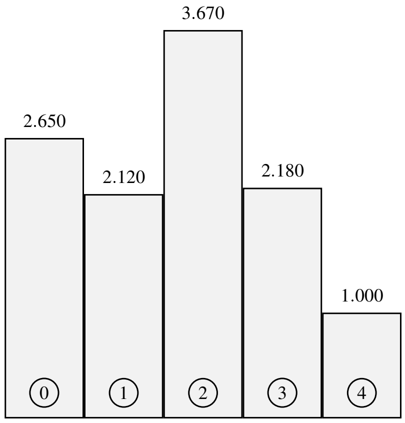


Figure 6.9: Result of benchmark A for the GNU compiler *with* optimization.



Compiler 1: `KCC`
Flags: `+K3 --abstract_float --abstract_pointer --restrict`

Compiler 2: `g++`
Flags: `-O3 -fstrict-aliasing -finline-functions -funroll-loops`

Parameters: `200 x 200 grid, 16 loops`

0 generic/Complex2D (KCC)
1 generic/Triang2D (KCC)
2 generic/Complex2D (g++)
3 generic/Triang2D (g++)
4 F77

Figure 6.10: Benchmark results for cell neighbor calculation. Times are normalized against the (optimized) `F77` version.

6.5.3 Benchmark B: Cell Neighbor Calculation

This benchmark compares a generic implementation of algorithm 3.2 (see page 222 ff.) with a Fortran implementation from Geompack[Joe91]. The latter routine works only for triangles, whereas the generic algorithms is applicable to grids with arbitrary cells (simple polygons). The code of the generic implementation is listed in the appendix.

This comparison of course has to be considered with some caution, because the two implementations differ considerably. However, both use hash tables internally, so performance should grossly be comparable.

As a result, we see a performance difference by a factor of about 2-3 in favor of the Geompack implementation, both for `KCC` and `g++`. This factor seems to reduce for increasing grid size, however, the Geompack routine has a bug and cannot be used for grids with ≥ 50.000 cells.

It is not quite clear where the reason for this difference lies; the generic implementation itself uses hash tables from the SGI STL [sgi96] which have not been compared separately with the ad hoc hash table of Geompack.

6.5.4 Benchmark C: Facet Normal Calculation

This benchmark tests the performance of geometric computations. For all facets of all cells in a grid, the outward facet normals (with the same lengths as the 2D facets themselves) are summed up. Here is the generic code:

```
coord_type normal(0,0);
for(CellIterator c(Grid); ! c.IsDone(); ++c)
  for(FacetOnCellIterator fc(*c); ! fc.IsDone(); ++fc)
    normal += Geom.outer_area_normal(fc);
// normal should be zero
```

The following versions were tested: Two versions of an implementation for `Triang2D`, one with low-level internal implementation, one using the grid interface, two versions using low-level access to `Complex2D` data, and three version using arrays for the geometry, one using the connectivity information stored in a grid of type `Complex2D`, one using the connectivity information stored in an array, and one exploiting in addition the fact that each cell is a triangle. Finally, a FORTRAN routine served as reference for the timings.

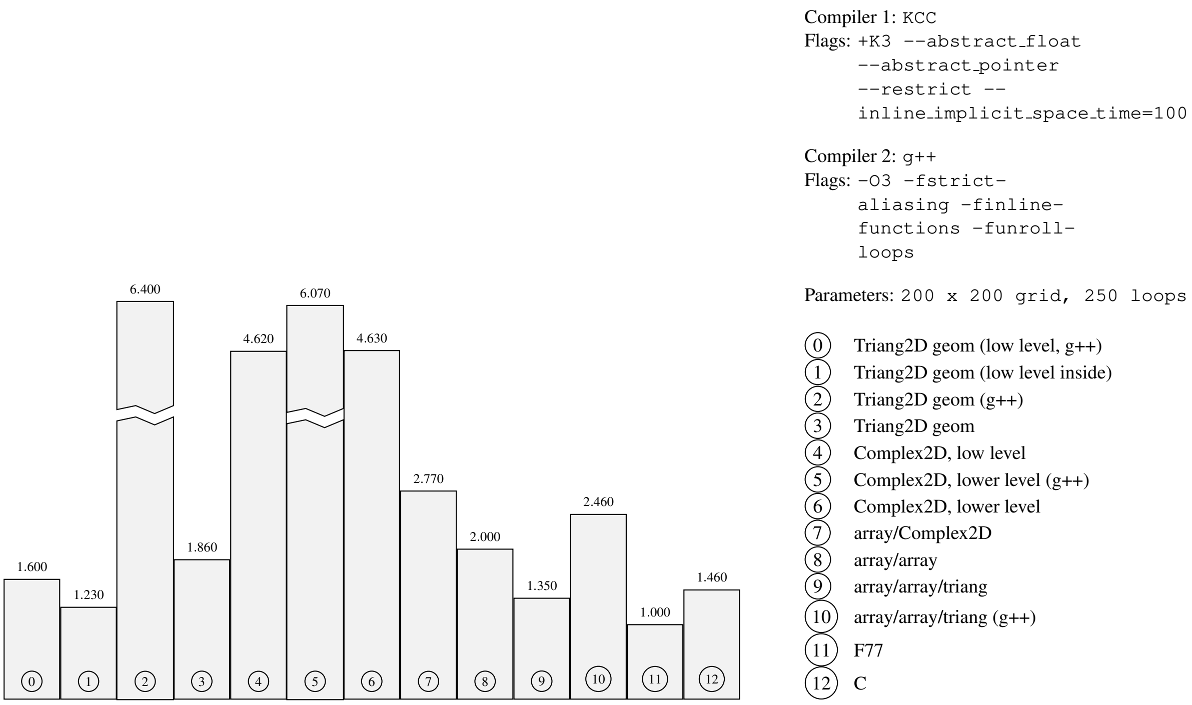


Figure 6.11: Benchmark results for outward normal calculation. Times are normalized against the `F77` version.

The performance gain when switching from a grid with variable cell types to fixed cell types (triangles in this case) is remarkable. The two compilers tested here perform very differently only on the high-level `Triang2D` code, where `KCC` beats `g++` by a factor of about 3.5. Generally, `KCC` performs by a factor of about 1.5 better than `g++`. The code generated by `KCC` for the `Triang2D` class is about twice as efficient than the code generated by `g++` for a plain array.

6.5.5 Final Remarks

We have compared only very small kernels. It is in general difficult to predict how much the differences at this level influence performance on the application-scale level — these kernels measure essentially overhead, as no costly calculations are performed.

On the other hand, comparing performance of complete applications faces the difficulty that one must have access to at least two applications — say, one written in a generic style, and one in a classical, efficient low level style — implementing exactly the same algorithms, or else the result will be of questionable significance. In practice, this would mean a re-coding of an entire application, which is generally not feasible.

One other final remark applies to the performance of C++ compilers. Had we done the same measurement about two years ago, using, say, the GNU `g++` available at that time, the results would have led to the conclusion that generic components are not competitive with respect to efficiency.

However, the scene has changed, compiler technology has matured, and still continues to do so. Because of this ongoing process, it is reasonable and fair to pick up always the best C++ result among all compiler and optimization options available, for this in general gives a better approximation to what is really possible. Thus, the results obtained with the KAI compiler indicate that the *abstraction penalty* is going to evaporate *if* the right abstractions are chosen.

Chapter 7

Summary, Discussion and Outlook

Mit dem Wissen wächst der Zweifel.
Johann Wolfgang von Goethe

7.1 Summary

In this thesis, we laid the foundations for using the new, algorithm-centered approach of generic programming in scientific computing software development. For the central field of computational grids and grid algorithms, the necessary analytic, conceptual and computational issues were worked out in detail.

In chapter 2, the shortcomings of software development techniques currently used in scientific computing were pointed out, in particular their inability to provide *reusable algorithmic components* that are *independent* on the representational issues of complex, irregular data structures like unstructured grids.

As a possible solution for decoupling algorithms from the inherent, inevitable variance of data representations, we investigated the paradigm of *generic programming*.

In chapter 3, a comprehensive *domain analysis* on grids was performed, involving mathematical aspects of grids, an analysis of grid algorithms, and a review of the computational aspects and variabilities of grid data structures. Comparing requirements of algorithms with computational capabilities of grid representations, three major categories of functionality were identified: The *combinatorial*, the *geometric*, and the *data association* category. This laid the foundations for applying the generic paradigm to the field of grid algorithms and data structures.

The next step was a more formal, detailed description of the basic concepts, leading to a grid *micro-kernel* (section 4.1). It was then shown that a number of general purpose generic components, like grid ranges, iterators or grid functions, can be based on this micro-kernel (section 4.2).

The next two chapters (5 and 6) contain examples for more advanced uses of the foundations laid in section 4.1.

Chapter 5 introduced a *domain-specific* approach to parallel computations on grids, using the geometric data partitioning paradigm. Several concepts for distributed overlapping grids were derived, among them a structural description of grid overlaps (section 5.5.2). This description both comes close to mental concepts, and is adequate to precisely describe the semantics of 'data-parallel', distributed algorithms. The notion of 'data access pattern' of algorithms on unstructured grids was formalized in a novel way by introducing *stencils* for such algorithms, building on the notation developed in section 3.1. In the sequel, we proved some basic facts about these stencils, (section 5.5.4).

Furthermore, algorithms were derived for efficient determination of overlaps, based on formal stencils, thus decoupling overlap generation from the concrete algorithms (sections 5.6.2 and 5.6.3).

These algorithms and the data structures resulting from the distributed grid concepts mentioned before were implemented using the generic paradigm, thus achieving independence of the basic, sequential grid data structures. This implementation provides an extensive application of the generic paradigm on a set of complex components, both algorithmic and data-centered, to solve a highly non-trivial problem in a very reusable way.

In chapter 6, the generic approach was assessed using different applications from the context of numerical PDE solution. In order to evaluate its soundness and breadth, we varied the type of the problems (elliptic and hyperbolic equations), the algorithmic archetype (finite element and finite volume methods), and the overall programming paradigms. In detail, we discussed two applications developed *completely* following the generic approach (sections 6.2 and 6.3), as well as the *selective use* of generic components in an *existing application* (section 6.4). Here a Navier-Stokes solver was parallelized using the components developed in chapter 5. Finally, performance of generic components was evaluated using a number of small computational kernels (section 6.5).

7.2 Discussion

For the first time (to the author's knowledge) it has been attempted successfully to create reusable components for *grid-based* algorithms that are *completely independent* of concrete grid data structures. This step allows to decouple algorithm development from representational issues, and must be considered a decisive progress.

Algorithmic developments using the methodology developed in this work are going to have a far broader potential for reuse than components based on a specific data layout, as is common practice today. Perhaps the most convincing example for the potential of the generic approach is — at present — provided by the distributed-grid family of components, demonstrated by two different parallel applications.

We will now discuss in more detail various aspects of generic components: In section 7.2.1, we study issues related to *using* generic components. Section 7.2.2 examines the generic approach from the implementation point of view. The specific case of parallel computing is discussed in section 7.2.3. Performance issues are judged in section 7.2.4.

7.2.1 Use and Reuse of Generic Components

Two basic scenarios can be distinguished when studying reuse of generic components: First, the *incremental* case, where applications written in a non-generic style reuse selected components. Here, typically some adaptation effort is necessary to map the 'native' grid data structures onto the micro-kernel. The prototype for this case is the parallelization of a Navier-Stokes solver, discussed in section 6.4.

Second, applications written in their entirety following the generic approach. Here all grid-related parts are based generically on the micro-kernel, in particular algorithms and grid data structures. A prototype has been discussed in section 6.2. This approach results in additional benefits in comparison with the scenario of incremental reuse. Some of the issues related to this case are similar to those for implementing generic components in general, and will be discussed in the next section.

We first consider some aspects related to the incremental scenario, and then turn to a general discussion of generic components reuse.

Incremental use and adaptation effort For an arbitrary grid representation, an adaptation has to be created to fulfill the requirement in the micro-kernel. A *full* adaptation (i. e. as far as the underlying representation allows) includes generic grid functions, grid geometry and the semi-generic copy/enlarge functions discussed in section 4.1.6. It consists in large parts of straightforward interface code, which can be modeled after existing examples. So, `Triang2D` (appendix B.1), can be regarded as an adaptation of an array-based representations, often found in C or FORTRAN implementations. As a rule of thumb, such an adaptation contains about 1–1.5 KLOC of code, which could probably be further reduced.

If only very few components are to be used, it suffices to implement just the required concepts; for example if the grid representation essentially has to be copied, the effort shrinks to a few dozens lines, as demonstrated in figure B.3, page 221.

Adaptation work is supported by a standardized testing procedure, checking e. g. iterators for correct operation.

A certain effort, however, is inherent in *any* reuse of grid-related components. The 'classical' way of converting between different data structures also demands a non-trivial effort, and generally is more error-prone. In case there is a *generic adapter* for a third-party component (see below) it is probably easier to first create a restricted adaptation to the micro-kernel , as just discussed.

The real power of the generic approach however comes into play when reusing a larger number of components, because the required adaptation effort is *constant*: Once done, *all* generic components can be used, whereas in the classical case, adaptations have to be done *ad hoc* for each new type of component.

The adaptation effort has to be weighed against the potential gain by reuse. For a 15-line algorithms, it certainly is not worth the trouble, for using the distributed grid components, it definitely is. Because the effort is constant, it often will pay off also for a set of *fine-grained* components, like additional iterators. In contrast, for ad-hoc adaptation in a 'traditional' context, only sufficiently *coarse-grained* components are worth reusing.

Off-the-shelf reuse of generic components For generic applications, on top of the micro-kernel, the above discussion does not apply: Generic components can be used *off-the-shelf* without any further labour.

The practical case studies presented in chapter 6 show that reuse of generic components indeed works on different scales of granularity.

The distributed-grids family of components is certainly an example of coarse-grained reuse, which internally contains many examples of fine-grained reuse, for example grid ranges. It is also case where 'traditional' approaches would have a hard stand: The necessary modifications of underlying grid representations would make a 'copy-approach' difficult to implement.

Simple visualization algorithms like iso-lines and color plots provide another proof-of-concept. Using them with different grids and types of simulation data causes no problem.

The ad-hoc adaptation effort necessary to use a conventional library can be encapsulated and therefore reused as well. A lot of such libaries exist, performing tasks badly needed by scientific applications, like grid generation, grid partitioning, or visualization. The adaptation effort required for using such a package is sometimes substantial, for example, the Visual3 visualization package [Hai99] requires a number of subroutines to be implemented, which can make up a total of a few hundred lines, and requires detailed study of documents. Such an adaptation has been implemented once and for all in a generic way, parameterized over the types of grid, geometry, and equation (see also page 160). Similar adaptations were created for the Metis partitioner [Kar99]. Thus, these packages can be used by any application without further labour.

If *whole applications* are written following the generic paradigm, highly parameterized programs result, which can be regarded as 'program families' or generators: Specific programs can be generated as needed, see for example the family of finite-volume codes for hyperbolic equations discussed in section 6.2. Among other, these applications allow to experiment with data structures, in order to find the best suited ones for the particular case.

In the future, this may lead to highly configurable systems which present a step beyond current programs. The main problem will be to coordinate the mutual relationship between components, for example discretizations requiring triangular grids or specific properties of equations, in order to achieve a sufficient breath, see [CE00] for a discussion of these configuration tasks.

Correctness and documentation of generic components A problem introduced by using generic components is that their correctness also depends on the type parameters satisfying the requirements. This is not always easy to verify, and requires some understanding from the side of the user. For complicated algorithms, it could help to provide additional components checking the validity of the parameters.

Many generic algorithms have quite a few parameters in their most general form. For helping a user not interested into this generality, several layers of interfaces are useful, substituting default values for the types, see for example the implementation of CELL NEIGHBOR SEARCH C.1.

Because a large number of components can be specified by using a limited number of well-documented concepts (see section 4.1 and 7.3), it is to be expected that users are able to quickly understand the interface of new components, once they have acquired the necessary fluency with these concepts. This is in contrast to more traditional types of components, for example subroutines, where the data layout must be specified anew for each function.

The taxonomy induced by the micro-kernel concepts also provides an easy means of classifying data structures on the basis of their functionality. Therefore, it can quickly be deduced whether a specific algorithm is supported by given data structure or if a more powerful one has to be used.

Technical issues At this point, we must mention some practical difficulties of a more technical nature which have to do with the implementation using C++'s template mechanism. Heavily templated program code often results in longer compile times, large memory consumptions for compiler runs, and sometimes problems due to long symbol names, leading to large executables and causing some assemblers to crash. Part of these problems are due to poor implementations of template-handling code in compilers, as noted by VELDHUIZEN [Vel99c].

A further point is the sometimes poor support for detecting errors due to incompatible template arguments, stemming from type parameters which do not model the concepts faithfully. Also, the actual support for all the ramifications of the template mechanism, including partial specialization and partial ordering of function signatures, has been a problem. This problem is now gradually fading, as more compilers implement the new C++ standard [Int98].

In sum, some of these problems may still be encountered when using the implementations, but there is reason to expect them to vanish in the future.

7.2.2 Implementing Generic Components and Applications

Implementing generic components certainly is harder than implementing 'ordinary' classes or routines, because there is additional source of variation. This is true even if the fundamental work, as presented in chapter 4, has already been accomplished.

Therefore, it will only be worthwhile in areas where sufficient variance of the parameters is to be expected. This clearly is the case for the grid domain.

The programmer's knowledge of the data structures passed as type parameters is very restricted, in comparison to concrete types. Because the concepts, obtained as a result of an extensive analysis, reflect fundamental properties of the domain, they are considerably more stable then more ad-hoc details of class interfaces, or even worse, of the bare data layout.

In our experience, using an abstract interface in general increases the readability and *intentionality* of implementations, because idiosyncratic details of concrete data structures (or class interfaces) are removed, compare for example the various implementations of a simple kernel shown in the appendix D. This also reduces the probability of errors. Of course, this has nothing to do with generic programming *per se*, however, as the syntax is required to be uniform across a wide range of concrete implementations, this uniformity also stretches over a wider range of code. Also, this higher level does *not* imply substantial performance losses.

Which algorithms can be implemented in a generic way, using the concepts we have developed?

It has been pointed out before that generally non-mutating algorithms are better suited then mutating ones. Here in particular, changing the internal structures of *grids* is critical; changing the values of *grid functions* causes no particular problems.

For *non-mutating* algorithms, the situation seems to be satisfactory: We did encounter only very few such algorithm that could *not* be implemented based on the concepts in the micro-kernel.

This certainly is due in part to the fact that *implementation of components* and *specification of concepts* grew up in parallel, strongly influencing each other. Overall, the concepts developed seem to faithfully reflect the mathematical structures of grids. This is also proven by the large number of very basic components like iterators and subgrids, which can be based generically on the micro-kernel, see section 4.2. The approach thus allows to develop really *scalable* grid software: New features can be implemented in a way that benefits *all* grid data structures simultaneously.

Yet, some refinements remain to be done, for example a taxonomy allow compile-time branching over the mathematical type of a grid, such as Cartesian or triangular. Furthermore, geometries still need some more work, because the number of geometric quantities required by algorithms is rather high.

The only group of non-mutating algorithms that has caused some problems are those that require more detailed knowledge on the *cell archetypes*: An example is refinement by bisection, which demands the cells to be simplexes, and to know the permutation of the vertices with respect to some model simplex. Here more knowledge has to be transferred to the algorithm than currently possible with the concepts. Similar difficulties can be expected for finite element matrix assembly, where in general local coordinates will be necessary. This is certainly a topic for further research, but cell archetypes seem to be the right abstraction to build on.

For *mutating* algorithms, an approach using coarse-grained mutating primitives has been presented in section 4.1.6. These have been used successfully for the distributed-grid components, and we also sketched how to implement grid refinement using them (page 165). In general, more experience has to be collected with mutating algorithm, which constitute a small, but important part of the algorithms relevant for self-adaptive solution of PDEs.

7.2.3 Parallel Computing with Generic Components

The concepts developed for describing the grid overlap structure (bilateral/total ranges, exposed/shared/copied ranges) have proven useful for a large class of algorithms exhibiting a grossly *data-parallel* nature.

First, for the more restricted class of data-parallel algorithms in the narrower sense, as defined in section 5.5.5, the approach guarantees the same results as in the sequential case. Such algorithms occur for example in the time-explicit solution of *hyperbolic* problem, where spatial information spreads with a *finite* speed, see section 6.2. In this case, no new algorithms result.

Furthermore, the approach can be used for algorithms that do not formally satisfy the conditions of sec. 5.5.5, like Gauss-Seidel iteration. In this case, one has to decide whether the *resulting* parallel algorithm is acceptable. The ranges offered by the distributed grid components allow to control algorithm execution to some extent, for example, one can choose to start iteration on boundary variables. Also, the fact that changing overlap width just means changing a few numbers in a parameter file, allows to easily experiment with overlap size.

In many cases, however, *new* algorithms have to be designed that perform efficiently in parallel. The major example here is the solution of systems of linear equations, occurring for example in the solution of elliptic problems. Characteristic is here the *infinite* speed of information: Changes at one location affect the solution everywhere. Good numerical algorithms in general reflect this property of the analytical problem.

Techniques to cope with this property in a distributed setting are the classical domain decomposition of SCHWARZ, resulting in an overlap as shown by fig. 5.2(b), p. 118, or modern *non-overlapping* methods, as discussed e. g. in [BRW93, Haa99]. Here *non-overlapping* means, in the terminology developed in this work, that the overlap consists of a shared range only (which does not contain any cell).

In general, these algorithms perform special action on boundary ranges, entities readily available in the overlap data structures. Typically, special *interface boundary conditions* are defined on the shared ranges. Thus, it seems very probable — although not yet backed by concrete software examples — that the *distributed overlapping grids* framework will considerably ease the implementation of such algorithms. There is always the option of further refining the concepts, for example, to allow more fine-granular interlocking communication and computation. ILU algorithms would be a possible candidate for this.

It still remains to be seen whether slightly more irregular types of computations, like particle tracing, grid refinement or grid generation, fit seamlessly into the framework, or if extensions of the concepts have to be made.

Hierarchical data structures, as they occur for example in the case of multigrid algorithms, have been tackled in the sequential case (sec. 6.3.3). Nothing prevents in principle the concepts from being applied to several grids at once; some attention has to be paid to vertical *inter-grid* dependencies.

No implementations have so far been created for the problem of dynamic grid migration and load balancing. However, a road-map for solving these tasks, containing an enumeration of the building blocks, have been given in section 5.6.5. There certainly remains work to be done, but based on what has been achieved so far, the difficulties do not seem unsurmountable.

Parallelization of hierarchical, dynamic grid computations is an involved topic, see the dissertations of BASTIAN [Bas94] or BIRKEN [Bir98] which are devoted to these problems.

Cell-based stencils, replacing the general incidence stencils defined in section 5.5.4, have proven sufficient in all practical circumstances encountered so far. This general, algorithm-independent description, together with the efficient algorithms 5.1 and 5.3, mark a substantial progress of practical impact. Earlier attempts to use a general description were constrained to preprocessing or compilation steps using ad-hoc tools ([Bir98] [GHR+93]), and thus were not well suited for *runtime* calculations.

With our results, overlap can be generated in *optimal time*. It is therefore now practical to use for example different overlaps for different grid functions / algorithms (to minimize exchanged data volume), to experiment easily with different stencils (for example for iterative methods), or to try out different combinations of stencils and synchronizations in an application, see page 158.

The *distributed overlapping grid* abstraction emphasizes the *logical* properties of distribution over *physical* properties. This allows, to some extent, to neglect the fact whether the grids reside in one and the same memory (*composite grid*) or are physically distributed (*physically distributed grid*), where details are encapsulated by a data transport layer (section 5.7.2).

In principle, this enables transparent adaptation or optimization to diverse machine architectures, like distributed or shared memory. However, some work remains to be done, because up to now, the distinction remains visible in the structure of global loops. Encapsulating the logic of global loops into generic '`foreach()`' components seems possible, and would allow further generalizations. However, some algorithms require more control over the order of execution. This would have to be taken into account in such an approach.

By virtue of the generic approach, the components can be used on top of any sequential grid data structure, possibly after an adaptation to the micro-kernel, see above. Therefore, existing applications can be parallelized with minimal changes to the code. This, of course, does not exclude the necessity that existing algorithms must be altered to perform efficiently in parallel, see the discussion above.

An advantage of practical importance of our approach is, that it does not depend on additional tools like preprocessors implementing language extensions, but only on a C++ standard-compliant compiler. In practice, extension-based approaches often suffer from interference with other development tools, like debuggers, or different extensions.

7.2.4 Performance of Generic Components

In general, generic components execute slower than their non-generic counterparts, because the extra layer of abstraction typically introduces additional layers of indirection and temporary objects which must be removed by a compiler to get equivalent code. This additional overhead has been termed *abstraction penalty*.

How much of this penalty can be removed is very much dependent on the concrete implementation, the number of additional layers, and, last not least, the compiler used, together with the options chosen. For some simple, but characteristic situations, quantitative results have been obtained (see section 6.5). The latter are to be taken with a pinch of salt, of course. These kernels represent to some extent a worst-case scenario, because not much real work is performed, and therefore the overhead is measured rather directly.

Some conclusions can be taken: In some cases, the overhead can be eliminated altogether, which definitely is good news. Also, the figures indicate once again the sensitivity on the compiler and particular optimization options. It remains open whether compiler technology will converge to a state where this issue is less important.

Often, a much more severe performance bottleneck may be bad data layout. In particular, 'classical' object-oriented approaches collecting small groups of heterogeneous data in objects sometimes seems to be a performance killer. Luckily, one can easily experiment with different data structures, if the generic approach is used consequently.

From an opposite point of view, a higher level of abstraction can *in practice* lead to more efficient code, namely if it encapsulates optimizations which are impractical on a lower level. This is the case with a `foreach()` type loop abstraction implementing overlapping communication and computation, or optimized traversals of multi-dimensional arrays for better data locality in stencil operations. Experimenting with data-locality optimized data structures for unstructured grids, for example, might be rewarding. Generic programming certainly helps in doing these experiments.

7.3 Outlook

The results obtained during this work encourage further research in the field. For exploiting the potential of the generic approach, many more generic components, in particular algorithms, are needed. In particular, substantial gains can be expected from generic implementations of parallel load balancing and partitioning, because here copying grid data structures is a serious bottleneck.

Many generic algorithms are implemented in a dimension-independent way (exception: most visualization algorithms); three-dimensional grids will be implemented to demonstrate this.

More work is needed for mutating algorithms on dynamic data structures, in order to evaluate the reach of the coarse-grained approach presented here.

On the side of data structures, the *generative* approach ([Cza98], [CEG$^+$]) could be employed to develop whole families of parameterized data structures, for example grid geometries allowing to specify which data is to be stored permanently, or grids / grid functions with various blocking strategies to enhance data locality.

Scientific computing is an interdisciplinary field with aspects from many domains. Therefore, the connection with software components from other algorithmic fields, like Algorithmic Graph Theory, Computer Graphics and Computational Geometry, has to be deepened. Some first preliminary steps towards using components from CGAL [pro99] (Computational Geometry) and GGCL [LSL99] (graph algorithms) have been done. This should be evaluated on a more systematic scale. In particular, generic libraries for graph algorithms are of interest, because a good deal of the algorithms relevant for scientific computing operate on graphs: Grid partitioning and bandwidth minimization are two examples we mentioned.

Above, we have repeatedly spoken of *program families* for the solution of partial differential equations. There are evidently many more aspects that could be left open until the actual instantiation of a program: Support for debugging, selection of the algorithms used, more general families of equations, and so on.

Done systematically, this would require a thorough *domain analysis* for the whole field of PDE solution, and possibly other fields of scientific computing, like nonlinear equations and optimization. Along the lines of generative programming [CE00], active libraries [CEG$^+$], or product-line approaches [Bat97], a new level of software design could be achieved in the field. We hope that the results of this work can contribute a small step towards this aim.

Bibliography

Up-to-date versions of the URLs will be published at `http://www.math.tu-cottbus.de/~berti/diss`.

[A+95] Ed Anderson et al. *LAPACK users' guide*. SIAM, 2nd edition, 1995.

[ABL97] Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen, editors. *Modern Software Tools in Scientific Computing*. Birkhäuser Press, 1997.

[AH35] Paul Alexandroff and Heinz Hopf. *Topologie*. Springer, 1935. Corrected reprint 1974.

[Åhl99] Krister Åhlander. *An object-oriented framework for PDE solvers*. PhD thesis, Uppsala university, Sweden, 1999.

[Ale98] Paul S. Alexandrov. *Combinatorial topology*. Dover Publications, 1998. Reprint of the three volumes published by Graylock Press, 1956, 1957, 1960. Translated from the russion edition from 1947.

[B+99] Peter Bastian et al. UG home page. `http://dom.ica3.uni-stuttgart.de/~ug/`, 1999.

[Ban98] Randolph E. Bank. *PLTMG: A Software Package for Solving Elliptic Partial Differential Equations - Users' Guide 8.0*. SIAM Publications., 1998.

[Bas94] Peter Bastian. *Parallele adaptive Mehrgitterverfahren*. PhD thesis, Universität Heidelberg, 1994.

[Bat95] Don S. Batory. Subjectivity and software system generators. Technical Report CS-TR-95-32, University of Texas, Austin, September 1, 1995.

[Bat97] Don S. Batory. Intelligent components and software generators. Technical Report CS-TR-97-06, University of Texas, Austin, April 1, 1997.

[BBK99] 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.

[BBar] Georg Bader and Guntram Berti. Design principles of reusable software components for the numerical solution of PDE problems. In Wolfgang Hackbusch and Gabriel Wittum, editors, *Concepts of Numerical Software*. Vieweg Verlag, To appear. Proceedings of the 14th GAMM Seminar, Kiel 1998.

[Ber98] Guntram Berti. External control — a pattern for mapping hierarchical structures to external control mechanisms. Technical report, Institut für Mathematik, Technische Universität Cottbus, October 1998.

[Ber99] Guntram Berti. Concepts for parallel numerical solution of PDEs. In Roland Vilsmeier, Fayssal Benkhaldoun, and Dieter Hänel, editors, *Finite Volumes for Complex Applications II*, pages 655–662. Hermès Science Publications, Paris, July 1999. Proceedings of FVCAII, July 19–22, 1999, Duisburg, Germany.

[Bey98] Jürgen Bey. *Finite-Volumen und Mehrgitter-Verfahren für elliptische Randwertprobleme*. PhD thesis, Institut für Mathematik, Universität Tübingen, 1998.

[Bey99] Jürgen Bey. AGM3D home page. `http://elc2.igpm.rwth-aachen.de/~bey/agm.html`, 1999. (last visit: 11/1999).

[BF99] Scott B. Baden and Stephen J. Fink. A programming methodology for dual-tier multicomputers. *IEEE Transactions on Software Engineering*, 1999.

[BGMS97] Satish Balay, William D. Gropp, Lois C. McInnes, and Barry F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *[ABL97]*, pages 163–202. Birkhäuser Press, 1997.

[BGMS98] Satish Balay, Bill Gropp, Lois Curfman McInnes, and Barry Smith. A microkernel design for component-based numerical software systems. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*. SIAM, October 1998.

[BGMS99] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc – The Portable, Extensible Toolkit for Scientific Computation. `http://www-fp.mcs.anl.gov/petsc/`, 1999.

[BHKF99] Thomas Brandes, Resi Höver-Klier, and Peter Faber. ADAPTOR – GMD's High Performance Fortran compilation system. `http://www.gmd.de/SCAI/lab/adaptor/`, 1999. (last visited: 01/2000).

[BHW97] James M. Boyle, Terence J. Harmer, and Victor L. Winter. The TAMPR program transformation system: Simplifying the development of numerical software. In Erlend Arge, Are Magnus Pettersen, and Hans Petter Langtangen, editors, *[ABL97]*, pages 353–372. Birkhäuser, 1997.

[Bir98] Klaus Birken. *Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen.* PhD thesis, Universität Stuttgart, October 1998.

[BK98] Thomas Breitfeld and Sven Kolibal. Tent: A CORBA based component architecture for MPI-parallel CFD simulations and their supporting tools. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, Athens, Georgia, 1998. C.S.R.E.A.

[BL97] Are Magnus Bruaset and Hans Petter Langtangen. Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software*, 23(1):50–80, March 1997.

[BL94] Randall Bramley and Tom Loos. EMILY: A visualization tool for large sparse matrices. Technical Report TR-412, Indiana University, Computer Science Department, July 94.

[BN95] John J. Barton and Lee R. Nackman. *Scientific and engineering C++.* Addison-Wesley, 1995.

[BO84] Marsha J. Berger and Joseph Ohliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[BQH99] 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.

[BR78] Ivo Babuška and Werner C. Rheinboldt. A-posteriori error estimates for the finite element method. *Int. J. Numer. Meth. Eng.*, 12:1597–1615, 1978.

[Bra] Kenneth A. Brakke. The surface evolver home page. `http://www.geom.umn.edu/software/evolver/`. (last visited: 01/2000).

[Bra73] Achi Brandt. Multi–level adaptive technique (MLAT) for fast numerical solution to boundary value problems. In H. Cabannes and R. Teman, editors, *Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics*, volume 18 of *Lecture Notes in Physics*, pages 82–89, Berlin, 1973. Springer–Verlag.

[Bra92] Kenneth A. Brakke. The surface evolver. *Experimental Mathematics*, 1(2):141–165, 1992.

[Bri89] Eric Brisson. Representing geometric structures in d dimensions: Topology and order. In *Proc. 5th Annu. ACM Sympos. Comput. Geom.*, pages 218–227, 1989.

[Bri99] Keith Briggs. The doubledouble homepage. `http://www-epidem.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`, 1999.

[BRW93] Georg Bader, Rolf Rannacher, and Gabriel Wittum, editors. *Numerische Algorithmen auf Transputer-Systemen.* Teubner, 1993.

[BSST93] Don S. Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable Software Libraries. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 191–199, December 1993.

[BZ94] Thomas Brandes and Falk Zimmermann. Adaptor — A transformation tool for HPF programs. In K. M. Decker and R. M. Rehmann, editors, *Programming environments for massively parallel distributed systems: working conference of the IFIP WG10.3, April 25–29, 1994, Ascona, Italy*, pages 91–96, Cambridge, MA, USA, 1994. Birkhäuser Boston Inc.

[C$^+$] Andrew Chien et al. Illinois Concert project. `http://www-csag.ucsd.edu/projects/concert.html`. (last visited: 01/2000).

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Techniques, and Applications.* Addison-Wesley, 2000.

[CEG$^+$] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries.

[CEKN99] Krzysztof Czarnecki, Ulrich Eisenecker, Johannes Knaupp, and Tobias Neubert. GMCL — generative matrix computation library. `http://electra.prakinf.tu-ilmenau.de/~czarn/gmcl/index.html`, 1999.

[Cop98] James O. Coplien. *Multi-paradigm design for C++.* Addison-Wesley, 1998.

[Cou98] Bernard Coulange. *Software reuse.* Springer Verlag, London, 1998.

[Cro96] Thomas W. Crockett. Beyond the renderer: Software architecture for parallel graphics and visualization. In A. Chalmers and E. Jansen, editors, *Proc. First Eurographics Workshop on Parallel Graphics and Visualisation*, pages 1–15, Bristol, U.K., September 1996. Alpha Books. Also available as ICASE Report No. 96-75 (NASA CR-201637).

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, TU Ilmenau, Germany, 1998.

[DEG98] Tamal K. Dey, Herbert Edelsbrunner, and Sumanta Guha. Computational topology. In B. Chazelle, J. E. Goodman, and R. Pollack., editors, *Advances in Discrete and Computational Geometry*, Contemporary Mathematics. AMS, 1998.

[DHH99] T. B. Dinesh, Magne Haveraaen, and Jan Heering. An algebraic programming style for numerical software and its optimization. Technical Report SEN-R9844, CWI, Amsterdam, 1999.

[DL89] David P. Dobkin and Micheal J. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4:3–32, 1989.

[DLPR99] Jack Dongarra, Andrew Lumsdaine, Roldan Pozo, and Karin Remington. IML++ (Iterative Methods Library). `http://math.nist.gov/iml++/`, 1999.

[DPW93] Jack J. Dongarra, Roldan Pozo, and David W. Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings, Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 162–171, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

[DQ99] Kei Davis and Dan Quinlan. ROSE II: An optimizing code transformer for C++ object-oriented array class libraries. In *Third World Multiconference on Systemics, Cybernetics and Informatics (SCI'99)*, 1999.

[Ede87] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer Verlag, 1987.

[Els97] Ulrich Elsner. Graph partitioning – a survey. Technical Report SFB393/97-27, Technische Universität Chemnitz, December 1997.

[FBK98] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1):61–82, April 10/May 1 1998.

[Fed64] Radii P. Fedorenko. On the speed of convergence of one iterative process. *USSR Comput. Math. and Physics*, 4:227, 1964.

[Fil96] Mark Filipiak. Mesh generation. Epcc technology watch report, Edinburgh Parallel Computing Centre, The University of Edinburgh, 1996.

[Fin97] Stephen Fink. The KeLP programming system. `http://www-cse.ucsd.edu/groups/hpcl/scg/kelp/index.html`, 1997. (last visited: 01/2000).

[FJP98] Lori Freitag, Mark Jones, and Paul Plassmann. Mesh component design and software integration within SUMAA3d. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*. SIAM, October 1998.

[FP90] Rudolf Fritsch and Renzo A. Piccini. *Cellular Structures in Topology*. Cambridge University Press, 1990.

[FP99] Joel H. Ferzinger and Milovan Perić. *Computational Methods for Fluid Dynamics*. Springer Verlag, 1999.

[FvFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice (2nd Ed.)*. Addison-Wesley, 1990.

[FWM94] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufmann Publishers, 1994.

[GC-99a] Grand challenge coupled field problems. `http://caswww.colorado.edu/CF.d/Home.html`, 1999. (last visited 11/1999).

[GC-99b] Grand challenge in petroleum reservoir modeling. `http://www.pe.utexas.edu/HPCC/`, 1999. (last visited 11/99).

[GC98] Bishwaroop Ganguly and Andrew Chien. High-level parallel programming of an adaptive mesh application using the Illinois Concert System. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505, pages 47–58, 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GHR$^+$93] Uwe Geuder, Monika Härdtner, Andreas Reuter, Bernhard Wörner, and Roland Zink. GRIDS — a parallel programming system for grid-based algorithms. *The Computer Journal*, 36(8):702–711, 1993.

[Gol93] Herman H. Goldstine. *The computer from Pascal to von Neumann*. Princeton University Press, Princeton, 1993.

[GR94] Christian Großmann and Hans-Görg Roos. *Numerik partieller Differentialgleichungen*. B. G. Teubner, 2nd edition, 1994.

[GRA99] GRAPE team. GRAPE - Graphics Programming Environment. `http://www-sfb256.iam.uni-bonn.de/grape/main.html`, 1999. (last visit: 11/1999).

[Gro98] William Gropp. Exploiting existing software in libraries: Successes, failures, and reasons why. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing*. SIAM, October 1998.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[Haa97] Gundolf Haase. New matrix-by-vector multiplications based on a nonoverlapping domain decomposition data distribution. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Parallel Processing (Proceedings of EuroPar '97)*, volume 1300 of *Lecture Notes in Computer Science*, pages 726–733. Springer Verlag, August 1997.

[Haa99] Gundolf Haase. *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen*. Teubner Verlag, Stuttgart, Leibzig, 1999.

[Hac85] Wolfgang Hackbusch. *Multigrid Methods and Applications*. Springer, Berlin, 1985.

[Hai91] Bob Haimes. Visual3 - a software environment for flow visualization. *Computer Graphics and Flow Visualization in Computational Fluid Dynamics,VKI Lecture Series 10*, 1991.

[Hai99] Bob Haimes. Visual3 homepage. `http://raphael.mit.edu/visual3/visual3.html`, 1999.

[Hen79] Michael Henle. *A combinatorial Introduction to Topology*. Dover Publications, N.Y., 1979.

[Hin83] Alan C. Hindmarsh. ODEPACK, A systematized collection of ODE solvers. In R. S. Stepleman et al., editors, *Scientific Computing*, volume 1 of *IMACS Transactions on Scientific Computation*, pages 55–64. North-Holland, 1983.

[Hir90] Charles Hirsch. *Numerical Computation of Internal and External Flows, Volume 2: Computational Methods for Inviscid and Viscous Flows*. John Wiley & Sons, Chichester, New York, Brisbane, Toronto, Singapore, (1990).

[HK98] Richard D. Hornung and Scott R. Kohn. The use of object oriented design patterns in the SAMRAI structured AMR framework. In Michael E. Henderson, Christopher R. Anderson, and Stephen L. Lyons, editors, *Object*

Oriented Methods for Interoperable Scientific and Engineering Computing. SIAM, October 1998.

[HL95] Walter Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, February24 1995.

[HMN93] Hans Hagen, Heinrich Müller, and Gregory M. Nielson, editors. *Focus on Scientific Visualization.* Springer Verlag, 1993.

[HPC99] High-performance computational methods for coupled fields and GAFD turbulence. `http://wwwmcb.cs.colorado.edu/home/gc/Home.html`, 1999. (last visited 11/99).

[HW95] Wolfgang Hackbusch and Gabriel Wittum, editors. *Numerical treatment of coupled problems*, volume 51 of *Note on Numerical Fluid Mechanics.* Vieweg Verlag, 1995.

[Int98] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++.* International Organization for Standardization, Geneva, Switzerland, 1998.

[Jac97] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods.* Cambridge University Press, 1997. For contents and other information see `http://www.radonc.washington.edu/prostaff/jon/z-book/`.

[Jän90] Klaus Jänich. *Topologie.* Springer Verlag, 3 edition, 1990.

[Joe91] Barry Joe. Geompack. A software package for the generation of meshes using geometric algorithms. *Advances in Engineering Software and Workstations*, 13(5–6):325–331, September 1991.

[Joe95] Barry Joe. Construction of three-dimensional improved-quality triangulations using local transformations. *SIAM J. Sci. Comput.*, pages 1292–1307, 1995.

[JP95] Mark T. Jones and Paul E. Plassmann. Blocksolve95 user's manual: Scalable library software for the parallel solution of sparse linear systems. ANL Report 95/48, Argonne National Laboratory, December 1995.

[K$^+$99] Steve Karmesin et al. POOMA: Parallel Object-Oriented Methods and Applications. `http://www.acl.lanl.gov/PoomaFramework/`, 1999.

[Kar95] Even-André Karlsson, editor. *Software Reuse – A Holistic Approach.* John Wiley & Sons, 1995.

[Kar99] George Karypis. METIS home page. `http://www-users.cs.umn.edu/~karypis/metis/`, 1999. (last visit: 11/1999).

[Ket97] Lutz Kettner. Designing a data structure for polyhedral surfaces. Technical Report Technical Report 278, ETH Zürich, Institute of Theoretical Computer Science, December 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[KMS81] Derek Kapur, David R. Musser, and Alexander A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Proceedings of a Workshop on Program Specification*, volume 134 of *LNCS*, pages 402–414, Aarhus, Denmark, August 1981. Springer.

[Knu93] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, Reading, MA, 1993.

[Knu96] Donald E. Knuth. *Selected papers on computer science*, volume 59 of *CSLI lecture notes*. Cambridge University Press, Cambridge, UK, 1996.

[KNW97] Dietmar Kühl, Marco Nissen, and Karsten Weihe. Efficient, adaptable implementations of graph algorithms. In *Proc. of Workshop on Algorithm Engineering WAE'97*. Venice University, Italy, September 1997.

[Kot99] Vladimir Kotlyar. *Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs*. PhD thesis, Department of Computer Science, Cornell University, February 1999. available as TR ncstrl.cornell/TR99-1732.

[KRYG82] David R. Kincaid, John R. Respess, David M. Young, and Roger G. Grimes. ITPACK 2C : A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Trans. Math. Software*, 8(3):302–322, 1982.

[Kuc99] Kuck & Associates, Inc. Introduction to KAI C++. `http://www.kai.com/C_plus_plus/`, 1999. (last visited 01/2000).

[Kue98] Frank Kuehndel. Software methods for avoiding cache conflicts. Technical Report CS-TR-98-16, University of Texas, Austin, September 1, 1998.

[KW96] Dietmar Kühl and Karsten Weihe. Iterators and handles for nodes and edges in graphs. Technical Report 15, Universität Konstanz, September 1996.

[KW97] Dietmar Kühl and Karsten Weihe. Data access templates. *C++ Report*, 9(7):15, 18–21, 1997. (also avail. as TR 9/1996, Univ. of Constance).

[LeV92] Randall J. LeVeque. *Numerical Methods for Conservation Laws*. Birkhäuser, 1992.

[LeV94] Randall J. LeVeque. CLAWPACK: A software package for solving multidimensional conservation laws. In J. Glimm, editor, *Proceedings of the Fifth International Conference on Hyberbolic Problems: Theorie, Numerics, Applications*. World Scientific, June 1994.

[LHKK79] Charles L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. Algorithm 539: Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:324–325, 1979.

[LLS99] Andrew Lumsdaine, Lie-Quan Lee, and Jeremy Siek. The Generic Graph Component Library, GGCL. `http://www.lsc.nd.edu/research/ggcl/`, 1999. (last visited 01/2000).

[LMR+97] Peter Luksch, Ursula Maier, Sabine Rathmayer, Matthias Weidmann, and Friedemann Unger. Sempa: Software engineering for parallel scientific computing. *IEEE Concurrency*, 5(3):64–72, July/September 1997.

[Loe76] Arthur L. Loeb. *Space Structures, their Harmony and Counterpoint*. Addison-Wesley, London, 1 edition, 1976.

[LS95] Meng Lee and Alexander A. Stepanov. The standard template library. Technical report, Hewlett-Packard Laboratories, February 1995.

[LS99a] Andrew Lumsdaine and Jeremy Siek. The Iterative Template Library (ITL). `http://www.lsc.nd.edu/research/itl/`, 1999. (last visited 11/1999).

[LS99b] Andrew Lumsdaine and Jeremy Siek. The Matrix Template Library (MTL). `http://www.lsc.nd.edu/research/mtl/`, 1999. (last visited 01/2000).

[LSL99] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *Proccedings of OOPSLA'99*, 1999.

[LW69] Albert T. Lundell and Stephen Weingram. *The Topology of CW Complexes*. Van Nostrand Reinhold, 1969.

[MA97] Fredrik Manne and Svein Olav Andersen. Automating the debugging of large numerical codes. In *[ABL97]*, pages 339–352. 1997.

[Män83] Martti J. Mäntylä. Computational topology: a study of topological manipulations and interrogations in computer graphics and geometric modeling. *Acta Polytech. Scand. Math. Comput. Sci. Ser.*, 37:1–46, 1983.

[McI68] Doug McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch-Partenkirchen, Germany*, pages 88–98, 1968.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[Mit97] William F. Mitchell. A parallel multigrid method using the full domain partition. In *Copper Mountain Conference on Multigrid Methods 97*, 1997. Available on MGNet.

[Mit99] William Mitchell. MGHAT netlib directory. `http://elib.zib.de/netlib/pdes/mgghat/`, 1999.

[MNU99] Kurt Mehlhorn, Stefan Näher, and Christian Uhrig. The LEDA home page. `http://www.mpi-sb.mpg.de/LEDA/`, 1999.

[Moi77] Edwin E. Moise. *Geometric Topology in Dimensions 2 and 3*. Springer-Verlag, New York, 1977.

[Mor97] Michael E. Mortenson. *Geometric Modeling*. Wiley, New York, 2nd edition, 1997.

[MPI] The Message Passing Interface (MPI) standard. `http://www-unix.mcs.anl.gov/mpi/`. (last visited 01/2000).

[MS89] David R. Musser and Alexander A. Stepanov. Generic programming. In Patrizia Gianni, editor, *Symbolic and algebraic computation: International Symposium ISSAC '88, Rome, Italy, July 4–8, 1988: proceedings*, number 358 in LNCS, pages 13–25. Springer, 1989.

[MS94] David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software Praxis and Experience*, 24(7):623–642, July 1994.

[MSD93a] Robert C. Malone, Richard D. Smith, and John K. Dukowicz. Climate, the ocean, and parallel computing. *Los Alamos Science*, 21, 1993.

[MSD93b] Robert C. Malone, Richard D. Smith, and John K. Dukowicz. New numerical methods for ocean modeling on parallel computers. *Los Alamos Science*, 21, 1993.

[Mye95] Nathan Myers. A new and useful technique: "traits". *C++ Report*, 7(5):32–35, June 1995.

[Nat95] National Coordination Office for Computing, Information, and Communications. Hpcc blue book 1996: Foundations for america's information future. `http://www.ccic.gov/pubs/blue96/index.html`, September 1995.

[Nor96] Charles D. Norton. *Object-Oriented Programming Paradigms in Scientific Computing.* PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1996.

[NW96] Marco Nissen and Karsten Weihe. Combining LEDA with customizable implementations of graph algorithms. Technical Report 17, Universität Konstanz, October 1996.

[OR97] Mario Ohlberger and Martin Rumpf. Hierarchical and adaptive visualization on nested grids. *Computing*, 59(4):365–385, 1997.

[Par76] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[PB99] Manish Parashar and James C. Browne. DAGH (distributed adaptive grid hierarchies) homepage. `http://www.caip.rutgers.edu/~parashar/DAGH/`, 1999.

[PD90] Ruben Prieto-Diaz. Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47, April 1990.

[PdKÜK83] Robert Piessens, Elise deDoncker Kapenga, Christoph Überhuber, and David Kahaner. *QUADPACK: a Subroutine Package for Automatic Integration.* Series in Computational Mathematics. Springer Verlag, 1983.

[Pet97] N. Anders Petersson. An algorithm for constructing overlapping grids. Technical report, Chalmers University of Technology, March 1997.

[PHS98] Steve Plimpton, Bruce Hendrickson, and James Stewart. A parallel rendezvous algorithm for interpolation between multiple grids. In *Proceedings of SC'98: High Performance Networking and Computing*, 1998. Electronic Proceedings available at `http://www.supercomp.org/sc98/proceedings/`.

[PIC99] PICS Consortium. Grand challenges on groundwater remediation. `http://www.isc.tamu.edu/PICS/PICS.html`, 1999.

[PNDN97] Preeti Ranjan Panda, Hiroshi Nakamura, Nikil D. Dutt, and Alexandru Nicolau. Improving cache performance through tiling and data alignment. In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolin, editors, *Solving Irregular Structured Problems in Parallel (Proceedings of 4th Intl. Symp. IRREGULAR97)*, volume 1253 of *LNCS*, pages 167–185. Springer Verlag, 1997.

[pro99] The CGAL project. The CGAL home page – Computational Geometry Algorithms Library. `http://www.cs.uu.nl/CGAL/`, 1999.

[PVM] PVM – Parallel Virtual Machine. `http://www.epm.ornl.gov/pvm/pvm_home.html`. (last visited 01/2000).

[PvW93] Frits H. Post and Theo van Wilsum. *Focus on Scientific Visualization*, chapter Fluid Flow Visualization. Springer, 1993. in [HMN93].

[PWJ97] Stephen G. Parker, David M. Weinstein, and Christopher R. Johnson. The SCIRun computational steering software system. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*. Birkhäuser Press, 1997.

[RB84] John R. Rice and Ronald F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer Verlag, New York, 1984.

[REL97] Rainer Roitzsch, Bodo Erdmann, and Jens Lang. An object-oriented finite element code: Design issues and application in hyperthermia treatment planning. In *[ABL97]*, pages 105–124. 1997.

[REL99] Rainer Roitzsch, Bodo Erdmann, and Jens Lang. Kaskade – a family of adaptive FEM codes. `ftp://elib.zib.de/pub/elib/codelib/kaskade/`, 1999.

[Riv84] María-Cecilia Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *Internat. J. Numer. Methods Eng.*, 20:745–756, 1984.

[Rob96] Arch D. Robison. C++ gets faster for scientific computing. *Computers in Physics*, 10(5):458–462, Sep/Oct 1996.

[Ros98] Jarek Rossignac. EDGEBREAKER: Connectivity compression for triangle meshes. Technical Report 98-35, GVU, 1998. Georgia Tech's Graphics, Visualization & Usability Center.

[RP97] Karin A. Remington and Roldan Pozo. The NIST Sparse BLAS. `http://math.nist.gov/spblas/`, 1997. (last visited 01/2000).

[RSS96] Martin Rumpf, Alfred Schmidt, and Kunibert G. Siebert. Function defining arbitrary meshes - a flexible interface between numerical data and visualiztion. *Computer Graphics Forum*, 15(2):129–141, 1996.

[Saa96] Yousef Saad. *Iterative methods for sparse linear systems*. PWS Publishing Company, 1996.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer Verlag, 1997.

[SBB97] Klaus Schenk, Georg Bader, and Guntram Berti. Analysis and approximation of multicomponent gas mixtures. In M. Feistauer, K. Kozel, and R. Rannacher, editors, *Proceedings of the 3rd Summer Conference Numerical Modelling in Continuum Mechanics*, Prague, 1997.

[Sch99] Joachim Schöberl. Netgen user manual. `http://www.sfb013.uni-linz.ac.at/~joachim/usenetgen/`, 1999. (last visit: 11/1999).

[SFdC+97] Mark S. Shephard, Joseph E. Flaherty, Hugues L. de Cougny, Carlo L. Bottasso, and Can Özturan. Parallel automatic mesh generation and adaptive mesh control. In M. Papadrakikis, editor, *Parallel Solution Methods in Computational Mechanics*. John Wiley & Sons, 1997.

[sgi96] SGI Standard Template Library Programmer's Guide. `http://www.sgi.com/Technology/STL`, since 1996. (last visit: 01/2000).

[She96] Jonathan R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[She99] Jonathan R. Shewchuk. Triangle: A two-dimensional quality mesh generator. `http://www.cs.cmu.edu/~quake/triangle.html`, 1999.

[SL98] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, volume 1505 of *Lecture Notes in Computer Science*, pages 59–70, 1998.

[Son93] Thomas Sonar. On the design of an upwind scheme for compressible flow on general triangulations. *Numerical Algorithms*, 4:135–149, 1993.

[SR97] Thomas Schmidt and Roland Rühle. An object oriented tool for visualization and debugging of finite volume methods. In *Winter School of Computer Graphics '97 Conference Proceedings, Plzen, Czech Republic*, 1997.

[SS95] Al Stevens and Alexander Stepanov. Al Stevens inteviews Alex Stepanov. *Dr. Dobbs Journal*, March 1995.

[Ste96] Alexander A. Stepanov. Generic programming. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, page 40. Springer Verlag, 1996.

[Str97] Bjarne Stroustrup. *The C++ Programming Language: Third Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1997.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, N.Y., 1998.

[Tan95] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[TGE97] Christian Teitzel, Roberto Grosso, and Thomas Ertl. Efficient and reliable integration methods for particle tracing in unsteady flows on discrete meshes. In W. Lefer and M. Grave, editors, *Eighth Eurographics Workshop on Visualization in Scientific Computing*, pages 49–56. The EuroGraphics Association, 1997.

[The00] The GCC team. GCC home page. `http://gcc.gnu.org/`, 2000. (last visited 01/2000).

[Tho99] Joe F. Thompson, editor. *Handbook of grid generation.* CRC Press, 1999.

[Tor97] Eleuterio F. Toro. *Riemann solvers and numerical methods for fluid dynamics.* Springer Verlag, 1997.

[TWM85] Joe F. Thompson, Zahir U.E. Warsi, and Charles W. Mastin. *Numerical grid generation.* North-Holland, 1985.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Vel95a] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[Vel95b] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[Vel99a] Todd L. Veldhuizen. Blitz++ home page. `http://oonumerics.org/blitz/index.html`, 1999.

[Vel99b] Todd L. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, 1999.

[Vel99c] Todd L. Veldhuizen. Techniques for scientific C++. `http://extreme.indiana.edu/~tveldhui/papers/techniques/`, August 1999.

[VG98] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[VJ97] Todd L. Veldhuizen and M. Ed Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[Wal99] Chris Walshaw. The JOSTLE home page. `http://www.gre.ac.uk/jostle`, 1999. (last visit: 11/1999).

[WD97] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, Department of Computer Science, University of Tennessee, December 1997.

[Wei98] Karsten Weihe. A software engineering perspective on algorithmics. Technical Report 50, Universität Konstanz, January 1998.

[Wie94] Monika Wierse. *Higher order upwind schemes on unstructured grids for the compressible Euler equations in timedependent geometries in 3D*. PhD thesis, Universität Freiburg, September 1994.

[Wie97] Christian Wieners. Conforming discretizations on tetrahedrons, pyramids, prisms and hexahedrons. Technical Report SFB 404 preprint 97/15, Universität Stuttgart,, 1997.

[WL96] Gregory V. Wilson and Paul Lu, editors. *Parallel programming using C++*. MIT Press, 1996.

[Wol98] Alex Wolfe. U.S. starts path to 30 TFlops computer. *HPCU news*, 1(3), January 1998.

[YFD97] T.-Y. Brian Yang, Geoffrey Furnish, and Paul F. Dubois. Steering object-oriented scientific computations. In R. Ege, M. Singh, and B. Meyer, editors, *Proceedings. Technology of Object-Oriented Languages and Systems, TOOLS-23*, pages 112–119, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. IEEE Computer Society Press.

[Zie94] Günter M. Ziegler. *Lectures on Polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg, 1994.

[Zom96] Albert Y. Zomaya, editor. *Parallel and distributed computing handbook*. McGraw Hill, New York, 1996.

Appendix A

Concepts

In this appendix, we provide a semi-formal definition of the concepts developed for the grid domain in chapter 4. For lack of space, it is not possible to present description of all concepts.

The full set of concept definitions, as well as source code for the components, are available online at `http://www.math.tu-cottbus.de/~berti/gral`.

All syntactical specifications are directly given in terms of the C++ programming language. The reason for this is simplicity and readability, because C++ is supposed to be known by the large majority of potential readers. On the other hand, specification languages like *Z* [Jac97] or *Tecton* [KMS81] have both support for generics, but are not known widely in the field of scientific computing. Their use would make the following sections widely unintelligible.

Thus, the format of the concept section closely follows that used in the SGI STL implementation [sgi96], which has proven its usefulness. Under the reference given, the reader will also find an introduction into the format.

A.1 Grid Entities

This section covers the basic entities associated with grids: Grid elements (like vertices, cells), element handles, grid sequence iterators and grid incidence iterators.

A.1.1 Grid Entity Concept

Description The *Grid Entity* concept is rather abstract in that it gives rise to rather separate sub-concepts, notably Grid Element → A.1.2, Grid Sequence Iterator → A.2.1, and Grid Incidence Iterator (→ p. 206). Its purpose is to bundle some fundamental properties, much like the STL concepts Assignable and Equality Comparable which it refines.

The fundamental property captured by *Grid Entity* is binding to a Grid → A.2.7 (the *anchor grid*), and global identity, in contrast to Element Handles → A.1.5, which are unique only within a single grid.

Conceptually, a grid is the owner of its entities; an entities lifetime cannot exceed that of its anchor grid. Also, an grid entity need not necessarily be permanently stored, it may be constructed only temporarily, for example a Grid Iterator or a Grid Element → A.1.2 obtained by dereferencing a Grid Iterator.

Refinement of: STL-Assignable · STL-Equality Comparable

Notation

`E` is a type which is a model of *Grid Entity*
`e, e1, e2` are objects of type `E`

Definitions An *anchor* of `e` is a logically superior entity. It is either an object of a type which is a model of Grid → A.2.7 or an object of a type which is a model of Grid Element → A.1.2, to which `e` is incident. This is defined more precisely depending on the subconcept involved. For example, for Grid Element → A.1.2 the anchor is the underlying grid; for a Grid Sequence Iterator → A.2.1 on a Grid Range → A.2.5, it is the grid range, and for a Grid Incidence Iterator → A.2.3, it is a grid element. So, the anchor of a Vertex-On-Cell Iterator → A.2.4, is the cell over whose incident vertices the iteration is performed.

A grid entity `e` is *bound* to a grid `g`, if `&g == &e.TheGrid()`. Else `e` is *unbound*.

A grid entity `e` is *valid*, if it is bound to a grid, and its anchor is valid (either refers to an existing grid, or is itself a valid Grid Entity), and its handle is a valid handle of that grid.

Associated types

Name	**Expression**	**Description**
Grid type	`E::grid_type`	type of the corresponding anchor grid → A.2.7
Anchor type	`E::anchor_type`	type of the corresponding anchor[1]
handle type	`E::handle_type`	type of the corresponding element handle → A.1.5

Valid Expressions

Name	**Expression**	**Type requirements**	**return type**
Anchor grid	`e.TheGrid()`		`grid_type const&`
Anchor entity	`e.TheAnchor()`		`anchor_type const&`
Handle	`e.handle()`		`handle_type`

Expression semantics

Name	**Expression**	**Precondition**	**Semantics**	**Postconditon**
Copy constructor	`E e1(e2)`			`e1 == e2`
Assignment	`e1 = e2`		`e1` is a copy of `e2`	`e1 == e2`
Anchor grid	`e.TheGrid()`	`e` is valid → A.1	get the grid to which `e` is bound	
Equality comp.	`e1 == e2`	`e1.TheAnchor() == e2.TheAnchor()` [1]	true if `e1` and `e2` refer to the same entity	

Invariants If `e` is valid, then `e` and its anchor are bound to the same grid:
`& e.TheGrid() == & e.TheAnchor().TheGrid()`.

Refinements: Grid Element → A.1.2 Grid Sequence Iterator → A.2.1 Grid Incidence Iterator → A.2.3

Notes

1. It can be discussed if this precondition is necessary. Formally, two entities compare as different if not `e1.TheAnchor() == e2.TheAnchor()`. However, requiring this equality allows to compare only the handles, which is faster.

See also: Grid → A.2.7 Element Handle → A.1.5

A.1.2 Grid Element Concept

Description A *Grid Element* is an entity, such as a Grid Vertex → A.1.3, that belongs to a *Grid*. To each grid element, there is associated a unique grid (the *anchor* grid). Two elements may be compared for equality, if they belong to the same grid.

Conceptually, a combinatorial grid consists of its elements of different dimension (Vertices, Edges and so on), plus an incidence relation between them. This does not imply, however, that the element constituing a grid must be stored permanently within the grid.

We name the element types of a grid consistently according to the following table, where we distinguish between names relating to element dimension and element codimension:[1]

k-Element	Dimension	Codimension
`Vertex`	0	
`Edge`	1	
`Face`[2]	2	
`Facet`		1
`Cell`		0

This naming scheme allows for a dimension-independent formulation of many algorithms: for example fluxes in finite volume algorithms are always defined on facets.

Refinement of: Grid Entity → A.1.1

The only refinement is that the anchor type must be equal to the grid type.

Refinements: Vertex → A.1.3 Edge Face Facet Cell → A.1.4

Notes

1. some of these types can coincide: for a concrete 2D-grid, the types `Edge` and `Facet` can be the same. But it is also possible to define them as distinct types.

2. There cannot be a type `Face` defined for 1D-grids.

See also: Grid → A.2.7 Grid Entity → A.1.1 Grid Element Handle → A.1.5

A.1.3 Grid Vertex Concept

Description A *Grid Vertex* represents the mathematical concept of a vertex — a 0-dimensional entity in a Grid → A.2.7.

Refinement of: Grid Element → A.1.2

Notation

`V` is a type which is a model of *grid vertex*
`v` is an object of type `V`
`g` is an object of type `V::grid_type`
`h` is an object of type `V::vertex_handle`
`ci` is an object of type `V::CellIterator`

Associated types

NOTE: The types and expression involving Incidence Iterators → A.2.3 are given below for the case of cell-on-vertex iteration. Analogous types and expressions can be defined for the other element types, like edge, facet, or cell. The tables are to be understood in the following sense:
If a vertex defines the incidence iterator over cells, *then* the requirements under *Optional part* apply. Analogous requirements take effect if 'cell' is replaced by another element type.

Name	Expression	Description
handle type	`V::vertex_handle`	type of the corr. Vertex Handle → A.1
	Optional part (as example)	
cell-on-vertex iterator	`V::CellIterator`	type of the corr. CellOnVertexIterator

Valid Expressions

Name	Expression	Type requirements	return type
handle	`v.handle()`		`V::vertex_handle`
	Optional part (as example)		
cell-on-vertex iteration	`v.FirstCell()`		`V::CellIterator`
no. of incident cells	`v.NumOfCells()`		`int`

Expression semantics

Name	Expression	Precond.	Semantics	Postcondition
handle	`h = v.handle();`	v is valid[2]	shorthand for `h = v.TheGrid().handle(v)`	`v == v.TheGrid().vertex(h)`
		Optional part (as example)		
cell-on-vertex iteration start	`ci = v.FirstCell()`	v is valid	`ci` points to first cell incident to `v`	`ci.TheVertex() == v`
number of incident cells	`n = v.NumOfCells()`	v is valid	n is the number of cells incident to `v`	

Complexity guarantees All operations are amortized constant time (if performed over all vertices).

Models: `Triang2D::Vertex` → B.1

See also: Grid → A.2.7 Grid Element Handle → A.1.5 Vertex Handle → A.1 Grid Element → A.1.2 Grid

[2] see A.1

Cell → A.1.4 Sequence Iterator → A.2.1 Incidence Iterator → A.2.3

A.1.4 Grid Cell Concept

Description The *Grid Cell* concept corresponds to the mathematical concept of a d-dimensional entity of a d-dimensional Grid → A.2.7

Refinement of: Grid Element → A.1.2

The Grid Cell concept alone does not introduce anything new in comparison with Grid Vertex. However, a concrete cell type will typically offer incidence iteration, and be a model of some refinements of Grid Cell.

Refinements: Vertex-Range Cell Edge-Range Cell Neighbor-Range Cell Full Grid Cell

Models: `Complex2D::Cell` defined in `cell2d.h`

See also: Grid Vertex → A.1.3 Grid Element → A.1.2 Vertex-On-Cell Iterator → A.2.4 Grid Range → A.2.5 Vertex Grid Range → A.2.6

A.1.5 Grid Element Handle Concept

Description A *Grid Element Handle* is a minimal representation of a Grid Element → A.1.2. They are unique only within a single Grid → A.2.7, which allows to map back and forth between handles and their corresponding elements.

Typical models of Element Handles are basic built-in types, such as integral types or pointers. As such, it is not required that the types of handles corresponding to different element types (such as vertices and cells) be themselves different.

Often, handles have to fullfil additional requirements. For example, implementations of Container Grid Functions → A.3.4 often exploit special properties of handles, such as being consecutively ordered integral types, or being a hashable type (that is, the `hash` template has been specialized for them).

Refinement of: STL Assignable · STL Equality Comparable

Refinements: Grid Vertex Handle Grid Edge Handle Grid Facet Handle Grid Cell Handle

These refinements to not add new requirements, they are just there for distinguishing between different handle types.

See also: Grid → A.2.7 Grid Entity → A.1.1 Grid Element → A.1.2

A.2 Grid Iterators

A.2.1 Grid Sequence Iterator Concept

Description A *Grid Sequence Iterator* lets a Grid Range → A.2.5 be seen as a sequence of the iterators element type.

Refinement of: STL Forward iterator · Grid Entity → A.1.1

Notation

`I` is a model of sequence iterator
`i, j` are objects of type `I`

Associated types

Name	Expression	Description
grid type	`I::grid_type`	type of the underlying grid, model of Grid → A.2.7
anchor type	`I::anchor_type`	type of the underlying grid range, model of Grid Range → A.2.5
element type	`I::element_type`	type of the underlying element, model of Grid Element → A.1.2
value type	`I::value_type`	synomym for `I::element_type`

Valid Expressions For consistency with Grid Incidence Iterator → A.2.3, an `IsDone()` member function is provided.

Expression semantics `IsDone()` is true iff the iterator is past-the-end.

Refinements: Grid Vertex Iterator → A.2.2 Grid Edge Iterator → A.2.2

See also: Grid → A.2.7 Grid Element → A.1.2 Grid Incidence Iterator → A.2.3

A.2.2 Grid Vertex (Edge, Facet, Cell ...) Iterator Concept

Description A Grid Vertex Iterator allows iteration over the vertices of a Vertex Grid Range → A.2.6.

Analogous concepts exist for edges, facets and cells; these are not listed seperately.

Refinement of: Grid Sequence Iterator → A.2.1

Associated types

Name	Expression	Description
vertex type	`I::Vertex`	model of Grid Vertex → A.1.3

Models: `Triang2D::VertexIterator` → B.1

See also: Grid Sequence Iterator → A.2.1 Vertex Grid Range → A.2.6

A.2.3 Grid Incidence Iterator Concept

Description A *Grid Incidence Iterator* allows to access all elements of a given type incident to an given element (the *anchor*), for example all vertices incident to a given cell, or all edges incident to a given vertex.

For each combination of Grid Element → A.1.2 types, there is a subconcept of incidence iterator, for example, the Vertex-On-Cell Iterator → A.2.4.

Refinement of: STL Forward Iterator · Grid Entity → A.1.1

Notation

`I` is a model of incidence iterator
`i, j` are objects of type `I`

Associated types

Name	Expression	Description
grid type	`I::grid_type`	type of the iterators underlying grid (range)
anchor type	`I::anchor_type`	type of the iterators anchor element
element type	`I::element_type`	type of referenced Element → A.1.2
value type	`I::value_type`	synomym for `I::element_type`

Valid Expressions

Name	Expression	Type requirements	return type
validity check	`i.IsDone();`[1]		`bool`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
constructor	`i(a);`	`a` is valid (→ p. 202)	set i to `a`'s first inc. element of type `I::element_type`	`i.TheAnchor() == a`
validity check	`i.IsDone();`		true iff `i` is past-the-end.	

Refinements: VertexOnCell Iterator → A.2.4 EdgeOnCell Iterator → A.2

Notes

1. For circular sequences, such as the vertices around a cell, there is no natural, predefined past-the-end value. Therefore it is more natural to let the iterator itself decide when it is invalid, instead of the standard comparison with a past-the-end iterator (which is defined as well, in order to be able to use STL algorithms).

See also: Grid → A.2.7 Grid Element → A.1.2 Grid Sequence Iterator → A.2.1

A.2.4 Vertex-On-Cell (-Facet, ...) Iterator Concept

Description A *Vertex-On-Cell Iterator* refines the concept of Grid Incidence Iterator → A.2.3 : It allows to access all vertices incident to a given cell. Thus, it can also be seen as a Grid Sequence Iterator → A.2.1 over the Vertex Grid Range → A.2.6 of that cell.

Similar iterators can be defined for other anchor elements, most notably facets. For these, just replace *cell* with *facet*.

Refinement of: Incidence Iterator → A.2.3

Notation

`V` is a model of Vertex-On-Cell Iterator
`v` is an object of type `V`

`c` is an object of type `V::Cell`

Associated types The types `V::element_type` and `V::anchor_type` can now be named more specifically. The names from the incidence iterator concept remain valid.

Name	Expression	Description
Vertex type	`V::Vertex`	synomym to `V::element_type`
Cell type	`V::Cell`	the cell type `V` operates upon, model of Grid Cell → A.1.4 model of Vertex Grid Range → A.2.6
Anchor type	`V::anchor_type`	synonym to `V::Cell`

Valid Expressions

Name	Expression	Type requirements	return type
anchor cell	`v.TheCell();`		`V::Cell const&`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
anchor cell	`Cell C = v.TheCell();`	`v is not singular`	get the anchor of `v`	`c` is a valid cell; `c == v.TheCell()`

Models: `Triang2D::VertexOnCellIterator`

See also: Grid → A.2.7 Grid Range → A.2.5 Grid Element → A.1.2 Grid Vertex → A.1.3 Grid Cell → A.1.4 Grid Sequence Iterator → A.2.1 Grid Incidence Iterator → A.2.3

A.2.5 Grid Range Concept

Description A *Grid Range* is a part of a grid, its *base grid*. The underlying mathematical concept is that of (a subset of) a (finite) CW-complex. Some well-known specializations of this concept are triangulations boundary complexes of convex polytopes and regular Cartesian grids.

A Grid Range behaves in most circumstances like a Grid → A.2.7. The main difference is that a Grid Range has reference semantics with respect to its underlying base grid, that is, the incidence relationship is determined by the base grid. This influences the behaviour of Incidence Iterators → A.2.3 associated with a Grid Range, which may visit grid elements → A.1.2 that are contained in the base grid, but not in the Grid Range.[3]

A Grid is a special case of a Grid Range.[1]

NOTE: A grid range as such does offer almost no functionality at all. Any useful model will be a specialization of one or more element ranges [4] like Grid Vertex Range → A.2.6.

Refinement of: STL Assignable

Notation

`R` is a type which is a model of grid range

`r` is an object of type `R`
`G` is `R::grid_type`

Associated types [2]

Name	Expression	Description
base grid	`R::grid_type`	type of the ranges' base grid

Valid Expressions

Name	Expression	Type requirements	return type
base grid reference	`r.TheGrid()`		`grid_type const&`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
base grid reference	`G const& g = r.TheGrid()`	`r` has not been default constructed	get the grid `r` references	`&g == &(g.TheGrid()) == &(r.TheGrid())`

Refinements: Vertex Grid Range → A.2.6 Edge Grid Range → A.2.6 Facet Grid Range → A.2.6 Cell Grid Range → A.2.6

Models: `enumerated_grid_range` `Triang2D` → B.1 `Complex2DInput`

Notes

1. If `R` is a model of Grid → A.2.7, then `R::grid_type` is identical to `R`. An object `r` of type `R` then references itself via `r.TheGrid()`, that is, it has value semantics.
2. Technically, these types are bundled in a struct `grid_types<R>` which is used by the algorithms to access these types. This opens up the possibility to parameterize algorithms by such a *traits class* like `grid_types<R>`, thereby introducing different iterator and element types, for example counting iterators or debug iterators producing graphical output. An example is the implementation of CELL NEIGHBOR SEARCH on page 222.

   In this case, it would be more precise to say that one "associates types with `R`", instead of speaking of "types associated with `R`".
3. This may seem to be an odd behaviour. However, it faithfully reflects what many locally operating grid-based algorithms are intuitively expected to do when given a proper subrange of a grid: The range restricts the region where some work is to be done, but on each element, the algorithm accesses also some meighboring elements via incidence iterators, which may or may not belong to the range. The most striking example for this occurs if grids are distributed with some overlap: On each part, the algorithm works only on the locally owned range, but it accesses also elements in the overlap which are copied from other parts.
4. It is not necessary to require all possible element types to be defined. Useful examples are Input Grids, such as mentioned in section 4.1.7.1 which are used just to read a grid from a specific file format.

See also: See also Grid → A.2.7 Grid Element → A.1.2 Grid Sequence Iterator → A.2.1

A.2.6 Vertex (Edge, Cell, ...) Grid Range Concept

Description A *Vertex Grid Range* is a sequence of objects of type `V` which is a model of Grid Vertex → A.1.3.

Virtually the same definitions can be made for the other element types, replacing `Vertex` with `Edge`, `Cell` etc. Therefore, Vertex Grid Range is chosen to stand generically for Edge Grid Range, Cell Grid Range and so on.

The Vertex Grid Range concept is seldom useful as such; its primary use is to define further refinements of the basic Grid Range → A.2.5 concept: Useful concrete grid types generally will be models of (at least) two different range concepts, such as Vertex Grid Range and Cell Grid Range.

Refinement of: Grid Range → A.2.5

Notation

`R` is a type which is a model of Vertex Grid Range
`r` is an object of type `R::grid_type`
`v` is an object of type `R::Vertex`
`vi` is an object of type `R::VertexIterator`

Associated types

Name	Expression	Description
Vertex	`R::Vertex`	type of vertex (model of Grid Vertex → A.1.3)
handle	`R::vertex_handle`	model of Grid Vertex Handle → A.1
Vertex iterator	`R::VertexIterator`	model of Grid Vertex Iterator → A.2.2

Valid Expressions

Name	Expression	Type requirements	return type
start of sequence	`vi = r.FirstVertex()`		`VertexIterator`
end of sequence	`vi = r.EndVertex()`		`VertexIterator`
size of sequence	`r.NumOfVertices()`		`size_type`
handle-to-element	`v = r.vertex(h)`		`Vertex`
element-to-handle	`h = r.handle(v)`		`vertex_handle`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
start of sequence	`v = r.FirstVertex()`		return iterator pointing to the first vertex of `s`	`v` is valid[3], or `v == r.EndVertex()`
end of sequence	`v = r.EndVertex()`			`v` is past-the-end.

[3] see A.1

Name	Expression	Precondition	Semantics	Postcondition
size of sequence	`n = r.NumOfVertices()`		get the number of vertices of `s`	`n == distance (r.FirstVertex(), r.EndVertex())`
handle-to-element	`v = r.vertex(h)`	`h` is a handle of `r.TheGrid()`	Get vertex corr. to `h`	`r.handle(v) == h`
element-to-handle	h = r.handle(v)	`v` is a vertex of `r.TheGrid()`	Get handle corr. to `v`	`r.element(h) == v`

Complexity guarantees The `r.NumOfVertices()` operation has complexity at most $O(V)$, where V is the number of vertices of `r`[1].
Element/handle conversions are $O(1)$.

Models: Virtually every concrete grid or grid range: `enumerated_grid_range` `Complex2D` `Triang2D` → B.1

Notes

1. The reason why Vertex Grid Range does not require $O(1)$ complexity for this operation is the following: Such a sequence might arise from a grid range given by a simple enumeration of its *cells* (or their handles, for example `enumerated_grid_range`). The elements of lower dimension belonging to `r` are then implicitely defined by the closure (in a topological sense) of the set of cells in the underlying grid. The determination of, for example, the set of vertices can be done in expected time O(V) (using partial grid functions → A.3.6, see also section 4.2.2.3). Requiring to do this determination at time of construction of the range would impose the cost of doing so also to clients not interested in this functionality.

A.2.7 Grid Concept

Description The mathematical concept underlying *Grid* is that of (a subset of) a (finite) CW-complex → 3.1.1. Some well-known specialization of this concept are triangulations, boundary complexes of convex polytopes and Cartesian grids.

Refinement of: Grid Range → A.2.5

The main difference to grid ranges is that grids stand for their own — there is no underlying base grid. This means that all grid entities produced by calls to member functions of a grid `g` refer to `g` with their grid anchor references.

Virtually all algorithms can do with grid ranges, they do not require grids.

Notation `G` is a type which is a model of grid

Associated types

Name	Expression	Description
base grid	`G::grid_type`	identical to `G`

Refinements: Grid-With-Boundary

Models: `Triang2D` → B.1 `Complex2D`

See also: Grid Range → A.2.5 Grid Element → A.1.2 Grid Sequence Iterator → A.2.1

A.2.8 Cell-Vertex Input Grid Range Concept

Description A *Cell-Vertex Input Grid Range* is a Grid Range which can be used to construct another grid. More precisely, its representation is cell-based: Supported is mostly iteration over cells and over vertices incident to cells.

Refinement of: Cell Grid Range → A.2 · Vertex Grid Range → A.2.6

Associated types

Name	Expression	Description
cell type	`R::Cell`	model of Grid Cell → A.1.4 defining `R::Cell::VertexIterator` (a model of VertexOnCellIterator → A.2.4)

Models: `enumerated_grid_range` `Triang2D` → B.1 `Complex2DInput`

See also: See also Grid Range → A.2.5 Cell Grid Range → A.2 Vertex Grid Range → A.2.6 VertexOnCellIterator → A.2.4 `CellNeighborSearch` → C.1

A.3 Grid Functions

A.3.1 Grid Element Function Concept

Description The *Grid Element Function* concept models the mathematical concept of a mapping from grid elements of some fixed type (vertex, edge, cell) which is a model of Grid Element → A.1.2, to values of some type `T`.

Refinement of: STL Adaptable Unary Function (AUF)

Notation

type `F` is a model of Grid Element Function
`f` is an object of type `F`
`e, e1, e2` are objects of `F::element_type`
`t` is an object of `F::value_type`

Associated types

Name	Expression	Description
element type	`F::element_type`	model of Grid Element → A.1.2 synonym for `F::argument_type` (from AUF)
value type	`F::value_type`	synomym for `F::result_type` (from AUF)

Valid Expressions

Name	Expression	Type requirements	return type
function evaluation	`t = f(e);`		`value_type`

Expressions Semantics

The semantics of function evaluation are more restrictive than those for AUF[1].

Name	Expression	Precondition	Semantics	Postcondition
evaluation	`t = f(e)`	`e` is in the domain of `f`	evaluate `f` at the argument `e`	`t` is equal to `f(e)`[1]

Invariants Argument identity: if `e1 == e2` then `f(e1)` is equal to `f(e2)`[2]

Refinements: Grid Function → A.3.2

Models: `cell_nb_degree<GRID>`

Notes

1. The important difference to STL function objects is that the latter are *not* guaranteed to deliver the same result for subsequent evaluations on the same argument.

2. The type `F::value_type` is not required to be STL Equality Comparable. If it is, then `e1 == e2` implies `f(e1) == f(e2)`.

A.3.2 Grid Function Concept

Description The *Grid Function* concept refines the Grid Element Function → A.3.1 concept in that it binds the function to a particular grid. Evaluating a grid function without a valid grid set or with an element whose grid is different from that of the grid function is considered an error.

By binding to a particular grid, it is possible to treat both domain and range of a grid function as sequences with associated iterators.

Refinement of: Grid Element Function → A.3.1 · STL (immutable) Container

Notation

`F` is a type which is a model of Grid Function
`f` is an object of type `F`

Definitions The *range* of a grid function `f` is the set of all elements of type `F::element_type` in `f.TheGrid()`.

The *domain* of a grid function is the set of all values of the form `f(e)` where `e` is in the range of `f`.

Associated types

Name	Expression	Description
Grid type	`F::grid_type`	type of the corresponding associated grid, model of Grid → A.2.7

Valid Expressions

Name	Expression	Type requirements	return type
Grid reference	`f.TheGrid();`		`F::grid_type const&`

Expressions Semantics

Name	Expression	Precondition	Semantics	Postcondition
Grid reference	`G& g = f.TheGrid();`	`f` is bound to a grid	get reference to the underlying grid	`&(g.TheGrid()) == &(f.TheGrid())`

Refinements: Mutable Grid Function → A.3.3

Models: `cell2handle_map<G>` `vertex2coord_map<Geom>`

A.3.3 Mutable Grid Function Concept

Description The *Mutable Grid Function* concept refines the Grid Function → A.3.2 concept. It allows to change function values, that is, to store values on elements. This is accomplished by an additional array-access operator `[]`.

Refinement of: Refinement of Grid Function → A.3.2 · STL (mutable) Container

Valid Expressions

Name	Expression	Type requirements	return type
write access	`f[e];`		`F::value_type&`

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
write access	`f[e] = t;`	`f` is bound to a grid	assign the value `t` to `f(e)`	`f(e)` is equal to `t`

Refinements: Container Grid Function → A.3.4

A.3.4 Container Grid Function Concept

Description The *Container Grid Function* concept refines the Mutable Grid Function → A.3.3 concept. A Container Grid Function can be created and filled with values by a client, much like an ordinary container. This of particular importance for algorithms needing temporary storage, such as boolean flags on grid elements.

Refinement of: Mutable Grid Function → A.3.3 · STL Assignable

Notation

`F` is a type which is a model of Container Grid Function
`f` is an object of type `F`
`G` is shorthand for `F::grid_type`

g is an object of type G.

Valid Expressions

Name	Expression	Type requirements	return type
Default construction	`F f();`		
Construction from grid	`F f(g);`		
Construction and initialization	`F f(g,t);`		
Binding to grid	`f.set_grid(g);`		

Expression semantics

Name	Expression	Precondition	Semantics	Postcondition
Default construction	`F f();`		default construct `f`	`f` is unbound w/r access is error
construction from grid	`F f(g);`		construct and bind `f` to `g`	`f` is bound to `g` write access ok read undef.
construction and initialization	`F f(g,t);`		construct and bind `f` to `g`, initialize all values to `t`	`f` is bound to `g` write access ok `f(e) == t` ∀ `e`
Binding to grid	`f.set_grid(g);`	f is unbound	bind `f` to `g`	`f` is bound to `g` write access ok, read undef.

Complexity Guarantees Default construction takes constant time.
Construction from grid and construction with initalization both take time at most O(`f.size()`), that is, the number of elements of type `F::element_type` of `g`.

Refinements: Total Grid Function → A.3.5 Partial Grid Function → A.3.6

A.3.5 Total Grid Function Concept

Description The *Total Grid Function* concept refines the Container Grid Function → A.3.4 concept. A total grid function reserves storage to hold a value for each element in its range.

Refinement of: Container Grid Function → A.3.4 The Total Grid Function concept is more specific on what happens on construction: Storage is allocated when initializing with a grid, and all memory locations are filled with a value if it is specified in the constructor.

Complexity Guarantees Default construction takes constant time.
Construction from grid and construction with initalization both take time at O(`f.size()`), that is, the number of elements of type `F::element_type` of `g`.

Models: `grid_function_vector<E,T>`

See also: See also Grid Element Function → A.3.1 Grid Function → A.3.2 Mutable Grid Function

→ A.3.3 Container Grid Function → A.3.4 Partial Grid Function → A.3.6

A.3.6 Partial Grid Function Concept

Description The *Partial Grid Function* concept refines the Container Grid Function → A.3.4 concept. A partial grid function reserves storage only for those values which are write-accessed explicitly. This is in contrast to Total Grid Function → A.3.5 which allocates storage for each value.

Partial grid functions are of particular importance for locally operating algorithms with sublinear runtime and memory requirements.

Refinement of: Container Grid Function → A.3.4 In addition to a Container Grid Function, a Partial Grid Function has methods to check wether an element has be explitely written to (`f.defined(e)`), and to set the default value (`f.set_default(t)`).

Complexity Guarantees Default construction takes constant time.
Construction from grid and construction with initalization both take constant time.
The `set_default(t)` operation takes constant time.
The access operations `f(e), f[e]` and the test `f.defined(e)` take at most logarithmic time, or amortized constant time, or expected constant time.
The memory requirements are at most proportional to the total number of different elements `e` for which `f[e]` has ever been called.

Models: `partial_grid_function_hash<E,T>` — a generic implementation of partial grid functions by hash tables.

Notes

1. This means that storage is also added if the access is meant read-only! Therefore, one should use the syntax `t = f(e)` in this case.

Appendix B

Concrete Grid Components

B.1 `Triang2D`

The `Triang2D` grid type (and its associated types) implements a very simple two-dimensional triangulation data structure. The only incidence information stored are vertices incident to cells. This happens in a single integer array where three consecutive entries contain the vertex indices of the cell.

Consequently, not every possible type of incidence iteration can be supported: Only VertexOnCell and FacetOnCell can be implemented without further information.

Likewise, sequential iteration can be implemented *directly* only over vertices and cells. Both are represented simply by a reference to their grid and by their handle, an integer between 0 and the number of grid vertices/cells minus 1. Iteration is obvious: Just increment the handle.

We can, however, implement iteration over edges (same as facets) as indicated in section 4.2.2.1: We iterate over cell and their incident facets, taking care of visiting each facet exactly once. As no cell neighbors are stored, we must explicitly mark visited facets by a partial grid function.

These, in turn, can be profit from a generic implementation, as discussed in section 4.2.3.

Edges (facets) themselves are represented by a `FacetOnCellIterator`, which allows both access to the incident vertices and one incident facet. The two incident vertices can also be used to compare and hash edges. Note that this works only if edges are uniquely determined by their vertex set, e. g. if the grid's poset is a lattice.

The presentation of the implementations has been shortened slightly, in order to fit on two pages. So it is possible that some typedefs are missing that should be there according to the concepts as specified in appendix 7.3. Also, code for checking pre- and postconditions has been omitted.

B.2 A file adapter component

This type of component has been discussed in section 4.1.7.1. The code is shown in figure B.3. It is very similar to the code for `Triang2D`, because the file is read into a buffer before. This decision is completely hidden from the client code.

```
class Triang2D : public grid_types_base_Triang2D {
private:
  int* cells;
  int  ncells;
  int  nvertices;
public:
  Triang2D()
    : cells(0),  ncells(0), nvertices(0) {}
  Triang2D(int* c, int nc, int nv);

public:

  friend class Triang2D_VertexOnCellIterator;
  friend class Triang2D_FacetOnCellIterator;
  friend class Triang2D_Cell;
  friend class Triang2D_Vertex;

  int NumOfCells   () const { return ncells   ;}
  int NumOfVertices() const { return nvertices;}

  VertexIterator FirstVertex() const
    { return VertexIterator(*this);}
  FacetIterator  FirstFacet()  const
    { return FacetIterator (*this);}
  CellIterator   FirstCell()   const
    { return CellIterator  (*this);}
};
```

the Grid class

```
class Triang2D_VertexOnCellIterator
  : public grid_types_base_Triang2D {

  typedef  Triang2D_VertexOnCellIterator self;
private:
  Cell c;
  int  vc;
public:
  Triang2D_VertexOnCellIterator() : vc(-1) {}
  Triang2D_VertexOnCellIterator(Cell const& cc)
    : c(cc), vc(0) {}

  self& operator++() { ++vc; return *this;}
  Vertex operator*()  const
    { return Triang2D_Vertex(*(c.g),
                         c.g->cells[3*c.c+vc]);}

  bool     IsDone() const { return (vc == 3);}
  operator bool()   const { return (vc < 3);}

  vertex_handle handle() const
    { return c.g->cells[3*c.c+vc];}

  typedef Cell  anchor_type;
  Cell const& TheCell() const { return c;}
  Cell const& Anchor () const { return c;}
};
```

the VertexOnCell Iterator class

```
class Triang2D_FacetIterator
  : public facet_set_of_cells_iterator<Triang2D_Cell>
{
  typedef facet_set_of_cells_iterator<Triang2D_Cell>
                                              base;
 public:
  Triang2D_FacetIterator() : base(CellIterator()) {}
  Triang2D_FacetIterator(CellIterator const& c)
    : base(c) {}
  Triang2D_FacetIterator(grid_type    const& g)
    : base(CellIterator(g)) {}
};
```

the FacetIterator class

```
class Triang2D_Vertex
  : public grid_types_base_Triang2D {

  typedef Triang2D_Vertex    self;
public:
  typedef Triang2D grid_type;
  typedef int      handle_type;
  typedef self     value_type;
private:
  grid_type const* g;
  int              v;
public:
  Triang2D_Vertex() : g(0), v(-1) {}
  Triang2D_Vertex(grid_type const& gg)
    : g(&gg), v(0) {}
  Triang2D_Vertex(grid_type const& gg, int vv)
    : g(&gg), v(vv) {}
  ~Triang2D_Vertex() {}

  bool IsDone()   const {return (v == g->nvertices);}
  operator bool() const {return (v <  g->nvertices);}
  self const& operator*()  const { return *this;}
  self      & operator++() { ++v; return *this;}

  grid_type const& TheGrid() const { return *g;}
  handle_type handle() const { return v;}
};
```

the Vertex/VertexIterator class

Figure B.1: Implementation of class `Triang2D`: Part I

```
struct grid_types_base_Triang2D {

  typedef Triang2D        grid_type;

  typedef Triang2D_Vertex Vertex;
  typedef Triang2D_Edge   Edge;
  typedef Edge            Facet;
  typedef Triang2D_Cell   Cell;

  typedef Cell            CellIterator;
  typedef Vertex          VertexIterator;
  typedef Triang2D_FacetIterator EdgeIterator;
  typedef EdgeIterator    FacetIterator;

  typedef Triang2D_VertexOnCellIterator VertexOnCellIterator;
  typedef Triang2D_FacetOnCellIterator  FacetOnCellIterator;
  typedef FacetOnCellIterator           EdgeOnCellIterator;

  typedef int vertex_handle;
  typedef int cell_handle;
};

struct grid_types<Triang2D>
  : public grid_types_base_Triang2D
{
  // ... omitted
};
```

the grid types class

```
class Triang2D_FacetOnCellIterator
  : public grid_types_base_Triang2D {
  typedef  Triang2D_FacetOnCellIterator self;
private:
  Cell c;
  int  fc;
public:
  Triang2D_FacetOnCellIterator() : fc(-1) {}
  Triang2D_FacetOnCellIterator(Cell const& cc)
    : c(cc), fc(0) {}

  self& operator++() { ++fc; return *this;}
  Facet operator*()  const
    { return Triang2D_Edge(*this); }

  bool IsDone()   const { return (fc == 3);}
  operator bool() const { return (fc <  3);}

  Vertex V1() const
   { return Vertex(*c.g, c.g->cells[3*c.c+  fc   ]);}
  Vertex V2() const
   { return Vertex(*c.g, c.g->cells[3*c.c+ (fc%3)]);}

  typedef Cell  anchor_type;
  Cell const& TheCell() const { return c;}
  Cell const& Anchor () const { return c;}
};
```

the FacetOnCell Iterator

```
class Triang2D_Edge : public grid_types_base_Triang2D {
  typedef Triang2D_Edge  self;
private:
  Triang2D_FacetOnCellIterator fc;
public:
  Triang2D_Edge() {}
  Triang2D_Edge(Triang2D_FacetOnCellIterator ffc)
    : fc(ffc) {}

  Vertex V1()      const { return fc.V1();}
  Vertex V2()      const { return fc.V2();}
  Cell   TheCell() const { return fc.TheCell();}

  friend bool operator==(self const& a, self const& b)
  {
    vertex_handle av1 = a.V1().handle();
    vertex_handle av2 = a.V2().handle();
    vertex_handle bv1 = b.V1().handle();
    vertex_handle bv2 = b.V2().handle();
    return( ((av1 == bv1) && (av2 == bv2))
         || ((av2 == bv1) && (av1 == bv2)));
  }

  friend bool operator<(self const& a, self const& b)
  {
    vertex_handle max1(max(a.V1().handle(),a.V2().handle()));
    vertex_handle max2(max(b.V1().handle(),b.V2().handle()));
    return ( (max1 < max2)
        || (   (max1 == max2)
            && (min(a.V1().handle(),a.V2().handle())
min(b.V1().handle(),b.V2().handle()))));
  }
};
```

the Edge class

```
class Triang2D_Cell
  : public grid_types_base_Triang2D  {
  typedef Triang2D_Cell     self;
  friend class     Triang2D_VertexOnCellIterator;
  friend class     Triang2D_FacetOnCellIterator;
public:
  typedef Triang2D grid_type;
  typedef int       handle_type;
  typedef self      value_type;
private:
  grid_type const* g;
  int              c;
public:
  Triang2D_Cell() : g(0), c(-1) {}
  Triang2D_Cell(grid_type const& gg)
     : g(&gg), c(0) {}
  ~Triang2D_Cell() {}

  bool IsDone()   const { return (c == g->ncells);}
  operator bool() const { return (c <  g->ncells);}
  self const& operator*() const { return *this;}
  self & operator++() { ++c; return *this;}

  grid_type const& TheGrid() const { return *g;}
  handle_type      handle()  const { return c;}

  VertexOnCellIterator FirstVertex() const
    { return VertexOnCellIterator(*this);}
  FacetOnCellIterator  FirstFacet () const
    { return FacetOnCellIterator(*this);}

  unsigned NumOfVertices() const { return 3;}
  unsigned NumOfEdges()    const { return 3;}
  unsigned NumOfFacets()   const { return 3;}
};
```

the Cell/CellIterator class

Figure B.2: Implementation of class `Triang2D`: Part II

```
class grid_file {
private:
  vector<point> coords;
  vector<int>   cells;
  // vertices of cell i =
  // {cells[i*3],cells[i*3+1],
  //  cells[i*3+2]}

public:
  point const& coord(Vertex const& v)
    { return coords[v.v];}
  unsigned NumOfVertices() const
    { return coords.size();}
  unsigned NumOfCells()    const
    { return (cells.size()/3);}
  VertexIterator FirstVertex const
    { return VertexIterator(this);}
  CellIterator   FirstCell   const
    { return CellIterator(this);}

  typedef int vertex_handle;
  typedef int cell_handle;

  struct  Vertex;
  typedef Vertex VertexIterator;
  struct  Cell;
  typedef Cell CellIterator;
  struct VertexOnCellIterator;

};
```

(a) the main class

```
struct grid_file::VertexOnCellIterator {
  int  c; // cell number
  int vc; // in [0,2]
  grid_file const* g;

  VertexOnCellIterator(Cell const& C)
    : g(C.g), c(C.c), vc(0) {}
  Vertex operator*() const
    { return Vertex(cells[3*c+vc],g);}
  bool   IsDone()    const
    { return (c >= g->NumOfVertices());}
  self&  operator++() {
    if(vc < 2)
        ++vc;
    else {
      vc = 0;
      ++c;
    }
    return (*this);
  }
};
```

(b) the vertex-on-cell iterator

```
struct grid_file::Cell {
  int c;
  grid_file const* g;

  Cell(grid_file const* gg)
    : g(gg), c(0) {}
  Cell      & operator++()
    { ++c; return (*this);}
  Cell const& operator*() const
    { return (*this);}
  bool        IsDone()    const
    { return (c >= g->NumOfCells());}

  VertexOnCellIterator FirstVertex() const {
    return VertexOnCellIterator(*this);
  }
};
```

(c) the cell class

```
struct grid_file::Vertex {
  int v;
  grid_file const* g;

  Vertex(grid_file const* gg)
    : g(gg), v(0) {}
  Vertex      & operator++()
    { ++v; return (*this);}
  Vertex const& operator*() const
    { return (*this);}
  bool          IsDone()    const
    { return (v >= g->NumOfVertices());}
};
```

(d) the vertex class

Figure B.3: Grid adapter for the simple file format of page 96

Appendix C

A Sample of Generic Components

C.1 Cell-neighbor-search Algorithm implementation

Declaration

The routine `CalculateNeighborCells` implements the algorithm Cell-neighbor-search discussed on page 61. It comes in three variants, which are all ultimately implemented by the third one:

```
[1] template<class NBF, class CELLRANGE>
    void CalculateNeighborCells
          (NBF            &  Nb,   // out
           CELLRANGE  const&  C);   // in

[2] template<class NBF, class CELLRANGE, class FACETMAP>
    void CalculateNeighborCells
          (NBF            &  Nb,   // out
           CELLRANGE  const&  C,    // in
           FACETMAP       &  F);   // inout

[3] template<class NBF, class CELLRANGE, class FACETMAP, class CGT>
    void CalculateNeighborCells
         (NBF            &  Nb,    // out
          CELLRANGE  const&  C,     // in
          FACETMAP       &  F,     // inout
          CGT        const&);      // in (only type used)
```

Description

`CalculateNeighborCells` takes a range of cells `C` and determines the neighbor relation on its cells. (Two cells are neighbors if they share a facet.) The output is stored in `nb`.

`CalculateNeighborCells` (versions [2], [3]) can be used incrementally in the variable `C`: If $C = C_1 \cup \ldots \cup C_n$ is a partitioning of a given cell set, then calling `CalculateNeighborRange` with the cell set C is equivalent to the successive calls with cell sets $C_1, \ldots, C_n$. (The remaining arguments stay identical.)

Requirements on types

- `CELLRANGE`: `CELLRANGE` is a model of CellGridRange (section A.2.6) .
- `CGT`: The following types must be defined by the type `CGT` (version [3]) or the template `grid_types<CELLSET>` (versions [1], [2]):
  - `Cell`: a model of Grid Cell (section A.1.4) and Facet Grid Range (section A.2.6)
  - `FacetOnCellIterator`: a model of Facet-On-Cell Iterator (section A.2.4) .
  - `Facet`: a model of Grid Facet (section A.1) and of Vertex Grid Range (section A.2.6)
  - `Vertex`: a model of Grid Vertex (section A.1.3) .

  In the following, we use `CGT` also for `grid_types<CELLSET>` in versions [1], [2].
- `NBF`: is a model of Mutable Mapping from `CGT::FacetOnCellIterator` to `CGT::cell_handle`.
- `FACETMAP`: is a model of Mutable Mapping from `vtuple<CGT::grid_type>` to `CGT::FacetOnCellIterator`. In addition, the following expressions must be defined, where `i` is of type `FACETMAP::iterator`, `f` is of type `CGT::FacetOnCellIterator`, and `F` is of type `FACETMAP`:
  - `FACETMAP::iterator i;`
  - `i = F.end();  i = F.find(f);  F.erase(i);`

  This is fullfilled if `FACETMAP` is a model of STL Pair Associative Container.

Notation

F is the set of facets of `C`.
I the set of interior facets of `C`.
B the boundary facets of `C` (such that $I \cup B = F$).
D is the domain of the map `F` before the call.
D' is the domain of the map `F` after the call.

Preconditions

The grid underlying `C` is a subgrid of a *manifold-with-boundary complex* (→ p. 49). Its facets must be uniquely determined by their vertex sets, which is guaranteed if the element poset is a *lattice* (→ p. 53).
For versions [2] and [3], the domain of the map `F` may already contain facets, but these must be boundary facets:

$$D \cap I = \emptyset$$

Postconditions

All pairs of cells sharing an interior facets $\in I$ have been found, as well as all cells having a Facet such that the corresponding vertex set has been in D.

After the call, F contains exactly the boundary facets, except those that have been in it before (versions [2], [3]). In version [1], the value of Nb is set to an invalid cell handle for boundary facets (that is, facets which have been visited by only facet-on-cell iterator).

$$D' = (D \setminus B) \cup (B \setminus D)$$

It follows that $D' \cap I = \emptyset$.

Complexity

Expected time $O(F)$. Execution time is dominated by the number of lookups in the map F; the complexity bound essentially depends on the map type FACETMAP allowing $O(1)$ lookup.

Example

```
#include "Grids/Algorithms/cell-neighbor-search.h"

Triang2D t;
typedef grid_types<Triang2D> gt;
...
hash_map<gt::FacetOnCellIterator> , gt::cell_handle> Nbs;
CalculateNeighborCells(nbf,T);

for(gt::CellIterator c(t); ! c.IsDone(); ++c)
  for(gt::FacetOnCellIterator fc(c); ! fc.IsDone(); ++fc)
    out << nbf[fc] << ' ';
  out << endl;
```

In special circumstances, the mapping from gt::FacetOnCellIterator to gt::cell_handle can be done more efficiently than with a hash table — often, it will be offered by the grid data structure itself.

The implementation

The generic implementation of the most general version [3] is shown below.

```
template<class NBF, class CellSet, class FACETMAP, class CGT>
void CalculateNeighbourCellsBench(NBF           &  NB,
                                  CellSet  const&  cell_set,
                                  FACETMAP      &  facet_map,
                                  CGT       const&)
{

  typedef typename CGT::CellIterator        CellIt;
  typedef typename CGT::Cell                Cell;
  typedef typename CGT::FacetOnCellIterator FacetOnCellIt;

  typedef typename FACETMAP::iterator       MapIt;
  typedef vtuple_2d<typename CGT::grid_type> vtuple;
  // must be equal to FACETMAP::data_type / result_type;

  CellIt c(cell_set);
  for(; !c.IsDone(); ++c){
    Cell C(*c);
    FacetOnCellIt f(C);
    for(; !f.IsDone(); ++f) {
      vtuple facet(*f);
      MapIt nb = facet_map.find(facet);
```

```
      if(nb != facet_map.end()){
        // facet found: nb has already been visited
        // do appropriate entries in the neighbourlists
        //  & remove facet from the map.

        FacetOnCellIt NbIt((*nb).second);
        NB[NbIt] = f   .TheCell().handle();
        NB[f   ] = NbIt.TheCell().handle();

        facet_map.erase(nb);
      }
      else // 1st time this facet is encountered: add it to map
        facet_map[facet] = f ;
    }
  }
}
```

C.2 `FacetIterator` component

Declaration

```
template<class FacetOnCellIt>
class FacetIterator;
```

Description

The class template `FacetIterator<>` implements a Grid Facet Iterator → A.2.2 based on cell iteration and local facet iteration. The iteration proceeds by iterating over each facet of each cell; in order to ensure visiting each facet only once, there is established an arbitrary order between the two possible occurences of an interior facet, and the greater one is silently skipped over.

In order to evaluate the comparison, an order must be defined on cells, and neighbor cells must be accessible[1].

Model of: Facet Iterator (→ p. 206)

Type requirements

`FacetOnCellIt` is a model of FacetOnCell Iterator → A.2.4. The grid type of `FacetOnCellIt` must be a model of Cell Grid Range → A.2.6 and of Grid-With-Boundary; the associated cell type must be a model of STL LessThanComparable.

Complexity

Each facet is visited at most 2 times. Memory consumption is constant — one `CellOnCell-Iterator` and one `FacetOnCellIterator`.

Example

```
a_grid_type g;
a_geometry_type geom(g); // a grid geometry for g

typedef grid_types<a_grid_type> gt;
grid_function<gt::Facet, double> volume(g);

typedef FacetIterator<gt::FacetOnCellIterator>  FacetIt;
for(FacetIt f(g); ! f.IsDone(); ++f)
  volume[*f] = geom.volume(*f);
```

Notes

1. If information about cell neighbors is not available, one has to mark visited facets explicitely, which is discussed in section 4.2.2.1 There could be a compile-time switch on the availability of cell neighbor information, selecting the appropriate version of `FacetIterator` automatically.

The implementation

```
template<class FacOnCellIt>
class FacetIterator  {
public:
  typedef typename FacOnCellIt::grid_type grid_type;
  typedef grid_types<grid_type>           gt;
  typedef FacOnCellIt                     local_it;
  typedef typename gt::CellIterator       global_it;

  typedef typename gt::Facet              Facet;
private:
  //----- DATA --------
  global_it         c;  // current cell
  local_it          fc; // current facet on c
  typedef FacetIterator<FacOnCellIt> self;
public:
  typedef grid_type anchor_type;
  typedef Facet     value_type;

  FacetIterator() {}
  FacetIterator(grid_type const& g) { init(g.FirstCell()); }
  FacetIterator(const global_it& cc) { init(cc) ;}

private:
  void init(const global_it& cc)
    {
      c = cc;
      if(!c.IsDone())
fc = (*c).FirstFacet();
      while(! IsDone() && ! is_new_facet())
advance();
    }

public:
  //---------- iterator operations -----------------

  inline self& operator++() {
    advance();
    while( ! IsDone() && ! is_new_facet())
      advance();
    return *this;
  }

  Facet operator*() const {return (*fc);}
  bool  IsDone() const {return (c.IsDone());}

  const grid_type& TheGrid() {return (c.TheGrid());}

  friend bool operator==(const self& ls, const self& rs) {
    return (    (ls.IsDone() && rs.IsDone())
     || ((ls.c == rs.c) && (ls.fc == rs.fc)));
  }
  friend bool operator<(const self& ls, const self& rs) {
    return ((ls.c < rs.c) || ((ls.c == rs.c) && (ls.fc < rs.fc )));
  }

private:
  void advance();
  bool is_new_facet()
    { return (TheGrid().IsOnBoundary(fc) || (fc.OtherCell() < *c));}

};

template<class FacOnCellIt>
```

```
inline void
FacetIterator<FacOnCellIt>::advance()
{
  ++fc;
  if(fc.IsDone()) {
    ++c;
    if(! c.IsDone()){
      fc = (*c).FirstFacet();
    }
  }
}
```

C.3 The `CopyGrid` primitive

Description

The `CopyGrid` familiy of functions allows to contruct a grid of type `G1` from an arbitrary grid type `G2`. In common cases, when `G1` is a cell-based grid type for unstructured grids, such as `Complex2D` or `Triang2D` (B.1). `G2` must be a model of Cell-Vertex Input Grid Range (→ p. 212).

The versions [2]-[5] make an *associative copy* of a source grid `Gsrc` to a destination grid `Gdest`. The `CopyGrid0XXX` functions ([1], [2]) do only a copy of combinatoric grids, the others copy also geometric information.

For each concrete type `G1`, there must exist specialized implementations. The more general grids the type `G1` can represent, the more useful these functions are. For very specialized grids, like Cartesian ones, they do not make much sense.

This algorithm is in some sense the analogue to the STL `copy` algorithm for sequences, only that it cannot be fully generic on the destination argument.

Declaration

```
[1]
template<class G2>
void
CopyGrid0(G1           & destG,
          G2      const& srcG);

[2]
template<class G2, class VertexMap, class CellMap>
void
CopyGrid0VC(G1           & destG,
            G2      const& srcG,
            VertexMap    & V21,
            CellMap      & C21);

[3]
template<class Geom1, class G2, class Geom2>
void
CopyGrid(G1        & destG,
         Geom1 const& destGeom,
         G2    const& srcG,
         Geom2 const& srcGeom);

[4]
```

```
template<class Geom1, class G2, class Geom2, class VertexMap>
void
CopyGridV(G1           & destG,
          Geom1        & destGeom,
          G2      const& srcG,
          Geom2   const& srcGeom,
          VertexMap    & V21);

[5]
template<class Geom1, class G2, class Geom2, class VertexMap, class CellMap>
void
CopyGridVC(G1           & destG,
           Geom1        & destGeom,
           G2      const& srcG,
           Geom2   const& srcGeom,
           VertexMap    & V21,
           CellMap      & C21);
```

Type requirements

`Geom1` is a model of Mutable Vertex Grid Geometry.
`G2` is a model of Cell-Vertex Input Grid Range (→ p. 212)
`Geom2` is a model of Vertex Grid Geometry
`VertexMap` is a model of Mutable Mapping from `G2::vertex_handle` to `G1::vertex_handle`
`CellMap` is a model of Mutable Mapping from `G2::cell_handle` to `G1::cell_handle`

Preconditions

`srcGeom` is bound to `srcG`.

Postconditions

`destG` contains a copy of `srcG`, and ([3]-[5]) `destGeom` contains a copy of `srcGeom`, such that the mappings `V21` and `C21` induce a grid isomorphism from `srcG` to `destG`.

Complexity

Linear in the size of `srcG`.

Example

```
RegGrid2D R(10,10);  // 10x10 Cartesian grid
mapped_geometry_reg2d<Linear2D>  geomR(R, Linear2D::identity); //
Complex2D  G;         // empty grid
stored_geometry_complex2d geomG(G);
CopyGrid(G,geomG, R,geomR); // copy R to G;
assert(G.NumOfVertices() == R.NumOfVertices());
assert(G.NumOfCells()    == R.NumOfCells());
for(Complex2D::CellIterator c(G), ! c.IsDone(); ++c)
  cout << "Cell " << c.handle() << "  "
       << "has "  << (*c).NumOfVertices() << " vertices";
```

The implementation

The implementation of variant [2] is shown below.

```
template<class G2, class VertexMap, class CellMap>
void Construct0(Complex2D     & CC,
                G2       const& Gsrc,
```

```
                VertexMap     & VertexCorr, // maps src::handles to
                CellMap       & CellCorr)   // complex2d::handles

{
  typedef grid_types<G2> source_gt;

  typedef typename source_gt::cell_handle          src_cell_handle;
  typedef typename source_gt::CellIterator         src_cell_it;
  typedef typename source_gt::VertexIterator       src_vertex_it;
  typedef typename source_gt::VertexOnCellIterator src_vertex_on_cell_it;

  typedef typename Complex2D::Cell          Cell;
  typedef typename Complex2D::cell_handle   cell_handle;
  typedef typename Complex2D::vertex_handle vertex_handle;

  CC.clear();

  //---  construct vertices ---------------------------

  for(src_vertex_it src_v = Connect.FirstVertex(); ! src_v.IsDone(); ++src_v) {
    VertexCorr[src_v.handle()] = CC._new_vertex();
  }

  //---- construct cells -----------------------------

  // use vertex mapping in VertexCorr to find the vertices
  // incident to newly created cells.
  copy_cells(CC,Gsrc,VertexCorr,CellCorr);

  //---- construct additional adjaceny information ----------------------

  CC.calculate_vertex_cells();    // unordered list of adjacent cells per vertex
  CC.calculate_neighbour_cells(); // calls the generic Cell Neighbor Search algorithm

}
```

Appendix D

The Benchmark Codes

D.1 Code for the vertex-cell incidence benchmark

This section presents the concrete loops utilized in the benchmarks of section 6.5.2. These loop kernels where all executed a number of times indicated in the benchmark result figures. Code for the other benchmarks is available on request.

The following variables and types occur in the code:

```
Complex2D CC;
Triang2D  T2D;

typedef grid_types<Complex2D>  gt;
typedef grid_types<Triang2D>   tgt;

grid_function<gt ::Vertex,int> NumCells(CC,0);
grid_function<tgt::Vertex,int> NumCellsT(T2D,0);
```

Variable `ntri` is the number of triangles in the grid, `til` is an integer array of size `3*ntri` containing the vertices of each cell.

`Complex2D`:

```
for(gt::CellIterator c = CC.FirstCell(); ! c.IsDone(); ++c)
  for(gt::VertexOnCellIterator vc = (*c).FirstVertex();  !vc.IsDone(); ++vc) {
    NumCells[*vc]++;
  }
```

`Complex2D` (low level):

```
for(int c = 0; c < ntri; ++c)
  for(int vc = 0; vc < 3; ++vc)
    num_cells[ CC._cells[c]._vertices[vc] ]++;
```

`Triang2D`:

```
for(tgt::CellIterator c(T2D); c; ++c)
  for(tgt::VertexOnCellIterator vc(c);  vc; ++vc) {
    NumCellsT[vc]++;
  }
```

`Triang2D` (unrolled):

```
for(tgt::CellIterator c(T2D); c; ++c) {
  tgt::VertexOnCellIterator vc(c);
  NumCellsT[vc]++;
  ++vc;
  NumCellsT[vc]++;
  ++vc;
  NumCellsT[vc]++;
}
```

`Triang2D` (ForEach):

```
IncrGF<grid_function<tgt::Vertex, int> > inc(NumCellsT);
for(tgt::CellIterator c(T2D); c; ++c) {
  ForAllVertices(c,inc);
}
```

The classes `ForAllVertices` and `IncrGF` are defined as

```
template<class F>
inline void ForAllVertices(Triang2D_Cell const& c, F & f)
{
  typedef grid_types<Triang2D> tgt;
  tgt::VertexOnCellIterator vc(c);
  f(vc);
  ++vc;
  f(vc);
  ++vc;
  f(vc);
}

template<class GF>
struct IncrGF {
private:
  typedef grid_types<Triang2D>   tgt;
  GF * gf;
public:
  IncrGF(GF & f) : gf(&f) {}
  void operator()(tgt::VertexOnCellIterator const& v) { ++((*gf)[v]);}
};
```

C array:

```
for(int c = 0; c < ntri; ++c)
  for(int vc = 0; vc < 3; ++vc)
    num_cells[ til[3*c + vc] ]++;
```

C array (unrolled):

```
for(int c = 0; c < ntri; ++c) {
  num_cells[ til[3*c]    ]++;
  num_cells[ til[3*c] +1 ]++;
  num_cells[ til[3*c] +2 ]++;
}
```

FORTRAN:

```
       DO 20 c = 1,ntri
          DO 10 vc = 1,3
             numcells( til(vc,c)) = numcells( til(vc,c)) + 1
10        CONTINUE
20     CONTINUE
```

D.2 Code for the facet normal benchmark

This section presents the concrete loops utilized in the benchmarks of section 6.5.4. The class for the `Triang2D` geometry is defined as follows:

```
template<class P>
class geom_triang2d {
public:
    typedef grid_types<Triang2D> gt;
    typedef P                    coord_type;
    typedef point_traits<P>      pt;
private:
    grid_function<Vertex,P> coords_;
    const Triang2D * const g;
    int *  cells;
public:
    geom_triang2d(Triang2D const& T)
        :  coords_(T),
        g(&T), cells(T.cells) {}

   // two implementations shown below
   inline P   outer_area_normal(FacetOnCellIterator const& fc) const;
};
```

The low-level version of the normal calculation (0,1) is defined as follows:

```
P   outer_area_normal(FacetOnCellIterator const& fc) const
{
  int v1 = cells[3*fc.c.c+fc.fc];
  int v2 = cells[3*fc.c.c+ (fc.fc)%3];
  return coord_type(pt::y(coords_[v2]) - pt::y(coords_[v1]),
                    pt::x(coords_[v1]) - pt::x(coords_[v2]));
}
```

Here is the corresponding higher-level version (2,3):

```
P  outer_area_normal(FacetOnCellIterator const& fc) const
{
   return coord_type(pt::y(coords_[fc.V2()]) - pt::y(coords_[fc.V1()]),
                     pt::x(coords_[fc.V1()]) - pt::x(coords_[fc.V2()]));
}
```

The measured loops (0-3) where:

```
for(int N = 0; N < nloop; ++N)
  for(tgt::CellIterator c(T2D); ! c.IsDone(); ++c)
    for(tgt::FacetOnCellIterator fc(*c); ! fc.IsDone(); ++fc)
       normal += GeomT2D.outer_area_normal(fc);
```

We used the following shortcut for the `Complex2D` loops:

```
#define  V1  CC.vertices[CC.cells[fc.C.pos].vertices[fc.lf         ]].coord
#define  V2  CC.vertices[CC.cells[fc.C.pos].vertices[(fc.lf+1) % nf]].coord
```

The loop for `Complex2D` (low level, 4):

```
for(int N = 0; N < nloop; ++N)
  for(gt::CellIterator c = CC.FirstCell(); ! c.IsDone(); ++c)
    for(gt::FacetOnCellIterator fc = (*c).FirstFacet(); ! fc.IsDone(); ++fc) {
      int nf = (*c).NumOfVertices();
      coord_type const& v1 = V1;
      coord_type const& v2 = V2;
      normal += coord_type(pt::y(v1) - pt::y(v2), pt::x(v2) - pt::x(v1));
    }
```

The loop for `Complex2D` (lower level, 5,6):

```
for(int N = 0; N < nloop; ++N)
  for(gt::CellIterator c = CC.FirstCell(); ! c.IsDone(); ++c)
    for(gt::FacetOnCellIterator fc = (*c).FirstFacet(); ! fc.IsDone(); ++fc) {
      int nf = (*c).NumOfVertices();
       normal += coord_type(pt::y(V1) - pt::y(V2), pt::x(V2) - pt::x(V1));
    }
```

The loop for the array/`Complex2D` combination (7), using an array `ll_geom` for vertex coordinates:

```
for(int N = 0; N < nloop; ++N)
  for(int c = 0; c < nc; ++c) {
    int nvc = NumVC[c]; // number of vertices of c
    for(int vc = 0; vc < nvc; ++vc) {
      int v1 = CC.cells[c].vertices[vc          ];
      int v2 = CC.cells[c].vertices[(vc+1) % nvc];
      normal += coord_type(pt::y(ll_geom[v1]) - pt::y(ll_geom[v2]),
                           pt::x(ll_geom[v2]) - pt::x(ll_geom[v1]));
    }
  }
```

The loop for array/array (8), using an array `connect` for the cell-vertex incidence relationship (note that it is only correct if all cells are triangles):

```
for(int N = 0; N < nloop; ++N)
  for(int c = 0; c < nc; ++c) {
    int nvc = NumVC[c]; // number of vertices of c
    for(int vc = 0; vc < nvc; ++vc) {
      int v1 = connect[3*c+vc];
      int v2 = connect[3*c+((vc+1)%nvc)];
      normal += coord_type(pt::y(ll_geom[v1]) - pt::y(ll_geom[v2]),
                           pt::x(ll_geom[v2]) - pt::x(ll_geom[v1]));
    }
  }
```

The same, but exploiting explicit knowledge on the constant number of vertices of a triangle (9,10):

```
for(int N = 0; N < nloop; ++N)
  for(int c = 0; c < nc; ++c) {
    for(int vc = 0; vc < 3; ++vc) {
      int v1 = connect[3*c+vc];
      int v2 = connect[3*c+((vc+1)%nvc)];
      normal += coord_type(pt::y(ll_geom[v1]) - pt::y(ll_geom[v2]),
                           pt::x(ll_geom[v2]) - pt::x(ll_geom[v1]));
    }
  }
```

The Fortran77 implementation (11):

```
      SUBROUTINE LOOPF(nloop,ntri, til, nv, geom, nx, ny)
      IMPLICIT NONE
      INTEGER nloop, ntri, nv
      INTEGER til(1:3,1:ntri)
      DOUBLE PRECISION  geom(1:2,0:nv-1)
      DOUBLE PRECISION nx, ny
      INTEGER c, vc, vc1, N
      DO 30 N = 1, nloop
         nx = 0.0
         ny = 0.0
         DO 20 c = 1,ntri
            DO 10 vc = 1, 3
               vc1 = MOD(vc+1,3)
               nx = nx + (geom(2,til(vc1,c)) - geom(2,til(vc ,c)))
               ny = ny + (geom(1,til(vc ,c)) - geom(1,til(vc1,c)))
 10         CONTINUE
 20      CONTINUE
 30   CONTINUE
      RETURN
      END
```

The C implementation (12):

```
void loopC(int nloop, int nc, int* til, int nv, double* geom, double* nx, double* ny)
{
    int N, c, vc;
    *ny = 0.0;
    *nx = 0.0;
    for(N = 0; N < nloop; ++N) {
        for(c = 0; c < nc; ++c) {
            for(vc = 0; vc < 3; ++vc) {
                int vc1 = (vc+1)%3;
                *nx += (geom[2*til[3*c+vc1] + 1] - geom[2*til[3*c+vc ] + 1]);
                *ny += (geom[2*til[3*c+vc ]    ] - geom[2*til[3*c+vc1]    ]);
            }
        }
    }
}
```

List of Figures

List of Algorithms

List of Tables

Symbol Table

General

$\mathbb{D}^n$	the $n-$dimensional closed unit ball $\{x \in \mathbb{R}^n \mid \|x\| \leq 1\}$	46
$\mathbb{S}^n$	the $n-$dimensional unit sphere $\{x \in \mathbb{R}^{n+1} \mid \|x\| = 1\}$	47
$\mathbb{H}^n$	the closed halfspace $\{x \in \mathbb{R}^n \| x_n \geq 0\}$	49
$\mathbb{RP}^2$	two-dimensional projective space	54

Complexes and Grids

$\mathcal{C}$	a complex (finite CW-complex)	46
$\|\mathcal{C}\|$	the underlying space of a complex	46
d	the dimension of a grid	47
$\mathcal{C}^k$	set of k-elements of complex $\mathcal{C}$	49
$\mathcal{C}^{\leq k}$	k-skeleton of $\mathcal{C}$, i. e. the set $\bigcup_{l \leq k} \mathcal{C}^l$	49
$\mathcal{G}$	a grid	
$\mathcal{V}(\mathcal{G})$, $\mathcal{E}(\mathcal{G})$, $\mathcal{C}(\mathcal{G})$	vertex (edge, cell) set of grid $\mathcal{G}$, $\mathcal{V}(\mathcal{G}) = \mathcal{G}^0$, $\mathcal{E}(\mathcal{G}) = \mathcal{G}^1$, $\mathcal{C}(\mathcal{G}) = \mathcal{G}^d$	100
$\mathcal{R}$	grid sub-range	97
$\mathcal{A}(\mathcal{C})$	abstract (combinatorial) complex of $\mathcal{C}$	54
e, f	elements of a complex (grid)	
c	a cell of a grid (d-dimensional element)	
v	a vertex of a grid (0-dimensional element)	
$\bar{f}$	closure of element f	47
$e < f$	e is a side of f	47
$e \lesseqgtr f$	e and f are incident	47
$f \prec c$	f side of c with $\dim f = \dim c - 1$	47
$\mathrm{st}(e)$	the open star of grid element e	48
$\mathrm{lnk}(e)$	the link of grid element e	48
$\mathcal{I}_k(f)$	the set of k-elements incident to f	47
$\mathcal{I}_{j \to k}$	$j > k$: downward incidence sequence	89
	$j < k$: upward incidence sequence	89

$\mathrm{seq}(A)$	the set of finite sequences over a set A	
$\chi(X)$	the Euler characteristic of space X	55

Γ	geometric realization of a complex	56
$\\|\mathcal{A}\\|_\Gamma$	image of abstract complex $\mathcal{A}$ under Γ	56
$\mathfrak{a}_c$	archetype of cell c	56
Φ_c	characteristic mapping for cell c	56

Posets and lattices

$\mathcal{S}$	poset	52
$\hat{0}$,$\hat{1}$	minimal / maximal element of poset	53
$[a,b]$	in interval in poset, $[a,b] = \{c \in \mathcal{S} \vert a \leq c \leq b\}$	53
$a \wedge b$	the meet of a and b in a lattice	53
$a \vee b$	the join of a and b in a lattice	53
$e_k^{\Diamond}$	in lattice interval $e_{k-1} \prec e_k \prec e_{k+1}$: The other element in $[e_{k-1}, e_{k+1}]$	91

Distributed grids

$\mathcal{G}_i$	local grid part	117
Φ_{ij}	correspondance map on $\mathcal{O}_{ij}$	117
$e^i \sim e^j$	correspondence relation on distributed grid	117
$\widehat{e}$	equivalence classes (global element)	117
$\widehat{\mathcal{G}}$	the global grid of a distributed grid, $\widehat{\mathcal{G}} = \left\{\widehat{e} \,\middle\vert\, e \in \bigcup_{i=1}^N \mathcal{G}_i\right\}$	118
Γ_i	local grid geometry	118
$\widehat{\Gamma}$	global grid geometry	118
N	the number of parts of a distributed grid	

$\mathcal{O}_{ij}$	bilateral overlap	117
$\mathcal{E}_{ij}$, $\mathcal{S}_{ij}$, $\mathcal{C}_{ij}$	bilateral exposed (shared, copied) overlap range	119
$\mathcal{O}_i$	total overlap range, $\mathcal{O}_i = \bigcup_{j=1}^n \mathcal{O}_{ij}$	119
$\mathcal{E}_i$	total exposed overlap range, $\mathcal{E}_i = \bigcup_{j=1}^n \mathcal{E}_{ij} \setminus (\mathcal{S}_i \cup \mathcal{C}_i)$	119
$\mathcal{S}_i$	total shared overlap range, $\mathcal{S}_i = \bigcup_{j=1}^n \mathcal{S}_{ij} \setminus \mathcal{C}_i$	119
$\mathcal{C}_i$	total copied overlap range, $\mathcal{C}_i = \bigcup_{j=1}^n \mathcal{C}_{ij}$	119
$\mathcal{P}_i$	private overlap range, $\mathcal{P}_i = \mathcal{G}_i \setminus \mathcal{O}_i$	120
$\mathcal{L}_i$	local overlap range, $\mathcal{L}_i = \mathcal{P}_i \cup \mathcal{E}_i \cup \mathcal{S}_i$	120
$\mathcal{X}_i$	exposed overlap range, $\mathcal{X}_i = \mathcal{S}_i \cup \mathcal{E}_i$	120
$\mathcal{W}_i$	owned overlap range, $\mathcal{W}_i = \mathcal{P}_i \cup \mathcal{E}_i$	120

$\mathfrak{G}$	quotient grid, short for $\mathcal{G}/_P$	121
$\mathcal{G}/_P$	quotient of grid $\mathcal{G}$ and partitioning P	121

$\sim_k$	equivalence relation for k-elements of quotient	121
$\mathfrak{e}$	quotient element corr. to grid element e, equivalence class $[e]$ of e with respect to $\sim_k$	121
I	incidence sequence (stencil)	123
$\lvert I \rvert$	length of stencil I	125
$\mathcal{L}_{(i,j)}(\mathcal{K})$	incidence layer	123
$\mathcal{H}_I(\mathcal{K})$	incidence hull of stencil I and germ $\mathcal{K}$	123
$\mathcal{H}_I^k(\mathcal{K})$	partial incidence hull of stencil I and germ $\mathcal{K}$	123

Index

Page numbers in **bold face** point to a definition.
Glossary entries are marked ≪*like this* ≫.

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

U

V

W

X

Z

Curriculum Vitae

Personal data

Name	Guntram Berti
Place of birth	Dortmund
Date of birth	December 8, 1967
Nationality	german
Marital status	single

Education

Primary school: Westricher Grundschule, Dortmund	1974 – 1978
Secondary school: Bert-Brecht-Gymnasium, Dortmund	1978 – 1987
Abitur	June 1987
Universität Dortmund	1987 – 1990, 1991 – 1994
Vordiplom Mathematik	1989
Diplom Mathematik	1994
Université François Rabelais, Tours	1990-1991
Maîtrise de mathématiques	1991

Employment

Scientific assistant, BTU Cottbus	since march 1995