gridlib: Flexible and Efficient Grid Management for Simulation and Visualization*

Frank Hülsemann[1], Peter Kipfer[2], Ulrich Rüde[1], and Günther Greiner[2]

[1] System Simulation Group of the Computer Science Department,
Friedrich-Alexander University Erlangen-Nuremberg, Germany,
frank.huelsemann@cs.fau.de,
[2] Computer Graphics Group of the Computer Science Department,
Friedrich-Alexander University Erlangen-Nuremberg, Germany,
kipfer@cs.fau.de

Abstract. This paper describes the *gridlib* project, a unified grid management framework for simulation and visualization. Both, adaptive PDE-solvers and interactive visualization toolkits, have to manage dynamic grids. The *gridlib* meets the similar but not identical demands on grid management from the two sides, visualization and simulation. One immediate advantage of working on a common grid is the fact that the visualization has direct access to the simulation results, which eliminates the need for any form of data conversion. Furthermore, the *gridlib* provides support for unstructured grids, the re-use of existing solvers, the appropriate use of hardware in the visualization pipeline, grid adaptation and hierarchical hybrid grids. The present paper shows how these features have been included in the *gridlib* design to combine run-time efficiency with the flexibility necessary to ensure wide applicability. The functionality provided the *gridlib* helps to speed up program development for simulation and visualization alike.

1 Introduction

This article gives an overview of the *gridlib*[1] grid management project, its aims and the corresponding design choices [5], [6], [7], [8]. The *gridlib* combines grid manipulation requirements of mesh based PDE-solvers and visualization techniques into one single framework (library). Thus it offers developers of simulation programs a common interface for the computational and graphical parts of a project.

For interactive computer graphics, the efficient manipulation of grids and the data attached has always been important. In the numerical PDE community, it is the development of adaptive h-refinement algorithms in several space dimensions that led to recognising grid management as a worthwhile task in its own right.

* This project is funded by a KONWIHR grant of the Bavarian High Performance Computing Initiative.

[1] This is a temporary name. Choices for the final name of the whole project are currently being considered.

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2331, pp. 652–661, 2002.

Despite the shared need to administer dynamically changing grids, there seems to be little joint work. Many PDE-packages, such as deal.II[2] or Overture[3] for example, include tools for the visualization of the results. However, these graphics components are usually tied to the solver part of the package and as such, they are too specific to be widely applicable. Likewise, although the numerous visualization libraries available, such as AVS[4] or VTK[5] for example, obviously display gridded data, they delegate the work of integrating the visualization into the solver to the solver developers. This assumes that an integration is possible at all, which is not obvious, given that some toolkits modify the submitted data for optimisation purposes.

The *gridlib* is a joint effort of three groups to exploit the synergy offered by one common grid management. The development is shared mainly between a visualization- and a simulation group, while the third, from computational fluid dynamics, provides valuable input from the users' perspective. Although the overall task of grid management is shared, the two communities, simulation and visualization, put different emphasis on the features of a grid administration software. The high performance computing community has demonstrated time and again that it is willing to put runtime efficiency (as measured in MFLOPS) above all other considerations. Visualization is a much more interactive process, which has to be able to respond to the choices of a user with very low latency. Consequently, visualization requirements result in higher emphasis on flexibility than is the norm (traditionally) in the HPC context, willing to trade CPU performance and memory usage for interactivity. This paper shows how the *gridlib* meets the demands from both sides. After an overview of the *gridlib* system architecture in Sect. 2, the topic of flexibility is discussed in Sect. 3. This is followed by the efficiency considerations in Sect. 4, before the main points of the paper are summed up in the conclusion in Sect. 5.

2 System Architecture of the gridlib

The *gridlib* is a framework library for the integration of simulation and visualization on adaptive, unstructured grids. Its infrastructure serves two main purposes. First, it supports developers of new simulation applications by providing subsystems for I/O, grid administration and grid modification, visualization and solver integration. Second, its parametrised storage classes allow (in principle) the re-use of any existing solvers, even those only available in binary format. For the special group of solvers that do not perform grid management themselves, the *gridlib* can provide plug-and-play functionality.

This high level of integration is achieved by three abstraction levels:

[2] deal.II homepage: `http://gaia.iwr.uni-heidelberg.de/~deal/`
[3] Overture homepage: `http://www.llnl.gov/CASC/Overture/overview.html`
[4] AVS homepage: `http://www.avs.com`
[5] VTK homepage: `http://public.kitware.com/VTK`

1. The lowest level offers an interface to describe the storage layout. This is the part of the library that has to be adapted when integrating an existing solver.
2. The level above implements abstraction of the geometric element type. Relying on the storage abstraction, it provides object oriented element implementations for the higher abstraction levels.
3. The highest level offers the interface to operations on the whole grid. It employs object oriented design patterns like functors for frequently needed operations.

3 Flexibility

The *gridlib* intends to be widely applicable. From a simulation perspective, this implies that the user should be able to choose the grid type and the solver that are appropriate for the application. For the visualization tasks, the *gridlib* must not assume the existence of any dedicated graphics hardware. However, if dedicated hardware like a visualization server is available, the user should be able to decide whether to use it or not. The following subsections illustrate how these aims have been achieved in the *gridlib* design.

3.1 Unstructured Grids

The scientific community remains divided as to what type of grid to use when solving PDEs. As a consequence, there are numerous different grid types around, ranging from (block-)structured over hybrid up to unstructured grids, each of them with their advantages and problems and their proponents. A grid software that intends to be widely applicable cannot exclude any of these grid types. Thus, the *gridlib* supports completely unstructured grids[6], which include all other more specialised grid types. Furthermore, the *gridlib* does not make any assumptions about the mesh topology nor the geometrical shape of the elements involved. Currently supported are tetrahedra, prisms, pyramids, octahedra and hexahedra. The *gridlib* is designed in such a way that other shapes can be added easily using object oriented techniques.

3.2 Integrating Existing Solvers

As mentioned before, the *gridlib* supports the re-use of existing solvers, even those only available in binary form. To this effect, the *gridlib* provides the grid data in the format required by a given solver. For example, this could imply storing the grid data in a particular data file format or arranging certain arrays in main memory to be passed as arguments in a function call. Clearly, for this approach to work, the input and output formats of the solver have to be known. In this case, the integration involves the following steps:

[6] One repeated argument against the use of unstructured grids in the scientific computing community is their alleged performance disadvantage. We will return to this point in Sect. 4.2.

1. Implementation of the storage format for the element abstraction.
2. Creation of an object oriented interface, which can be inherited from a provided, virtual interface. This step effectively "wraps" a potentially procedural solver into an object oriented environment.
3. Link the solver together with the *gridlib*.

Note that in many cases, the second step can be performed automatically by the compiler through the object-oriented template patterns already provided by the *gridlib*. If the source code of the solver can be modified, the first two steps can be combined, which results in the native *gridlib* storage format to be used throughout.

3.3 Visualization Pipeline

In the *gridlib*, the visualization is based on a attributed triangle mesh which in turn is derived from the original data or a reduced set of it. By working directly on the grid data as provided by the grid administration component of the library, the visualization subsystem can exploit grid hierarchies, topological and geometrical features of the grid and the algorithms for grid manipulation. This approach provides a common foundation for all visualization methods and ensures the re-usability of the algorithmic components.

In the visualization pipeline, the data is represented in the following formats:

1. As simulation results on the compute grid
2. As data on a modified grid (reduced, progressive, changed element types, ...)
3. As visualization geometries (isosurfaces, stream lines, ...)
4. As bitmap or video (stored in a file or displayed immediately)

These stages can be distributed across several machines. In the context of large scale simulations, a common distribution of tasks involves a *compute node* for the first step, a *visualization server* for the second and third, and lastly, the user's workstation for the forth. For a given project, these three functions, compute server, visualization server and front end workstation, have to be assigned to the available hardware. The *gridlib* makes provisions for different configurations that help the user to adequately match the given hardware to the tasks. The following factors influence the visualization pipeline:

1. Availability and performance of an interactive mode on the compute node. This is often an issue on batch-operated super computers.
2. Bandwidth and latency of the involved networks.
3. Availability and performance of a dedicated visualization server.
4. Storage capacity and (graphics-) performance of the front end workstation.

Given the concrete configuration, it is the user who can decide how to trade-off requirements for interaction with those for visualization quality. Conceptually, the *gridlib* supports different scenarios:

- Remote rendering on the compute node. Being based on the complete set of high resolution simulation results, this approach yields the maximum visualization quality. However, on batch-operated machines, no form of interaction is possible.
- Postprocessing of the simulation results on the compute node and subsequent transfer of a reduced data set to the visualization server or front end. Once the simulation results are available, this strategy offers the maximum of interaction in displaying the results but places high demands the servers and the networks, as even reduced data sets can still be large in absolute terms.
- Local rendering of remotely generated visualization geometries. The user experiences (subjective) fast response times but can only work on a given number of data sets. This approach allows high visualization quality but requires fast networks and high storage facilities.
- Two stage rendering. First, the user determines the visualization parameters (view point, cut plane, ...) on a reduced quality set, then transfers these parameters to the compute node, where they will be used for remote rendering at maximum quality.

Supporting all these scenarios is ongoing work. Several components have already been implemented. Progressive mesh techniques allow to trade visualization quality for faster response time (resolution on demand), see [5]. Slice- and isosurfaces geometries can be computed and displayed via various rendering options, see [8]. The most generally applicable renderer is a software-only implementation, which is useful on machines without dedicated graphics hardware. It can be run transparently in parallel on any multiprocessor machine with MPI support. Figure 1 illustrates the data flow for the parallel software renderer. The alternative is tuned for hardware accelerated OpenGL environments. Thus the *gridlib* lets the user choose a compromise between visualization quality and interaction.

4 Efficiency

This section introduces the two main features of the *gridlib* that are useful in the development of high performance solvers, for which maximum runtime efficiency is important. These two features are the provision of grid adaptation techniques and the concept of hierarchical hybrid grids.

4.1 Grid Adaptation

Adaptive h-refinement techniques, usually based on error estimators, have attracted considerable interest in the numerical PDE community over the last twenty years, see, for instance, [2], [1]. For many applications, these techniques are well-established and reliable error-estimators are available [1], [9], [3]. By providing functions for the uniform, adaptive or progressive subdivision and coarsening of the mesh, the *gridlib* is a well-suited platform for the implementation of

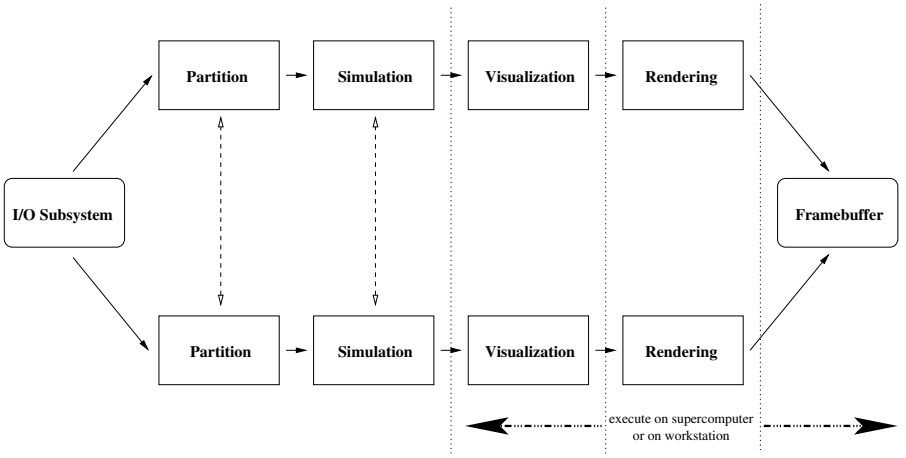

Fig. 1. Data flow for the parallel software renderer: The renderer processes the distributed simulation results concurrently before merging the individual parts together into the final picuture. The diagram emphasises the various stages in the visualization pipeline that can be assigned to the available hardware.

h-refinement algorithms. The user need only specify a criterion that marks the grid cells to be subdivided. The *gridlib*'s refinement algorithm performs the subdivision and ensures that the resulting grid is consistent and that hanging nodes are avoided (red-green refinement). For subdividing tetrahedra, the algorithm of Bey [4] has been chosen because of the small number of congruency classes it generates.

Provided that the user contributes a sharp error estimator, the *gridlib* features make it easy to generate solution adapted unstructured grids. Such grids are the essential tool to improve the accuracy of the solution for a given number of grid cells.

4.2 Efficiency Limits of Unstructured Grids and What to Do about It

It is important to note that adaptive refinement of unstructured grids (alone) cannot overcome the problem of low MFLOPS performance when compared to (block-)structured approaches.

The performance problem of solvers on unstructured grids results from the fact that the connectivity information is not available at compile time. Hence the resulting program, although very flexible, requires some form of book-keeping at run time. In structured codes, the connectivity is known at compile time and can be exploited to express neighbourhood relations through simple index arithmetic.

The following, deliberately simple example illustrates the difference between the two approaches. Given the unit square, which is discretised into square cells of side length h using bi-linear elements. An unstructured solver "does not see"

the regularity of the grid and hence has to store the connectivity data explicitly. In pseudo code, an unstructured implementation of a Gauss-Seidel step with variable coefficients in the unstructured solver reads as follows:

```
for i from first vertex to last vertex:
  rhs = f(i)
  for j from 1 to number_of_neighbours(i)
    rhs = rhs - coeff(i,j)*u(neighbour(i,j))
  u(i) = rhs/coeff(i,i)
```

Contrast this to a structured implementation (assuming that this ordering of the for-loops is appropriate for the programming language):

```
for i from first column to last column:
  for j from first row to last row:
    u(i,j)=(f(i,j)-c(i,j,1)*u(i-1,j-1)-c(i,j,2)*u(i-1,j)
                  -c(i,j,3)*u(i-1,j+1)-c(i,j,4)*u(i+1,j-1)
                  -c(i,j,5)*u(i+1,j)  -c(i,j,6)*u(i+1,j+1)
                  -c(i,j,7)*u(i,j+1)  -c(i,j,8)*u(i,j-1))/c(i,i)
```

The work as measured in floating point operations is the same in both implementations, but their run-time performance differs significantly as the second version, being much more explicit, lends itself much better to compiler optimisation than the first one. On one node (8 CPUs) of a Hitachi SR8000 at the Leibniz Computing Centre in Munich, the MFLOPS rate of the (straightforward) structured version is a factor of 20 higher than the one of the similarly straightforwardly implemented unstructured algorithm.

The *gridlib* introduces the concept of hierarchical hybrid grids to overcome the performance penalty usually associated with unstructured grids while retaining their geometric flexibility.

The main idea behind the hierarchical hybrid grids is to deal with geometric flexibility and computing performance on different grid levels. The coarse grid levels are in general unstructured and ensure the geometric flexibility of the approach. The coarse grids are nested in the sense that the finer ones are generated through uniform or adaptive refinement from the coarser ones. The finest unstructured grid is assumed to resolve the problem domain adequately and is therefore referred to as the geometry grid. The fine grids, on which the computations are to be carried out, are generated through regularly subdividing the individual cells of the geometry grid. Figure 2 illustrates the concept.

As shown above, it is essential for high floating point performance that the implementation of the computing algorithms takes the regular structure of the compute grid within each cell of the geometry grid into account. Given that the compute grid is only patchwise regular, some fraction of the computations still require unstructured implementations. Obviously, the finer the compute grid, the more the overall floating point performance is dominated by the contribution from the structured parts.

The following discussion confirms this expectation for a vertex based algorithm like Gauss-Seidel. Let N_c be the number of vertices in the (unstructured)

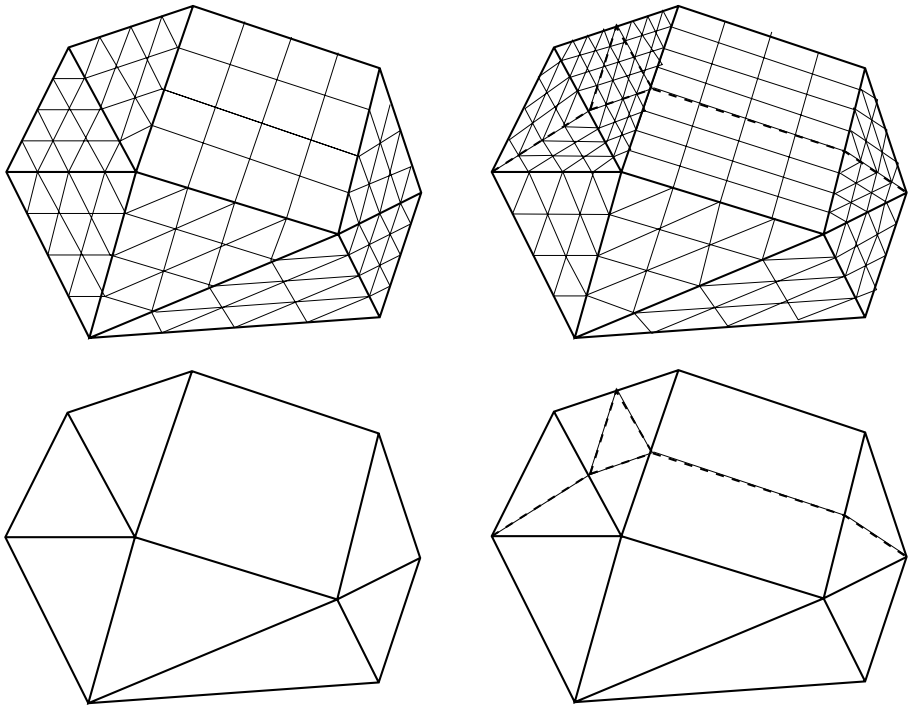

Fig. 2. Bottom left: coarsest base grid, bottom right: geometry grid after one unstructured refinement step, top row: compute grids after two regular subdivision steps of the respective coarse grids below

geometry grid and N_f be the number of vertices in the structured refinements. The unstructured algorithm achieves M_c MFLOPS while the structured part runs at M_f MFLOPS. Under the assumption that N_{op}, the number of floating point operations per vertex, is the same for both grid types (as it was in the Gauss-Seidel example above), then the execution time of one Gauss-Seidel iteration over the compute grid is given by

$$\frac{N_c \times N_{op}}{M_c} + \frac{N_f \times N_{op}}{M_f}. \tag{1}$$

Dividing the total number of operations, $N_{op} \times (N_c + N_f)$, by this execution time, one finds the MFLOPS value for the whole grid, M say, to be

$$M = \frac{(N_c + N_f) \times M_c M_f}{N_c M_f + N_f M_c}. \tag{2}$$

Introducing the fine to coarse ratio i

$$i = \frac{N_f}{N_c} \iff N_f = i \times N_c$$

and the speed-up factor s for structured implementations over unstructured ones

$$s = \frac{M_f}{M_c} \iff M_f = s \times M_c,$$

M is given by

$$M = \frac{s(i+1)}{s+i} M_c, \tag{3}$$

which for $i \to \infty$ tends to

$$\lim_{i \to \infty} M = \lim_{i \to \infty} \frac{s(i+1)}{s+i} M_c = sM_c = M_f. \tag{4}$$

In other words, provided the structured part is sufficiently large, the floating point performance on a hierarchical hybrid grid is dominated by its structured part, while retaining the geometric flexibility of its unstructured component.

The interface to the hierarchical hybrid grids is still under construction. However, as the experience from the Hitachi shows, the speed-up factor s can be as large as 20. This shows that the extra work of tuning the algorithm to the regularity of the grid inside the coarse grid cells is well worth the effort.

5 Conclusion

The paper presented the main features of the grid management framework *gridlib*. It combines the grid management requirements of both, visualization and simulation developers, into a single framework. By providing subsystems for frequently needed tasks in PDE solvers, such as I/O, adaptive grid refinement and, of course, visualization, the *gridlib* helps to speed up the development of such programs.

The article described the main features of the *gridlib* from the two perspectives of flexibility and (run-time) efficiency. Through its support of unstructured grids and numerous cell geometries, the *gridlib* is widely applicable. In case a particular cell geometry is not already included, the object-oriented design of the *gridlib* ensures that the user can add the required object easily. It was shown how existing solvers, that do not include any grid management, can be combined with the *gridlib*, so that these solvers, too, can benefit from the visualization facilities of the framework. For the visualization of large scale simulations, the *gridlib* supports different hardware scenarios, from which the user can choose to meet the project-specific requirements concerning visualization quality and interactivity. Its provision of algorithms for the consistent, adaptive subdivision of unstructured grids in three space dimensions makes the *gridlib* an ideal platform for implementing and experimenting with adaptive h-refinement methods. To close the gap in MFLOPS performance between unstructured and structured grids, the *gridlib* introduces the concept of hierarchical hybrid grids. This approach employs a hierarchy of two different grid types on the different levels to combine the advantages of the unstructured grids (geometric flexibility) with those of structured ones (high floating point performance). The coarse levels are

made up of nested, unstructured grids. The patchwise structured grids on the finer levels are constructed through repeated *regular* subdivision of the cells of the finest, unstructured grid. Adapting the algorithm to take the grid structure into account increased the floating point performance of a Gauss-Seidel iteration inside the patches on a Hitachi SR8000 by a factor of twenty. The promise of the approach is therefore evident. However, more work in specifying user interfaces for the hierarchical hybrid grids among other things has to be done.

6 Acknowledgements

The authors wish to thank Dr. Brenner from the fluid dynamics group of Erlangen University for helpful discussions, U. Labsik, G. Soza from the Computer Graphics Group at Erlangen University for their input concerning geometric modelling and mesh adaptivity, S. Meinlschmidt from the same group for his work on the various options in the visualization pipeline, M. Kowarschik from the System Simulation Group, also at Erlangen University, for his insights in exploiting grid regularity for iterative methods. As mentioned in the introduction, this project is funded by a KONWIHR grant of the Bavarian High Performance Computing Initiative, which provided the access to the Hitachi SR8000.

References

1. Ainsworth, M., Oden, J.T.: A posteriori error estimation in finite element analysis. Comp. Methods Appl. Mech. Engrg. **142** (1997) 1–88
2. Babuska, I., Rheinboldt, W. C.: Error estimates for adaptive finite element computations. SIAM J. Numer. Anal. **15** (1978), 736–754
3. Becker, R., Rannacher, R.: Weighted A posteriori error control in FE methods. IWR Preprint 96-1, Heidelberg, 1996
4. Bey, J.: Tetrahedral Grid Refinement. Computing **55** (1995), 355–378
5. Labsik, U., Kipfer, P., Meinlschmidt, S., Greiner, G.: Progressive Isosurface Extraction from Tetrahedral Meshes. Pacific Graphics 2001, Tokio, 2001
6. Labsik, U., Kipfer, P., Greiner, G.: Visalizing the Structure and Quality Properties of Tetrahedral Meshes. Technical Report 2/00, Computer Graphics Group, University Erlangen, 2000
7. Greiner, G., Kipfer, P., Labsik, U., Tremel, U.: An Object Oriented Approach for High Performance Simulation and Visualization on Adaptive Hybrid Grids. SIAM CSE Conference 2000, Washington, 2000
8. Kipfer, P., Greiner, G.: Parallel rendering within the integrated simulation and visualization framework “gridlib”. VMV 2001, Stuttgart, 2001
9. Süli, E.: A posteriori error analysis and adaptivity for finite element approximations of hyperbolic problems. In: Kröner, D., Ohlberger, M., Rohde C. (Eds.): An Introduction to Recent Developments in Theory and Numerics for Conservation Laws. Lecture Notes in Computational Science and Engineering **5**, 123–194 Springer-Verlag, 1998