Algorithms of Scientific Computing
Hierarchical Methods and Sparse Grids

Michael Bader

Technische Universität München

Summer Term 2013
Part VII

Algorithms and Data Structures for Sparse Grids
We will now look at typical sparse grid algorithms
Can, e.g., be used for hierarchization/dehierarchization,
integration, data mining, solution of PDE, . . .

important: *adaptive* representation

Algorithms depend on data structure:
- Efficient traversal of sparse grid necessary
- Thus, we deal with data structures for sparse grids, too
Data Structures \((d = 1)\)

- How to store function \(u : [0, 1] \rightarrow \mathbb{R}\) in hierarchical representation (i.e. surplusses \(v_{i,j}\))? 

- Order and store grid points and associated values in binary tree
  - Root is node \(x_{1,1} = 1/2\)
  - Children of node \(x_{l,i}\) are – if existent – the grid points \(x_{l+1,2i-1}\) and \(x_{l+1,2i+1}\) of level \(l + 1\)
  - Alternative point of view if child does not exist: Complete subtree of binary tree starting from child with all surplusses set to 0
Data Structures \((d = 1)\) (2)
Typical Algorithms \((d = 1)\)

Hierarchization and Dehierarchization

- Prototype for typical algorithm (c.f. worksheet 4)
- Our data structure has to allow
  1. Iteration over all grid points, considering the hierarchical relations
     - E.g. for hierarchization: first handle all grid points in the support of \(\phi_{l,i}\), then compute \(v_{l,i}\)
  2. Access to *hierarchical neighbors*: grid points at interval boundaries of support of \(\phi_{l,i}\) (if possible – exception for points 0 and 1 as not in the tree), e.g. to compute

\[
v_{l,i} = u_{l,i} - \frac{1}{2}(u_l + u_r).
\]

Michael Bader: Algorithms of Scientific Computing
Hierarchical Methods and Sparse Grids, Summer Term 2013
Typical Algorithms ($d = 1$) (2)

- Hierarchical neighbors are easy to find geometrically
  \[ x_{l,i-1}, \quad x_{l,i+1} \]

- But have even indices \(\Rightarrow\) really are on another level (\(< l\))

- In the binary tree structure:
  - Can be found on way from root to node
  - One is parent node

- For hierarchization/dehierarchization: pass hierarchical neighbors as additional parameters

- Developing algorithms:
  - Try to store all information to process one node at the node and its hierarchical neighbors
  - Access to other nodes typically expensive
  - Tree traversal with “supply of hierarchical neighbors” only linear in number of nodes
What data structure to use in more than one dimension?

Algorithmically: use construction of basis functions as product of one-dimensional hats. Ideally:
- Use a loop \(1, \ldots, d\) over the dimension
- Apply \(1d\) algorithm on one-dimensional structures in each dimension (see also worksheet 7)

⇒ Need access to hierarchical neighbors in each spatial direction; implies to create binary tree structure in each dimension

Disadvantages:
- Storage requirements \((2d\text{ pointers})\)
- High effort to keep structure consistent when inserting or deleting points
If you could recognize anything, it would be binary tree structures for rows (black) and columns (magenta)
Data Structures and Typical Algorithms \((d > 1)\) (3)

Often better:

- Store in a node only two pointers for one direction (e.g. \(x_1\))
- A binary tree of nodes is a row (a 1\(d\) structure parallel to the \(x_1\) axis)
- For next spacial direction \(x_2\), only a binary tree in \(x_2\) direction required
- Stores one plane parallel to \(x_1–x_2\) coordinate plane; nodes are the binary trees with 1\(d\) structures
- For each additional spatial direction \(x_d\) build binary tree with \((d – 1)\)-dimensional structures as nodes
- Disadvantage: Access to hierarchical neighbors not that easy any more (except for \(x_1\)-direction)
- But can be achieved without much more computational effort by suitable reordering of loops and tree traversals
Already more clear: One plane (two-dimensional structure) consists of one binary tree (magenta) of which the nodes are binary trees (black) for each row.
Hash table

- Much more comfortable (and not too inefficient) alternative
- Store coefficients as target values, with, e.g., \((\vec{l}, \vec{i})\) as keys
- No need to care about tree structures
- Only requires computation of indices of accessed nodes (hierarchical neighbor, . . . )

⇒ Best solution for your own sparse grid experiments

Further assumptions on data structures

- Algorithms will assume that all hierarchical neighbors exist for each grid point

⇒ If creating grid points adaptively, create them if necessary
- No further assumptions
Summary

Data Structures

- array-based for regular sparse grids and combination technique (see tutorials)
- hierarchical adaptivity reflected by tree-based data structures (but: more complicated in higher dimensions)
- hash-based data structures

Algorithms

- hierarchisation and dehierarchisation: tree-based recursion plus “hierarchical neighbours”
- archimedes quadrature $\rightarrow$ recursion on dimensions
- much more complicated algorithms, if we want to use sparse grids for solution of partial differential equations