Efficient Storage and Processing of Adaptive Triangular Grids Using Sierpinski Curves

M. Bader and Ch. Zenger

Dept. of Informatics, TU München, 80290 München, Germany

Abstract. We present an algorithm to store and process fully adaptive computational grids requiring only a minimal amount of memory. The adaptive grid is specified by a recursive decomposition of triangular grid cells; the cells are stored and processed in an order that is given by Sierpinski's space filling curve. A sophisticated system of stacks is used to ensure the efficient access to the unknowns. The resulting scheme makes it possible to process grids containing more than one hundred million cells on a common workstation, and is also inherently cache efficient.

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 discretisation with finite element or similar methods. One of the key requirements in the generation of the respective computational grids, is the possibility for adaptive refinement. Unfortunately, introducing adaptive refinement most often leads to a trade-off between memory requirements and computing time. This is basically a result of the need to obtain the neighbour relationships between grid cells both during the grid generation and during the computation on the grids. Storing these neighbour relations explicitly allows arbitrary, unstructured meshes, but requires a considerable memory overhead. It is not uncommon for codes using unstructured grids to use up more than 1 kilobyte of memory per unknown, a large part of which is due to the explicit storage of the grid structure. In this paper, however, we want to address a situation where memory should be saved as far as possible, which requires the use of a strongly structured grid. The structure helps to save most of the memory overhead, but the neighbour relations now have to be computed, instead. In a strongly adaptive grid, this can be a rather time-consuming task.

In that context, Guenther[3], Mehl[4], and others have recently introduced an approach that allows full adaptivity using recursively substructured rectangular grids (similar to octrees). To store such adaptive grids requires only a marginal amount of memory. To efficiently process the grids they presented a scheme that combines the use of space-filling curves and a sophisticated scheme of stacks. The stack-like access even leads to excellent cache efficiency. Moreover, parallelisation strategies based on space-filling curves are readily available (see [8], for example).

In this paper, we will present a similar approach for grids resulting from a recursive splitting of triangles. Such grid generation strategies are, for example,

used by Stevenson[6] or Behrens et.al.[1]. Behrens even uses a space-filling curve (Sierpinski curve) for parallelisation, and observes an overall benefit which is a result of the locality properties induced by the space-filling curve. In this article, we will present an algorithm to efficiently process these kinds of triangular grids. The algorithmic scheme will be similar to that used by Guenther[3] and Mehl[4]: it is also based on the use of stack structures, and shows equally promising properties with respect to cache efficiency.

2 Recursively Structured Triangular Grids and Sierpinski Curves

For description of the grid generation, we stick to the simplest case and use a right, isosceles triangle as the computational domain. The computational grid is then constructed in a recursive process. Starting with a grid cell that consists of the entire computational domain, we recursively split each triangle cell into two congruent subcells. This recursive splitting is repeated until the desired resolution of the grid has been reached. The grid may be adaptive and may even contain hanging nodes—only in section 5 will we place some restrictions on the structure of the grid, which result from the application of a multigrid scheme.

Figure 1 shows a simple grid constructed by this scheme. The respective substructuring tree is shown next to it. Note that a respective uniformly refined recursive construction is used to define the so-called *Sierpinski curve*[5]. Thus, we can use the Sierpinski curve to generate a linear order on the grid cells (see Figure 1). This corresponds to a depth-first traversal of the substructuring tree. To store the grid structure therefore requires only one bit per node of the construction tree: this bit indicates whether a node (i.e. a grid cell) is a leaf or whether it is adaptively refined.

Fig. 1. Recursive construction of the grid on a triangular computational domain

There are several straightforward extensions to this basic scheme, which offer a quite flexible treatment of reasonably complicated computational domains[1]:

1. Instead of only one initial triangle, a simple grid of several initial triangular cells may be used.

2. The cells can be arbitrary triangles instead of right, isosceles ones as long as the structure of the recursive subdivision is not changed: one leg of each triangular cell will be defined as the *tagged edge* and take the role of the hypotenuse.
3. The subtriangles do not need to be real subsets of the parent triangle: whenever a triangle is subdivided, the tagged edge can be replaced by a linear interpolation of the boundary (compare Figure 2).

Fig. 2. Subdividing triangles at boundaries

3 Discretisation

After specifying the grid generation strategy, the discretisation of the given PDE is the next step in the simulation pipeline. Consider, for example, a discretisation using linear finite elements on the triangular grid cells. The discretisation will generate an element stiffness matrix and a corresponding right-hand side for each grid cell. Accumulation of these local systems will lead to a global system of equations for the unknowns, which are placed on the nodes of the grid.

Sticking to our overall principle to save memory as far as possible, we assume that storing either the local or the global systems of equations is considered to be too memory-consuming. Instead, we assume that it is possible to compute the stiffness matrix on the fly or even hardcode it into the software. Then our memory requirements are constricted to a minimal amount of overhead to store the recursive grid structure plus the inevitable memory we need to store the values of the unknowns.

Iterative solvers typically contain the computation of the matrix-vector product between stiffness matrix and the vector of unknowns as one of the fundamental subtasks. In a classical, node-oriented approach, this product is evaluated line-by-line using a loop over the unknowns. This requires, for each unknown, to access the unknowns of all neighbouring nodes. However, in a recursively structured grid, as it is described in section 2, not all neighbours will be easily accessible: a neighbour may well be part of an element that lies in an entirely different subtree. Therefore, the grid should be processed in a cell-oriented way, instead. Proceeding through the grid cell-by-cell, we can compute the local contributions of the respective element stiffness matrices, and accumulate the local contributions to compute a correction for the unknowns.

4 Cache Efficient Processing of the Computational Grid

In such a cell-oriented processing of grids, the problem is no longer to access all neighbours of the currently processed unknown. Instead, we need to access

all unknowns that are situated within the current element. A straightforward solution would be store the indices of the unknowns for each element, but the special structure of the grids described in section 2 offers a much better option: it requires that the elements are processed along the Sierpinski curve.

Then, as we can see in Figure 3, the Sierpinski curve divides the unknowns into two halves – one half lying to the left of the curve, the other half lying to the right. We can mark the respective nodes with two different colours, for example red and green. Now, if we process the grid cells in Sierpinski order, we recognize an interesting pattern in the access to the unknowns: the pattern is perfectly compatible with the access to a stack. Consider, for example the unknowns 5 to 10 in Figure 3. During the processing of the cells left of these unknowns, they are accessed in ascending order. During the processing of the cells to the right, they are accessed in descending order. In addition, the unknowns 8, 9, and 10 are in turn placed on top of the respective stack.

Fig. 3. Marching through a grid of triangular element in Sierpinski order. The nodes to the left and right of the curves are accessed in an order that motivates the use of a stack to store intermediate results.

In the final algorithm, a system of four stacks is required to organise the access to the unknowns:

- one read stack that holds the initial values of the unknowns;
- two helper stacks, a *green* stack and a *red* stack, that hold intermediate results for the unknowns of the respective colour;
- and finally a write stack to store the updated values of the unknowns.

In addition, whenever we move from a processed cell to the next one, two unknowns can always be directly reused, because they are adjacent to the common edge. Thus, we only have to deal with the two remaining unknowns – one in the cell that was just left, the other one in the entered cell –, both lying in a corner opposite to the common edge.

The remaining unknown in the exited cell will either be put onto the write stack (if its processing is already complete) or onto the helper stack of the correct

colour (if it still has to be processed by other cells). To decide whether the processing is completed, we can simply use a counter for the number of accesses. To decide the colour of the helper stack, we need to know whether an unknown is lying to the left of the curve or to its right. This decision can be easily made, if we know where the Sierpinski curve enters and leaves the triangle element. As the Sierpinski curve always enters and exits a grid cell at the two nodes adjacent to the hypotenuse, there are only three possible scenarios:

1. the curve enters through the hypotenuse – then it can only leave across the opposite leg (it will not go back to the cell it just left);
2. the curve enters through the adjacent leg and leaves across the hypotenuse;
3. the curve enters and leaves across the opposite legs of the triangle element.

Figure 4 illustrates these three scenarios, and also shows that obtaining the colouring of a node is then trivial.

Fig. 4. Three scenarios to determine the colouring of the edges: nodes to the left of the curve are *red* (circles), nodes to the right are *green* (boxes)

Next, we have to decide how to obtain the remaining unknown in the entered grid cell. It will be either be taken from the read stack, if it has not been used before, or otherwise from the respective coloured helper stack. This decision solely depends on whether the unknown has already been accessed before. Therefore, we consider whether the adjacent triangle cells have already been processed or not. For two out of the three neighbour cells, this property is already known: the cell adjacent to the entering edge has already been processed; the cell adjacent to the exit edge has not. The third cell can be either *old* (processed already) or *new* (not yet processed). Consequently, we split each of our existent three scenarios according to this additional criterion, and obtain six new scenarios, such as illustrated in Figure 5. The strategy where to obtain the remaining unknown is then straightforward:

- if at least one of the adjacent edges is marked as *old*, we have to fetch the unknown from the respective coloured stack;
- if both adjacent edges are marked as *new*, we have to fetch the unknown from the read stack.

This strategy works fine, even though it does not consider all elements adjacent to the node. Combinations that would make the scheme fail cannot occur due to the special recursive construction of the Sierpinski curve. Figure 5 also shows that the scenarios for the two respective subcells are always determined.

Fig. 5. Determination and recursive propagation of edge parameters

The processing of the grid can thus be managed by a set of six recursive procedures—one for each possible scenario. Each procedure implements the necessary actions in one cell of the grid tree. If the grid cell is subdivided into two child cells, the procedure will call the two respective procedures recursively. On the leaves, the operations on the discretisation level are performed.

5 Example: An Additive Multigrid Solver for Poisson’s Equation on an Adaptive Grid

To add a multigrid solver to the existing scheme, we adopted an approach based on a hierarchical generating system such as introduced by Griebel [2]. The respective ansatz functions were set up according to the hierarchical basis functions introduced by Yserentant [7]. The coarse level ansatz functions – that is the coarse level grids – are integrated into our recursive discretisation tree in a way, such that every node of the tree will contain a grid cell: either one of the finest level or one of a coarser level. Thus, the tree not only provides an adaptive computational grid, but at the same time provides a hierarchy of coarse grids for the multigrid method. When processing the grid cells on the finest level, all parent cells (i.e. the coarse grid cells) are automatically visited, as well. Therefore, an additive multigrid method can easily be integrated into the numerical scheme. Interpolation operations are invoked before the recursive calls to child cells; restriction operations are invoked after these calls are finished. According to the choice of linear finite elements, simple linear interpolation and full-weighted restriction was used.

However, not all of the cell corners should actually carry an unknown in the multilevel sense. Consider the node opposite to the hypotenuse of a cell: in the next refinement step, the adjacent right angle will be split, and the node will be adjacent to two rectangles of half size (belonging to the next finer level). Note that the support of the respective ansatz functions will be of the same extent on these two levels. It is therefore unnecessary to generate both of these duplicate unknowns – we chose to place unknowns only at corners that are adjacent to the hypotenuse; at the opposite corners (adjacent to the right angle) values will be interpolated and no unknowns will be situated there. As a result, a corner that

is adjacent to both right angles and acute angles would belong to two different levels. Such situations are therefore forbidden, and the grid generation strategy was modified to prevent these situations.

The resulting multigrid solver was tested for solving Poisson’s equation on a triangular domain of unit size. A respective a priori adaptive grid, such as illustrated in figure 6, was prescribed, and the equation was discretised using linear finite elements. The additive multigrid solver was able to achieve convergence rates between 0.7 and 0.8, largely independent of the resolution and adaptive refinement of the mesh. In our test runs, we applied 40 iterations of the multigrid method, which reduced the maximum error to about 10^{-4} . Table 1 lists the run-time and memory requirements for several sample grids. It is remarkable that the algorithms requires only about approximately 19 bytes of memory per degree of freedom. To process one multigrid iteration requires approximately 1.8 seconds per one million degrees of freedom.

Fig. 6. Computational domain and refined grid for the solution of Poisson’s equation (for this illustration, a low resolution grid was used)

Table 1. Run-time and memory requirements for several sample grids on a standard workstations (Pentium4, 3.4 GHz, 1GB main memory)

grid	depth		number of cells	degrees of freedom	memory [in MB]	proc. time [per iteration]
	min.	max.				
uniform	26	26	$67.1 \cdot 10^6$	$22.4 \cdot 10^6$	397	39.7 s
adaptive	24	28	$54.3 \cdot 10^6$	$18.1 \cdot 10^6$	323	32.9 s
adaptive	22	30	$69.5 \cdot 10^6$	$23.2 \cdot 10^6$	410	41.4 s
adaptive	12	33	$82.8 \cdot 10^6$	$27.6 \cdot 10^6$	487	49.4 s
adaptive	13	33	$91.3 \cdot 10^6$	$30.4 \cdot 10^6$	538	54.7 s
adaptive	24	30	$116 \cdot 10^6$	$38.8 \cdot 10^6$	684	69.4 s

6 Conclusion

We have presented an algorithm that is able to solve Poisson's equation on an *adaptive* grid of more than 100 million cells in just a few minutes on a common PC. Thus, it has shown to be a promising approach for using full adaptivity at a memory cost and (to a certain extent) computational speed that is competitive to algorithms that are based on regular grids. This is also due to the fact that multigrid methods can be easily integrated.

Encouraged by this proof-of-concept, we are currently implementing the algorithmic scheme for the grid generator *amatos*[1] to prove its applicability in a full-featured grid generation and simulation package. In the respective applications, the current limitation to 2D grids is still not a severe restriction. Nevertheless, we are also working on an extension of the scheme to the 3D case.

References

1. J. Behrens, et.al. *amatos: Parallel adaptive mesh generator for atmospheric and oceanic simulation*. Ocean Modelling 10 (1–2), 2005.
2. M. Griebel. *Multilevel algorithms considered as iterative methods on semidefinite systems*. SIAM Journal of Scientific and Statistical Computing 15 (3), 1994
3. F. Günther, M. Mehl, M. Pögl, C. Zenger. *A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves*. SIAM Journal of Scientific Computing, accepted.
4. M. Mehl, C. Zenger. *Cache-oblivious parallel multigrid solvers on adaptively refined grids*. In: Proceedings of the 18th Symposium Simulationstechnique (ASIM 2005), Frontiers in Simulation, Erlangen, 2005.
5. H. Sagan. *Space Filling Curves*. Springer, 1994.
6. R.P. Stevenson. *Optimality of a standard adaptive finite element method*. Technical Report 1329, Utrecht University, May 2005, submitted.
7. H. Yserentant. *On the multilevel splitting of finite element spaces*. Numerische Mathematik 49 (3), 1986
8. G. Zumbusch. *Adaptive Parallel Multilevel Methods for Partial Differential Equations*. Habilitationsschrift, Universität Bonn, 2001.