Parallelizing Adaptive Triangular Grids with Refinement Trees and Space Filling Curves

Daniel Butnaru
butnaru@in.tum.de
Advisor:
Michael Bader
bader@in.tum.de
JASS '08
Computational Science and Engineering

Technical University Munich

April 16, 2008

Contents

1 Motivation

In numerically solving a PDE usually a grid is layed over the domain of the problem. If the problem is time dependent or the domain has singlarities we can use an adaptive grid. This allows to better reflect the domain caracteristics but it also adds more work in storing/accessing the grid information.

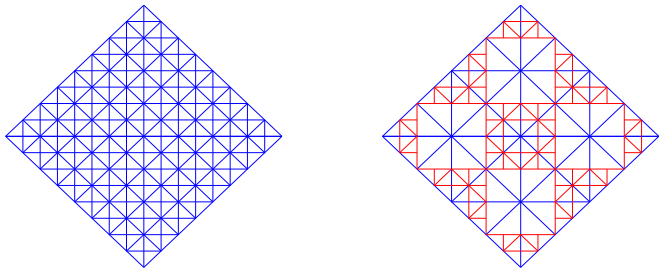

Figure 1: Regular and non-regular(adaptive) grids

1.1 Iterative Solver

The typical scenario of a solver that works on a grid is to iterate over the grid nodes and update the unknowns in each node. For this, information about the neighboring unknowns is needed. Figure 2 shows a typical stencil used in a finite difference scheme to update the central node.

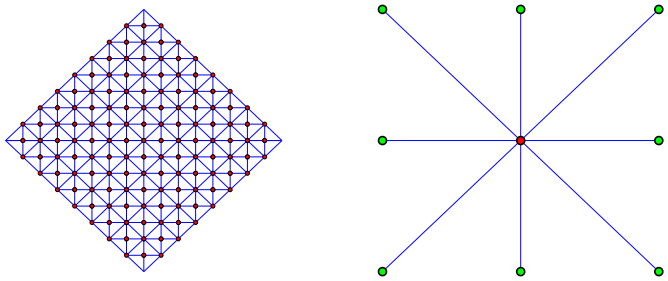

Figure 2: Grid nodes and node stencil

The solver can traverse the grid in several ways but depending on the traversal strategy and the grid structure, neighborhood information can be easily computed or needs to be stored. For a regular grid a simple incrementing of a counter can give us the next node. With an adaptive grid, depending of how the grid is stored (e.g a tree structure), some additional work is needed to move to the next node.

1.2 Parallelization

In a multiple CPU environment we can partition the grid and split the computational effort on several processors. Some requirements can be posed for this approach:

- load balancing. Each processor should work on equally sized partitions.
- reduced inter-processor communication. To reduce the communication latency the CPUs should work as independent as possible. Having short boundaries between the partitions insures that a minimal amount of update information gets sent across.
- fast repartioning. If refinements of the grid occur, the complexity of the repartitioning algorithm should not slow down a fast solver.
- avoid redistributing existing nodes to other processors. Partitions not affected by the refining should be handled by the same CPU.

In the following we will consider the finite element method as a solver. The grid is in this case traversed cell by cell (and not node-wise).

2 Triangular Grids

One type of an adaptive-regular grid is the triangular grid. Each cell of the grid is a (right angle) triangle.

2.1 Grid generation

To construct such a grid we start with a triangular domain. We then split the initial triangle along the marked edge in two equal triangles. The hypothenuse is always marked for splitting. To further refine we call the procedure recursively on the two new triangles until the desired refinement level is reached.

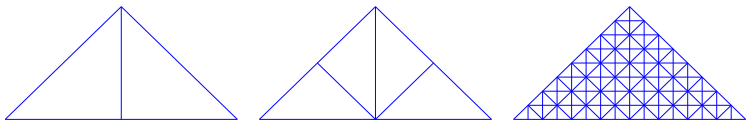

Figure 3: Triangular grid after 1 ,2 and 7 iterations

To reflect the singularities of the domain we can call the recursive bisection only on certain cells (local splitting) where more refinement would be needed.

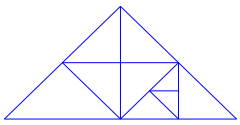

Figure 4: Adaptively refined triangular grid

2.2 Refinement Tree. Representing the Grid as a Binary Tree

To represent and store the grid we could use geometric information. However, do to the recursive structure of the grid generation process we can represent the entire grid structure as a simple binary tree.

- the root corresponds to the initial domain.
- the children of any other node correspond to the elements that were created when that node was refined
- the leafs are the individual triangles in the last refinement.

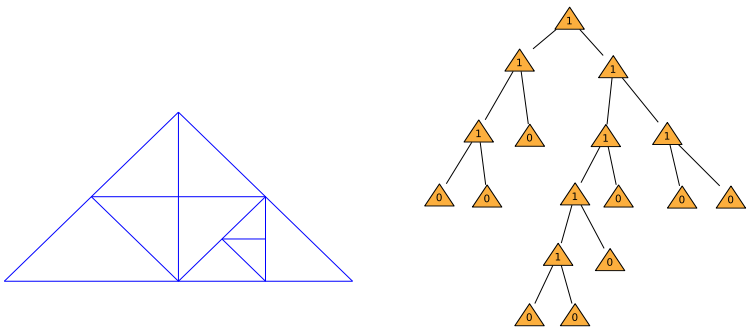

Figure 5: Representing an adaptive grid by a binary tree

Figure 6 presents another example of a more complex grid. The numbers show correspondence between triangles of the grid and tree nodes. The unlabeled nodes have the same numbers as in the grid above them.

2.3 Conforming Grid

Situations where neighbor elements share no common edge have to be avoided for numerical reasons. Refining the blue triangle in figure 7 along the hypothenuse will create two new triangles and a new unknown on the hypothenuse.

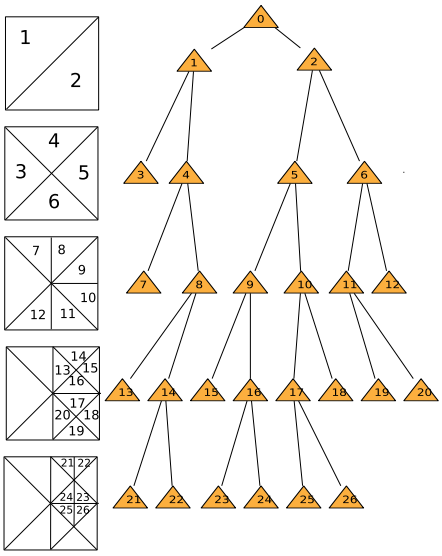

Figure 6: Representing an adaptive grid by a binary tree

However, this new unknown is not shared by the red triangle (hanging node) and as such gets no influence from that neighbor. Such a situation leads to a non-conforming grid.

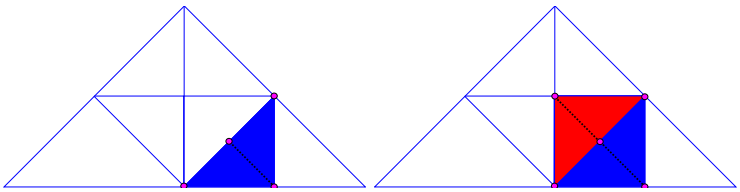

Figure 7: Non-nonforming vs conforming grids

To obtain a conforming grid we must also refine the red triangle. In the case of a more complex grid refining one triangle might need a cascade of refinements to avoid hanging nodes. Figure 8 shows the changes in the tree representation caused by the cascading refinements.

2.4 Storing the Refinement Tree

We can further simplify the storage of the grid structure information by using a bit vector. Each node in the tree can be marked with a bit value. 1 if the node is refined and 0 otherwise. We can this way store and work on the the corresponding structure of the adaptive grid in a single bit vector.

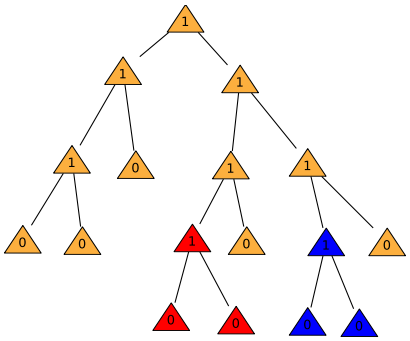

Figure 8: Tree representation after the refinement step

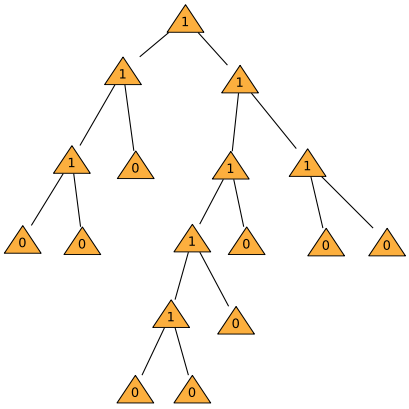

Figure 9: Bit vector representation: 11100011110000100

The Depth-First traversal of the tree gives us the bit vector `11100011110000100` for the tree in figure 9. An appropriate depth first traversal (DFS) of the tree can give a sequential order of grid cells. In the next section we will see that traversing the grid along the Sierpinski curve gives us such an order.

3 Sierpinski Curve

3.1 Sierpinski Curve. Definition and Construction

The Sierpinski curve, together with the Hilbert and Peano curves, are examples of `space-filling curves`. If $f : I \longrightarrow \mathbb{R}^n$ is a continuous function and $I \subset \mathbb{R}$ is compact then the image $f_*(I) := \{f(x) \in \mathbb{R} \mid x \in I\}$ is called a curve[3].

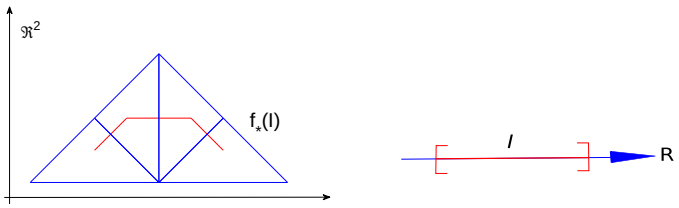

Figure 10: Mapping a 2D domain to 1D

We can further add that if $f_*(I)$ is a curve and the area (volume in 3D) is positive ($\neq 0$), then the curve is called a space filling curve.

To intuitively construct the curve we proceed by determining the curve for small triangle and then we connect the subcurves in the the correct order. A set of grammar rules can be used to construct a curve of a certain level.

One important feature of a space filling curve is that it provides a way to traverse the entire domain in a sequential way without returning to an already visited cell. Another feature is the quality of the partitions. By stopping at any point along the curve and partitioning the visited cells we get a compact partition (with a small border).

3.2 Grid Traversal along the Sierpinski Curve

Numerical algorithms need neighborhood information to evaluate stencils. Our refinement tree representation does not easily allow getting the immediate neighbors. It would not be memory efficient to store these neighborhood relations.

One approach is to traverse the grid triangle by triangle in the order given by the Sierpinski curve. This way two neighbors are easily available from the DFS traversal of the refinement tree. Also grid partitioning by curve partitioning leads now to a good partition quality (small borders). For the other neighbors

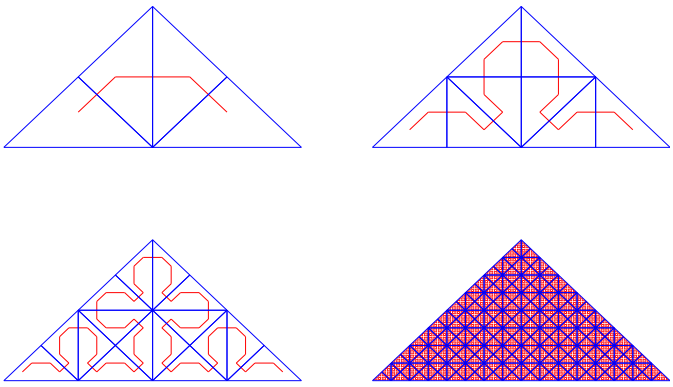

Figure 11: Sierpinski Curve for several iterations: 1, 2, 3 and 7

could use the concept of stacks (introduced bellow) which significantly lowers the memory requirements.

Figure 12 shows an example of a grid traversal, triangle by triangle. The order of traversal if given by the Sierpinski curve fitted to the triangular domain.

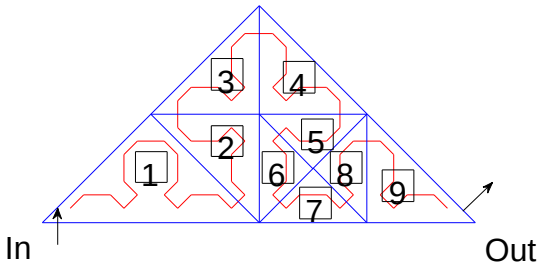

Figure 12: Example of a grid traversal, triangle by triangle.

The red curve depicted in figure 12 illustrates the idea of the curve traversal. However, we don't need to explicitly construct the curve. Using the refinement tree together with some small modifications is enough to get such a traversal by a simple DFS of the tree.

Start with the refinement tree. Fix the Depth First Search to go first to the left child and then to the left child. Set the left and right children in a consistent manner, i.e the left child is represented by the triangle where the curve enters the parent triangle and the right one by the exit point of the parent.

3.3 Splitting Template

We can infer the entry and exit vertices(edges) of a triangle by observing the parent triangle. Consider a triangle template with marked input-vertex (I) and output-vertex (O) (figure 13):

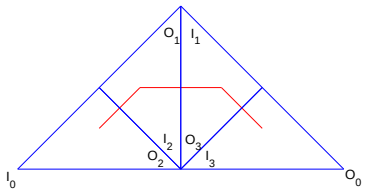

Figure 13: Bisection template

After the bisection the subtriangle which contains the input-vertex of the parent is the left child in the refinement tree. The subtriangle which contains the output-vertex of the parent is the right child. With this convention the DFS traversal of the refinement tree will now run along the Sierpinski curve.

3.4 Stacks

The Sierpinski curve approach does provide a way to traverse and partition a grid. However, unlike the regular grid (line by line) traversal some unknowns will be need at a later step along the curve. When moving to a cell oriented processing we must store the intermediary values of the unknowns.

A `stacks`[2] approach can be used to reduce the memory footprint. We start by processing cells along the Sierpinski curve. For each cell we need access to the unknowns (nodes). We start by classifying the unknowns. The ones left of the curve are Blue and the ones right of the curve are classified as Red.

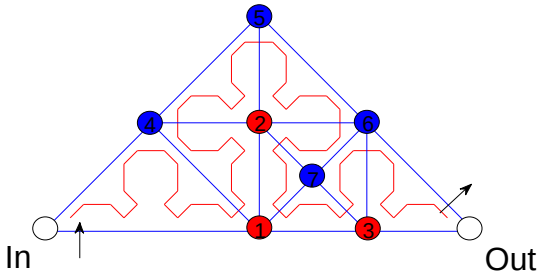

Figure 14: Stacks

The idea is to use a system of four stacks for fast and memory efficient access to unknowns. One stack is used to hold the initial values of the unknowns. Two helper stacks, one for each color - red and blue - are used to hold intermediary

values of the unknowns. Also a write stack holds the final values of the unknowns.

To shortly illustrate the idea begind the stacks approach consider the example grid in figure 14. The nodes on the left of the curve are blue and the ones on the right are red. We start with the first cell and update the unknowns 4, 1 and the white node. We took these unknowns from the input stack and place them as follows:

- the white node will not be needed by any other cell so it has the final value. We put it on the write (final) stack
- node 4 will need to get a contribution from the next cell so we place its intermediary value on the blue stack
- similar to node 4, node 1 is placed on the top of the red stack

As we move from cell to cell we take and put unknowns on the stacks. The advantage of this approach is that at each cell along the path to top of the stacks will always provide the necessary unknowns from the current cell. To decide which vertex belongs to which stack and it is needed by a later cell a set of rules is used. This handout does not cover this construction process.

The table bellow presents the content of the stacks when processing the grid in figure 14.

4	1	
blue	red	final
	2	
4	1	
blue	red	final
	2	
5	1	4
blue	red	final
	2	5
6	1	4
blue	red	final
7	2	5
6	1	4
blue	red	final
		2
7		5
6	1	4
blue	red	final

continues...

4 Parallelization

4.1 Tree Partitioning

In a parallel environment we can split the computational effort on several processors. To get the most of this possibility some requirements need to be taken into consideration:

- each of the processors should receive an equal slice of the grid. We want to achieve load balancing and avoid wasting CPU time
- partitioning information should be handled globally and then sent to each processor
- adaptive refinement on a cell can lead to repartitioning. We may need to repartition and redistribute some pieces of the grid to maintain load balancing.
- processor inter-communication needed (when redistributing triangles)
- triangles need not necessarily require the same computational effort (like elements on the Dirichlet boundaries). The algorithm could also take into consideration the computational effort for individual cells and assign cell weights for better load balance.

4.2 REFTREE Algorithm

In the following the `REFTREE` [1] algorithm will be presented. It partitions a grid along the Sierpinski curve and satisfies the requirements listed above.

REFTREE Algorithm Phase 1 (Labelling):

1. start with the refinement tree.
2. leaf nodes get a weight of 1, intermediary nodes 0.
3. each other node is labelled with the sum of the weights in the subtree rooted in that node.

REFTREE Algorithm Phase 2 (Partitioning):

1. partition size P is cumulated sum at the root divided by number of processors
2. with a DFS examine the summed weights of the nodes. Initial sum S is 0.
3. consider the summed weight S_n at node n
4. if $S + S_n \leq P$, add the the entire tree rooted in n to the partition. $S = S + S_n$.

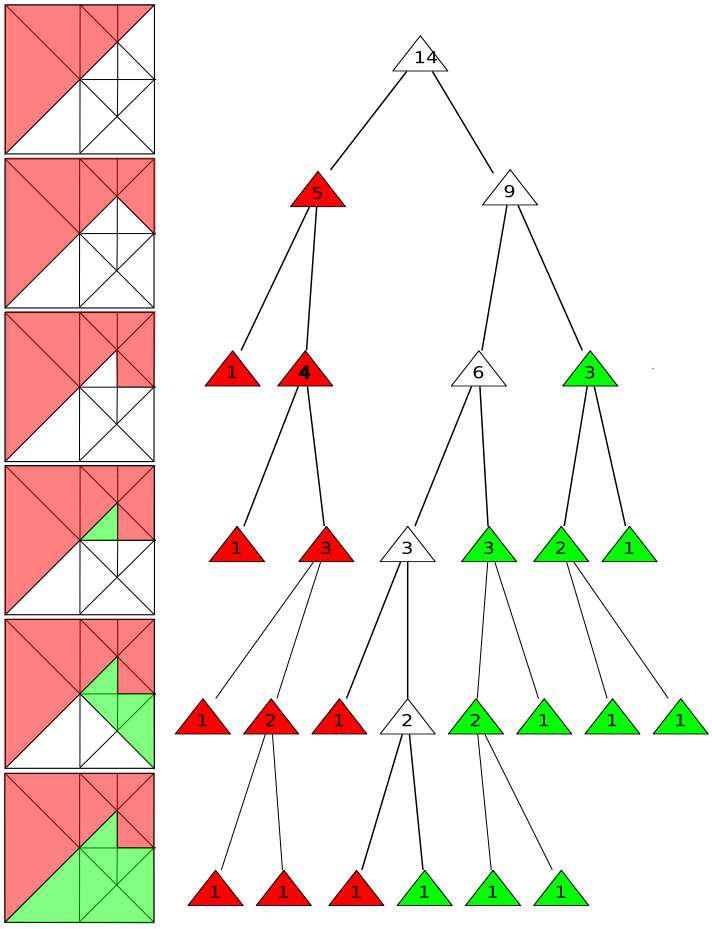

Figure 15: Labeling phase

5. else if $S + S_n > P$, continue with the subtrees of node n
6. stop when $S = P$

One DFS is enough to label the entire tree ($O(n)$). One truncated DFS is also required to partition the entire tree in p partitions $O(p * log(N))$.

4.3 Inter-Processor Communication

Two types of information need to be communicated between the CPUs. When updating the cells on the interior border of a partition the contribution of the neighbors in other partition needs to be communicated. Also, after a refinement step, the new labels of the tree nodes have to be communicated.

In figure 16 information for the pruning nodes (white) will be exchanged with all other CPUs. Each processor sends the weights it has for the pruning nodes and all responses from other CPUs are summed up to give the weighted sum from the pruning points. Now each processor has enough information to perform another iteration on its allocated partition.

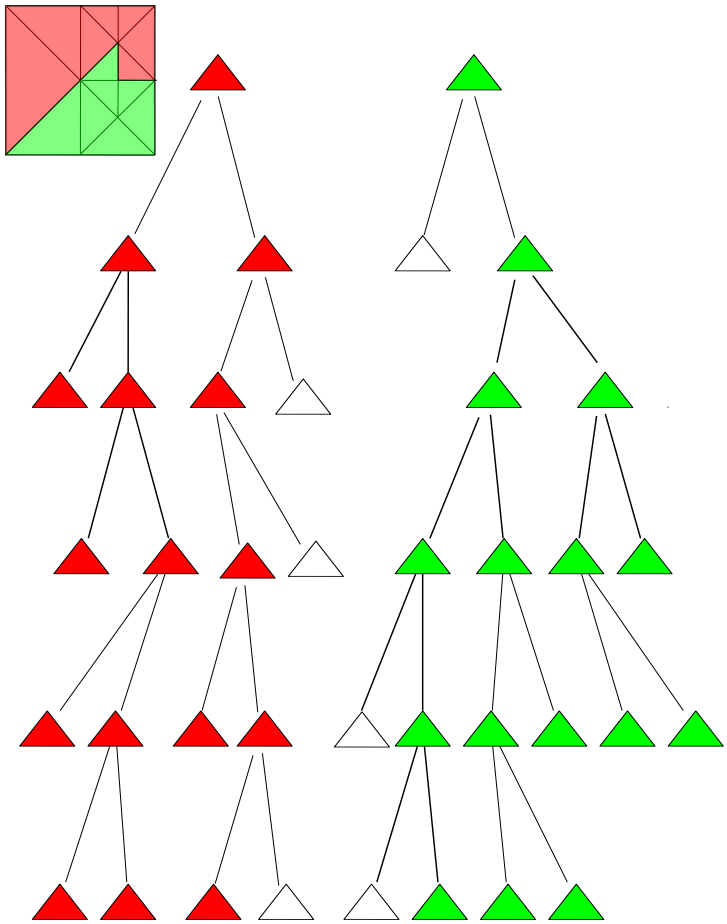

Figure 16: Partitioning phase

The REFTREE algorithm presents the following advantages:

- it produces almost equally sized partitions.
- for most of the used types of grids it creates connected partitions
- it is fast enough not to affect the benefits of a fast multigrid solver

- after refining, the new partitions are similar to previous ones. This way the redistribution of cells is reduced with a positive impact on the communication overhead

4.4 Wrap up

One type of discretization grids are the (right angle) triangular grids. By recursive bisection the grid can be refined until the desired grid resolution is achieved. We can significantly simplify the grid representation by using a refinement tree. By assigining a certain order to the depth first search traversal of the refinement tree, a Sierpinski curve-like traversal of the grid cells is possible. A system of stacks can be used to add cache efficiency. The REFTREE algorithm builds on the previous concepts and is fast in creating equal and compact partitions and preserves this way the advantages of a fast solver.

References

[1] [Mitchell, 1991] William F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids

[2] [Bader, 2007] Bader, M., Schraufstetter, S., Vigh, C.A. and Behrens, J. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves

[3] H. Sagen Space-filling Curves