Object-oriented Design of an AMR-algorithm for Distributed Memory Computers

Ralf Deiterding
Institute of Mathematics, Technical University Cottbus, Germany
e-mail deiterding@math.tu-cottbus.de

Abstract

The design of an object-oriented framework for the blockstructured Berger-Oliger AMR-method is presented. It simplifies the definition of concrete applications and allows a natural formulation of the parallel AMR-algorithm. A distribution strategy especially tailored for AMR is described. Gas dynamical computations of instabilities of the Kelvin-Helmholtz type in two and three space dimensions are employed to demonstrate the efficiency of the approach.

1. Introduction

The adaptive mesh refinement method (AMR) by M. Berger and J. Oliger [4] is widely used for the adaptive computation of hyperbolic conservation laws on blockstructured grids. It is designed especially as a general adaptive framework for time-explicit finite-volume methods on high-performance computers. Due to its limitation to blockstructured grids the AMR-method is restricted to relatively simple computational domains, but it allows extraordinarily highly resolved computations [1].

Instead of replacing single cells by finer ones the AMR-method follows a patch-wise refinement strategy. Cells being flagged by various error estimators are clustered into rectangular boxes of appropriate size. They describe refinement regions geometrically and subgrids with refined mesh spacing in space and time are generated according to them. Refined grids are derived recursively from coarser ones and an entire hierarchy of successively embedded grid patches is therefore constructed. All grid patches are logically rectangular and only a specific integration method for single rectangular grids is required. The adaptive algorithm calls this application dependent routine automatically. Further on it uses conservative interpolation functions to transfer cell values between refined subgrids and their coarser parents appropriately.

It is important to note, that refined grids overlay the coarser subgrids from which they have been created. The numerical solution on a particular level is first of all advanced independently. Values of cells covered by refined subgrids are overwritten by averaged fine grid values subsequently. The resulting extra work is usually negligible compared to the computational costs for integrating the superimposed refinement grids.

Replacing coarse cell values by averaged fine grid values modifies the numerical stencil on the coarse grid. In general the import property of conservation is lost. A flux correction replacing the coarse grid flux at the affected side of a neighboring

cell by accumulated fine grid fluxes is necessary to ensure conservation. This so called conservative fixup is usually implemented as a correction pass. In two and three space dimensions hanging nodes additionally have to be considered. See [2] for details.

Up to now, various reliable implementations of the AMR-method for single processor computers have been carried out [3, 5, 6]. Even implementations for parallel computers with shared memory architecture have reached a stable state [1]. Parallelism is an inherent feature of the AMR-algorithm and in a shared memory environment simply the numerical solution on the whole sequence of grids has to be advanced in parallel to achieve a sufficient load-balancing.

The question for an efficient parallelization strategy becomes more delicate for distributed memory machines, because the costs of communication can not be neglected anymore. Due to the technical difficulties in implementing dynamical adaptive methods in a distributed memory environment only few parallelization strategies have been tried out in practice yet, c. f. [7, 8, 10].

We follow a parallelization strategy proposed by M. Parashar and J. Browne [9, 10]. All data assigned to grid patches are stored in hierarchical grid functions which are automatically distributed with respect to a global grid hierarchy. Dynamic distribution is carried out under the restriction that higher level data must reside on the same computing node as the coarsest level data.

In this paper, we demonstrate, that a full, parallel AMR-algorithm for hyperbolic conservations laws can be implemented efficiently on top of these data structures. We describe an object-oriented design, that allows the formulation of all components of the parallel AMR-method almost like in the serial case. By employing ghost cells regions, which are synchronized automatically whenever the algorithm applies boundary conditions, an overlap between subgrids is constructed that allows most AMR-operations to be carried out strictly local.

Exemplary computations of our parallel AMR implementation in two and three dimensions will be discussed in detail. Although the framework already has been applied successfully to more complex multicomponent gas flows, intentionally simple examples are chosen to yield more understandable results.

2. An object-oriented design

In AMR-methods three main abstraction levels can be identified. At the top level, the specific application is formulated. Our AMR implementation relies on standardized interface-objects to application specific components like initial and boundary conditions or numerical integration routines. Defining a new application does not require any knowledge of AMR. The mere AMR-solver and its components for grid generation, error estimation, interpolation and flux correction make up the second level. This level naturally utilizes methods of the base level, which supplies the hierarchical data structures. The base level is divided into the preparation of elementary functionality for single grid patches and the implementation of various lists that store these patches hierarchically.

All implementation approaches that try to combine the clarity of an object-oriented design with the execution speed necessary for highly resolved simulations come to similar designs for the application level and the definition of a single grid patch inside the base level [1, 5, 6, 7, 10]. These elementary objects are neglected in the object diagram of the complete design in fig. 4. We describe them briefly instead.

A *box*-object defines a rectangular box in a global integer index space. Methods for geometric operations on boxes like concatenation or intersection are available. A *patch*-object adds consecutive data storage to a box-object. When the AMR-algorithm passes through the hierarchy, single patches are parameters for the *integrator* that calls the numerical solution routines. It is common practice to use Fortran-functions for computationally intensive single patch operations, like numerical integration, restriction or prolongation and C++ for the implementation of the objects themselves. Hence, the data inside a patch-object is in Fortran-format.

The geometrical description of all refinement areas is stored in hierarchical lists of box-objects inside a single *grid hierarchy*. Several "distributed" *grid functions* of various storage types create patch-objects according to these lists. The grid hierarchy consists of global lists storing the complete refinement information and lists for each processor's local contribution. During computation, the grid hierarchy is dynamically repartitioned under the restriction that higher level data must be assigned to the same processor as the coarsest level data. As patch-objects are generated directly from these local box-lists, all grid functions become equally distributed following the actual "floor plan". Redistribution over processors is carried out automatically as a natural part of the AMR-algorithm whenever the grid hierarchy changes (see fig. 1).

Note that the described data representation model assigns multiple patch-objects stored in different grid functions to a single subgrid. The AMR-algorithm utilizes rectangular data blocks of various storage type to handle a single refinement grid, but in our design these blocks are accessed via patch-objects kept in different grid functions. This special design follows from the perception that the tedious technical overhead in implementing the AMR-method in a distributed memory environment can be simplified significantly, if commonality in organizing rectangular data blocks independent of their storage type is exploited. We concentrate this common functionality in a single C++ base-class and by employing template data types and compile-time parameters carefully all grid-function-objects are derived from this base class without a loss of computational performance.

Usually AMR-implementations employ ghost cells for the setting of boundary conditions, because their application allows a similar treatment of internal and physical boundaries. In our approach distributed grid functions enlarge their patches automatically by ghost cell regions of suitable size. Ghost cell regions of neighboring patches are synchronized transparently even over processor borders, whenever the AMR-method applies boundary conditions. Thus, a proper parallel synchronization of neighbors is guaranteed by the algorithm itself.

The use of overlapping ghost cell regions combined with the distribution strat-

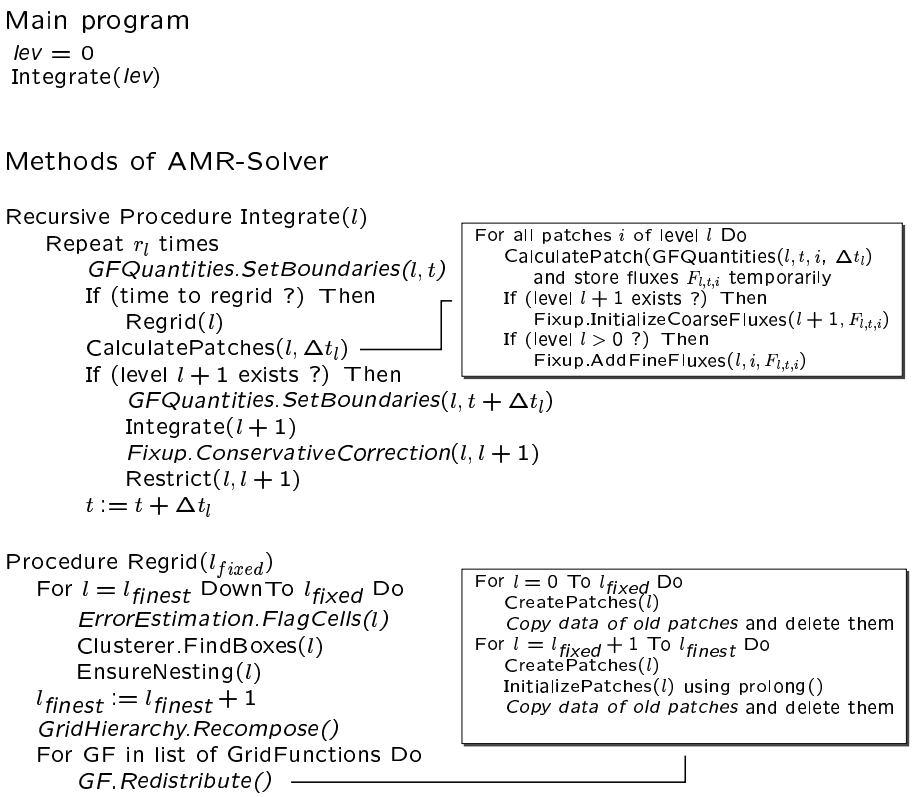

Figure 1: An AMR-algorithm for distributed memory computers based upon the design shown in fig. 4. The procedures in italics might require interprocessor communication.

egy outlined above ensures that almost all computational operations of the parallel AMR-algorithm do not require interprocessor communication. Technical details of communication can be hidden completely against the AMR-algorithm and each grid function just has to supply methods that initiate ghost cell synchronization and patch redistribution.

AMR requires hierarchical grid functions of various spatial dimensions, with complex storage data types and with differently sized overlap regions. Deriving all grid functions from one base class guarantees a common interface and we managed to code most parts of our parallel AMR-algorithm independently of spatial dimensions and size of the vector of state.

3. A parallel AMR-algorithm

Using the described programming abstractions, a parallel AMR-algorithm for hyperbolic conservation laws can be formulated. Following [1] the mere algorithm is split into a function that updates the levels recursively by employing the numerical solution routines and a function that controls the regridding procedure (see fig. 1). We implement these functions as methods of a central *AMR-solver*-object. Mostly, they operate on *GFQuantities*, a distributed grid function attached to AMR-solver

storing patch-objects for the vector of state. As mentioned above, GFQuantities synchronizes its patches transparently over processors, when boundary conditions are applied. Additionally, it automatically fills ghost cells at internal boundaries with appropriately prolongated (locally available) coarse grid values.

After setting boundary conditions, the numerical solution is computed locally. At coarse-fine interfaces the fluxes are used to calculate correction terms that ensure conservation. The correction terms are saved in grid functions *GFFixup* of lower spatial dimension that are assigned to the boundaries of fine grids. E.g. in two dimension four of these grid functions are necessary. They are initialized with the corresponding coarse grid flux, fine grid fluxes at the particular boundary are added during recursive computation. Flux correction in detail depends heavily onto the specific numerical method employed, c.f. [3]. Hence, an exchangeable *Fixup-object* with a well defined interface guarantees the required flexibility.

The automatic redistribution of all grid functions combined with the use of ghost cells ensures that correction terms can be computed strictly local. Only their application in form of a correction of coarse grid cell values must be done with respect to neighboring patches on other processors. Finally, coarse cell values are replaced by restricted values where finer grid patches overlap. Due to the chosen distribution strategy this operation is strictly local.

When a level and all finer levels need regridding, cells are flagged for refinement locally. The refinement criterion, e.g. a combination of error estimation and approximated gradients has to be interchangeable and is consequently defined in a further object outside of AMR-solver. The resulting flags are kept in a integer-valued grid function *GFFlags*. Its overlap region between patches corresponds to the size of the buffer region around flagged cells. Synchronizing this grid function allows a parallel execution of the algorithm that clusters flagged cells into suitable rectangles. Each processor thus generates properly nested box-lists of new refinement regions in his actual contribution of the computational domain.

These lists are used to update the global grid hierarchy. Before the grid hierarchy can be partitioned appropriately, the estimated workload on all levels is added up and assigned to cells of the coarsest level. The algorithm used for partitioning the computational domain has to meet several requirements. It must balance the estimated workload, while maintaining patches of sufficient size. The algorithm should be fast, because it is executed after each regrid operation. Additionally, communication for redistributing grid functions and synchronizing them during calculation should be minimal. M. Parashar and J. Browne proposed partitioning algorithms that are based on generalized space-filling curves [9]. We employ such a partitioner in our computational examples.

After recomposing the grid hierarchy's local box-lists all grid functions have to be redistributed. It depends on the type of the particular grid function how the newly created patches have to be initialized. Fig. 1 shows the general case which applies to the grid function for the vector of state. In case of the grid functions for correction terms no prolongation is necessary; the new patches of GFFlags need not even be initialized.

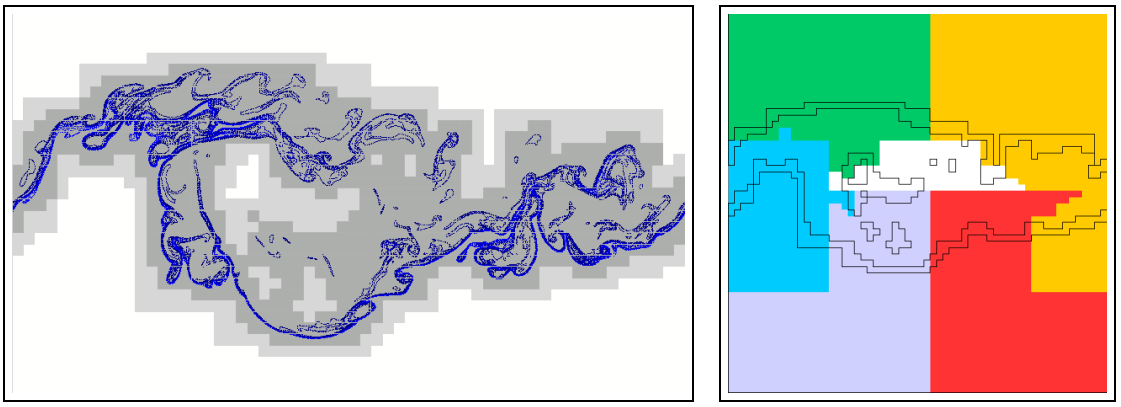

Figure 2: Growing perturbations after 1011 time steps of the 2D-computation and dynamical adaption of the parallel AMR-algorithm. *Left:* Isolines of density on refinement grids of 1st (light grey) and 2nd (dark grey) level. *Right:* Distribution of the computational domain to 6 computing nodes. Estimated workload differs between 96.4% and 106.1%.

4. Computational results

The efficiency of the approach is demonstrated by sample computations of instabilities of the Kelvin-Helmholtz type in an ideal gas. This well understood instability develops, if an initial planar contact discontinuity over flown by a tangential shear flow is slightly disturbed. In this case, the initial flat interface shows evolving perturbations during simulation time and starts to form the well know roll-up. We compute a two- and a three-dimensional example by solving the compressible Euler equations for a single ideal gas. To allow comparisons the popular CLAWPACK-routines are employed as numerical integration routines at application level. Special versions of the routines returning the fluctuations are necessary to enable the calculation of correction terms [3].

In both computations the interface separates a low density region from a higher density area below. At the upper and lower boundaries solid-wall boundary conditions are used, while all other boundaries are assumed to be periodic. In the two-dimensional case the interface initially is defined by one period of sine wave. The shear flows in both phases are equally directed, while the lower flow is forty times slower than the flow above. To illustrate this example fig. 2 shows the interface after 1011 time steps calculated with a CFL-No. around 0.95. The benchmark uses the first 216 time steps of this computation. The base grid has 120×120 cells, while two refinement levels each with refinement factor 4 are allowed. Instead of 3.7M cells for a uniform grid, the adaptive computation uses 167k up to 280k cells.

The initial interface for the three-dimensional computation is defined periodically as a product of two sine waves. Diagonally directed opposed shear flows are applied. The magnitude of the velocity of the lighter phase is five times large than the velocity of the flow below. Fig. 3 shows how the initially nearly flat interface has

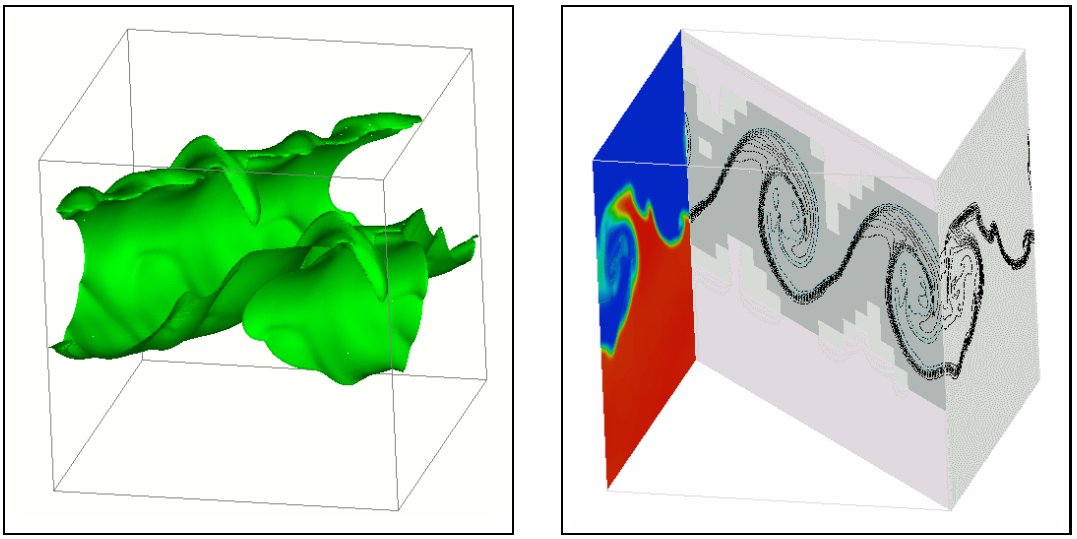

Figure 3: Roll-up of the interface after 686 times steps of the 3D-calculation. *Left:* An isosurface of the density distribution shows the interface. *Right:* Refinement grids of 2nd level on a diagonal cut parallel to the streamlines of the flow.

formed an enormous roll-up after 686 time steps using an approximate CFL-No. of 0.90. For this benchmark the first 102 time steps are calculated. A base grid with $40 \times 40 \times 40$ cells and two refinement levels with refinement factor 2 are employed. The adaptive computation restricts itself to 806k up to 877k cells instead of 4.1M cells for a corresponding uniform grid.

Tab. 1 shows the fractions of computational time spent in different code units, while an increasing number of computing nodes is used. Patch initialization and the setting of boundary cells at coarse-fine interfaces are accumulated to "Interpolation". "Boundary setting" accounts of physical boundary conditions and synchronization of patches of the same level. "Recomposition" denotes the load-balanced reconstruction of the grid hierarchy, while the mere parallel exchange of grid patches due to load-imbalance is called "Redistribution".

The omitted absolute time value of both computations show, that integration and interpolation scale up well. Obviously, numerical integration in three dimensions is more costly than in two and especially the costs for parallel synchronization are significantly higher. The portion for integration rises additionally in the three-dimensional computation, because it uses much fewer refinement grids than the two-dimensional one. Thus, the overhead for recomposition and redistribution becomes negligible against the expense for advancing the numerical solution. In contrast to the two-dimensional case the parallel efficiency of this example depends mainly on synchronization. The two-dimensional benchmark splits up much more refinement patches to achieve a sufficient load balancing. Hence, its portions for recomposition and setting of boundary values increase more drastically.

Task / %	2D-computation					3D-computation			
	P=1	P=2	P=4	P=8	P=16	P=1	P=2	P=4	P=6
Patch integration	77.5	72.9	63.7	54.1	42.4	93.3	88.0	84.4	77.5
Recomposition	5.8	5.1	6.7	8.1	10.3	0.6	0.4	0.5	0.4
Redistribution		3.5	5.3	9.9	13.4		0.3	0.4	0.4
Boundary setting	0.7	1.6	3.2	5.2	8.1	0.4	0.5	0.7	0.9
Synchronization		2.4	7.5	10.7	14.6		5.7	8.6	15.9
Interpolation	5.7	5.5	4.9	4.2	3.5	2.4	2.4	2.4	2.0
Conservative Fixup	2.9	2.8	2.8	2.5	2.6	1.3	1.2	1.3	1.0
Synchronization		0.1	0.7	1.0	1.0		0.1	0.3	0.5
Clustering	3.9	2.8	2.1	1.4	1.3	0.5	0.5	0.4	0.4
Not measured	3.5	3.3	3.1	2.9	2.8	1.5	0.9	1.0	1.0
Parallel Eff.		95.5	86.9	73.6	58.3		95.0	85.9	82.5

Table 1: Decomposition of computational time. Interprocessor communication using the MPI-library is set off in italics. The 2D-computation was carried out on SP2. The 3D-computation was run on a Pentium-based PC-Cluster connected with a high-speed network. *Parallel efficiency* $= T_1 \,/\, (P \cdot T_P)$.

5. Conclusions

An object-oriented design for the Berger-Oliger AMR-method on distributed memory machines has been developed. It divides into abstraction levels for the concrete application, adaptive method and parallelized data structures. At the top, numerical solvers are separated from the general AMR-framework. A specific problem mainly requires routines for advancing the numerical solution and setting of initial and boundary conditions which are implemented for a single blockstructured grid. The adaptive framework automatically generates a sequence of blockstructured subgrids which are passed to these functions successively.

The formulation of the mere AMR-algorithm itself is enormously simplified by encapsulating technical parallelization details completely inside the lowest level of hierarchical data structures. The design of these data structures has been described briefly, especially the chosen dynamic distribution strategy has been sketched. The described strategy places the data of all levels on the same processor as the data of the base level. Suitable regions of overlapping ghost cells guarantee a proper synchronization between subgrids. It has been demonstrated, that this strategy allows a formulation of the AMR-method similar to the serial case, because most AMR-operations remain strictly local and synchronization and redistribution can be carried out as natural parts of the algorithm. Especially the technically difficult conservative correction, which is mandatory for hyperbolic conservation laws, can be realized efficiently even in a distributed memory environment.

Sample computations with an implementation based upon the described design have been presented in detail. They confirm the efficiency of the whole approach

and show that even for fast Fortran-solvers the AMR-method can reach sufficient performance on distributed memory machines.

References

[1] J. Bell, M. Berger, J. Saltzman, and M. Welcome. Three-dimensional adaptive mesh refinement for hyperbolic conservation laws. *SIAM J. Sci. Comput.*, 15(1):127–138, 1994.

[2] M. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82:64–84, 1988.

[3] M. Berger and R. LeVeque. Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. *SIAM J. Numer. Anal.*, 35(6):2298–2316, 1998.

[4] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, 1984.

[5] W. Crutchfield and M. L. Welcome. Object-oriented implementation of adaptive mesh refinement algorithms. *J. Scientific Programming*, 2:145–156, 1993.

[6] H. Friedel, R. Grauer, and C. Marliani. Adaptive mesh refinement for singular current sheets in incompressible magnetohydrodynamics flows. *J. Comput. Phys.*, 134(1):190–198, 1997.

[7] S. R. Kohn and S. B. S. B. Baden. A parallel software infrastructure for structured adaptive mesh methods. In *Proc. of the Conf. on Supercomputing '95*, December 1995.

[8] M. Lemke, K. Witsch, and D. Quinlan. An object-oriented approach for parallel self adaptive mesh refinement on block structured grids. In W. Hackbuch and G. Wittum, editors, *Adaptive Methods-Algorithms, Theory and Applications*, pages 199–220, Braunschweig/Wiesbaden, 1994, January 22-24 1993. Proceedings of the Ninth GAMM-Seminar, Vieweg & Sohn Verlagsgesellschaft mbH.

[9] M. Parashar and J. C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, January 1996.

[10] M. Parashar and J. C. Browne. System engineering for high performance computing software: The HDDA/DAGH infrastructure for implementation of parallel structured adaptive mesh refinement. In *Structured Adaptive Mesh Refinement Grid Methods*, August 1997. IMA Volumes in Mathematics and its Applications, Springer-Verlag.

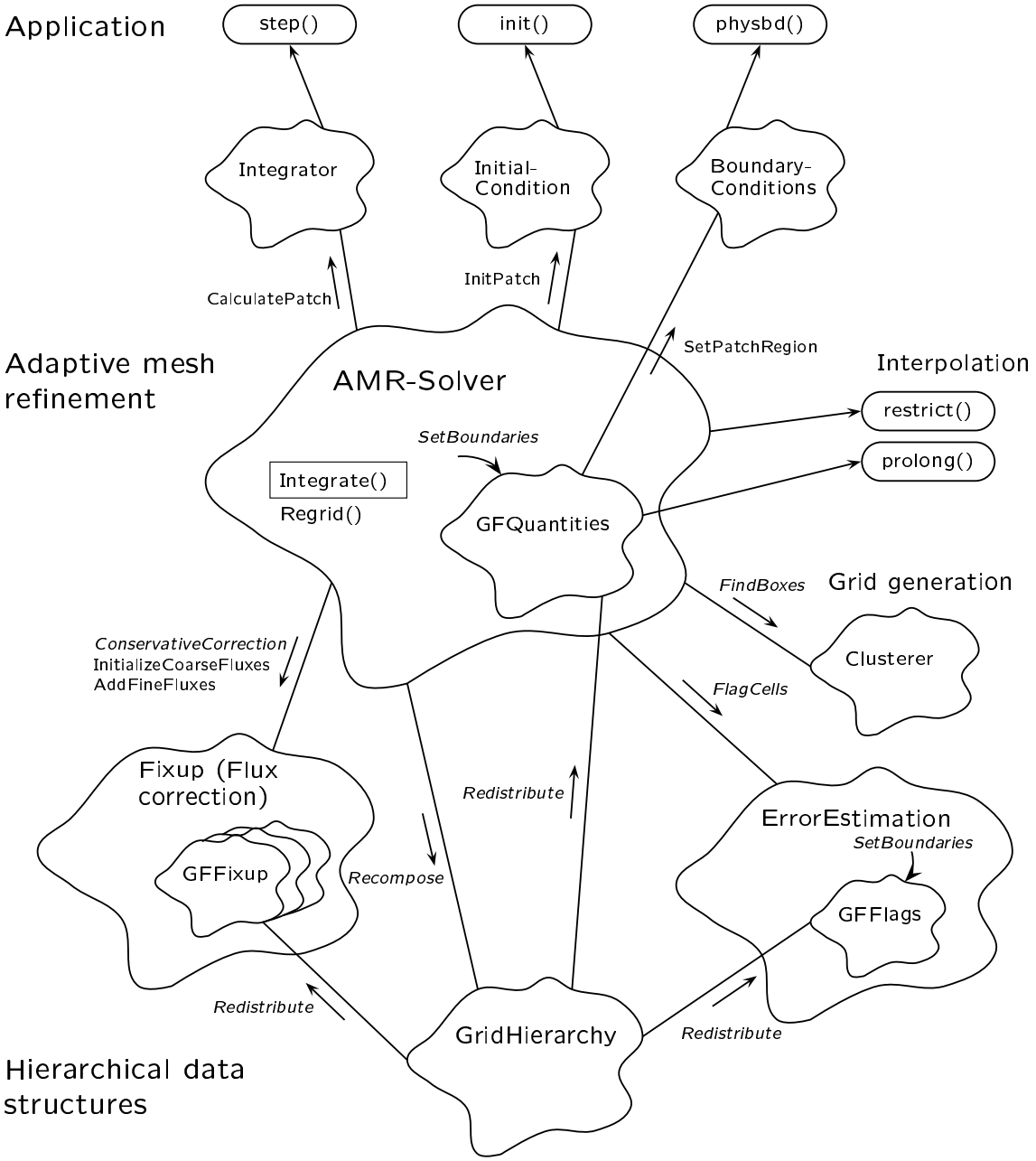

Figure 4: Object-oriented design for the AMR-method. The adaptive algorithm is implemented in AMR-Solver. See fig. 1.