PARALLEL UNSTRUCTURED GRID COMPUTATIONS

Peter Bastian[1], Klaus Birken[1], Klaus Johannsen[1], Stefan Lang[2],
Nikolas Neuß[2], Henrik Rentz–Reichert[1], Christian Wieners[1]

[1] Institut für Computeranwendungen III
Universität Stuttgart
Allmandring 5b, D-70569 Stuttgart

[2] Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg
Im Neuenheimer Feld 368, D-69120 Heidelberg

SUMMARY

The numerical solution of partial differential equations (PDEs) on unstructured grids using parallel computers leads to an increase in software complexity of several orders of magnitude when compared to a sequential, structured mesh code. Consequently, the design of simulation software with respect to code reuse over problem domains is of great importance.

In this paper we review the steps of the PDE solution process with respect to parallel computing and discuss the modular structure of the UG software toolbox. The high–level object–oriented design of the numerical algorithms is shown in some detail to give the reader an impression how new components can be incorporated into the UG framework.

INTRODUCTION

The numerical solution of partial differential equations involves a sequence of related steps starting with geometric modeling and ending with the visualization of the results as shown in Fig. 1. Arrows in the figure indicate the flow of control, links in gray are optional. Although the steps are the same for structured and unstructured grids as well as sequential and parallel computation, programming effort can vary from almost nothing to man–years, as e. g. in mesh generation.

In the following we comment each of the basic building blocks from Fig. 1:

Geometric Modeling. Holds a representation of the (three–dimensional) body in which the PDE is to be solved. Access methods include ways to find points in the interior, on

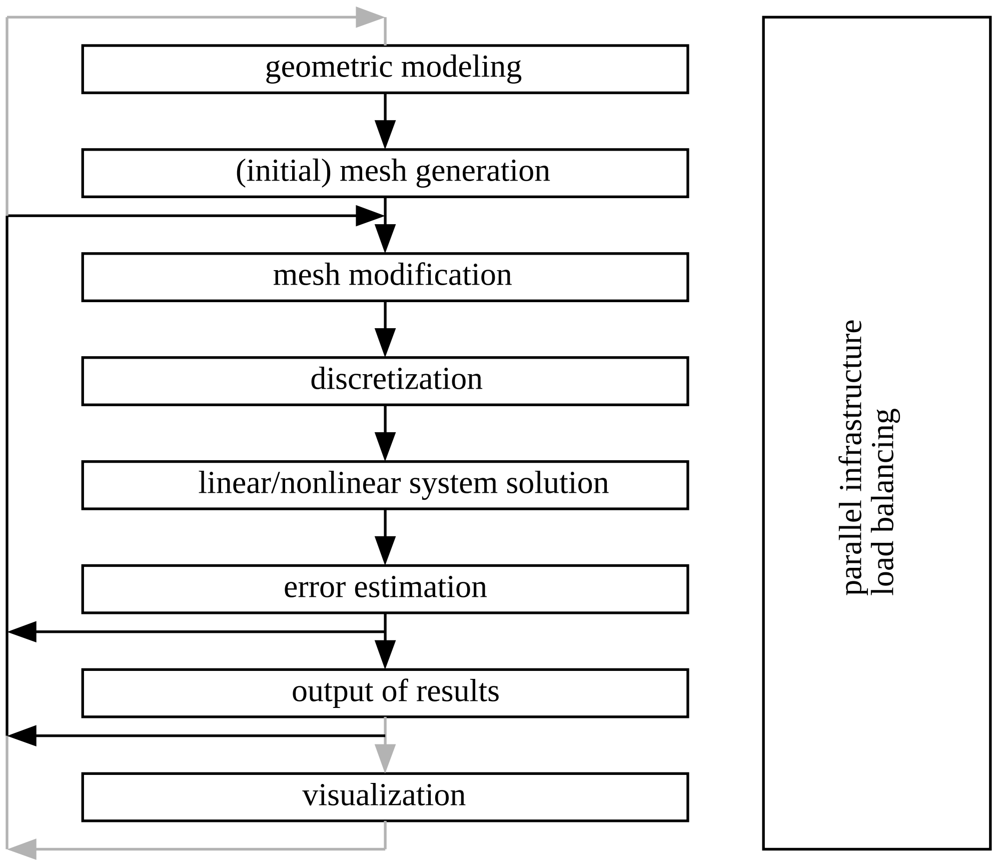

Figure 1: Basic building blocks of the PDE solution process.

(internal and external) surfaces and on manifolds where two or more surfaces intersect. In a parallel environment the geometric model might be duplicated on each processor if it is small enough, otherwise it has to be distributed together with the mesh data.

(Initial) Mesh Generation. Constructs a volume mesh approximating the domain given by the geometric model. Small details, e. g. a well or a tiny region of highly conductive material, must be resolved by the mesh if they are critical for the solution of the PDE. Other parameters to be controlled are mesh quality (angle condition), mesh size and anisotropy. In the parallel case load balancing/domain decomposition is notoriously difficult.

Mesh Modification. Given a mesh, construct a new mesh that is finer in some regions and possibly coarser in other regions of the domain without doing a complete remesh. The regions are indicated by the error estimator step. A very effective way to do this is the hierarchical approach where individual elements of the given mesh are subdivided according to certain rules. Coarsening is achieved by recombination of previously subdivided elements. This results in local operations and an efficient data–parallel implementation is possible, see [2], [11] or [9]. Mesh modification requires dynamic load redistribution in order to balance the load after the refinement step.

Discretization. Sets up a finite–dimensional approximation of the differential equation. Operations are typically trivially parallel on element level. Difficulties in load balance arise if amount of work is variable per element.

(Non-)Linear System Solution. Large systems in 3D are typically solved with iterative solvers. It is important to maintain a low iteration count independent of the size of the

mesh and the number of processors (and possibly other parameters). Multilevel and domain decomposition methods (often) have this property, see [15] for an introduction.

Error Estimation/Refinement Strategy. Determine how accurately the discrete solution approximates the differential equation. Provide information where the mesh has to be refined or coarsened. Operations are typically parallel on element level requiring at most access to data in neighboring elements.

Output of Results. Store geometry/mesh/solution information to a disk file for subsequent restart or visualization. Huge amounts of data are produced by parallel computations necessitating the use of clever file formats (suppress redundant information) and parallel file I/O.

Visualization. Although sequential visualization software can be improved to handle fairly large data sets (e. g. about five million nodes in GRAPE on a workstation with 1 GB of memory [13]), ultimately also the rendering process will have to be parallelized.

In order to avoid sequential bottlenecks all components that handle large amounts of data have to be parallelized. Furthermore, all components must work on the same distributed data structures (geometric model, unstructured mesh, matrices and vectors). At full scale this requires the incorporation of all components into an integrated environment. The interaction of the components and reuse of code for the different distributed data structures is simplified by providing a "parallel infrastructure" which is drawn as a vertical box in Fig. 1 since it is intended to support all components.

In the rest of this paper we will review the design of the UG (shorthand for *U*nstructures *G*rids) software which intends to provide such an integrated environment. The next section describes the modular structure, then we look at the parallel infrastructure part (DDD) and at the unstructured mesh data structure. Finally, we discuss the high–level object oriented interface to the numerical algorithms and give some conclusions.

UG MODULE STRUCTURE

The UG software is structured into several layers shown in Fig. 2. We will browse through the layers from bottom to top.

The Dynamic Distributed Data (DDD) layer provides the parallel infrastructure for creating and maintaining the distributed unstructured mesh data structure. It uses the Parallel Processor Interface (PPIF) for portability to many platforms. DDD is described in more detail below.

The next layer provides basic sequential functionality. The domain manager offers an abstract geometry interface to the grid manager. Two different implementations of this interface are available. The output devices module offers a portable graphics interface which is implemented for X11, postscript and other formats. The inter–module database is used by modules to exchange data with each other in a standardized way. Finally, the graph partitioner CHACO, see [8], has been included for use in the load balancing routines.

The grid manager module is responsible for creation and modification of the unstructured mesh data structure. Creation of initial meshes is done sequentially by 2D/3D

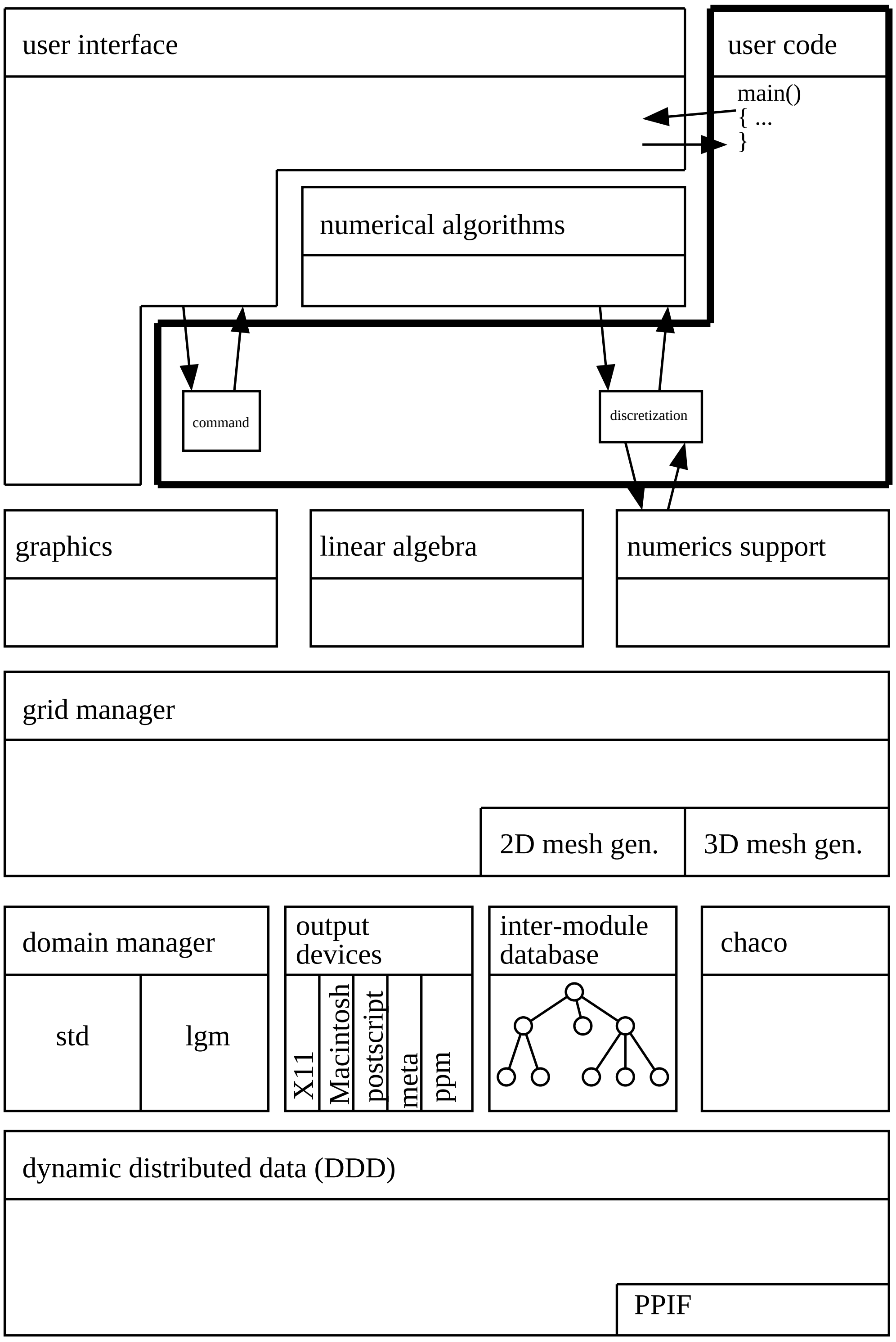

Figure 2: Modular structure of UG.

Diffusion Equation
- Linear conforming P1
- Quadratic conforming P2
- Linear non-conforming CR
- mixed RT0,RT1
- mixed BDM

Linear Elasticity
- Linear conforming P1
- Quadratic conforming P2
- Non-conforming (Falk)
- Stabilized BDM

Elastoplasticity
- Linear conforming P1
- Quadratic conforming P2

Biharmonic Equation
- Morley
- Argyris Element

Convection–Diffusion–Reaction Equations
- Finite–Volume

Stokes
- Taylor–Hood Element

Navier–Stokes
- Finite–Volume stabilized
- stationary–instationary
- compressible–incompressible
- laminar, turbulent ($k - \varepsilon$)

Density Driven Flow
- Finite–Volume

Multi–phase Flow
- Finite–Volume
- Global Pressure
- Transition Conditions
- Fractured Porous Media
- Multicomponent Flow

Figure 3: PDE problems and discretizations currently implemented in the UG toolbox.

advancing front mesh generators. The 3D mesh generator has been contributed by J. Schöberl, Linz, and is described in [14]. UG supports six different element types: Triangles, quadrilaterals, tetrahedra, pyramids, prisms and hexahedra. Degrees of freedom can be placed in vertices, edges, faces and elements of the mesh.

On top of the grid manager we have the graphics module enabling 2D and 3D visualization of meshes and solutions on planar cuts. Parallel 3D hidden surface removal is included, see [10]. Graphical output can be sent to any output device. The linear algebra module provides kernels for sparse matrix–vector operations and iterative solvers. Numerics support includes useful functionality for many finite volume and finite element discretizations.

The numerical algorithms module consists of a large variety of numerical methods such as linear and nonlinear solvers, time–stepping schemes. From the point of view of the application programmer UG provides a framework for building specialized simulator applications. The numerical algorithms are implemented in a set of classes which can be used directly or from which the application programmer can inherit in order to add new components or to replace existing ones. In the implementation of his new classes the programmer can use functionality offered by other UG modules (e. g. numerics support) in the traditional form of subroutine libraries. The object oriented design of the numerical algorithms is described in more detail below.

A large number of PDE problems have been solved using the UG framework. A partial list is given in Fig. 3. UG has been implemented in the C programming language. Most of its design follows the modular programming style, except for the numerical algorithms which have been designed with object oriented methods.

DYNAMIC DISTRIBUTED DATA

The DDD layer provides the parallel infrastructure to create and maintain the distributed unstructured mesh data structure as well as the distributed sparse matrices and vectors. The underlying idea of DDD is that an arbitrary data structure consisting of user defined data types referencing each other can be mapped to a directed graph where each node corresponds to an object (e. g. a vertex or an element) and each edge in the graph corresponds to a reference (pointer) from one object to another.

For the purpose of parallel processing we want to assign parts of the graph to different processors. Since we aim at distributed memory architectures an edge can only be stored by a processor if is also assigned the two corresponding nodes (no pointers to objects in another processor's memory are possible). In order for each edge to be stored in at least one processor some nodes have to be stored in several processors, resulting in an overlapping decomposition of the graph. Different forms of overlap are possible and are determined by the needs of the application. Nodes of the abstract graph that are stored on more than one processor are called "distributed objects" in DDD notation.

DDD allows the application to compose distributed objects from objects created seperately on each processor (a process called identification, used e. g. in mesh refinement) and it provides abstractions for efficient communication among the copies of a distributed object. The most powerful feature of DDD is its ability to dynamically migrate object copies from one processor to another while *automatically* updating the references to neighboring objects and the corresponding communication data structures.

DDD objects correspond to individual vertices or elements of the mesh data structure. All operations are designed to handle hundreds of thousands of objects per processor efficiently. Memory overhead is 12 bytes in each object and an additional 12 bytes for each remote copy of an object. DDD only stores information about local objects and copies of these local objects on other processors, no component of DDD has global information about all objects.

For more information about DDD we refer to the thesis of Klaus Birken [6].

HIERARCHICAL MESH DATA STRUCTURE

The central idea of the UG's approach to scalability is the use of a hierarchical mesh data structure. It is assumed that the geometry is simple enough to be duplicated on each processor and that a reasonable initial mesh can be constructed that is much coarser than the final mesh that is used to compute the solution of the differential equation. Thus it is possible to generate an initial mesh sequentially which is then distributed to (a subset of) the processors for (adaptive) refinement in parallel.

The mesh refinement procedure is an extension of the algorithms of [1] (2D, triangles) and [5] (3D, tetrahedra) to multiple element types. An efficient data–parallel implementation is enabled through a level–wise formulation (only elements of one grid level at a time are modified) and the use of a complete set of rules (there is a refinement rule for any possible refinement of edges and faces of an element), see [2] and [11].

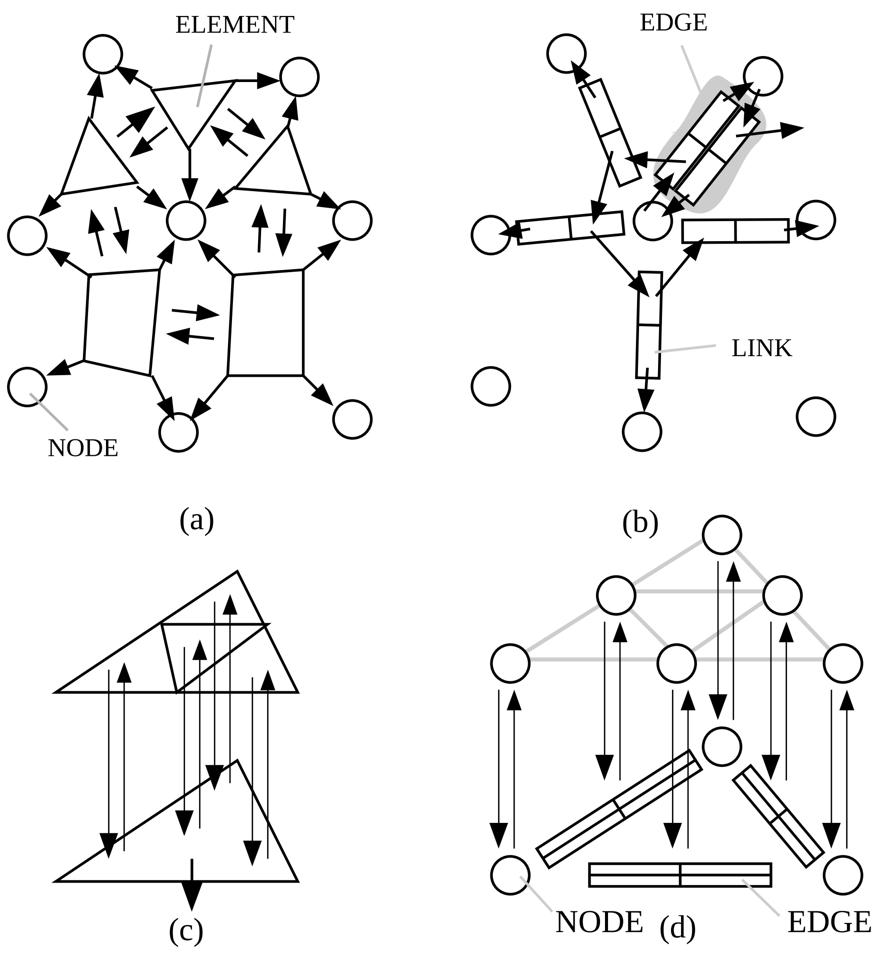

Figure 4: UG unstructured mesh data structure.

Besides in mesh generation, the hierarchical mesh structure is also of central importance to other steps of the PDE solution process: It is used to define a hierarchy of finite element spaces to be used in the multigrid solver, it can be used to obtain good initial guesses in the nonlinear solver (nested iteration) and it is useful for reduction of the complexity of the load balancing problem, see [3]. Furthermore, the hierarchical structure allows for tremendous savings in the size of output files, see [7], and can be used for an efficient parallel solution of the 3D hidden surface problem, see [10].

We will now briefly consider the data structure used to represent the hierarchical mesh (it is covered in more detail in [4]). The `MULTIGRID` data type represents a complete hierarchical unstructured mesh consisting of several grid levels. Each grid level is accessible via the `GRID` data type holding elements (the `ELEMENT` data type), vertices (`NODE` and `VERTEX` data types) and edges (`LINK` and `EDGE` data type).

References between objects are shown in Fig. 4. Elements have access to their nodes and neighboring elements (part (a)), nodes have access to neighboring nodes via the edge list (part (b)). The hierarchical relationships of elements and nodes are shown in parts (c) and (d) of the figure.

Typical operations on the data structure include browsing, tagging elements for refinement/coarsening and mesh modification.

SPARSE MATRIX–VECTOR DATA STRUCTURE

In finite element or finite volume methods the solution of a PDE problem is approximated in a finite–dimensional function space equipped with a local basis. This means that any function in that space is determined locally on each element by degrees freedom related to that element and its faces, edges or vertices. Two elements share degrees of freedom at common vertices, edges and faces.

Typically, the whole solution process requires several solutions and/or right hand sides to be stored. Therefore the grid manager allows a variable number of floating point values to be associated with each geometric location (node, edge, face, element) at run–time. Note that degrees of freedom forming for example the solution vector are not stored in one big array but rather all floating point values related to a geometric location are stored in a small block. This prevents the use of efficient, array–based matrix–vector operations but on the other hand enables easy addition/deletion of degrees of freedom as the mesh is refined/coarsened. Furthermore it allows the direct use of DDD for the parallelization of matrix–vector operations.

Matrix entries are collected in blocks that couple all degrees of freedom in a geometric location with those in another geometric location. Each location stores a list of all matrix blocks coupling this location with other locations (i. e. a block compressed row storage scheme). Matrix blocks are allowed to be sparse as well, see [12].

OBJECT–ORIENTED DESIGN OF NUMERICAL ALGORITHMS

The solution of nonlinear, time–dependent problems involves several cooperating numerical algorithms. An implicit time discretization requires the solution of a system of nonlinear algebraic equations per time step. Solving that by Newton's method requires the solution of a system of linear equations per iteration. As a linear solver one might consider a Krylov subspace method which requires a preconditioner, e. g. multigrid. A multigrid iteration needs a smoothing iteration, grid transfer operators and a coarse grid solver which in turn might be another preconditioned Krylov method or an algebraic multigrid scheme if the coarse grid is not small enough to solve the equations exactly. The Newton scheme may also require an interpolation scheme to transfer an initial guess from coarse to fine grid. In the adaptive case an error estimator is required. Even more complex scenarios can be imagined when using decoupled solution strategies for systems of PDEs.

The "numerical procedures" in UG have been designed to support this kind of flexible composition of solver components. In particular we wanted the design to possess the following properties:

- Components should be reusable across problem domains. E. g. the time–stepping code should be the same regardless of the PDE to be solved.

- Components should not use outside knowledge. E. g. the nonlinear solver should not know whether it solves a nonlinear problem within a time–step or a stationary problem.
- The components should be configurable from script file to be able to quickly test different configurations.

In order to achieve these goals the numerical algorithms have been realized as a class hierarchy. Part of the class diagram related to discretization and time–stepping schemes is shown in Fig. 5. Classes are denoted by rectangular boxes having the class name at the top. Classes with names in italics denote abstract classes, a class name in regular text denotes concrete classes. An arrow with a triangle denotes class inheritance, a regular arrow denotes usage (reference) of a class.

A few words about implementation may be in order here since UG is written in C, not in C++. Classes are implemented as structs containing function pointers. Every member function receives a pointer to the object as first parameter (the `this` pointer). All function pointers are included in every instance of a class. A virtual function table has been omitted since memory requirements are not critical. Inheritance is implemented by including the "base class" in the "derived class".

The class design is illustrated by showing the interaction of the implicit time–stepping scheme and the nonlinear solver in more detail.

Let us consider the solution of a system of ordinary differential equations (ODEs) in the form

$$\frac{d\,(m(y(t)))}{dt} = f(t, y(t)), \qquad y : \mathbb{R} \to \mathbb{R}^N, y(t^0) = y^0. \tag{1}$$

Solving (1) with an implicit Euler scheme leads to the following nonlinear algebraic system to be solved per time step:

$$F_{IE}(y^{n+1}) = m(y^{n+1}) - \Delta t f(t^{n+1}, y^{n+1}) - m(y^n) = 0. \tag{2}$$

We assume that the general form of the nonlinear system occurring in an implicit solution of (1) is

$$F(y^{n+1}) = \sum_{k=k_0}^{1} \left[\alpha_{n,k} m(y_{n+k}) + \beta_{n,k} f(t_{n+k}, y_{n+k})\right] = 0 \tag{3}$$

with $\alpha_{n,1}$ normalized to 1.0. Abstract class ***NP_T_ASSEMBLE*** (see Fig. 5) defines the interface to the discretization scheme based on this general form. The two main functions are ***TAssembleDefect()*** which does one step of (3),

$$d = d + \alpha m(y) + \beta f(t, y) \tag{4}$$

(given α, β, t and y) and ***TAssembleMatrix()*** which is required to provide some linearization $J \in \mathbb{R}^{N\times N}$ of (3), e. g. the full linearization

$$(J)_{ij} = \frac{\partial m_i}{\partial y_j} - \beta_{n,1} \frac{\partial f_i(t, y)}{\partial y_j} \tag{5}$$

This interface is general enough to support a wide variety time discretization schemes including one step–θ, fractional step–θ, backward difference formulas and diagonally implicit Runge–Kutta schemes.

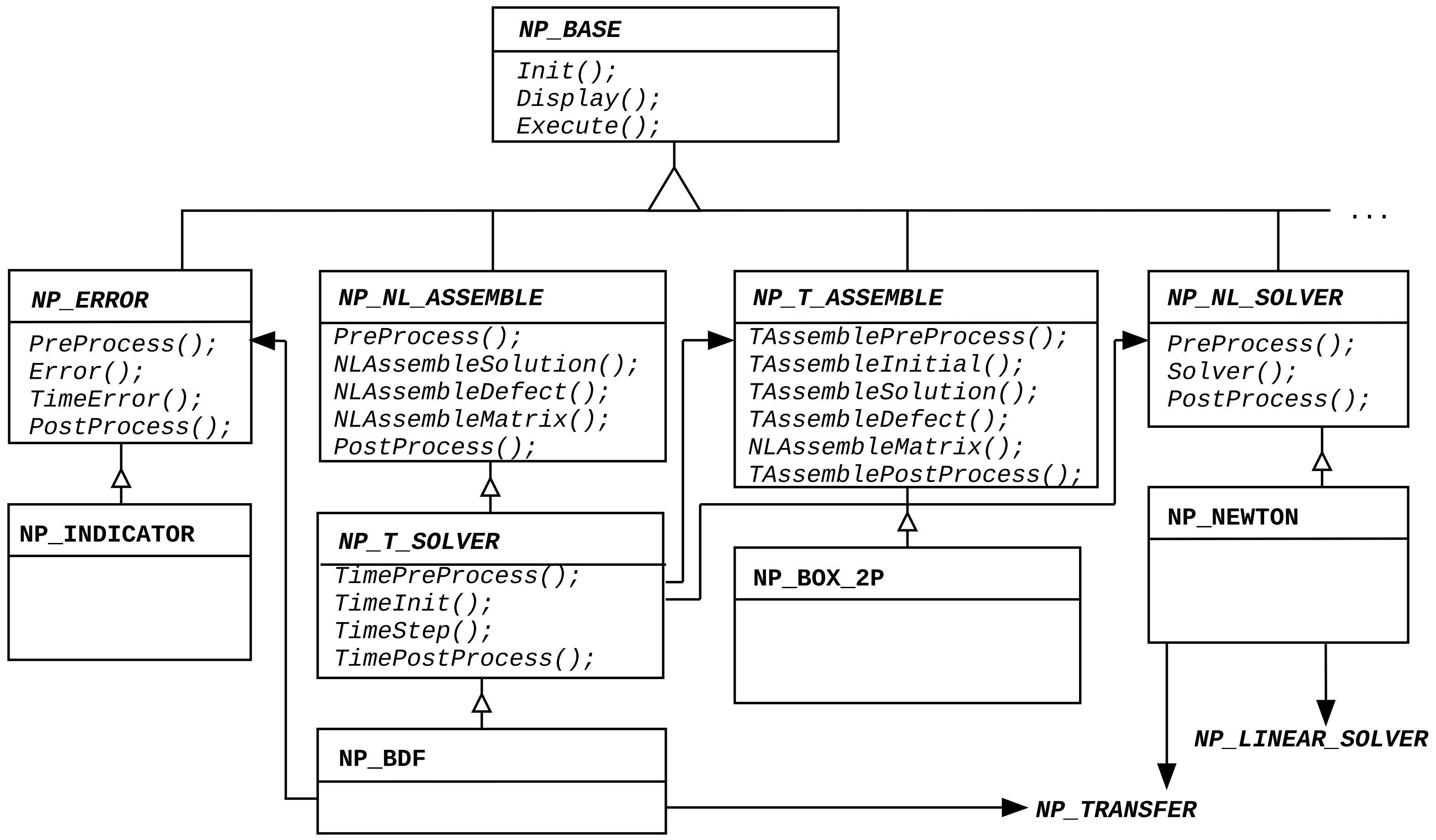

Figure 5: Numerical algorithms class diagram: Assemble, time–stepping and nonlinear solver classes.

The discretization interface for *stationary* nonlinear problems of the form

$$F(y) = 0 \tag{6}$$

is given by the virtual class `NP_NL_ASSEMBLE`. It is very similar to the time–dependent case with `NLAssembleDefect()` computing $d = F(y)$ and `NLAssembleMatrix()` providing a linearization. A nonlinear solver from class `NP_NL_SOLVER` (not shown in the figure) expects an object of type `NP_NL_ASSEMBLE` as an argument to its `Solver()` member function:

```
NP_NL_SOLVER::Solver(...,NP_NL_ASSEMBLE *problem,...);
```

The time–stepping scheme interface is defined by class `NP_T_SOLVER`. It is derived from `NP_NL_ASSEMBLE` and uses an object of type `NP_T_ASSEMBLE` to implement the `NP_NL_ASSEMBLE` interface for the nonlinear problem to be solved in a time step. When the time–stepping scheme calls the nonlinear solver it passes itself as the `problem` parameter. When the nonlinear solver then executes a member function of the `problem` object, control will return to the time–stepping scheme which has all the information available in order to compute the defect and linearization correctly. Hence, the nonlinear solver does not need to know whether it solves a nonlinear problem within a time step.

All nonlinear, time–dependent problems listed in Fig. 3 have been implemented conforming to the `NP_T_ASSEMBLE` interface and therefore share the same time–stepping code as well as linear and nonlinear solvers.

CONCLUSIONS

Due to lack of man–power and expertise probably no single group of researchers will ever have the fully–integrated parallel adaptive PDE software package. It is therefore mandatory to define standardized interfaces for the PDE software components such that each group can contribute modules from its area of expertise and use the modules of other groups in the remaining areas. Module interfaces should be general enough to allow competing implementations concentrating on different aspects such as speed, memory requirements or generality. Algorithms and data structures should be decoupled wherever possible.

The biggest challenges in the construction of these interfaces are "design for change" and the "combination of flexibility and efficiency". As new (numerical) algorithms are developed it should be able to incorporate them into the framework. This requires a lot of experience in the design of the interfaces. Flexibility and efficiency can be achieved by combining a high–level object–oriented design with optimized low–level kernels.

REFERENCES

[1] BANK, R., A. SHERMAN, and A. WEISER: "Refinement algorithms and data structures for regular local mesh refinement", In Scientific Computing, IMACS, North–Holland, 1983.

[2] BASTIAN, P.: "Parallele adaptive Mehrgitterverfahren", Teubner, Stuttgart, 1996.

[3] BASTIAN, P.: "Load balancing for adaptive multigrid methods", SIAM J. Sci. Stat. Comput., 19(4), 1303–1321, 1998.

[4] BASTIAN, P., K. BIRKEN, S. LANG, K. JOHANNSEN, N. NEUSS, H. RENTZ-REICHERT, and C. WIENERS: "UG: A flexible software toolbox for solving partial differential equations", Computing and Visualization in Science, 1, 27–40, 1997.

[5] BEY, J.: "Tetrahedral grid refinement", Computing, 55, 355–378, 1995.

[6] BIRKEN, K.: "Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen", Ph. D. thesis, Universität Stuttgart, 1998.

[7] FEIN E. (ED.): "d^{3}f – Ein Programmpaket zur Modellierung von Dichteströmungen", GRS, Braunschweig, ISBN 3-923875-97-5, 1998.

[8] HENDRICKSON, B. and R. LELAND: "The CHACO user's guide 1.0", Technical Report SAND93–2339, Sandia National Laboratories, 1993.

[9] JONES, M. and P. PLASSMANN: "Parallel algorithms for adaptive mesh refinement", SIAM J. on Scientific Computing, 18, 686–708, 1997.

[10] LAMPE M.: "Parallelisierung eines Grafiksubsystems in einem Paket zur numerischen Lösung partieller Differentialgleichungen", Master's thesis, Uni Stuttgart, 1997.

[11] LANG, S.: "Parallele adaptive Mehrgitterverfahren für dreidimensionale instationäre Berechnungen", Ph. D. thesis, Universität Heidelberg, to appear.

[12] NEUSS, N.: "A new sparse matrix storage method for adaptive solving of large systems of reaction-diffusion-transport equations", Technical Report 1999–04, IWR, Uni Heidelberg, 1999.

[13] RUMPF, M., R. NEUBAUER, M. OHLBERGER, and R. SCHWÖRER : "Efficient visualization of large–scale data on hierarchical meshes", In W. Lefer and M. Grave (Eds.), Visualization in Scientific Computing '97. Springer, 1997

[14] SCHÖBERL, J.: "A rule–based tetrahedral mesh generator", Computing and Visualization in Science, 1, 1–26, 1997.

[15] SMITH, B., P. BJØRSTAD, and W. GROPP: "Domain Decomposition", Cambridge University Press, 1996.