IO-Efficient Point Location and Map Overlay in Low-Density Subdivisions

Shripad Thite ..... sthite@win.tue.nl

Department of Computer Science
Technische Universiteit Eindhoven
The Netherlands

Joint work with
Mark de Berg, Herman Haverkort, and Laura Toma

Point location

- Map = polygonal subdivision of the plane
- Given a point in the plane, identified by its coordinates, find the region of the map that contains the point

Map overlay

- Combine various attributes of data from different maps or map layers to compute the interaction of these attributes

- Given two polygonal subdivisions of the plane, red and blue, compute all intersections between a red edge and a blue edge

Geographic Information System (GIS)

- A GIS is a spatial database with algorithms for managing, analyzing, and displaying geographic information
- Applications with tremendous environmental, social, and economic impact—infrastructure planning, social engineering, facility location, agriculture
- Require algorithms for fundamental problems well-studied in Computational Geometry—adjacency, containment, proximity . . .

  . . . with a twist—geographic data is huge!

Geometric algorithms for GIS

- Conventional analysis of algorithms accounts for worst-case behavior, often for inputs that do not occur in practice
- Complex algorithms are too hard to implement and make little impact on applications
- Simplifying assumptions about the computational model are not valid or hold only approximately
- **Goal:** Design theoretically efficient practical algorithms accompanied by an analysis of the algorithm complexity on a refined model of computation for realistic inputs

Massive data

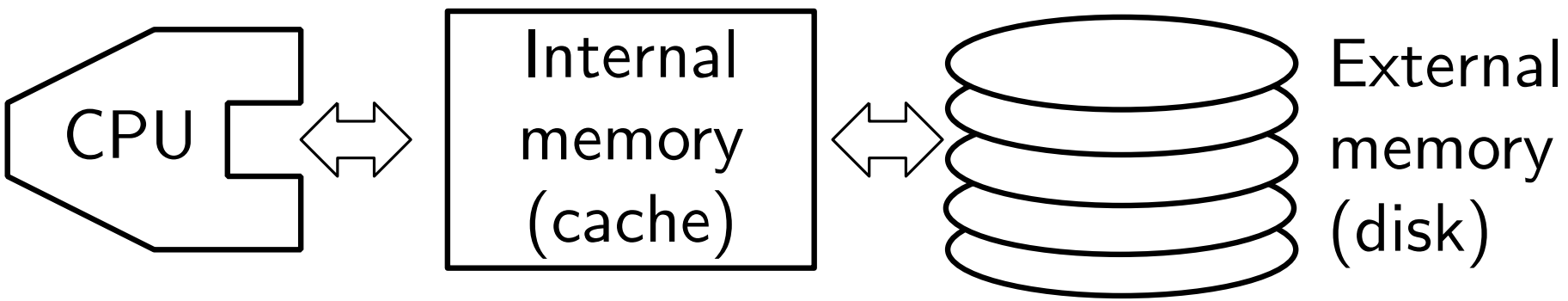
- Practical inputs have gigabytes and terabytes of data
- We need algorithms whose performance scales well for increasingly large input data sets encountered in practice
- Traditional algorithms suffer from poor memory usage
- Poor cache behavior causes thrashing where excessive time is spent transferring data in and out of memory cache

External-memory algorithms

- The cost of data transfer significantly influences the real cost of an algorithm, often dominating CPU operations
- External-memory algorithms seek to minimize data transfer, by utilizing *locality of reference*
- **Goal:** Develop external-memory algorithms and data structures for geometric problems, where it is often harder to exploit locality

External-memory model

- Model of computation where memory is organized in two levels—internal and external memory [Aggarwal & Vitter]
- CPU operations can take place only on data in internal memory, which is limited in size to M words



- External memory is large enough for input, working space, and output

External-memory model

- Both internal and external memory organized in *blocks* of B words each
- One input/output operation (one IO) transfers one block of B words between internal and external memory
- The *IO-cost* of an algorithm is the number of IO-operations it performs
- IO-complexity accurately models cost of data transfer between disk and main memory, as a function of memory architecture parameters B and M

External-memory algorithms

- Designed to minimize Input/Output (IO) operations between slow but large external memory and fast but small internal memory
- Each IO operation reads or writes B words stored in a block; internal memory of size M holds M/B blocks
- Two-level memory model introduced by Aggarwal and Vitter has become a popular design and analysis tool
- Lots of IO-efficient algorithms developed and proved useful in practice

Remember ...

- Map = polygonal subdivision of the plane
- **Point location:** Given a point in the plane, identified by its coordinates, find the region of the map that contains the point
- **Map overlay:** Given two polygonal subdivisions of the plane, red and blue, compute all intersections between a red edge and a blue edge

Previous work

- **External-Memory Algorithms for Processing Line Segments in Geographic Information Systems**
  Arge, Vengroff, and Vitter; ESA'95
  - overlay two maps in $O(sort(n) + t/B)$ optimal IOs where $t =$ number of intersections
  - batched point location in $O((n+k)/B \log_{M/B}(n/B))$ IOs, where $k =$ number of query points
  - using $\Theta(n \log_{M/B}(n/B))$ blocks of storage (???)
- We improve on space usage as well as query time, for low-density maps, *at the expense of* $O(sort(n))$ *pre-processing*; our algorithms are simpler to implement

Challenges

- Creating a *linear* size index supporting queries in logarithmic time

  usual hierarchical decompositions support $O(\log n)$ query time but using $O(n \log n)$ space

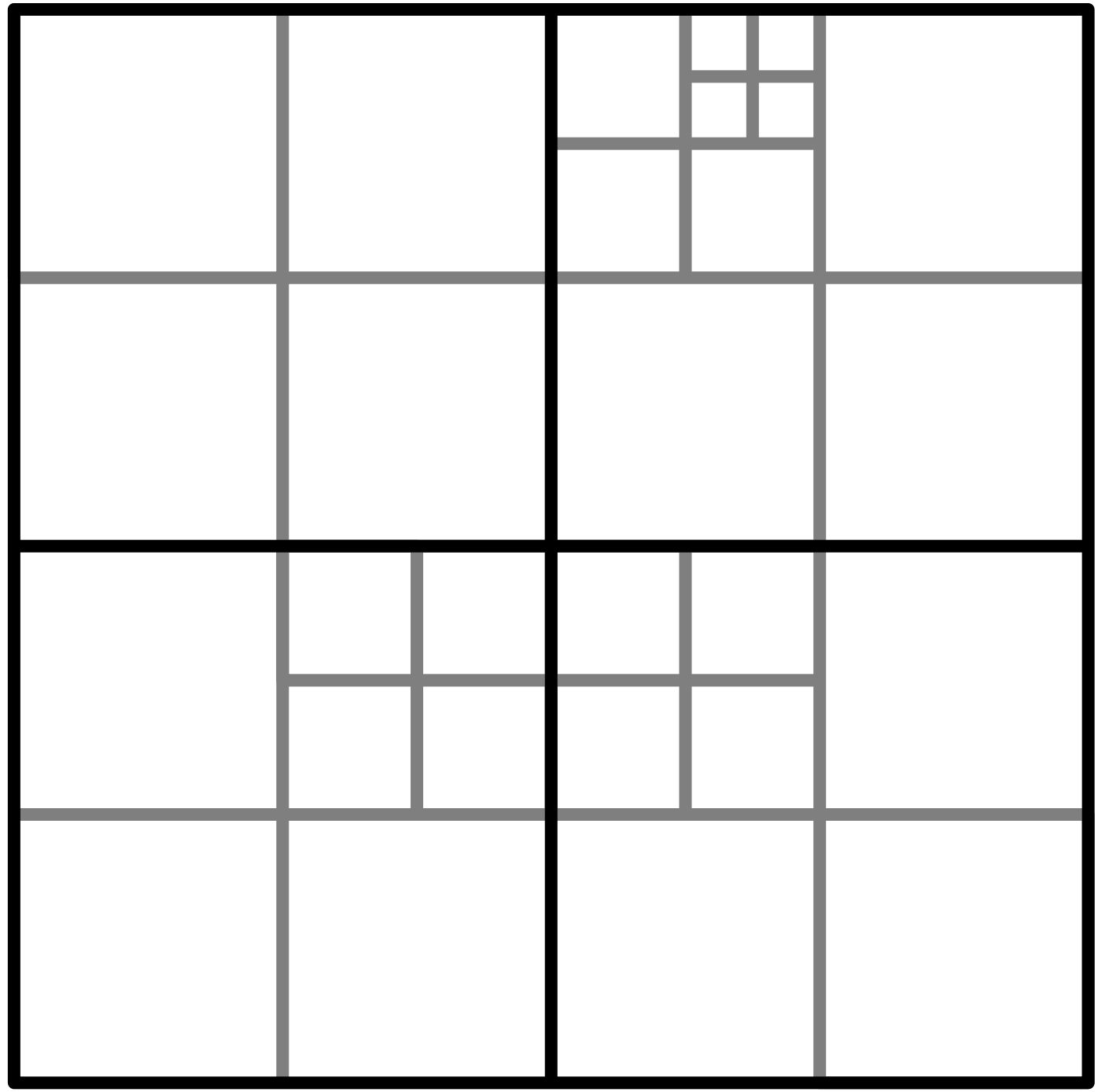
- Support efficient batched queries on the index

  to answer k queries presented in a batch more efficiently than k individual queries

- Can we overlay two maps in $O(scan(n))$ IOs?

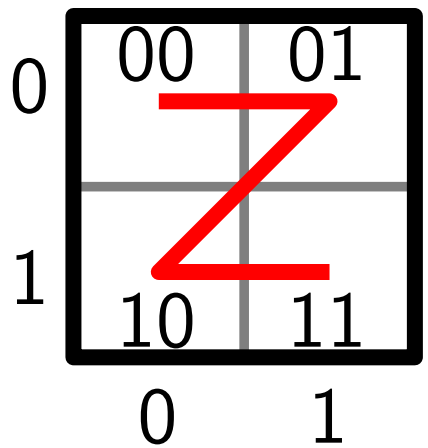
  Existing solutions too complicated and/or not IO-optimal

Quadtree

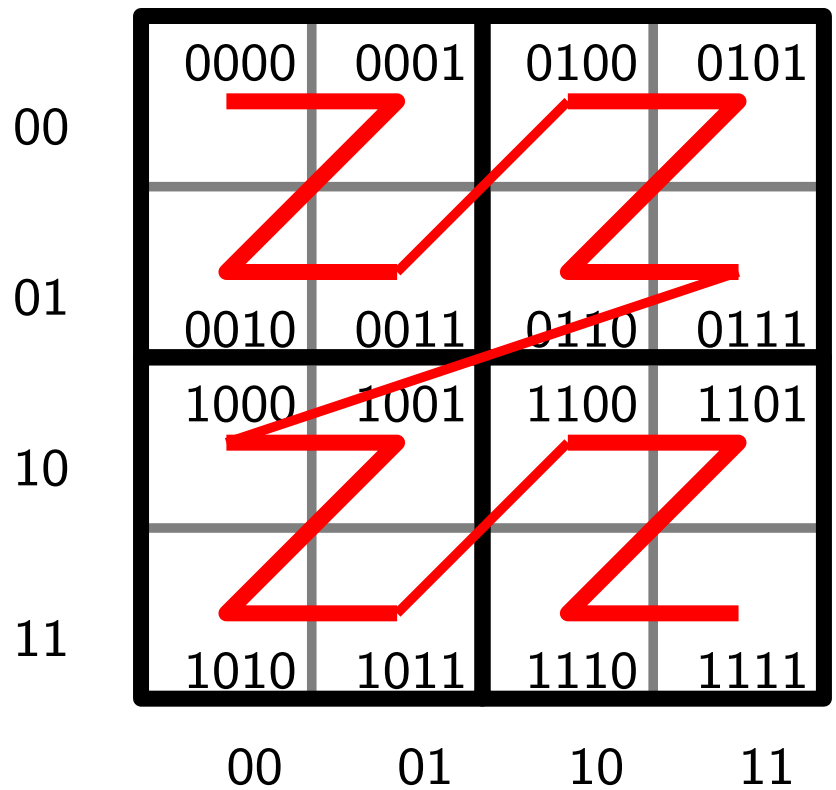


Z-curve

- Space-filling curve visits points in order of their Z-index (a.k.a. Morton block index)



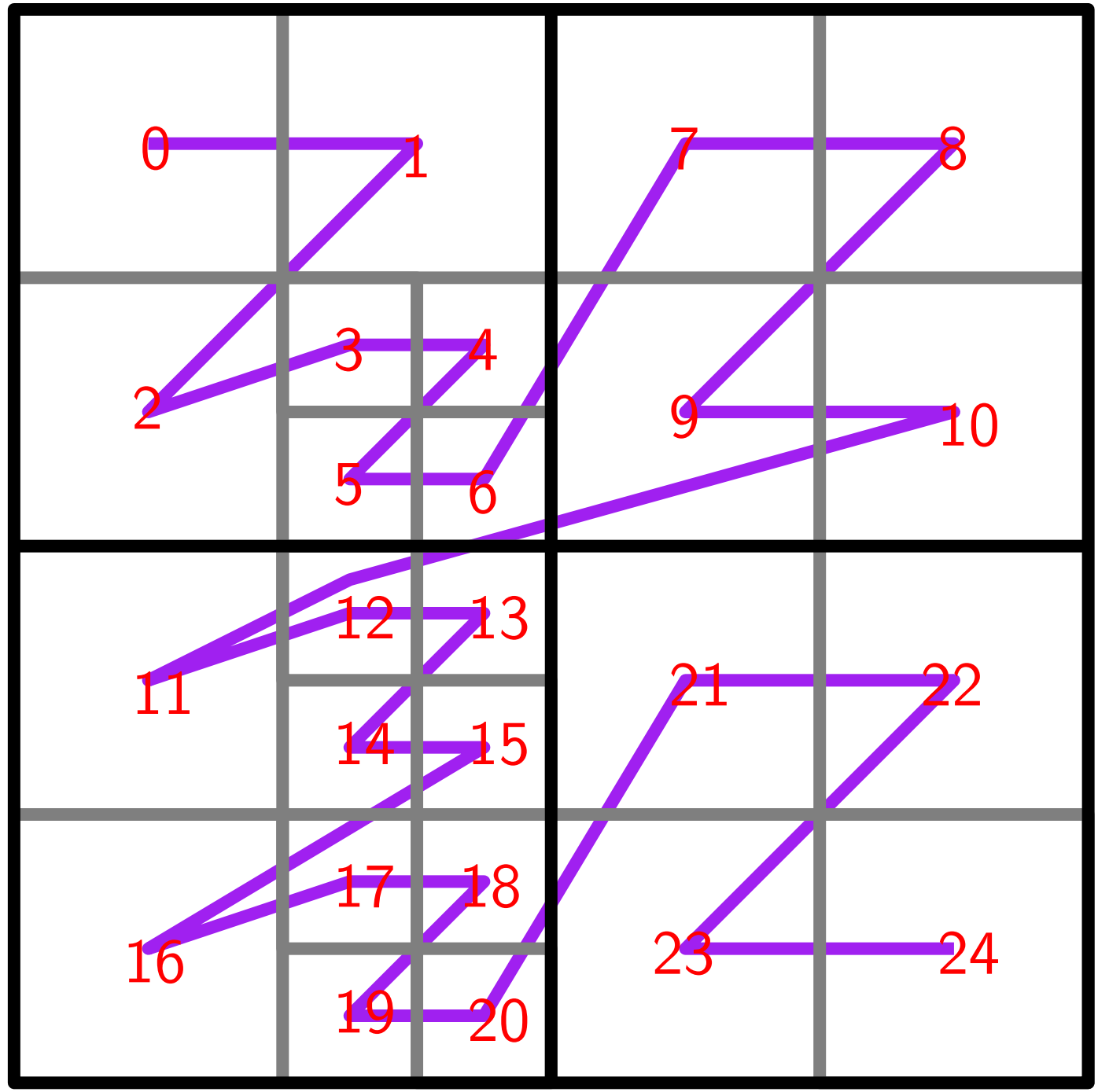
bit-interleaved order



Quadtree meets Z-curve

- Z-curve visits every quadtree cell in a contiguous interval
- The leaves of a quadtree define a subdivision of the Z-curve
- Two quadtree cells are either disjoint or nested
- Z-intervals of two quadtree cells are either disjoint or nested

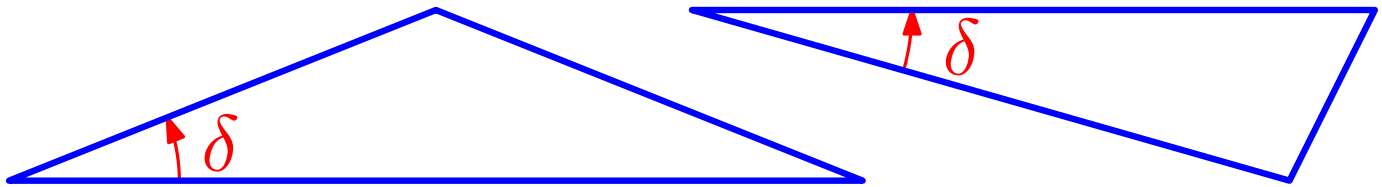
Example



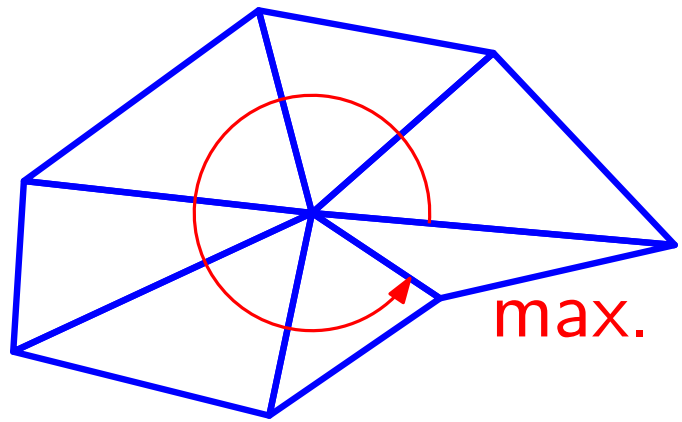
I. Fat Triangulations

Fat triangulation

- A *δ-fat triangulation* is one whose minimum angle is at least $\delta > 0$



- Our input is a triangulation with fatness δ



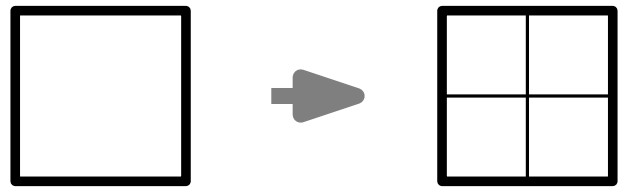
- We assume $B = \Omega(1/\delta)$ and $M = \Omega(1/\delta^3)$

Linear quadtree

- Our data structure is a *linear* quadtree:
  - a linear quadtree stores only leaves (no pointers)
  - internal nodes are represented implicitly and can be computed as required
- We store quadtree leaves in Z-order

Linear quadtree

- Recursively partition the bounding box into four quadrants



- **Novel stopping condition:**

  Stop splitting a quadtree cell when all edges intersecting the cell are incident on a common vertex

- **Lemma:** Quadtree contains $O(n/\delta^2)$ cells, each cell intersected by at most $2\pi/\delta$ triangles; total number of triangle-cell intersections is $O(n/\delta^2)$.

Building local quadtrees

- Top-down recursive algorithm to build quadtree not IO-efficient

  quadtree may have depth $\Theta(n)$, hence IO-cost is $O(n^2/B)$

- Instead, for each vertex v, build a *local* quadtree for the triangles incident on v

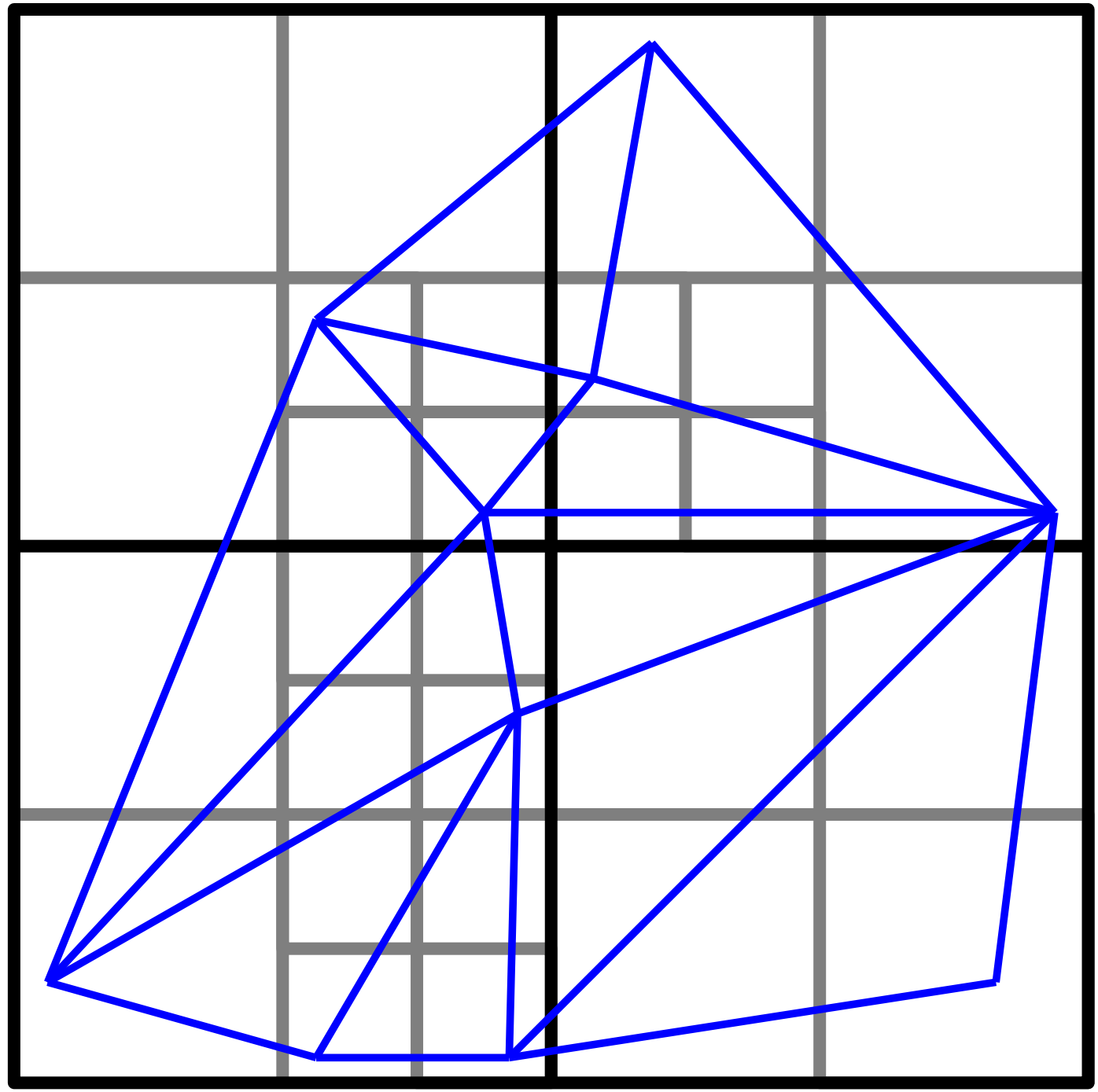
- Since vertex degree is at most $2\pi/\delta$, a local quadtree can be built entirely in internal memory

Building local quadtrees

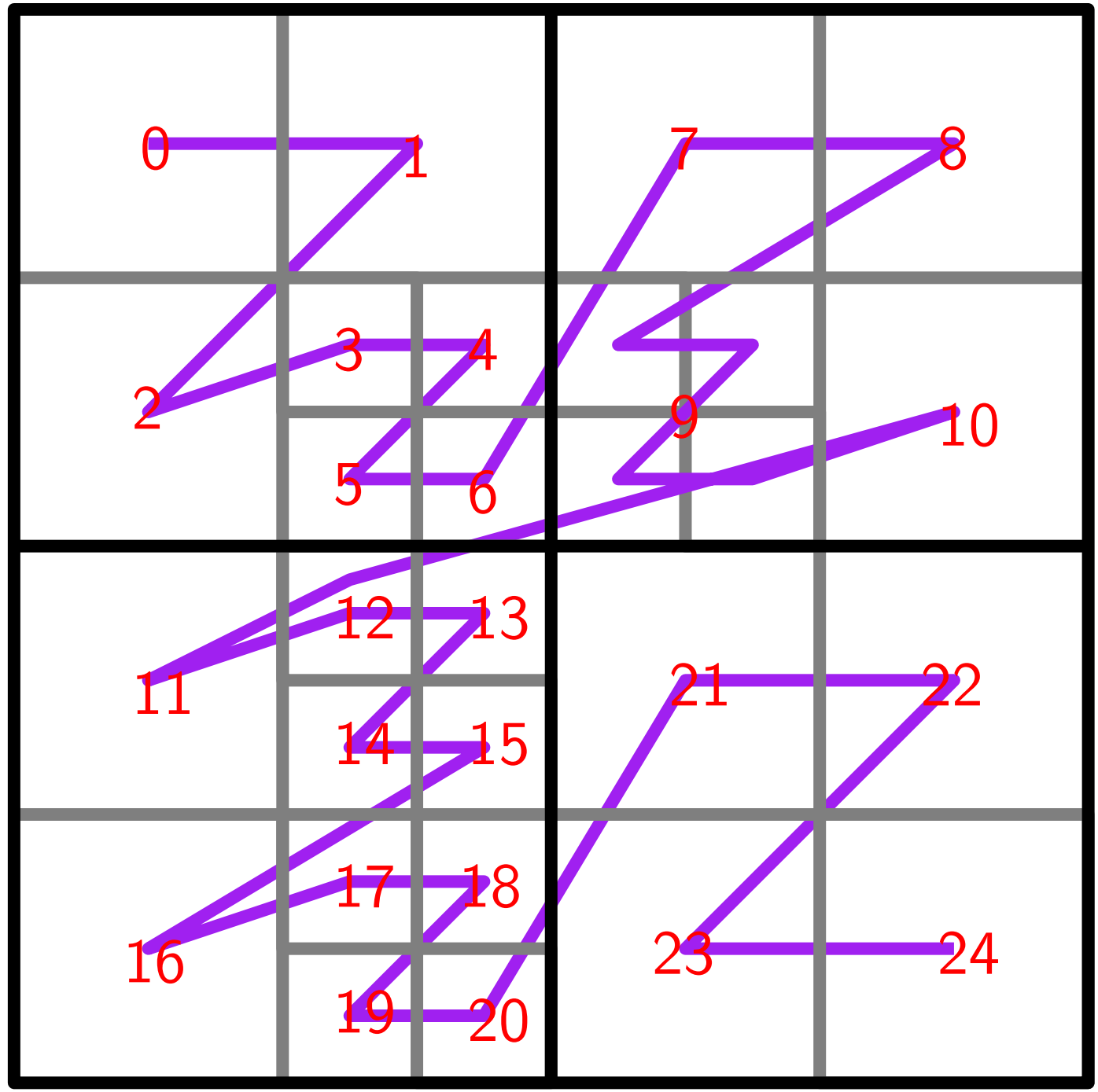
- **Lemma:** The union of all local quadtrees is identical to the global quadtree
- We need to show that every cell in the global quadtree appears in some local quadtree

  Proof: Every triangle T intersects a cell C of the global quadtree if and only if C belongs to the local quadtree of at least one of the vertices of T.

Example



Example



Building an index

- Each triangle stored with every quadtree cell that it intersects
- The *Z-index* of a cell is its order along the space-filling Z-curve
- Whenever triangle T intersects cell C, the pair (T, C) is stored with associated key equal to the Z-index of C

Indexing triangles

- Sort the $O(n/\delta^2)$ cell-triangle pairs in Z-order of cells

  $= O(sort(n/\delta^2))$ IOs

- Build a *cache-oblivious B-tree* on the set of cell-triangle pairs sorted by key (Z-index of cell)

- B-tree has size $O(n/\delta^2)$ and depth $O(\log_B(n/\delta^2))$

How to locate a single point

- Search the B-tree from root to leaf with Z-index of p for quadtree cell containing point p

$$= O(\log_B(n/\delta^2)) \text{ IOs}$$

- Check p against all triangles intersecting the cell (at most $2\pi/\delta$) in internal memory; all these triangles have the same key and are stored together

How to locate a batch of k points

- Sort the k query points by Z-index

$= O(sort(k))$ IOs

- Merge the sorted query points and the sorted leaf cells by scanning in parallel

$= O(scan(n/\delta^2 + k))$ IOs

How to overlay two triangulations

- Quadtree leaves subdivide the Z-curve into disjoint intervals
- Since quadtree leaves are sorted in Z-order, the intervals are in sorted order
- Merge the two sorted sets of intervals, corresponding to the quadtrees of the two triangulations

$= O(scan(n/\delta^2))$ IOs

How to support updates

- Each of the following operations affects $O(1/\delta^4)$ entries in the B-tree:
  - insert/delete a vertex
  - flip an edge
- Each update affects a local quadtree; perform corresponding changes to the global quadtree

$= O(\frac{1}{\delta^4} \log_B(n/\delta^2))$ IOs per update

Summary: fat triangulations

- We build a *linear quadtree*, from local quadtrees of small neighborhoods, using a *novel stopping condition*
- The quadtree leaves are stored in a *cache-oblivious B-tree*, indexed by their order along the *Z-order* space-filling curve
- The B-tree has linear size and logarithmic depth, thus supporting efficient queries and updates
- Two such quadtrees can be overlaid by scanning; the two indexes are merged in the process

II. Low-Density Maps

Low-density maps

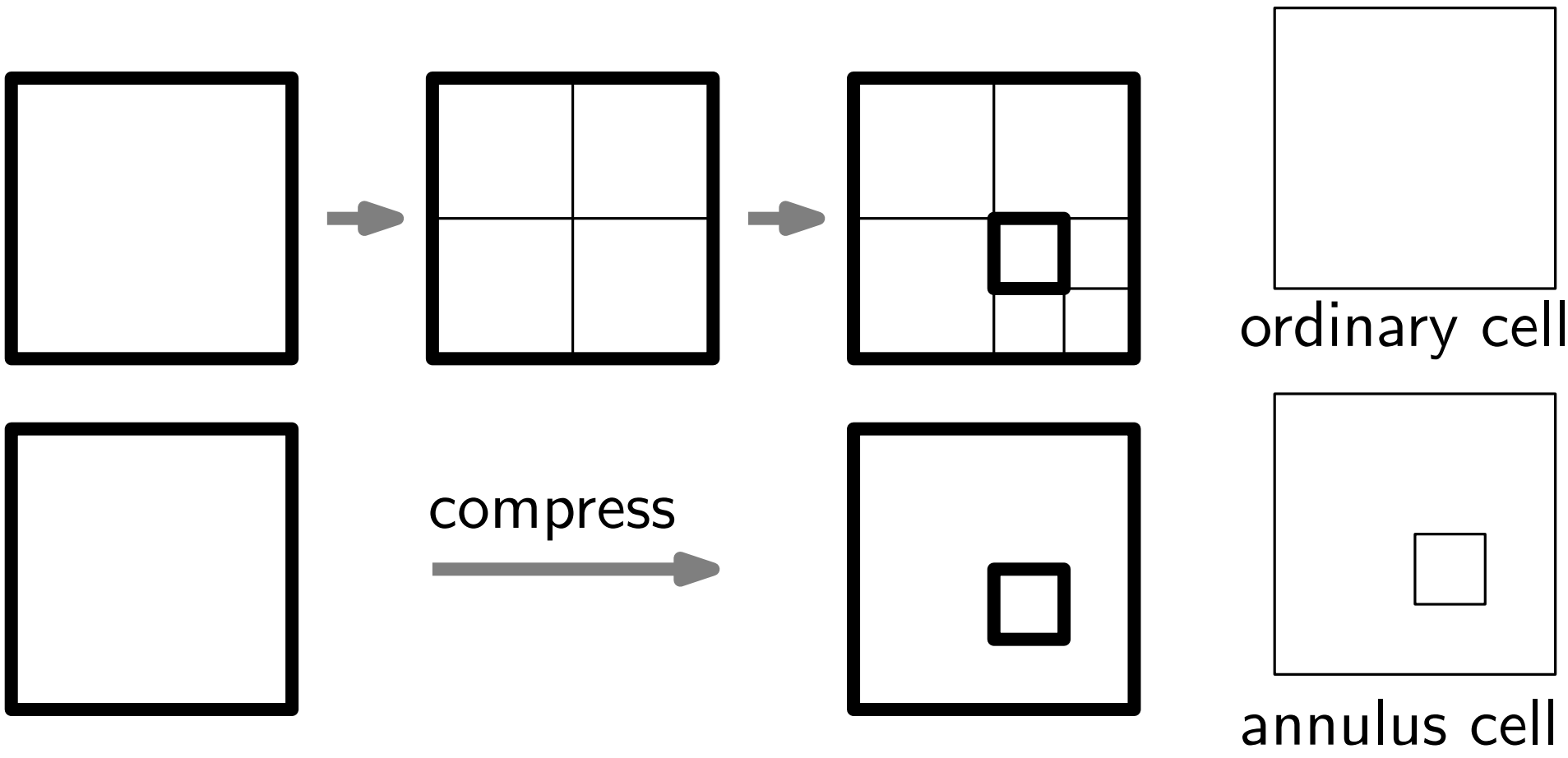
- The *density* of a set S of objects is the smallest number λ such that every disk D intersects at most λ objects of S whose diameter is at least the diameter of D

- The density of a planar map is the density of its *edge set*
- Our input is a map with density λ
- We assume $B = \Omega(\lambda)$

A δ-fat triangulation has density $\lambda = O(1/\delta)$

Compressed quadtree

- An *annulus* is the set-theoretic difference of two ordinary nested cells



- An annulus can be represented by *two nested Z-intervals*

Compressed linear quadtree

- We introduce *compressed* linear quadtrees:
  - a compressed quadtree has many fewer nodes than an ordinary quadtree
  - a compressed quadtree has more complicated cells (annuli); our storage scheme handles such cells

Quadtree of guarding points

- Build a compressed quadtree of guarding points of edges

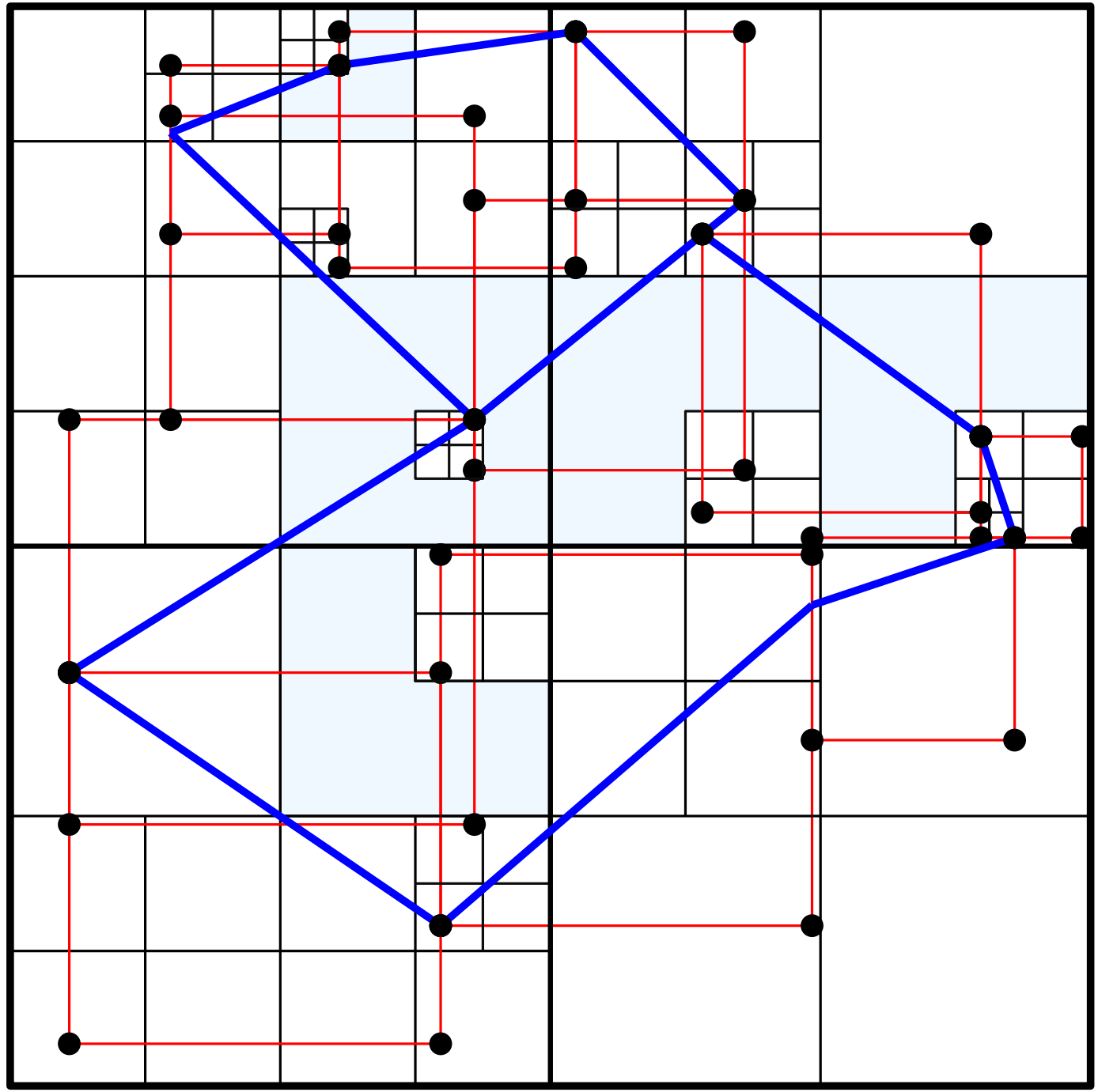
  Guarding points of an edge = vertices of the axis-aligned bounding square

- **Stopping condition:**

  Stop splitting a quadtree cell when it contains only one guarding point

- **Lemma [de Berg et al.]**: A square containing g guarding points intersects at most $g + 4\lambda$ edges

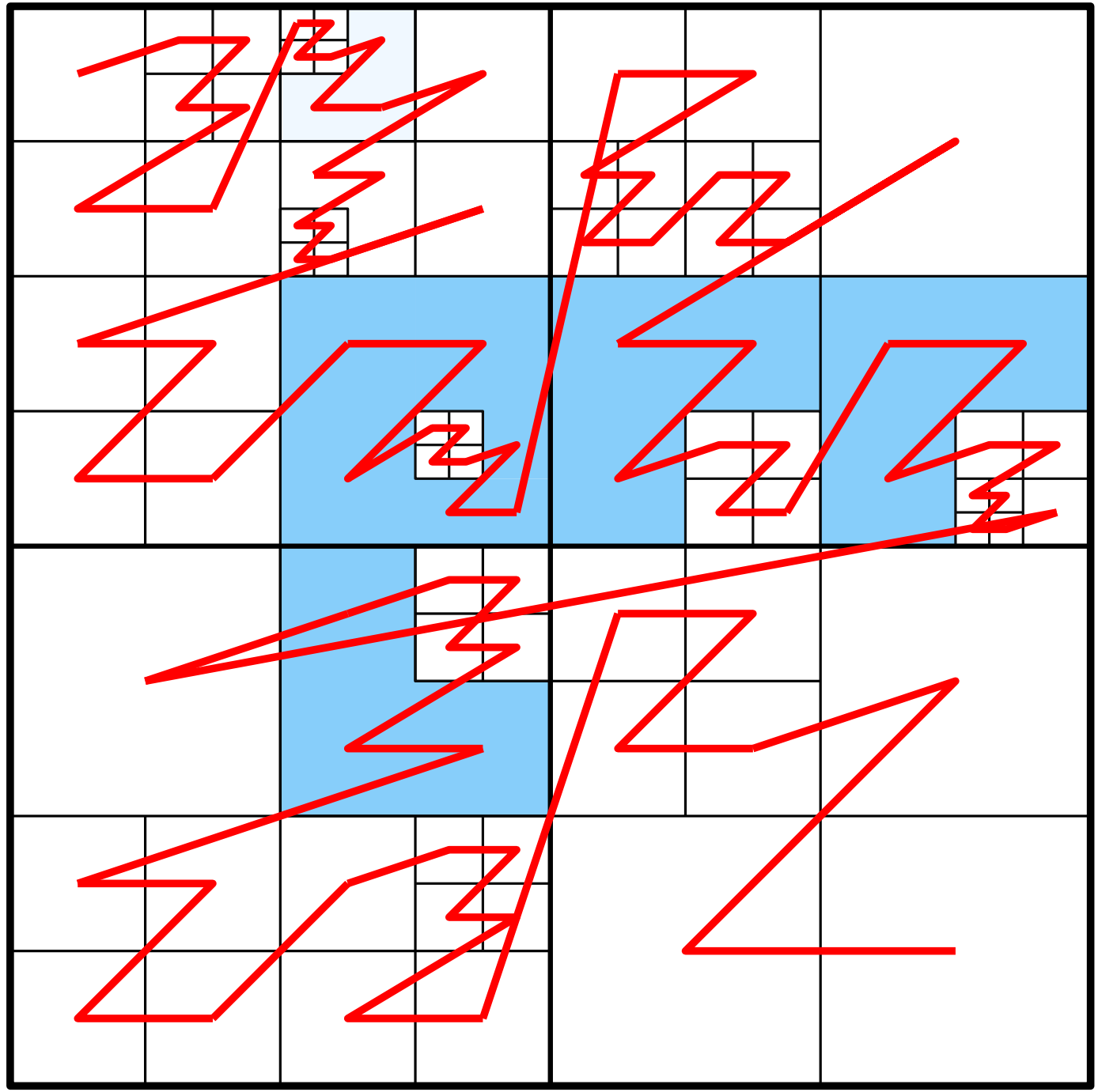
Example



How to build a quadtree of points

- Sort guarding points in Z-order
- For each consecutive pair of points, output their local quadtree: their canonical bounding square and its four children
- Sort all cells and remove duplicates
- **Result:** Compressed quadtree of guarding points in $O(sort(n))$ IOs, where leaf cells are sorted in Z-order

Example



Computing cell-edge intersections

- We distribute the edges of the subdivision among the quadtree leaf cells

  For each edge e, we compute the quadtree cells that it intersects in a batched filtering

  use cache-oblivious distribution sweeping?

- A quadtree leaf cell not intersected by any edge is repeatedly merged with a predecessor or successor cell in Z-order

Small-size quadtree

- **Lemma:** Compressed quadtree of guarding points contains $O(n)$ leaf cells, each leaf intersected by at most $O(\lambda)$ faces; total number of face-cell intersections is $O(n\lambda)$.

- Build a B-tree on the set of cell-edge pairs sorted by key (Z-index of cell)

- B-tree has $O(n)$ leaves and depth $O(\log_B n)$

How to locate a single point

- Search the B-tree from root to leaf with Z-index of p for quadtree cell containing point p

$= O(\log_B n)$ IOs

- Check p against all $O(\lambda)$ faces intersecting the cell, in internal memory; all these faces have the same key and are stored together

How to locate a batch of k points

- Sort the k query points by Z-index

$= O(sort(k))$ IOs

- Merge the sorted query points and the sorted leaf cells by scanning in parallel

$= O(scan(n + k))$ IOs

How to overlay two maps

- Quadtree leaves subdivide the Z-curve into disjoint intervals
- Since quadtree leaves are sorted in Z-order, the intervals are in sorted order
- Merge the two sorted sets of intervals, corresponding to the quadtrees of the two maps

$= O(scan(n))$ IOs

Summary: low-density maps

- We introduce *compressed* linear quadtrees
- We build a compressed linear quadtree of the set of $O(n)$ *guarding points* for the edges of the subdivision
- We store the quadtree leaves (only) in sorted order along the Z-order space-filling curve
- We build a 1D index, a B-tree of linear size, on the quadtree leaves supporting efficient queries
- Making construction and update algorithms cache-oblivious remains an open problem

Implementation

- The Z-order of a point is its bit-interleaved order

$$Z(x_0x_1\ldots x_b, y_0y_1\ldots y_b) = \underbrace{x_0y_0x_1y_1\ldots x_by_b}_{2b\text{-bit integer}}$$

- The canonical bounding box of two points is computed from the longest common prefix of the bitstring representing their coordinates
- Several optimizations described in our paper

Summary

- We preprocess a fat triangulation or low-density subdivision in $O(sort(n))$ IOs so we can:
  - answer k batched point location queries in $O(scan(n) + sort(k))$ IOs
  - overlay two maps in $O(scan(n))$ IOs
- We give simple, practical, implementable, fast, scalable algorithms!
- Our algorithms for triangulations are cache-oblivious

To read more ...

I/O-Efficient Map Overlay and Point Location in Low-Density Subdivisions
Mark de Berg, Herman Haverkort, ST, Laura Toma

http://www.win.tue.nl/~sthite/pubs/

Condensed version to appear at EuroCG 2007

Thanks to Sariel Har-Peled for valuable discussions

Future work

- Implementation (in TPIE?)
- IO-efficient range searching in low-density subdivisions
- IO-efficient overlay of general subdivisions, not assuming fatness or low density

Tak!