

Algorithms for Scientific Computing

Parallelization using Space-Filling Curves

Michael Bader
Technical University of Munich

Summer 2018



TUM Uhrenturm

Parallelisation using Space-filling Curves

Problem setting:

- “mesh” (2D, 3D, ...) of N unknowns ($N \gg 1000$)
- solve linear system(s) of equations (maybe repeatedly with varying right-hand side)
- in the system, only spatially neighbouring unknowns are coupled

Parallelisation:

Distribute N unknowns to p partitions, such that

- each partition contains the same number of unknowns (*load balancing*)
- for as many unknowns as possible, all neighbours are in the same partition (\Rightarrow avoids communication between partitions)

Parallelisation using Space-filling Curves (2)

Further demand: adaptivity

- add further unknowns (during/depending on intermediate results) or drop unknowns
- (re-)partitioning required to be **fast**:
must not cost more computation time than going on with a bad load balance
- “shape preserving”: if only few unknowns are added or dropped, the shape of partitions should not change strongly
⇒ only few unknowns then need to migrate to another partition

⇒ popular strategy: use **space-filling curves**

Hölder Continuity of Space-filling Curves

Definition: (*Hölder continuous*)

A function f is called **Hölder continuous with exponent r** on the interval I , if a constant $C > 0$ exists, such that for all $x, y \in I$:

$$\|f(x) - f(y)\|_2 \leq C|x - y|^r$$

Importance for space-filling curves:

- $|x - y|$ is the distance of the indices
- $\|f(x) - f(y)\|$ is the distance of the image points (in “space”)
- To prove: the Hilbert curve is Hölder continuous with exponent $r = d^{-1}$, where d is the dimension:

$$\|f(x) - f(y)\|_2 \leq C|x - y|^{1/d} = C\sqrt[d]{|x - y|}$$

Hölder Continuity of the 3D Hilbert Curve

Proof analogous to simple continuity proof:

- given $x, y \in \mathcal{I}$; find an n , such that $8^{-(n+1)} < |x - y| < 8^{-n}$
- 8^{-n} is the interval length for the n -th iteration
 $\Rightarrow [x, y]$ covers at most two neighbouring(!) intervals.
- per construction of the 3D Hilbert curve, the function values $h(x)$ and $h(y)$ are in two adjacent cubes of side length 2^{-n} .
- the length of the space diagonal through the two adjacent cubes is $2^{-n} \cdot \sqrt{1^2 + 1^2 + 2^2} = 2^{-n} \cdot \sqrt{6}$, hence:

$$\begin{aligned} \|h(x) - h(y)\|_2 &\leq 2^{-n} \sqrt{6} = (8^{-n})^{1/3} \sqrt{6} = \left(8^{-(n+1)}\right)^{1/3} 8^{1/3} \sqrt{6} \\ &\leq 2\sqrt{6} |x - y|^{1/3} \quad \text{q.e.d.} \end{aligned}$$

Hölder Continuity and Parallelisation

- for the Hilbert curve (also Peano curve and all connected, recursive SFC), we have:

$$\|f(x) - f(y)\|_2 \leq C \sqrt[d]{|x - y|}$$

- relates the distance $|x - y|$ between indices to the distance $\|f(x) - f(y)\|$ of (mesh) points
 - also: $|x - y|$ is the area of the respective space-filling-curve partition
 - hence: relation between volume (number of grid cells/points) and extent (e.g. radius) of a partition
- ⇒ Hölder continuity gives a quantitative estimate for **compactness** of partitions

Teil I

Parallelisation Using Space-Filling Curves

Generic Space-filling Heuristic

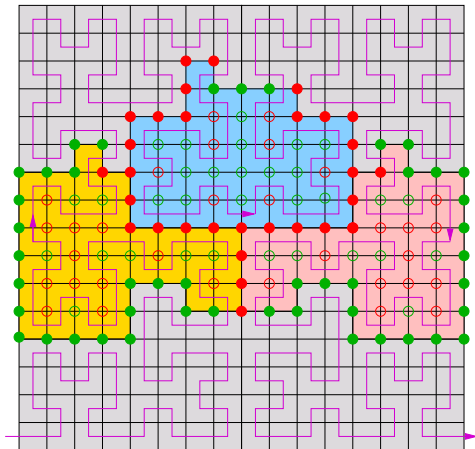
Bartholdi & Platzman (1988):

1. Transform the problem in the unit square, via a space-filling curve, to a problem on the unit interval
2. Solve the (easier) problem on the unit interval

For parallelisation: strategy to determine partitions

1. use a space-filling curve to generate a sequential order on grid cells
2. do a 1D partitioning on the list of cells
(cut into equal-sized pieces, or similar)

Hilbert-Curve Partitions on a Cartesian Grid



- Hilbert order traversal provides sequential order on grid cells
- Hilbert curve splits vertices into right/left (red/green) set

Partitioning Based on SFC Mapping

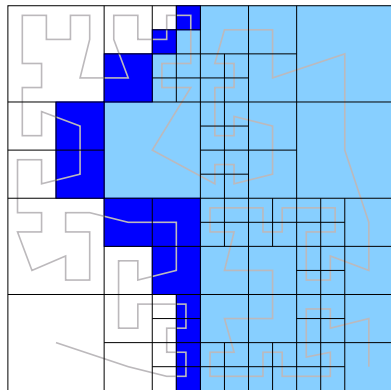
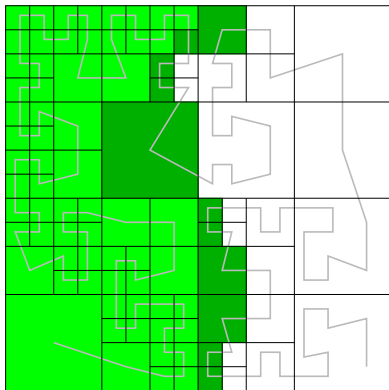
Computation of SFC Index:

- remember inverse mapping \bar{h}^{-1}
- given any (characteristic) coordinate $c_i = (x, y)$ of a grid cell (e.g., cell center), particle, etc., compute SFC index $t_i = \bar{h}^{-1}(c_i)$

Partitioning Algorithm:

1. compute $t_i = \bar{h}^{-1}(c_i)$ for all c_i
 2. sort all c_i according to index t_i
(requires efficient parallel sorting algorithm)
 3. split resulting “string of pearls” (i.e., 1D list of coordinates) into equal-sized (in general: equal-balanced) partitions
- ⇒ Hölder continuity promises compact partitions
- ⇒ index computation and sorting have complexity $\mathcal{O}(N \log N)$;
works for unstructured grids, particles, etc → highly flexible

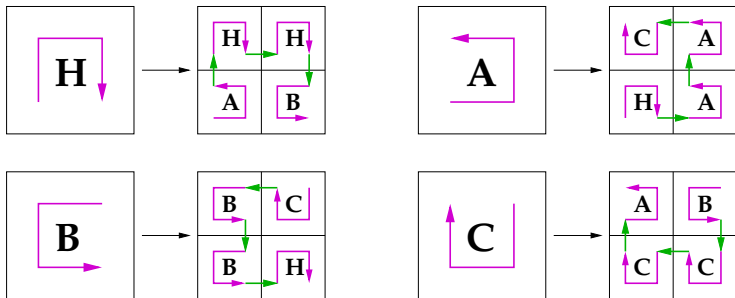
Hilbert-Curve Partitions on Quadrees



- here: with so-called **ghost cells** (data exchange with neighbours)
→ processed in identical order in both partitions

Recall: Grammar to Describe the Hilbert Curve

Construction of the iterations of the Hilbert curve:



Question:

Can this grammar be used to generate **adaptive** Hilbert orders?

Problem:

Words generated by the grammar do not allow reproduction of the refinement info!

A Grammar for Hilbert Orders on Quadrees

- Non-terminal symbols: $\{H, A, B, C\}$, start symbol H
- terminal characters: $\{\uparrow, \downarrow, \leftarrow, \rightarrow, (,)\}$
- productions:

$$H \leftarrow (A \uparrow H \rightarrow H \downarrow B)$$

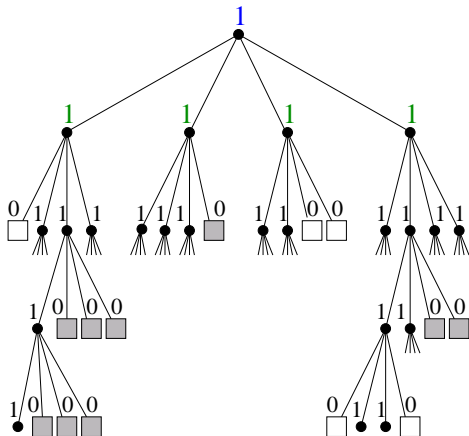
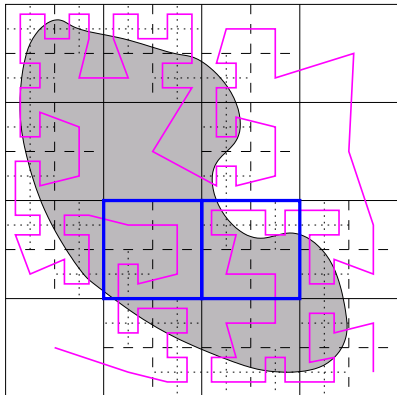
$$A \leftarrow (H \rightarrow A \uparrow A \leftarrow C)$$

$$B \leftarrow (C \leftarrow B \downarrow B \rightarrow H)$$

$$C \leftarrow (B \downarrow C \leftarrow C \uparrow A)$$

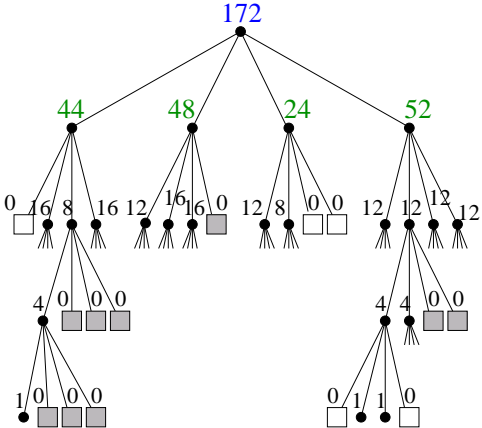
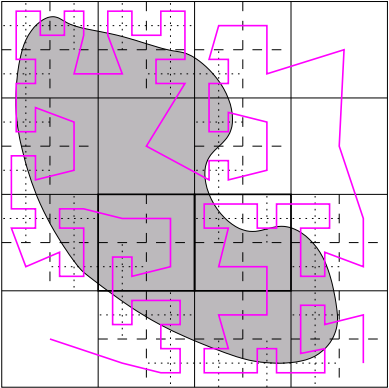
- ⇒ arrows describe the iterations of the Hilbert curve in “turtle graphics”
- ⇒ terminals (and) mark change of levels: “up” and “down”
- ⇒ cmp. algorithm in Python script *sfc_hilbert_plotter_adap*

Hilbert-Order Bitstream-Encoding of a Quadtree



1 1 0 1 \leftarrow 1 1 1 0 0 0 0 0 0 1 \leftarrow 1 1 \leftarrow 1 \leftarrow 1 \leftarrow 0 1 1 \leftarrow 1 \leftarrow 0 0 1 \leftarrow 1 1 0 1 1 1 0 1 \leftarrow 0 0 1 \leftarrow 1 \leftarrow

Refinement-Tree Encoding of a Quadtree



172
44
0
16
≪
8
4
1
0
0
0
0
0
0
0
0
0
16
≪
48
12
≪
16
≪
16
≪
0
24
12
≪
8
≪
0
0
52
12
≪
12
≪
12
≪
12
≪
12
≪
12

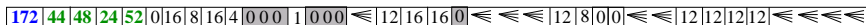
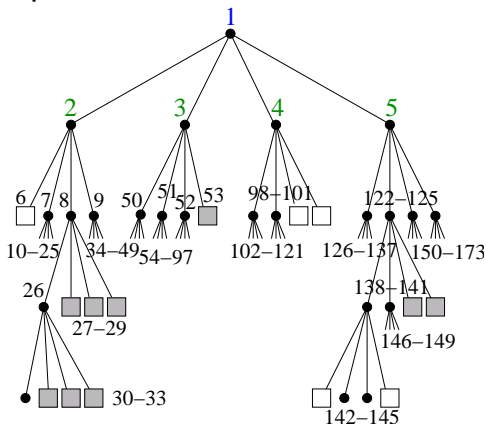
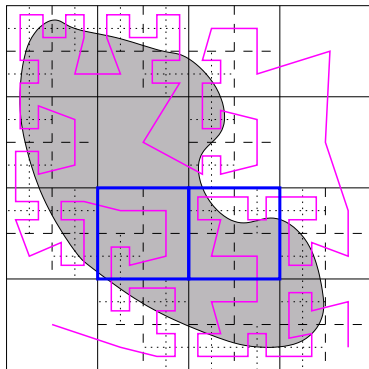
Refinement-Tree Encoding of a Quadtree (2)

REFTREE algorithm for partitioning:

- attributed quadtree with number of leaves/nodes of the subtree for each node
- allows to determine whether a certain node/subtree may be skipped by the current partition:
 - index of partition's first & last leaf/node are given
 - track first & last leaf of subtree from attributed info
 - decide whether to skip or (partially) traverse subtree
- disadvantage of data structure: required information spread across several locations in the stream
⇒ may be fixed by modified depth-first order

Towards Parallelisation

Refinement-Tree Encoding with Modified Depth-First Order



(numbers in the tree represent position of resp. node information in the stream)