Memory Efficient Adaptive Mesh Generation and Implementation of Multigrid Algorithms Using Sierpinski Curves

Michael Bader* TU München
Stefanie Schraufstetter TU München
Jörn Behrens† AWI Bremerhaven

Abstract

We will present both underlying ideas and concepts, and first results and experiences gained within an ongoing project to implement a highly memory efficient version of the grid generator `amatos` [BRH+05]. Our focus will be on algorithmic and implementational approaches to realize multigrid algorithms on recursively structured adaptive triangular grids. The key concept is to use a cell-oriented processing of the grids and space-filling-curve techniques to get rid of the need to store neighbourship relations on adaptive grids.

1 Generating and Storing Recursively Structured Adaptive Grids

The possibility to adaptively refine the discretisation mesh is an issue that has to be addressed in many simulation codes. Apart from numerous numerical questions that have to be answered, such as establishing criteria where refinement is necessary, there are also several challenges concerning the implementation of adaptive grids in simulation software. In [BZ06] we presented an approach that facilitates both the efficient storage of adaptively refined grids, and the efficient implementation of fast multigrid solvers on these data structures. This approach is currently being integrated into the grid generator `amatos` [BRH+05]. Our focus in this paper will be the algorithmic and implementational details of the implemented solver, which is a multigrid preconditioned conjugate gradient method.
The grid generator `amatos` is a simulation tool for generating finite-element meshes that are based on the recursive bisection of triangular grid cells. Starting from a triangular super cell, each cell of the computational grid is subdivided at a *marked edge* until a certain resolution or level of adaptive refinement is reached. The recursive structure motivates the representation of such a grid using the corresponding tree structure (see figure 1). This tree structure, and thus the corresponding adaptive grid, may be stored in memory using a linearisation of the tree based on a depth-first traversal, where the grid cells are visited in the order of a Sierpinski space-filling curve, as is illustrated in figure 1.

*Institut für Informatik, TU München, Boltzmannstr. 3, D-85748 Garching, bader@in.tum.de
†Alfred-Wegener-Institut, Am Handelshafen 12, D-27570 Bremerhaven, jbehrens@awi-bremerhaven.de

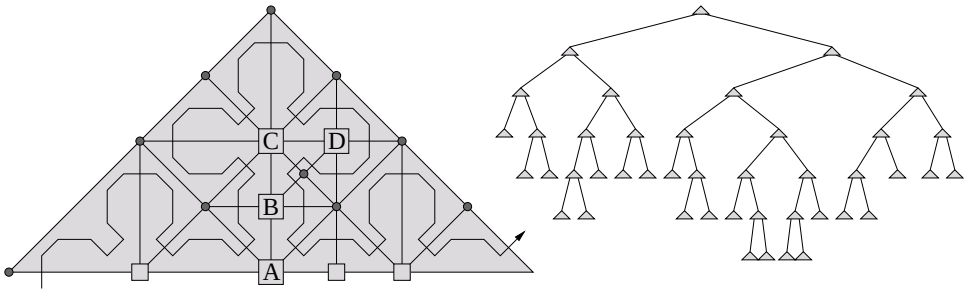

Figure 1: Recursively constructed triangular grid and its corresponding tree structure. The Sierpinski order leads to a FIFO-type access to the unknowns A, B, C, and D.

However, up to now, such data structures were at best used for grid generation, but not for implementing numerical algorithms. To implement these, the neighbour relationships between cells, nodes, and edges have to be known to evaluate local discretisation stencils, for example. From the tree structure alone, these neighbour relationships are not easily available, which is why most grid generators that use adaptive grids, including the current version of `amatos`, invest precious amounts of memory to store these neighbour relations explicitly.

2 Using Sierpinski Curves for Element-Wise Processing

The algorithmic scheme presented in [BZ06] leads to implementations of iterative solvers without the need to store neighbour relations. The respective algorithm is based on an element-wise processing of the grid cells in an order given by a space filling curve—for the recursively structured triangular grid cells, a Sierpinski curve is used. The processing on each cell requires access to the unknowns, which may be located on nodes, edges or in the interior of the cells. Unknowns on nodes and edges will be used in several grid cells, so intermediate results have to be stored. Figure 1 motivates the use of stacks as data structures to hold these intermediate values: for example, the unknowns at the nodes A to D are accessed in ascending order when processing cells in the left half of the domain, but are accessed in descending order, when the right half of the domain is processed.
In the final algorithm, two streams for input and output data are used, plus two stacks for intermediate results—a *red* stack for unknowns left of the curve, and a *green* one for unknowns right of the curve. To decide on which stack an unknown is to be buffered, or from which stack a buffered unknown needs to be retrieved, we use a set of rules which is derived from the local course of the Sierpinski curve within the current element. There are three basic types of elements, V, K, and H, depending on whether the Sierpinski curve enters and leaves an element via a leg or via the hypotenuse (i.e. the marked edge) of the triangle (see figure 2). The element types enable us to decide for all unknowns, whether

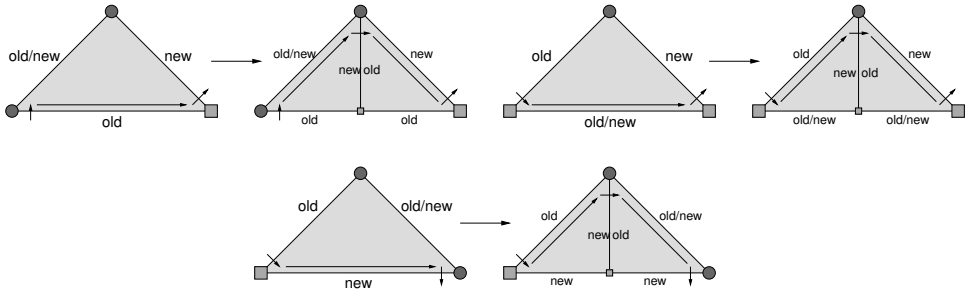

Figure 2: The 2×3 different element-types and the respective types of the child cells.

they should be buffered on the *red* or on the *green* stack. Adding information on whether an edge has been visited before, the types also determine whether an unknown is used for the first time, and has to be taken from the input stack, or whether its processing is finished, and it should be put on the output stack.[BZ06]

3 Multigrid Algorithms on Recursively Structured Triangular Grids

The recursive construction of the computational grid leads directly to a hierarchy of coarse grids, which can be used to design a multigrid algorithm. During the depth-first traversal of the construction tree, all coarse grid cells will be automatically visited as well. Hence, implementing an additive multigrid method is rather straightforward. Here, we present the implementation of a conjugate gradient method with a BPX-type preconditioner. Our approach follows the idea of using hierarchical generating systems as introduced by Griebel [Gri94]. The respective ansatz functions can be chosen similar to the hierarchical basis functions introduced by Yserentant [Yse86].
In the multigrid hierarchy, the coarse grid cells will not carry unknowns on each of their nodes. As illustrated in figure 3, each multigrid level consists of two levels of recursion in the grid generation tree. Whether a coarse grid node carries an unknown can be decided from its element type – V, H, or K. We use the following rules:

- a node opposite to the marked edge will never hold a multilevel unknown;
- in a K-type cell, only the entry node carries an unknown;
- in an H-type cell, only the exit node carries an unknown;
- in a V-type cell, both nodes adjacent to the marked edge, i.e. entry and exit node, are unknowns.

Entry and exit nodes are the nodes where the Sierpinski curve enters and leaves the cell. The combination of two recursion levels to one multigrid level makes sure that all multilevel

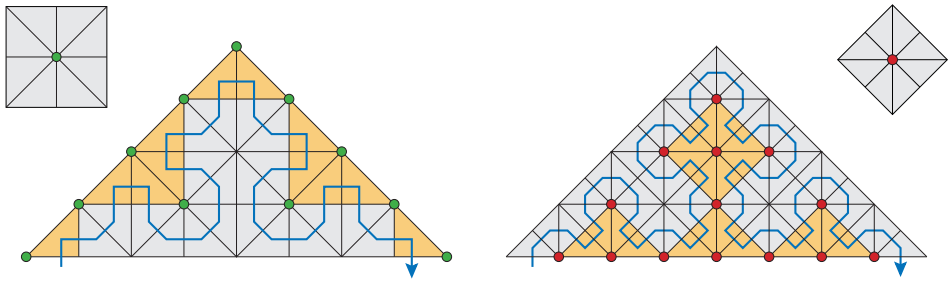

Figure 3: Two subsequent refinement levels with the respective multilevel unknowns. Note the different orientation of the ansatz functions and their supports.

unknowns are associated with a unique ansatz function; if unknowns were defined on each node of every cell, duplicate ansatz function would be generated. The respective duplicate unknowns are not only superfluous, but would also interfere with the stack principle by blocking out fine-level unknowns on the respective stack.
On adaptive grids, situations may occur where a node is supposed to carry a coarse grid unknown in some of the adjacent coarse grid cells, but other adjacent cells are not sufficiently refined to carry an unknown of this level. Such conflict situations have to be detected and avoided by the algorithm. In those cases, no coarse level unknown must be generated, and only the unknowns of the finest level are used. On the finest level, of course, each node of the conforming grid will carry an unknown.

4 Implementation of the Multigrid-Preconditioned Conjugate Gradient Method

The implementation of the preconditioned conjugate-gradient (PCG) method requires two traversals of the grid tree—one for the matrix-vector multiplication within the CG method, and one for the multigrid preconditioning step.[1] During these recursive, depth-first traversals of the cell-tree, both *element updates* and *node updates* may be performed on each cell. An *element update* works on the local element stiffness matrices, and accumulates local contributions to compute matrix-vector products for residuals, etc. A *node update* can be done either right after an unknown is popped from the input stream, or just before an unknown is written to the output stream. Node updates serve to compute vector operations, which can not be implemented in an element-oriented manner. Table 1 shows how the PCG's individual operations are executed throughout the two traversals.

[1]The two traversals may be combined, but this requires some duplicate computations, and extra storage.

Cell-oriented PCG:

1st traversal:

node update: $d_e \leftarrow z_e + \beta d_e$

element update: $a_e \leftarrow a_e + A_e d_e$

$\delta \leftarrow \delta + d_e^T A_e d_e$

restriction: $A_e d_e$ to coarse grid

$\alpha = \gamma_1 / \delta$

2nd traversal:

node update: $x_e \leftarrow x_e + \alpha d_e$

$r_e \leftarrow r_e - \alpha a_e$

prolongation: c_e from coarse grid

$z_e \leftarrow z_e - \alpha c_e$

$\gamma_2 \leftarrow \gamma_2 + r_e z_e$

$\beta \leftarrow \gamma_2 / \gamma_1$; $\gamma_1 \leftarrow \gamma_2$; $\gamma_2 \leftarrow 0$

Regular PCG algorithm:

for $k = 0, 1, 2, \ldots$:

$a^{(k)} = A d^{(k)}$

$\delta = (d^{(k)})^T A d^{(k)}$

$\alpha = (r^{(k)})^T z^{(k)} / \delta = \gamma_1 / \delta$

$x^{(k+1)} = x^{(k)} + \alpha d^{(k)}$

$r^{(k)} = r^{(k)} - \alpha a^{(k)}$

$z^{(k+1)} = M^{-1} r^{(k+1)}$

$\quad = z^{(k)} - \alpha M^{-1} a^{(k)}$

$\gamma_2 = (r^{(k+1)})^T z^{(k+1)}$

$\beta \leftarrow \gamma_2 / \gamma_1$; $\gamma_1 \leftarrow \gamma_2$

$d^{(k+1)} = z^{(k+1)} + \beta d^{(k)}$

Table 1: Element and node updates during the cell-oriented implementation of the PCG method. Initialisation of vectors has been left out for better readability.

5 Numerical Examples

The presented iterative algorithm has been tested on two simple example problems: solving Poisson's equation on the two domains given in figure 4 – the unit square and a domain with a re-entrant corner featuring an edge singularity. As illustrated in figure 4, the computational grids consist of few (two or three, resp.) initial triangular coarse-grid cells, which are connected in a way as to ensure the correctness of the stack-based approach.

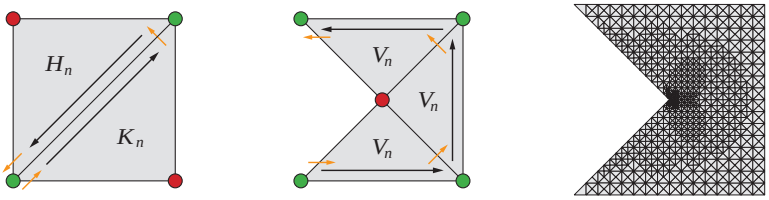

Figure 4: Computational domains and their initial triangulations (left and center), and an adaptive mesh for the re-entrant-corner domain (right).

The convergence of the PCG algorithm has been tested on uniformly refined grids on both computational domains, and for an a-posteriori refined grid for the re-entrant-corner problem. Table 2 lists the number of required PCG iterations, and shows that the convergence rates are as can be expected for an additive multilevel method. The convergence for the re-entrant corner problem deteriorates slightly, which is due to the edge singularity. The convergence for the adaptive grids is similar to that of the finest uniform grid.

unit square			re-entrant corner			re-entrant corner (adaptive)		
level	unknowns	iter.	level	unknowns	iter.	max. level	unknowns	iter.
17	66 049	22	15	24 897	24	20	153601	28
19	263 169	21	17	98 945	26	20	299479	28
21	1 050625	21	19	394 497	27	23	163531	28
23	4 198 401	21	21	1 575 425	29	23	347494	29

Table 2: PCG iterations required to reduce the error to (single precision) machine accuracy.

6 Current and Future Work

The potential of the presented algorithm lies in its ability to dramatically reduce the memory requirements for numerical simulation on adaptive grids. For example, on a computational grid of refinement level 20, consisting of approx. 500 000 unknowns, the memory requirements for a simple CG iteration is reduced from 126 MB to only 21 MB for the cell oriented approach (and 34 MB for the cell-centered PCG), because the system matrix is no longer stored explicitly.

We are currently working to reduce `amatos`' internal data structure to a memory-efficient storage of the bisection-tree, only, which should lead to an even more dramatic reduction of memory requirements. Currently, `amatos` requires over 900 MB to store neighbourship relations of the grid cells. The results gained from the prototype implementation in [BZ06] indicate that it is possible to get by with less than 10 MB.

Our goal for the presented approach is therefore to make it possible to use fully adaptive grids containing numbers of unknowns that can otherwise only be reached when strongly structured grids are used that offer only a very limited potential for adaptive refinement. In this paper, we demonstrated that it is possible to implement fast, multilevel methods on these memory-efficient data structures, which provides another key component towards this new approach to implement finite-element solvers.

References

[BRH+05] J. Behrens, N. Rakowsky, W. Hiller, D. Handorf, M. Läuter, J. Päpke, and K. Dethloff. Parallel adaptive mesh generator for atmospheric and oceanic simulation. *Ocean Modelling*, 10, 2005.

[BZ06] M. Bader and C. Zenger. Efficient storage and processing of adaptive triangular grids using sierpinski curves. In *Computational Science – ICCS 2006*, volume 3991 of *Lecture Notes in Computer Science*. Springer, May 2006.

[Gri94] M. Griebel. Multilevel algorithms considered as iterative methods on semidefinite systems. *SIAM Journal of Scientific and Statistical Computing*, 15(3), 1994.

[Yse86] H. Yserentant. On the multilevel splitting of finite element spaces. *Numerische Mathematik*, 49(3), 1986.