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

March 13, 2008

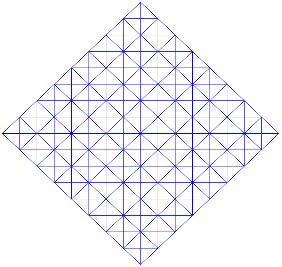
Outline

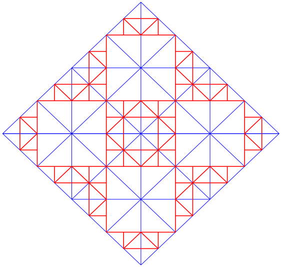
Motivation

Domain Discretization

In numerically solving a PDE usually (1):

- a grid is layed over the domain.
- if the domain has singularities or changes in time using an adaptive grid is a better choice.



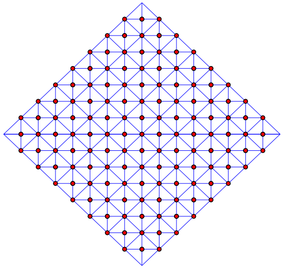


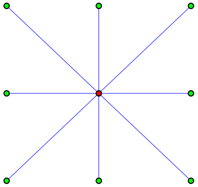
Motivation

Iterative Solver

In numerically solving an elliptic PDE usually (2):

- a solver iterates over the grid nodes
- neighborhood information is used to update the nodes
- 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





Motivation

Parallelization

We can partition the grid and split the computational effort on several processors.

Parallelization Requirements

- load balancing. Each processor should work on equally sized partitions.
- reduced inter-processor communication. Short boundaries between the partitions.

Adaptive Grids Constraints

- fast repartioning
- avoid redistributing existing nodes to other processors.

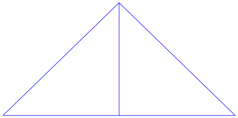
Outline

Triangular Grids

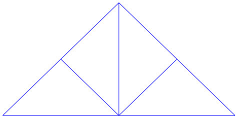
Grid generation

Constructing a triangular grid:

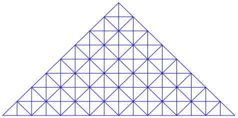
- start with a triangular domain
- recursively split the triangle by bisection
- the hypothenuse is always marked for splitting



(a) G_1

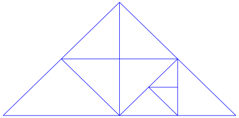


(b) G_2



(c) G_7

To adaptively refine call the recursive bisection on the desired triangle (local splitting):

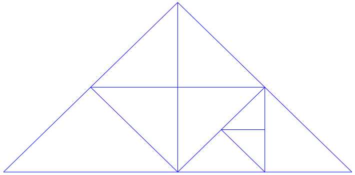


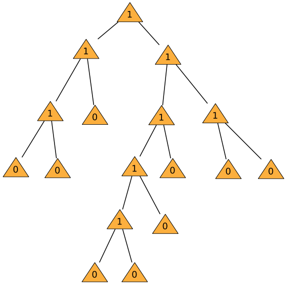
Refinement Tree

Representing the Grid as a Binary Tree (1)

We can represent the refinement of the grid as a 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.



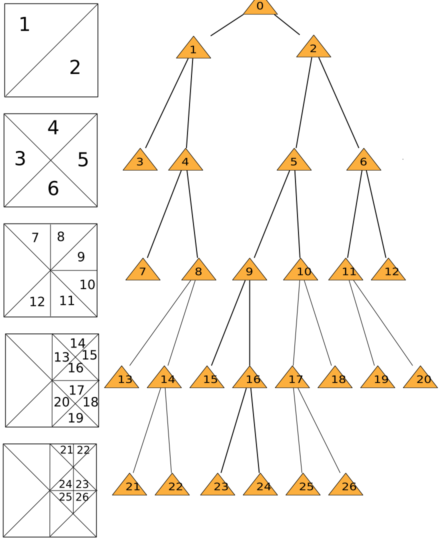


Refinement Tree

Representing the Grid as a Binary Tree (2)

Further example:

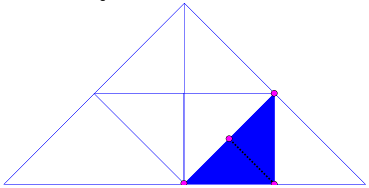
- the numbers show correspondence between triangles of the grid and tree nodes
- unlabeled nodes have the same numbers as in the grid above them.



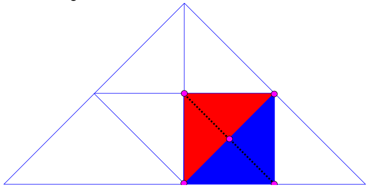
Refinement Tree

Conforming Grid

non-conforming

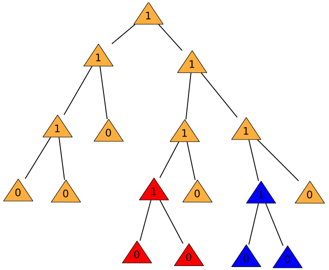


conforming



For numerical reasons avoid situations where neighbour elements share no common edge (causing hanging nodes).

Refining one triangle might need a cascade of refinements to avoid hanging nodes.

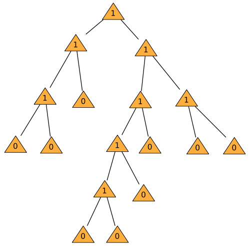


Refining the blue triangle triggers also a refinement of the red one.

Storing the Refinement Tree

Bit Vector

Mark the nodes in the tree with a bit value. 1 if the node is refined and 0 otherwise. We can this way store the corresponding structure of the adaptive grid in a single bit vector.



(d) 11100011110000100

An appropriate Depth First Traversal of the tree can give a sequential order of grid cells $\longrightarrow$ Sierpinski Curve.

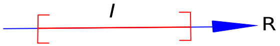
Outline

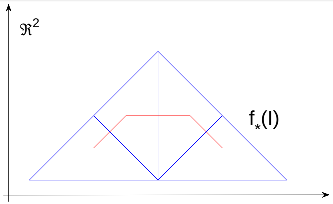
Sierpinski Curve

Space Filling Curves

Definition

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.





Definition

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.

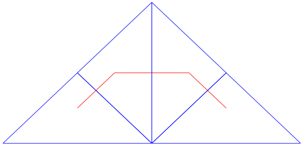
Sierpinski Curve

Construction

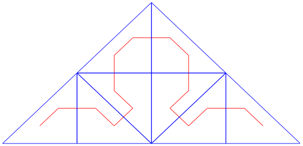
To (intuitively) construct the curve:

- determine the curve for each small triangle
- connect the subcurves in the correct order

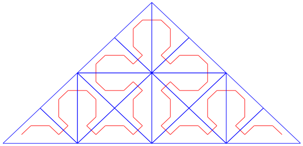
Sierpinski Curve for several iterations:



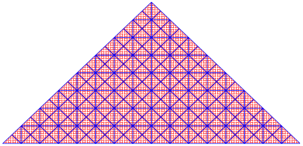
1



2



3



7

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 neighbours
- would need a lot of memory to store these neighborhood relations

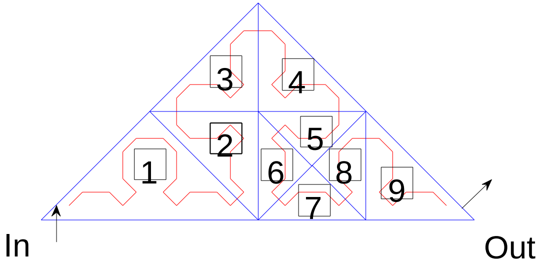
Idea: traverse the grid triangle by triangle in the order given by the Sierpinski curve:

- two neighbours are easily available (DFS traversal)
- grid partitioning by curve partitioning leads to good partition quality
- for the other neighbours we could use the concept of stacks (introduced later)

Grid Traversal along the Sierpinski Curve

Example of a Grid Traversal

Example of a grid traversal, triangle by triangle.



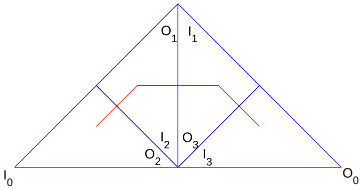
Note that we don´t need to explicitly construct the curve:

- use only the refinement tree
- fix the Depth First Search to go first to the left child.
- set the left and right children in a consistent manner

Grid Traversal along the Sierpinski Curve

Splitting Template

Consider a triangle template with marked input-vertex (I) and output-vertex (O). To identify the order of the children start at the input-vertex of the parent.



After the bisection:

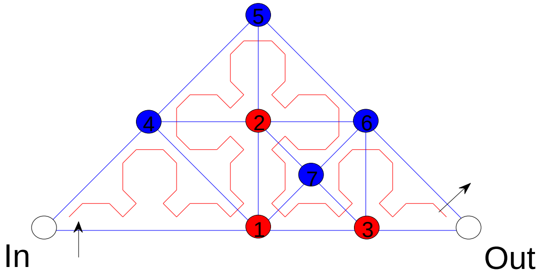
- the subtriangle which contains the input-vertex of the parent is the left child in the refinement tree.
- the subtriangle which contains the output-vertex of the parent is the right child
- the DFS traversal will now run along the Sierpinski curve

Grid Traversal along the Sierpinski Curve

Stacks - Cache Efficient Processing (1)

Cell Oriented Processing:

- process cells along the Sierpinski curve
- for each cell we need access to the unknowns (nodes)
- classify the unknowns. Left of the curve - Blue. Right of the curve - Red.



Grid Traversal along the Sierpinski Curve

Stacks - Cache Efficient Processing (2)

Use a system of four stacks for fast and memory efficient access to unknowns:

- a read stack holds initial values of the unknowns
- two helper stacks - red and blue - to hold intermediary values
- write stack holds final values of the unknowns

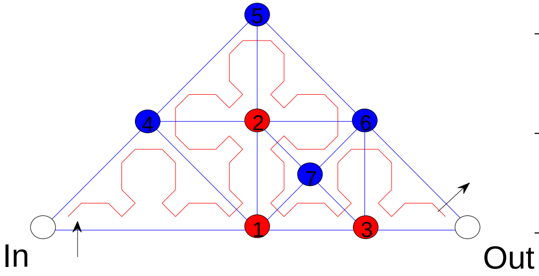
Processing:

- the 2 unknowns on common edge can be reused
- the 2 unknowns opposite to the common edge must be processed
- the top of the stacks will always have the unknowns needed to process the current triangle

Grid Traversal along the Sierpinski Curve

Stacks - Cache Efficient Processing (3)

Based on some rules we decide from which stack to take the needed unknowns and on which stack to put them on.



```
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...

Outline

Grid Partitioning

Tree Partitioning

In a parallel environment we can split the computational effort on several processors. Some observations:

- each of the processors should receive an equal slice of the grid.
- partitioning information can be handled globally and then sent to each processor
- adaptive refinement on a cell can lead to repartitioning
- processor inter-communication needed (when redistributing triangles)
- triangles need not necessarily require the same computational effort (like elements on the Dirichlet boundaries) $\longrightarrow$ node weights

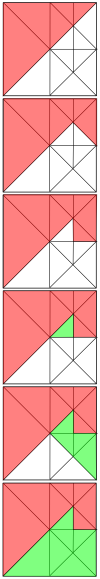
Grid Partitioning

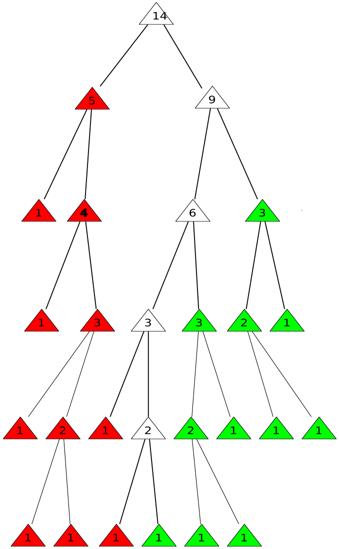
REFTREE Algorithm Phase 1

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.

One Depth First Search is enough to label the entire tree ($O(n)$)



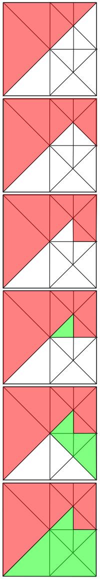


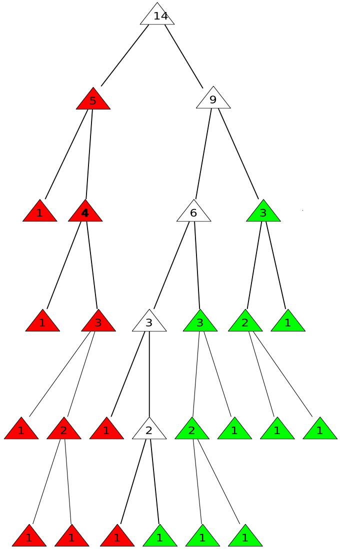
Grid Partitioning

REFTREE Algorithm Phase 2

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



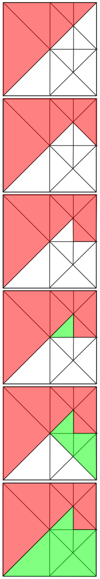


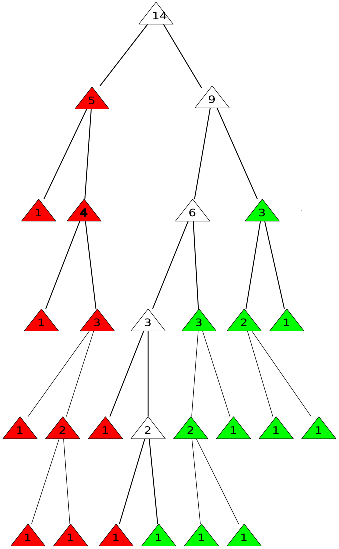
Grid Partitioning

REFTREE Algorithm Phase 2

4. if $S + S_n \leq P$, add the the entire tree rooted in n to the partition. $S = S + S_n$.
5. else if $S + S_n > P$, continue with the subtrees of node n
6. stop when $S = P$

One truncated Depth First Search is required to partition the entire tree. $O(p * log(N))$

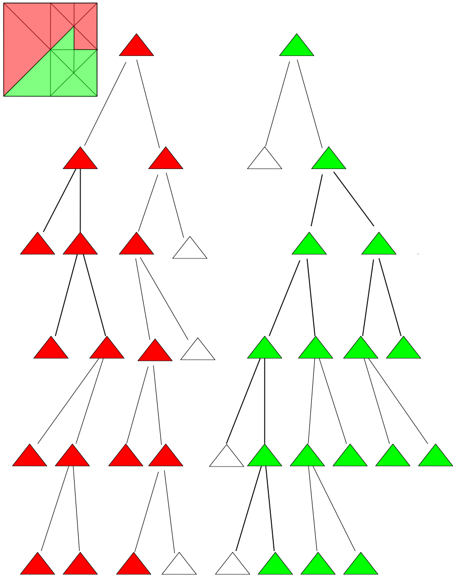




Grid Partitioning

Inter-Processor Communication

- for the pruning nodes (white) information will be exchanged
- each processor sends the weights it has for the pruning nodes
- 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.

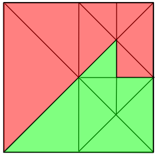


REFTREE Algorithm

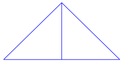
Nice Properties

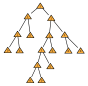
Advantages of the REFTREE algorithm:

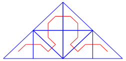
- produces almost equally sized partitions (with a difference of at most 1)
- creates connected partitions (for most of the used types of grids)
- fast enough not to affect the benefits of a fast multigrid solver
- after refining, the new partitions are similar to previous ones (reduces redistribution)

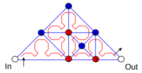


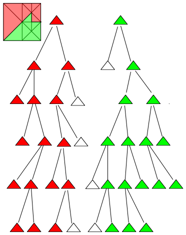
Wrap up











Important points:

- Triangular grids created by recursive bisection
- grid can be represented and stored as a refinement tree
- Sierpinski Curve gives a linear traversal of the grid cells, thus simplifying storage
- Stacks add cache efficiency
- REFTREE algorithm is fast in creating equal and compact partitions using the refinement tree

References

- (Mitchell, 1991) William F. Mitchell.
  A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids
- (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
- H. Sagen
  Space-filling Curves

Thank you!
Questions?