Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves

M. Bader*, S. Schraufstetter and C.A. Vigh

Institut für Informatik,
TU München, Germany
E-mail: bader@in.tum.de
E-mail: schraufs@in.tum.de
E-mail: vigh@in.tum.de
*Corresponding author

J. Behrens

Alfred Wegener Institute for Polar and Marine Research,
Bremerhaven, Germany
E-mail: Joern.Behrens@awi.de

Abstract: We will present an approach to numerical simulation on recursively structured adaptive discretisation grids. The respective grid generation process is based on recursive bisection of triangles along marked edges. The resulting refinement tree is sequentialised according to a Sierpinski space-filling curve, which leads to both minimal memory requirements and inherently cache-efficient processing schemes. The locality properties induced by the space-filling curve are even retained throughout adaptive refinement of the grid. We demonstrate the efficiency of the approach by implementing a multilevel-preconditioned conjugate gradient solver for a simple, yet adaptive, test problem: solving Poisson's equation on a re-entrant corner problem.

Keywords: adaptive grid generation; space-filling curves; cache efficiency; simulation.

Reference to this paper should be made as follows: Bader, M., Schraufstetter, S., Vigh, C.A. and Behrens, J. (2008) 'Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves', *Int. J. Computational Science and Engineering*, Vol. 4, No. 1, pp.12–21.

Biographical notes: Michael Bader is Senior Research Assistant at the chair of Scientific Computing in Computer Science (SCCS) at TU München (TUM). He received a PhD in Informatics at TUM in 2001. His research interests include hardware-aware algorithms in scientific computing based on space-filling curves.

Stefanie Schraufstetter received a Degree in Mathematics at TUM in 2006; her degree thesis was part of the work presented in this paper. She is now working on her PhD at SCCS/TUM on the topic of efficient numerical algorithms for PDE.

Csaba A. Vigh received a Degree in CSE at TUM in 2007. He holds a PhD scholarship within the International Graduate School of Science and Engineering at TUM, and is part of the research training group *Hardware-aware simulation and computing*.

Jörn Behrens received a PhD in Mathematics at University of Bremen in 1996, and finished his Habilitation at TUM in 2006, where he was working on adaptive atmospheric modelling. Currently, he heads the tsunami modelling research group at Alfred Wegener Institute, Bremerhaven.

1 Introduction

One of the most common approaches to modelling and simulation in Computational Science is based on Partial Differential Equations (PDEs) and their numerical treatment with Finite Element or similar methods. From a user's point of view there are usually only a few simple demands on a successful simulation code: it has to be fast, accurate, and easy to use. Of course, the two demands *fast* and *accurate* already generate a remarkable list of features that have to be implemented:

- To increase accuracy requires refinement of the grid. In particular, the need for *adaptive refinement* will arise wherever memory is too short or computing

time is too long to allow the use of uniformly refined grids.

- Nevertheless, the computational grid should be stored efficiently. To achieve this, a *structured grid* (at least to some extent) will generally help to reduce the amount of grid information that has to be stored.
- Fast solution of the systems of equations obtained from the discretisation process often requires *multigrid* or multilevel methods.
- Moreover, a code which has an optimal computing time in theory, may still not behave well on a real computer: aspects like *cache efficiency* and *vectorisation* can have an enormous influence on the efficiency of an algorithm.
- In addition, efficient *parallelisation* of the code will be essential on any high performance computer.

The list of features is further extended by requests for methods of higher order, treatment of complicated geometries, and probably several more.

Not surprisingly, these features are usually in conflict with each other. Therefore, there is probably no code that excels in all of the given 'disciplines'. Codes that are based on the use of unstructured grids have to invest precious amounts of memory for the storage of the grid structure and sacrifice cache efficiency and vectorisation properties for the sake of maximal freedom in adaptive refinement of the grid. On the other hand, codes on structured grids strive for speed and memory efficiency, but have to follow a laborious path to include adaptivity. Very often, only block structured approaches can be included without compromising the intrinsic advantages of the structured grids–see for example Bergen et al. (2006).

Recently, our research group have presented a family of methods based on recursively substructured grids (similar to octrees), which allow the implementation of iterative multigrid solvers on fully adaptive grids, but only require a marginal amount of memory–see Günther et al. (2006) and Mehl et al. (2006) for rectangular grids, and Bader and Zenger (2006) for triangular grids. For efficient processing on these grids, the presented schemes combine the use of space-filling curves (Peano curves or Sierpinski curves, respectively) and a sophisticated scheme of stacks, and thus implement iterative solvers in a way such that almost no topological information needs to be stored and that it is no longer necessary to explicitly store a system of equations. Moreover, the stack-based memory access leads to excellent cache efficiency, and parallelisation strategies are readily available by use of well-established techniques based on space-filling curves, such as presented by Zumbusch (2001) or Mitchell (2007), for example.

In this paper, we will present first experiences and results from the integration of this approach into the grid generator `amatos` (Behrens et al., 2005). `amatos` combines a grid generator and a respective Finite Element library, which is primarily intended for applications related to ocean and atmospheric modelling. The grid generation technique adopted in `amatos` is that of recursive bisection of triangles along *marked edges* (Mitchell, 1991; Behrens, 2005). `amatos` already used the Sierpinski space-filling curve for parallelisation, which made it an ideal candidate for integrating the memory-efficient, stack-based approach.

The aim of this project was to greatly reduce the storage requirements of `amatos` to make it, in that aspect, competitive with packages that are restricted to strongly structured grids. On the other hand, the potential for adaptive refinement was required to remain unaffected. Even more, the new implementation was intended to be able to cope with strongly dynamically adaptive grids, where the focus of adaptive refinement may change rapidly during a simulation. In Section 2, we will give a short introduction to the grid-generation technique, the sequential storage scheme based on the Sierpinski space-filling curve, and the stack-based element-wise processing of unknowns on these grids. Afterwards, in Section 3, we will present an algorithm for refinement of the conforming grid. In Section 4, we will discuss the integration of a multigrid-preconditioned conjugate gradient method as a fast iterative solver. Section 5 will present results for an example problem, before Section 6 gives some conclusions and an outlook to current and future work.

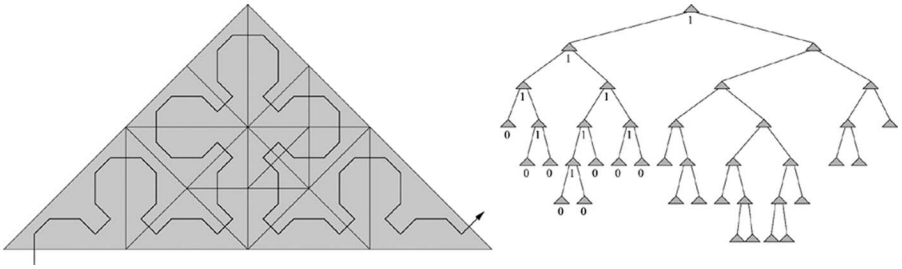
2 The Sierpinski-curves approach

2.1 Generating and storing recursively structured adaptive grids

`amatos` generates Finite Element meshes that are constructed from recursive bisection of triangular grid cells. Starting from a triangular parent cell, each cell of the generated computational grid is recursively subdivided at a *marked edge* until a certain resolution or level of adaptive refinement is reached. This grid generation technique was also, for example, introduced and discussed by Mitchell (1991), and is also referred to as *newest-vertex* bisection. In this paper, we will restrict the discussion to using right triangular grid cells, only. Then the hypotenuse defines the marked edge in each grid cell. However, the cells are actually allowed to be quite arbitrarily shaped, as long as the topological structure of the recursive bisection is preserved (Behrens, 2005).

The recursive structure motivates the representation of such grids using a corresponding binary tree (see Figure 1), which we will call a *refinement tree*. The refinement tree in Figure 1 is organised such that a depth-first traversal of it will generate a sequential order on the grid cells that is equivalent to the order given by a Sierpinski space-filling curve (Sagan, 1994). Using this linearisation of the refinement tree, we can store the corresponding adaptive grid by representing the refinement information, only. To store whether a corresponding grid cell is refined or not, only one bit per node of the refinement tree is sufficient. Hence, we can use a bit vector to store this linearised refinement tree, and thus the structure of our

Figure 1 Recursively constructed triangular grid and its corresponding refinement tree. The Sierpinski order, here depicted by a regularly refined iteration of the Sierpinski curve, corresponds to a depth-first traversal of the tree. In the refinement tree, refined nodes are labelled by 1; leaf nodes by 0. The data structure to store the refinement tree is based on a traversal of this 0/1-information



adaptive grid. For the refinement tree given in Figure 1, this bit vector would start as follows:

1 1 10100 111000100 . . .

(compare the labels in Figure 1; only the left half of the refinement tree is labelled and represented here).

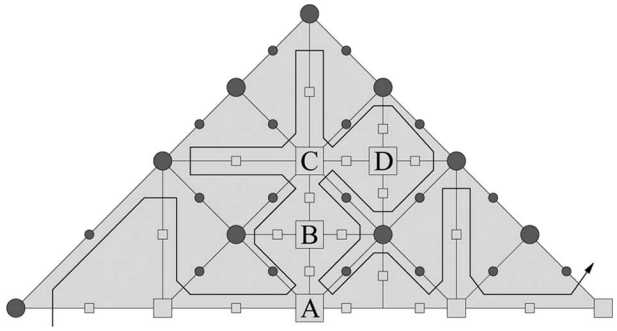
Such data structures – either explicit or linearised refinement trees – were, up to now, at best used for grid generation, because implementation of numerical algorithms requires knowledge on the neighbour relationships between cells, nodes, and edges to be able to evaluate local discretisation stencils, for example. However, from the tree structure alone, these neighbour relationships are not easily available. Therefore, most grid generators that use adaptive grids, including the existing version of `amatos`, invest precious amounts of memory to store these neighbour relations explicitly.

2.2 Using Sierpinski curves for element-wise processing

The algorithmic scheme presented in Bader and Zenger (2006) enables us to implement iterative solvers without the need to store neighbour relationships. The respective algorithm is based on traversals of the refinement tree. Hence, processing of the adaptive grid is not performed node-by-node, but always cell-by-cell, where the cells are visited in the order given by the Sierpinski space-filling curve. For this cell- or element-based processing, we also require an element-oriented discretisation of the underlying problem. In Finite Element discretisations, the element stiffness matrices and corresponding right-hand sides will provide such element-oriented discretisations automatically. A global system of equations will never be assembled. Instead, we work directly on the element systems. All operations that require the discretised operator, such as the computation of residuals, e.g., will therefore compute local contributions on each element, which then have to be accumulated for the entire grid.

Thus, in each cell, we need to access its respective unknowns, which may be located on nodes, edges or in the interior of the cells. Unknowns on nodes and edges will, during the accumulation process, be accessed by all adjacent grid cells. Thus, intermediate results have to be stored. Figure 2 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 that are left neighbours of the sequence A–D, but are accessed in descending order, when cells to the right of A–D are processed. We see from Figure 2 that two stacks are required for the intermediate results: a *red* stack for unknowns left of the curve, and a *green* stack for unknowns right of the curve. Note that this classification and also the stack-based access still works, when unknowns on cell edges are added. For that purpose, we represent the Sierpinski curve by an adaptive approximating polygon that enters and leaves a grid cell close to the nodes adjacent to the hypotenuse – we will call these nodes *entry node* and *exit node*, respectively. Then, also unknowns on edges are clearly assigned to either the red stack or the green stack, and the stack system can be used to store intermediate values for edge unknowns, as well.

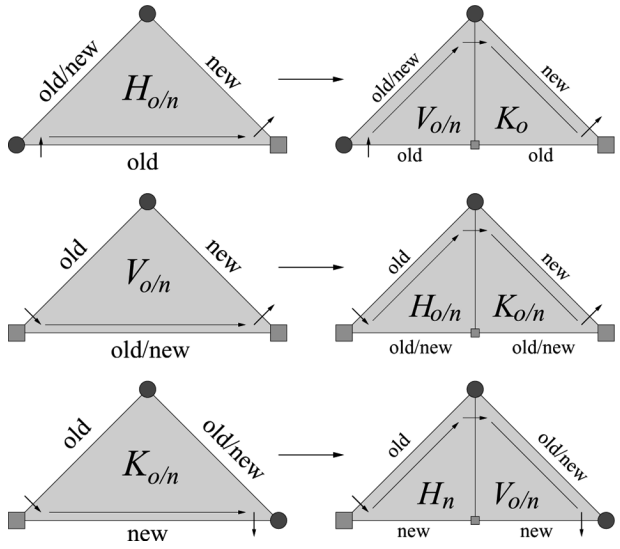
Figure 2 Colouring of the unknowns for a stack-based access: unknowns denoted by discs will be stored on the *red* stack; unknowns denoted by boxes will be stored on the *green* stack



The final algorithm uses two additional streams for input and output data. In iterative schemes, the input stream will contain the current values of the unknowns before their processing. Variables will be put onto the output stream, once their entire processing in an iterative step has been completed. To decide from which stack or stream an unknown is to be retrieved, or whether it needs to be put onto a coloured stack or onto the output stream, we require a suitable set of rules. The appropriate colour of a stack used for buffering a variable can be derived from the local course of the Sierpinski curve within the current

cell. 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 3). The element types enable us to decide for all unknowns, whether they should be buffered on the *red* or on the *green* stack.

Figure 3 The 2×3 different element-types and the respective types of the child cells. The element-types are determined by where the Sierpinski curve enters and leaves the cell



In addition, we require information on whether an unknown has already been accessed during a traversal or whether it will have to be updated within another cell. If an unknown is used for the first time, we will obtain it from the input stream. In the same way, if an unknown's processing during the current iteration step is finished, we will put it onto the output stream. For that purpose, we mark the edges of each cell: if the adjacent neighbour cell has been already processed, the respective edge is marked as *old*, otherwise as *new*.

Hence, we obtain the following set of rules for unknowns on cell corners:

- if a cell corner is adjacent to two *new* edges, the respective unknown is obtained from the input stream
- if a cell corner is adjacent to two *old* edges, the respective unknown will be put onto the output stream
- otherwise the respective unknowns will be buffered on or retrieved from the colour stacks.

For unknowns on cell edges, we obtain the following rules:

- unknowns on a *new* edge will be obtained from the input stream, and will be put onto a colour stack after being updated
- unknowns on an *old* edge will be obtained from a colour stack, and will be put onto the output stream after being updated.

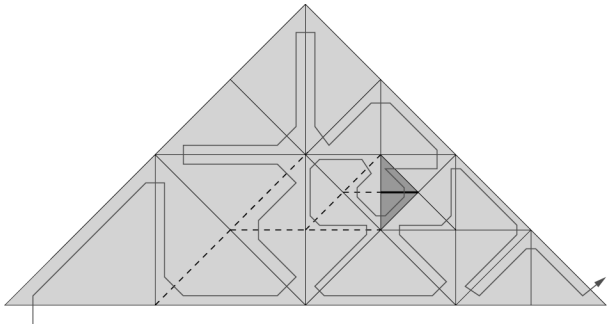
Unknowns in the interior of a cell are used in this cell only. Thus, input and output stream are sufficient for their processing.

The element types V, K, and H can be augmented to include the old/new-classification (see the indices in Figure 3), such that these rules can be enforced according to these six augmented element types, only. During bisection, the element type for each generated child cell is inherited from the parent's element type. In the final implementations, one nested-recursive procedure will be generated for each of the six element types, and each recursive procedure will implement the desired iterative scheme or grid manipulation task for one given element type.

3 Adaptive refinement

In `amatos`, the generated adaptive triangular grids are always *conforming* grids, which means that so-called *hanging nodes* are forbidden. Hanging nodes would occur, if only one of the two cells adjacent to a marked edge were bisected. Thus, to avoid hanging nodes requires communication during refinement of the adaptive grid: if a given cell is chosen for refinement, usually one of the adjacent cells will have to be refined as well to preserve conformity. This forced refinement might even force refinement of a further cell, and so on. Thus, a cascade of refinement operations may be initiated. Figure 4 shows such an example.

Figure 4 Refinement cascade: the requested refinement of the dark-coloured cell (thick line) forces the refinement of four further cells (dashed lines)



Following and executing such a refinement cascade in our linearised refinement tree would be a rather inefficient operation. Besides, in a typical adaptive simulation we will usually mark an entire set of grid cells for refinement. Our approach is therefore based on a refinement traversal with the following 'features':

- The traversal will take a refinement vector as parameter, which can mark any given grid cell for refinement.

- Information on refinement will be communicated using the colour stacks. Each edge carries a respective unknown that serves as a refinement marker, and thus helps to eliminate hanging nodes.
- During the refinement traversal, the additional cells generated by bisection can be integrated into the linearised refinement tree such that the Sierpinski order is also preserved in the refined grid.

If only a single traversal would be performed for refinement, only those cells of a refinement cascade visited after the initiating cell could actually be refined. But the example given in Figure 4 shows that a cascade may well contain cells in both upward and downward direction. The cells visited before the initiating cell would consequently not have obtained the necessary refinement information, yet. The entire treatment of all refinement cascades therefore requires two traversals of the refinement tree–one forward traversal, whose main task is to mark cells for refinement, and one backward traversal, which in addition has to execute the actual refinements by updating data structures and interpolating the solution.

Hence, the forward traversal's first task is to determine cells that have to be adaptively refined for numerical reason, which can be either determined by a refined vector given as parameter, or the result of an integrated error estimator. In addition, the forward traversal will also distribute refinement information across cell edges to ensure conformity of the grid. Each edge will therefore carry a boolean variable that represents whether it will be bisected in the refined conforming grid. We again use the stack approach to synchronise these boolean variables, and thus transport refinement information to neighbouring cells. Altogether, the following actions have to be performed for each cell during the forward traversal:

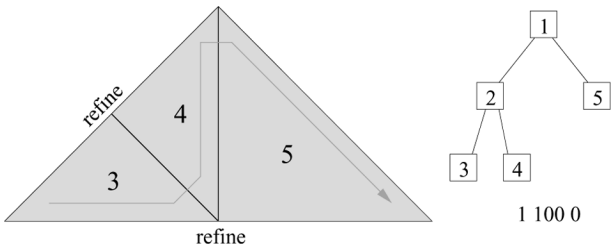
1 If the cell is determined for refinement by the refinement vector or an error estimator, the tagged edged will be marked for refinement
2 For all edges that are *old*, i.e., which are adjacent to cells that have already been visited during the traversal, the refinement info is obtained from the respective colour stack
3 The refinement status of the cell is updated according to Steps 1 and 2
4 For all edges that are *new*, and will therefore be visited again during the traversal, their refinement status will be stored on the respective colour stack.

The backward traversal first has to distribute refinement information also in the opposite direction and, thus, complete the process of ensuring conformity of the grid. Again, refinement of edges is communicated via the colour stacks. In addition, the backward traversal can finally execute the actual refinement of the grid. The bitstream representation of the refinement tree has to be updated to include the new grid cells. At the same time, the values of the new unknowns can be computed by interpolation, for example. Thus, the backward traversal consists of the following actions for each cell:

1 For edges that are *old*, the refinement info is obtained from the respective colour stack
2 The refinement status of the cell is updated, and now determines the final refinement of the cell
3 For edges that are *new*, the refinement status will be stored on the respective colour stack to ensure backward propagation of refinement information
4 The bitstream representation of the refinement tree is updated according to the cell's refinement status, and the new unknowns could be computed by interpolation, if required.

Figure 5 gives an example for Step 4. A virtual traversal of the newly generated cells determines where to and in which order the new data has to be written.

Figure 5 Refinement of a single cell: refinement info on edges, the respective refinement subtree, and its bitstream representation



4 A multigrid solver

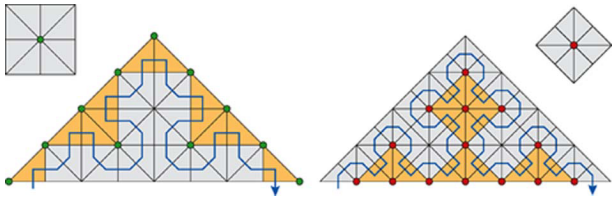
The recursive construction of the computational grid leads directly to a hierarchy of coarse grids, which can be used to design a multigrid algorithm. During the depth-first traversal of the refinement tree, all coarse grid cells will be automatically visited. Hence, implementing an additive multigrid 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 (1994). The respective ansatz functions can be chosen according to the hierarchical basis functions introduced by Yserentant (1986).

4.1 Multigrid hierarchies on recursively structured triangular grids

In the multigrid hierarchy, the coarse grid cells must not carry unknowns on each of their nodes. Otherwise, too many coarse grid unknowns would be generated. As illustrated in Figure 6, each multigrid level consists of two levels of recursion in the refinement tree, instead. 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 coarse-grid 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, carry unknowns.

Figure 6 Two subsequent refinement levels with the respective multilevel unknowns. Note the different orientation of the basis functions and their supports (see online version for colours)



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 unknowns are associated with a unique basis function; if unknowns were defined on each node of every cell, duplicate basis functions would be generated. The respective duplicate unknowns are not only superfluous, but would also interfere with the stack principle by blocking fine-level unknowns on the respective stack.

On the finest levels, however, this determination of coarse-grid unknowns is in conflict with the conforming grid approach: on the finest level, every node will carry an unknown. As a node might well carry a coarse-grid unknown on the penultimate level, as well, the respective node would carry unknowns on two subsequent levels of the refinement tree. This would again cause problems with the stack approach, hence, we enforce a so-called *level condition*, which requires that a node must not carry a coarse-grid unknown, if it will be a fine-grid unknown in the next level of the refinement tree.

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. We solved this problem by simply storing the number of existing coarse-grid levels for each node of the finest grid.

4.2 *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. 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 taken from the input stream, or just before an unknown is written to the output stream. Node updates serve to compute vector operations, which, naturally, can not be implemented in an element-oriented manner. Table 1 shows how the PCG's individual operations are mapped to element and node updates throughout the two traversals.

Table 1 Element and node updates during the cell-oriented implementation of the PCG method. Operations to initialise vectors have been left out to improve readability

Cell-oriented PCG	*Regular PCG algorithm*
1st traversal:	for $k = 0, 1, 2, \ldots$:
node update: $d_e \leftarrow z_e + \beta d_e$	
element update: $a_e \leftarrow a_e + A_e d_e$	
$\delta \leftarrow \delta + d_e^T A_e d_e$	$a^{(k)} = Ad^{(k)}$
restriction: $A_e d_e$ to coarse grid	$\delta = (d^{(k)})^T Ad^{(k)}$
$\alpha = \gamma_1/\delta$	$\alpha = (r^{(k)})^T z^{(k)}/\delta = \gamma_1/\delta$
2nd traversal:	
node update: $x_e \leftarrow x_e + \alpha d_e$	$x^{(k+1)} = x^{(k)} + \alpha d^{(k)}$
$r_e \leftarrow r_e - \alpha a_e$	$r^{(k)} = r^{(k)} - \alpha a^{(k)}$
prolongation: c_e from coarse grid	$z^{(k+1)} = M^{-1} r^{(k+1)}$
	$= z^{(k)} - \alpha M^{-1} a^{(k)}$
$z_e \leftarrow z_e - \alpha c_e$	
$\gamma_2 \leftarrow \gamma_2 + r_e z_e$	$\gamma_2 = (r^{(k+1)})^T z^{(k+1)}$
$\beta \leftarrow \gamma_2/\gamma_1; \gamma_1 \leftarrow \gamma_2; \gamma_2 \leftarrow 0$	$\beta \leftarrow \gamma_2/\gamma_1; \gamma_1 \leftarrow \gamma_2$
	$d^{(k+1)} = z^{(k+1)} + \beta d^{(k)}$

5 Numerical examples

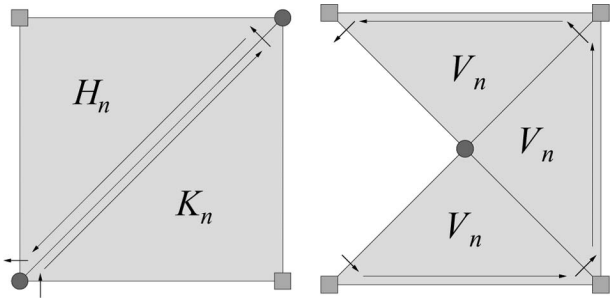
We tested the presented iterative algorithm on two common example problems: solving Poisson's equation,

$$-\triangle u(x, y) = f(x, y), \tag{1}$$

with homogeneous Dirichlet boundary conditions on the two domains given in Figure 7 – the unit square and a domain with a re-entrant corner featuring a point singularity. The right-hand-side function was chosen to be $f(x, y) = 2(x + y - x^2 - y^2)$, which leads to an exact solution of $u(x, y) = xy(1 - x)(1 - y)$ on the unit square. As illustrated in Figure 7, 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.

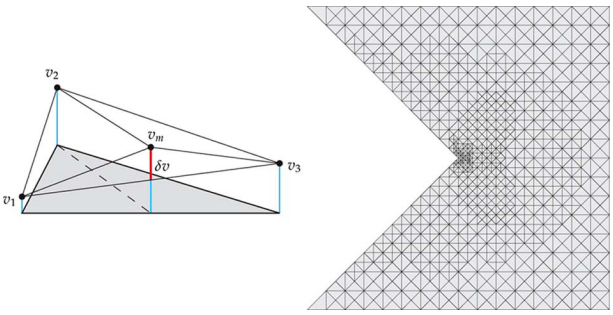
We used a Finite Element discretisation, where the element stiffness matrices and right hand sides were computed from linear test and basis functions on the triangular elements. As all cells of the discretisation grids can be classified into only a few different types of element shapes, the element systems needed only to be computed and stored once for each type.

Figure 7 Computational domains and their initial triangulations. The element types of the initial triangles are chosen as to ensure the correctness of the stack-based access



For the re-entrant corner problem, we set up an a-posteriori adaptive computation in addition to computations on uniformly refined grids. We used a simple approach based on computing the hierarchical surplus as an a-posteriori error estimator. In a given cell, the hierarchical surplus was computed as the difference between the linear interpolation of the two unknowns adjacent to the hypotenuse and the finest-level unknowns sitting on the midpoint (see Figure 8). This surplus was computed in all cells of the refinement tree that only had leaf cells as children; if the surplus was larger than a given threshold, both child cells were marked for refinement. For Poisson's equation, this simple approach is known to be a suitable error estimator (Deuflhard et al., 1989) and the given refinement tolerance may be used to control the accuracy of the computed solution. A resulting a-posteriori adaptive grid is shown in Figure 8. In our examples, we used refinement tolerances from 10^{-5} down to 10^{-7}; we started with uniformly refined grids of level 15, and allowed adaptive refinement up to level 30.

Figure 8 Computation of the hierarchical surplus as error estimator and a resulting adaptive mesh for the re-entrant-corner domain (see online version for colours)



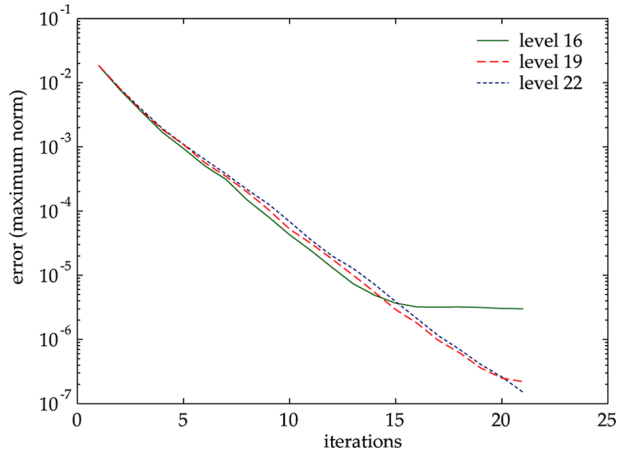
5.1 Performance of the multigrid preconditioner

We will first present convergence results for the multigrid PCG algorithm. Respective experiments were performed on uniformly refined grids of different resolution on both computational domains. Table 2 lists the average convergence rates, i.e., the average reduction rate of the residual norm over 10 PCG iterations, for different numbers of unknowns. The convergence rates are in a range that can be expected for an additive multilevel method; about 25–30 PCG iterations are required to reduce the residual norm by a factor of 10^{-6}. Figure 9, in addition, shows the reduction of the actual error throughout the PCG iterations, which is consistent with the reduction of the residuals. The convergence for the re-entrant corner problem deteriorates noticeably, if the resolution of the uniform grid is refined, which we believe to be an effect of the point singularity.

Table 2 Average convergence rates over 10 PCG iterations on uniformly refined grids with different resolution

Unit square			*Re-entrant corner*		
Level	*Unknowns*	*Conv.*	*Level*	*Unknowns*	*Conv.*
18	263,169	0.571	18	197,633	0.800
20	1,050,625	0.568	20	788,481	0.830
22	4,198,401	0.572	22	3,149,825	0.851
24	16,785,409	0.607	24	12,591,105	0.869

Figure 9 Reduction of the maximum norm of the error for the unit-square Poisson problem (see online version for colours)



Thus, for the re-entrant corner problem we additionally determined the convergence rates for computations on a-posteriori refined grids resulting from different refinement tolerances. Table 3 again lists the average convergence rates of the PCG iterations. The convergence is much better than what was achieved on the uniform grids for the re-entrant corner problem, and the speed of convergence is only slightly worse than that for the unit square problem using uniformly refined grids. Hence, our multigrid preconditioner proves to be reasonably efficient for our test problems.

Table 3 Average convergence rates for the PCG solver on a-posteriori refined grids with different refinement tolerance

Re-entrant corner (adaptive)		
Tolerance	*Unknowns*	*Conv. rate*
1.0e-05	179,439	0.684
3.0e-05	432,640	0.629
1.0e-06	959,190	0.650
3.0e-06	2,594,456	0.671
1.0e-07	6,455,242	0.688

5.2 Computational efficiency

In Table 4, we list the memory requirements of the current implementation. Due to the integration of the stack-based processing with the storage of the grid structure, we cannot strictly classify memory requirement into parts that are used for grid storage, only, and parts that are used for other organisational tasks within the algorithm. Hence, we compare the memory requirement when using single vs. double precision floating point variables for the unknowns. From the difference of these two values, we can infer the total memory requirement for grid storage, grid manipulation, and implementation of traversal algorithms. For uniform refinement, this 'integer part' of the memory requirement is about 10 bytes per unknowns, and seems to consolidate at only around 8 bytes per unknown for very fine grids. For adaptively refined grids, the 'integer part' of the memory requirements is a bit larger. For example, we require additional stack systems to hold information about the refinement status of each edge of the grid to ensure conforming grid refinement. The total memory requirement, as observed from the measurements listed in Table 4, is about 20–30 bytes per unknown. Our theoretical estimates for the current implementation indicate that approximately 26 bytes per unknown are needed for grid generation and processing. This could be further reduced, for example by using single bits instead of full bytes for boolean data, to less than 10 bytes per unknown.

Table 5 lists the runtimes for a single PCG iteration on uniformly refined grids of different resolution and for several adaptively refined grids with varying refinement tolerance. All computations were executed on a personal computer equipped with an Intel Pentium4 processor (Prescott, 3.4 GHz, 1 MB L2-cache) and 2 GB of main memory. Results are given for single and double precision, respectively. For both, single and double precision, we computed the execution time per unknown.

The almost identical execution times per unknown demonstrate that performance does not drop for larger problem sizes, which is a result of the excellent cache performance of the algorithm (for the level 2 cache, we

Table 4 Memory requirement for the re-entrant corner problem on uniform (above) and adaptively refined (below) grids with different refinement level and refinement tolerance (for adaptive grids). Given is the memory requirement (total and per unknown) when using single or double precision variables for the unknowns. The integer part reflects the amount of memory used for grid storage, grid processing, and other administrative tasks

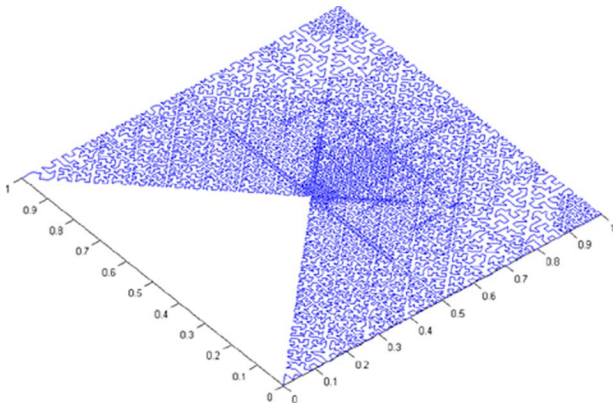
Refinement		*Number of*	*Number of*	*Memory* [MB]			*Bytes per unknown*		
Level	*Tol.*	*grid cells*	*unknowns*	*Float*	*Double*	*Int. part*	*Float*	*Double*	*Int. part*
18	–	393,216	197,633	7.2	12	2.4	38.2	63.7	12.7
20	–	1,572,864	788,481	26	47	5	34.6	62.5	6.6
22	–	6,291,456	3,149,825	105	186	24	35.0	61.9	8.0
24	–	25,165,824	12,591,105	420	741	99	35.0	61.7	8.2
15–30	1.0e-05	357,588	179,439	21	35	4	122	204	23.4
15–30	3.0e-05	863,296	432,640	34	56	12	82	136	29.1
15–30	1.0e-06	1,915,368	959,190	60	96	24	65.4	105	26.2
15–30	3.0e-06	5,184,010	2,594,456	145	219	61	57.8	88.5	24.3
15–30	1.0e-07	12,902,390	6,455,242	366	510	222	54.6	82.8	33.1

Table 5 Runtime for a single PCG iteration using single or double precision floating point variables. The results are presented for the same problems as used for Table 4

Ref.	*Number of*	*Runtime single precision*		*Runtime double precision*	
level	*unknowns*	*Per iteration* (s)	*Per it. and unknown* (μs)	*Per iteration* (s)	*Per it. and unknown* (μs)
18	197,633	0.59	2.99	0.68	3.43
20	788,481	2.33	2.96	2.74	3.48
22	3,149,825	9.30	2.95	10.8	3.42
24	12,591,105	38.0	3.02	42.4	3.37
15–30	179,439	0.54	2.99	0.61	3.40
15–30	432,640	1.29	2.99	1.48	3.41
15–30	959,190	2.88	3.00	3.28	3.42
15–30	2,594,456	7.89	2.99	8.84	3.41
15–30	6,455,242	21.2	3.01	22.2	3.44

measured hit rates of well over 99%). These excellent cache hit rates are a result of the preserved locality of the unknowns even after strong adaptive refinement, which is illustrated in Figure 10.

Figure 10 Sequential order of the unknowns in an adaptively refined grid. The sequence of the unknowns is identical to that on the output stream (see online version for colours)



Interestingly, there is only a minor difference between runtimes for single vs. double precision. It is definitely much smaller than the factor 2 that can often be observed in such situations. This reveals the fact that performance is to a considerable part dominated by the overhead to implement the tree traversals, and by the data handling on stacks and streams. Floating point performance is therefore limited, an effect which was also observed in similar approaches on quadrilateral grids (Mehl et al., 2006), but which was not yet the focus of this paper.

A further aspect of the overall performance is the computation time spent to set up and adaptively refine the computational grid. In the original version of `amatos`, the time to build the initial grid and set up the global system matrix was, even for small grids, usually more expensive than to compute the actual solution. For larger grids, this part more and more dominated the total computation time, apparently because poor cache efficiency during grid generation affected the relative performance for large grids. In contrast, the computation time to build a uniform initial grid using the new approach is, even for large grids, typically less than 5% of the time needed to perform one PCG iteration. Also for adaptive grids, the total amount of time spent for grid generation and conforming refinement only accounts for 10–15% of the total runtime. In the two largest adaptive examples given in Tables 4 and 5 (double precision), the computational time spent for the most expensive adaptive refinement step was 13.5 s (grid with 2,594,456 unknowns) and 34.78 s (grid with 6,455,242 unknowns), resp., to fulfil all refinement requests, ensure conformity of the grid, and insert the new unknowns with their interpolated values into the input/output stream. Hence, with respect to computation time, one refinement operation is equivalent to approximately 1.5 PCG iterations. Moreover, there is still some potential to reduce these refinement costs. For the presented examples, each refinement step usually required four grid traversals, which could be reduced to two traversals (the four-traversal implementation also allows for coarsening of the grid, which is not needed for the present test problem).

6 Conclusion

We presented an approach for the generation of recursively structured, adaptive grids and the implementation of multigrid algorithms on these grids. The locality properties of the Sierpinski space-filling curve are used for memory-efficient storage of the adaptive grid. Moreover, its locality properties lead to an inherently cache-efficient algorithm. Finally, we demonstrated that it is possible to implement fast, multilevel methods on these memory-efficient data structures, which provides another key component towards highly efficient Finite Element solvers.

As a general approach to solve partial differential equations, the presented algorithms, above all, make it possible to use fully adaptive grids containing numbers of unknowns that can otherwise only be reached when strongly structured grids with a very limited potential for adaptive refinement are used. In addition, we will be able to use the Sierpinski curve for load balancing during the planned parallelisation of the simulation code. For that purpose, the block-recursive scheme also provides a rather straightforward interface for the integration of domain decomposition methods.

In our opinion, the presented approach will be especially useful in problems, where highly dynamic adaptive refinement is required, i.e., where the hotspots of adaptive refinement constantly change their position. Then, approaches where grid generators and problem solvers are decoupled would suffer from a severe overhead, if, for example, a system of equations has to be set up explicitly every time the grid refinement changes. Also, implementing computations directly on adaptive grids is only feasible, if certain locality properties of the grid cells can be preserved. Otherwise, cache efficiency will quickly become a major problem. In the presented approach, the stack- and stream-based memory access ensures a near-optimal locality not only for the grid data-structure, but also for the respective unknowns during the computation. And, most importantly, this locality will be preserved even for strong and dynamically changing adaptive refinement.

References

Bader, M. and Zenger, C. (2006) 'Efficient storage and processing of adaptive triangular grids using sierpinski curves', *Computational Science – ICCS 2006*, Lecture Notes in Computer Science 3991, pp.673–680.

Behrens, J., Rakowsky, N., Hiller, W., Handorf, D., Läuter, M., Päpke, J. and Dethloff, K. (2005) 'Parallel adaptive mesh generator for atmospheric and oceanic simulation', *Ocean Modelling*, Vol. 10, pp.171–183.

Behrens, J. (2005) 'Adaptive atmospheric modeling – scientific computing at its best', *Computing in Science and Engineering*, Vol. 7, No. 4, pp.76–83.

Bergen, B., Gradl, T., Hülsemann, F. and Rüde, U. (2006) 'A massively parallel multigrid method for finite elements', *Computing in Science and Engineering*, Vol. 8, No. 6, pp.56–62.

Deuflhard, P., Leinen, P. and Yserentant, H. (1989) 'Concepts of an adaptive hierarchical finite element code', *IMPACT of Computing in Science and Engineering*, Vol. 1, pp.3–35.

Griebel, M. (1994) 'Multilevel algorithms considered as iterative methods on semidefinite systems', *SIAM Journal of Scientific and Statistical Computing*, Vol. 15, No. 3, pp.547–565.

Günther, F., Mehl, M., Pögl, M. and Zenger, C. (2006) 'A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves', *SIAM Journal of Scientific Computing*, Vol. 28, No. 5, pp.1634–1650.

Mehl, M., Weinzierl, T. and Zenger, C. (2006) 'A cache-oblivious self-adaptive full multigrid method', *Numerical Linear Algebra with Applications*, Vol. 13, Nos. 2–3, pp.275–291.

Mitchell, W.F. (1991) 'Adaptive refinement for arbitrary finite-element spaces with hierarchical bases', *Journal of Computational and Applied Mathematics*, Vol. 36, pp.65–78.

Mitchell, W.F. (2007) 'A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids', *Journal of Parallel and Distributed Computing*, Vol. 67, No. 4, pp.417–429.

Sagan, H. (1994) *Space Filling Curves*, Springer, New York, Heidelberg, Berlin.

Yserentant, H. (1986) 'On the multilevel splitting of finite element spaces', *Numerische Mathematik*, Vol. 49, No. 3, pp.379–412.

Zumbusch, G. (2001) 'Adaptive parallel multilevel methods for partial differential equations', *Habilitationsschrift*, Universität Bonn.