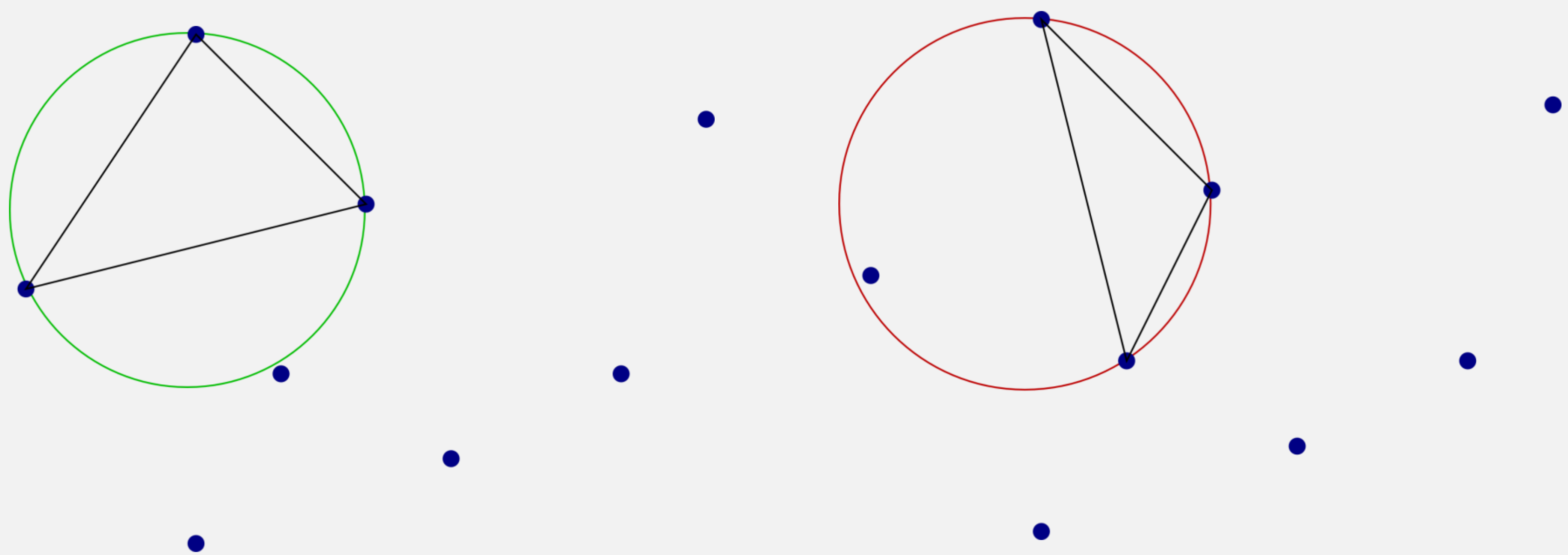
High-Performance Delaunay Triangulation for Many-Core Computers

V. Fuetterling and C. Lojewski

Contact: valentin.fuetterling@itwm.fhg.de

Basic property of the Delaunay triangulation (DT)

- No other points inside the circumcircle of a triangle



Applications for the DT

- point location
- path finding
- image processing
- mesh generation
- etc…

Contribution of the talk

- DT implementation for 2D point sets
  - **Multi-threaded**
  - High single-threaded performance
  - Big data sets

➡ Results 40-50x faster than previous implementations

Problem

DIFFICULTIES FOR A PARALLEL DT IMPLEMENTATION

Looking at previous implementations CGAL and Triangle

CGAL

- Point-insertion

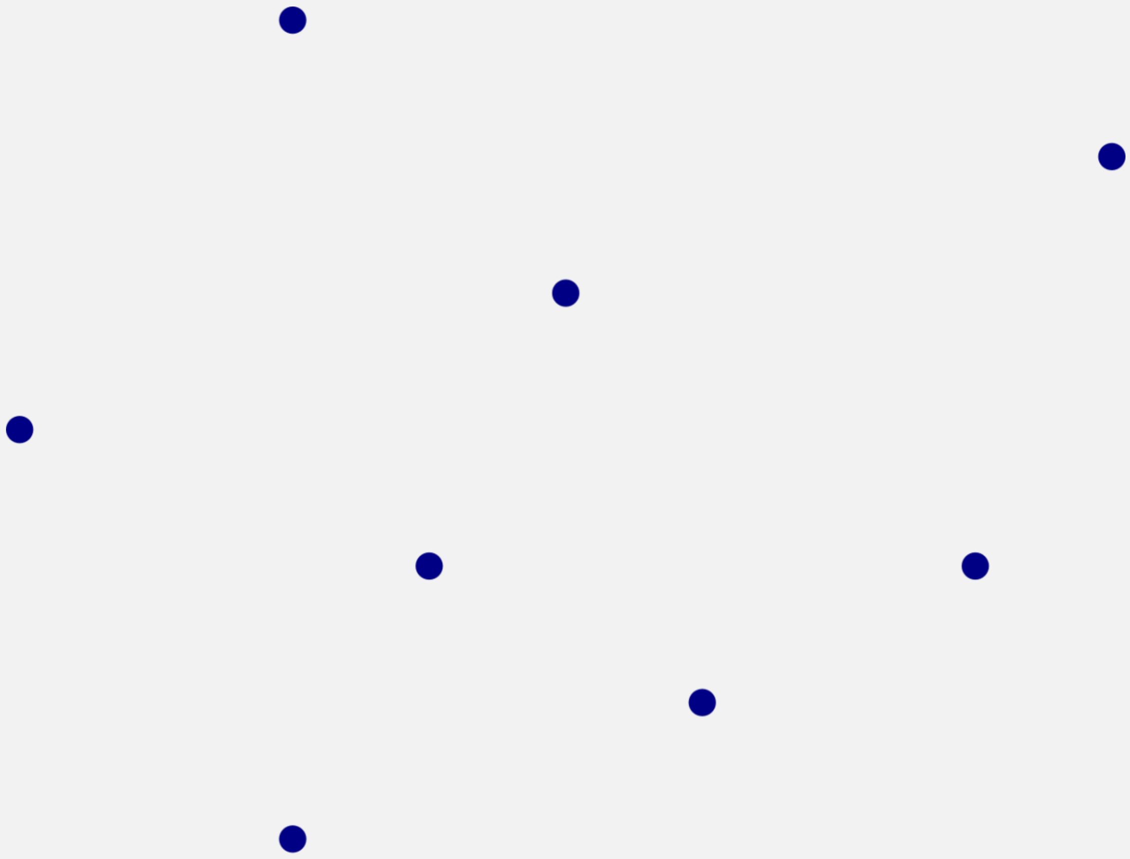
(www.cgal.org)

Triangle

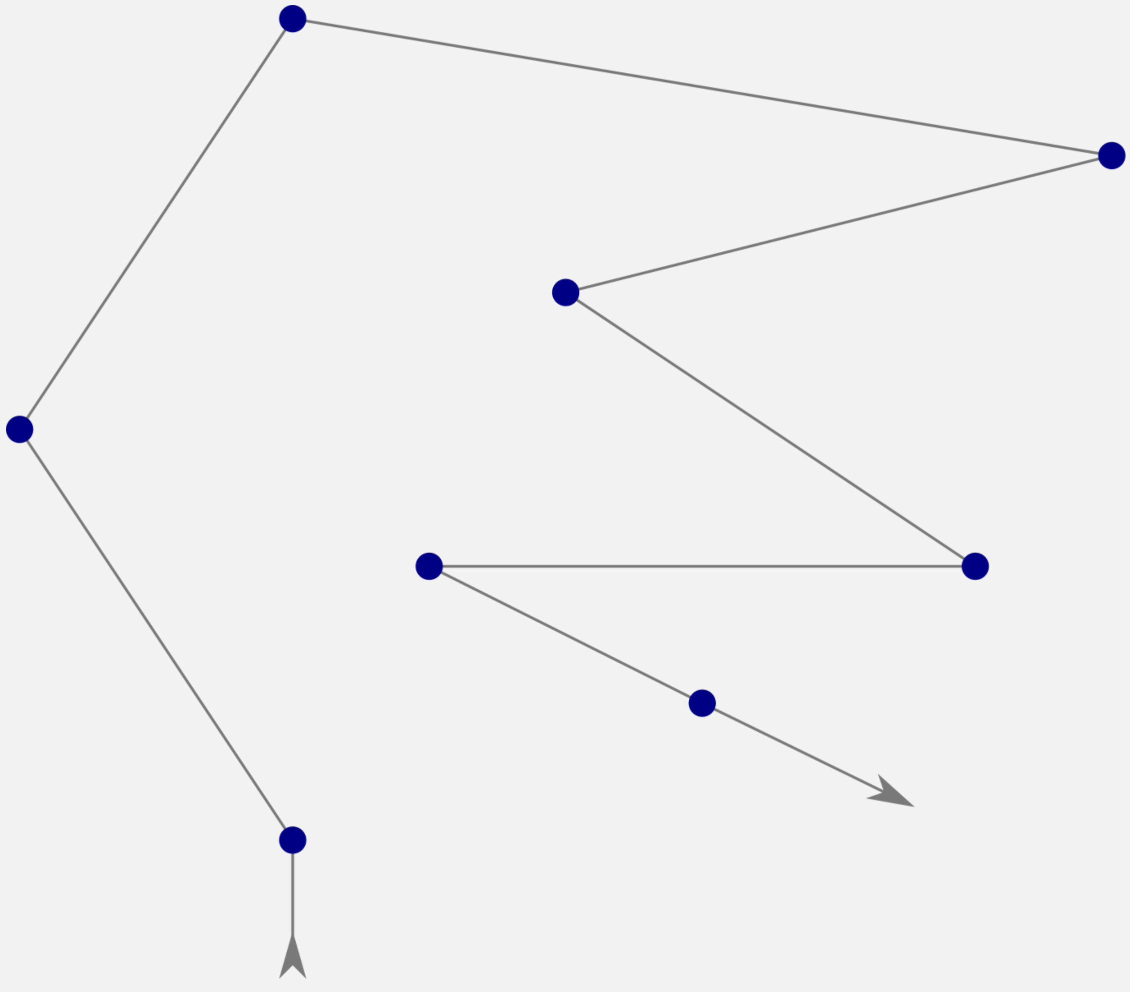
- Divide-and-conquer (Dwyer's algorithm)

(www.cs.cmu.edu/~quake/triangle.html)

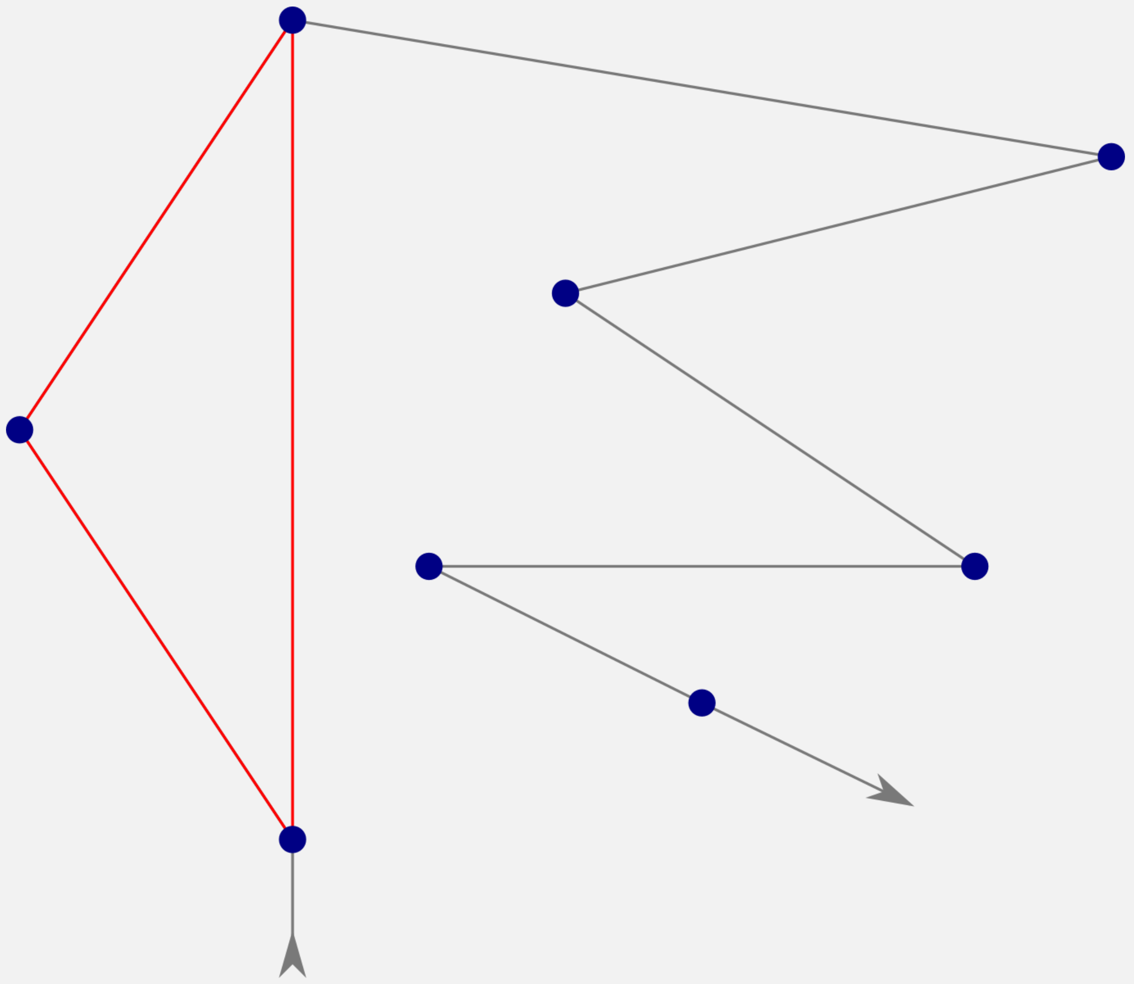
CGAL algorithm



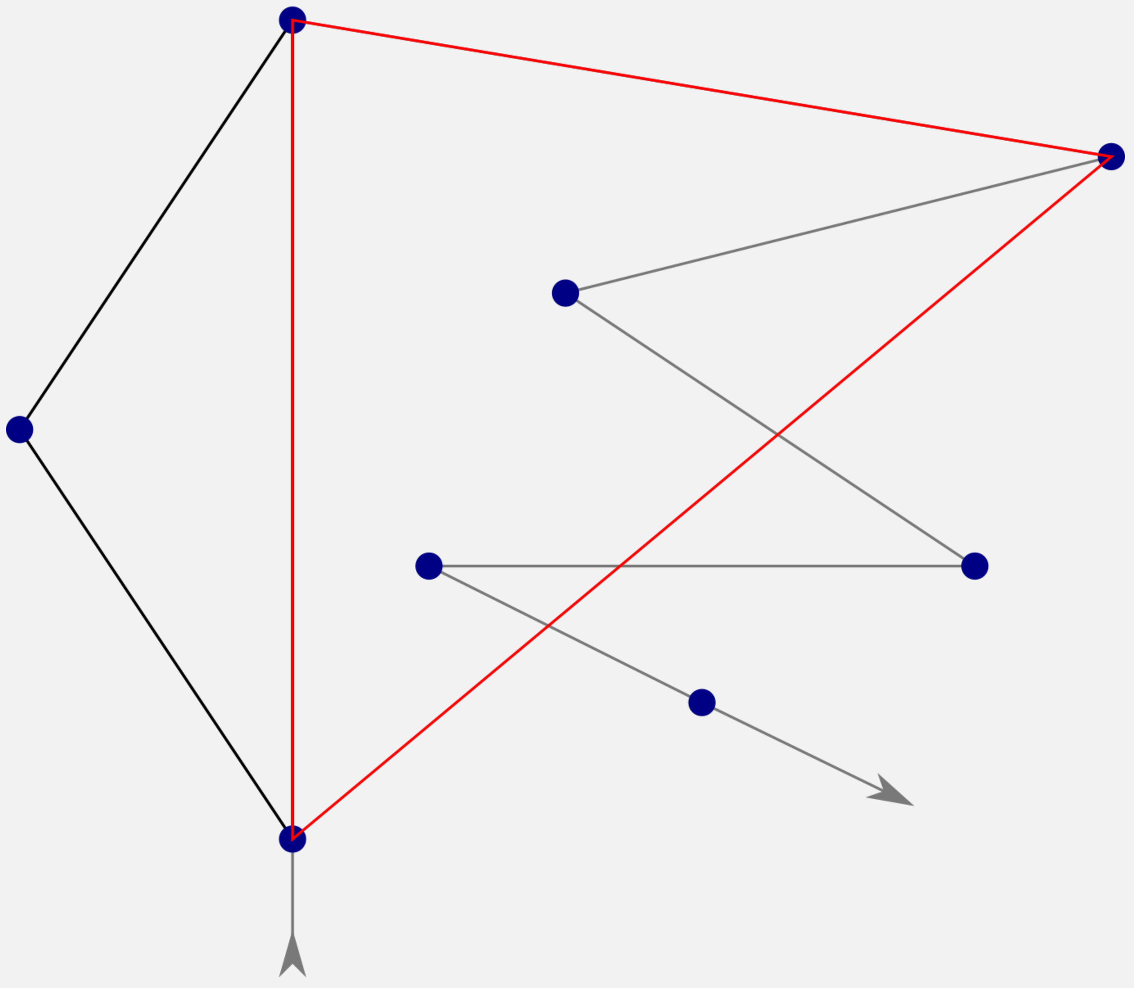
CGAL algorithm



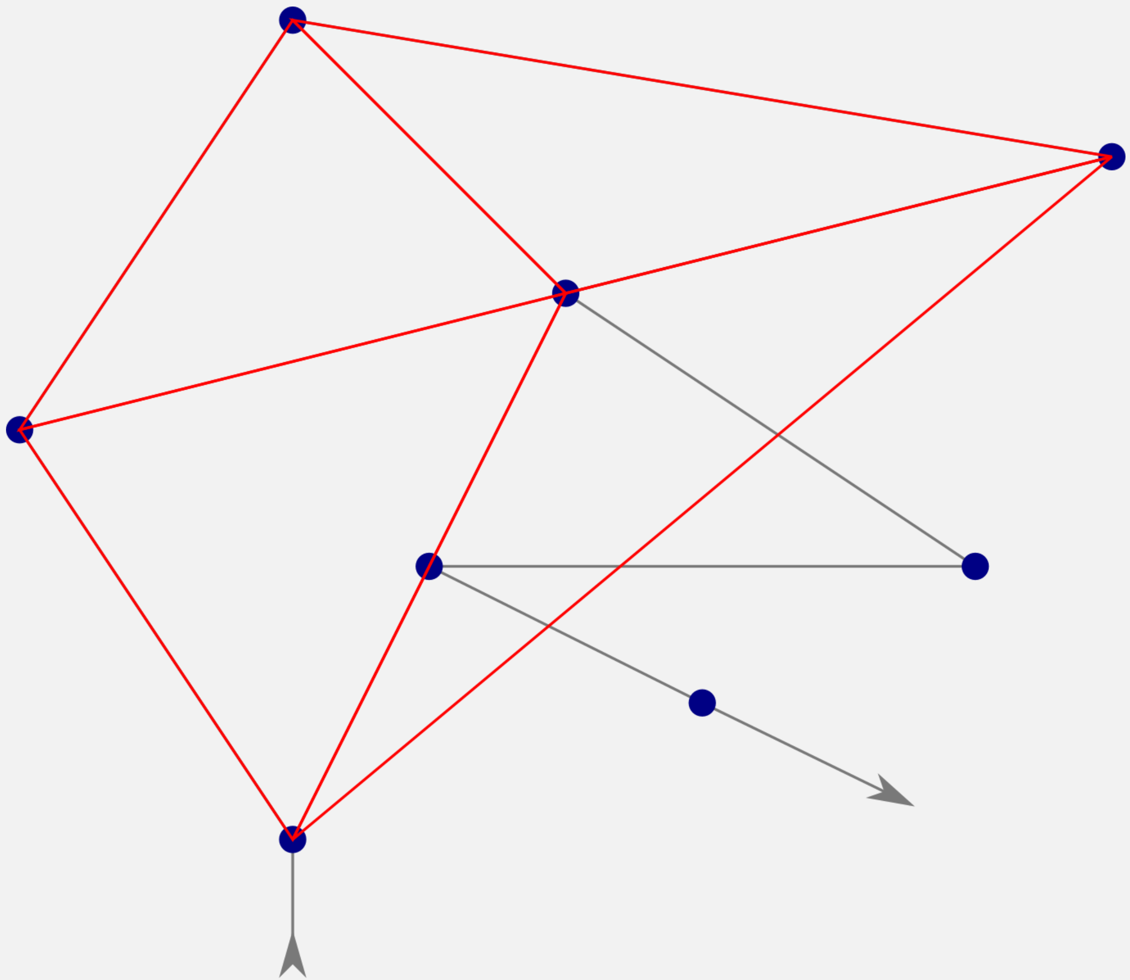
CGAL algorithm



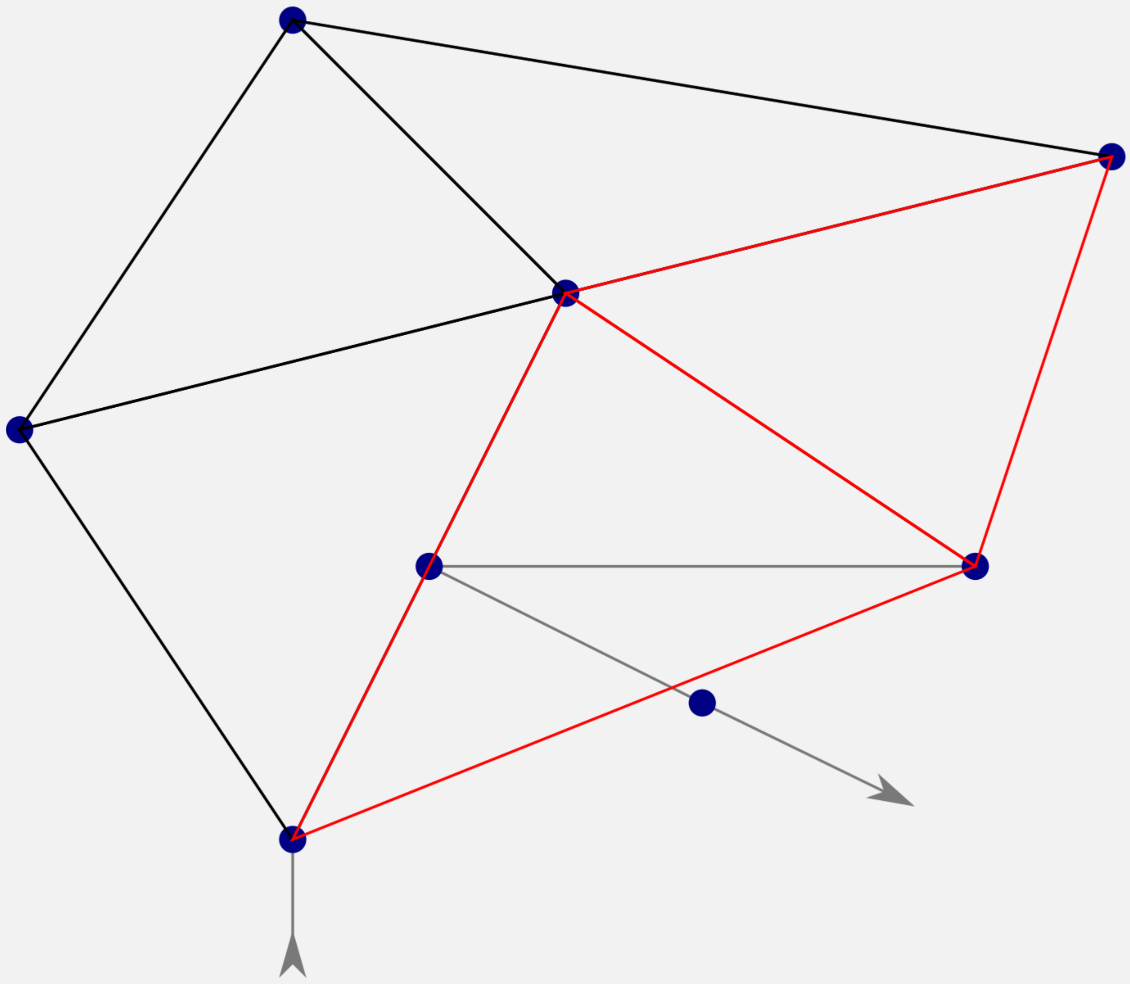
CGAL algorithm



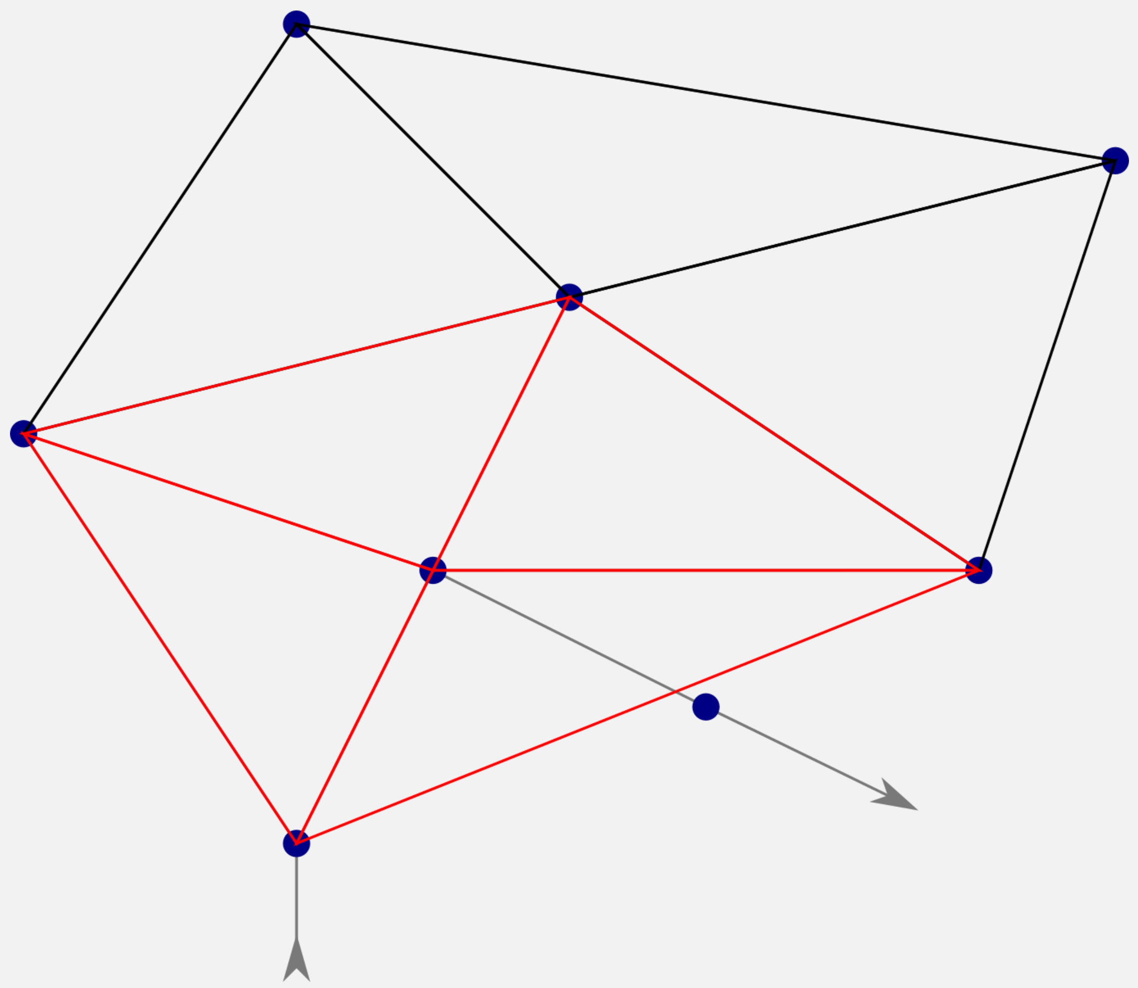
CGAL algorithm



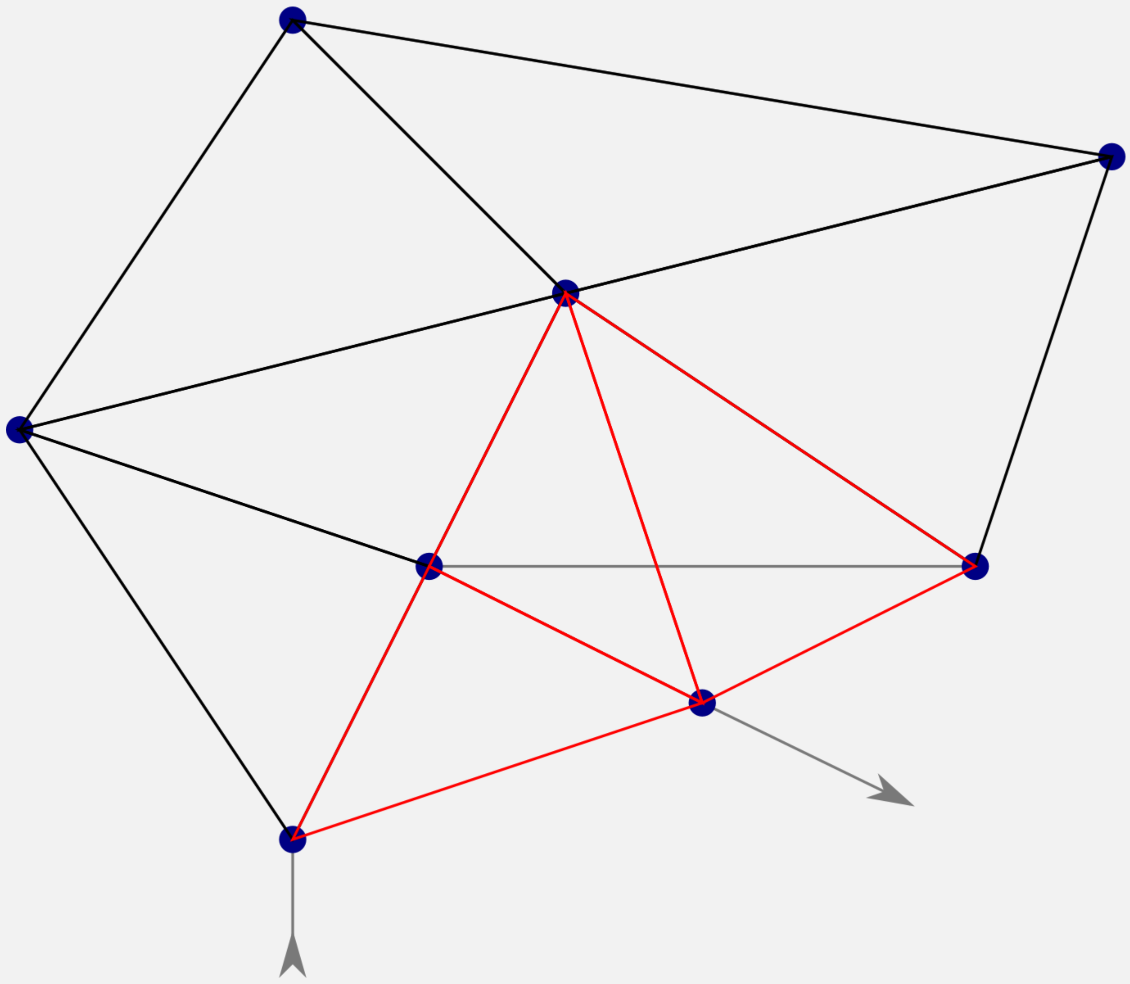
CGAL algorithm



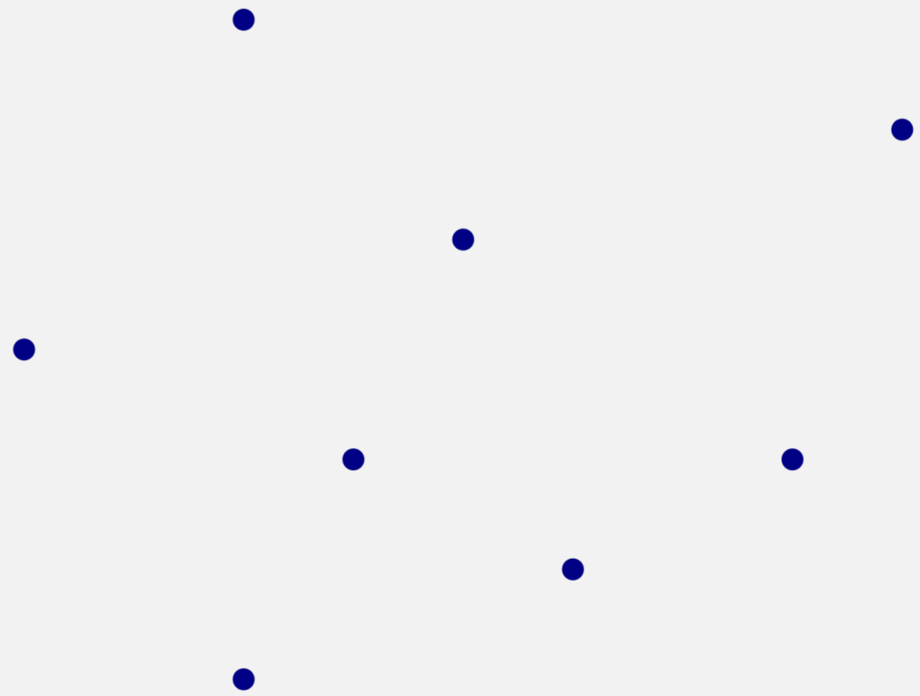
CGAL algorithm



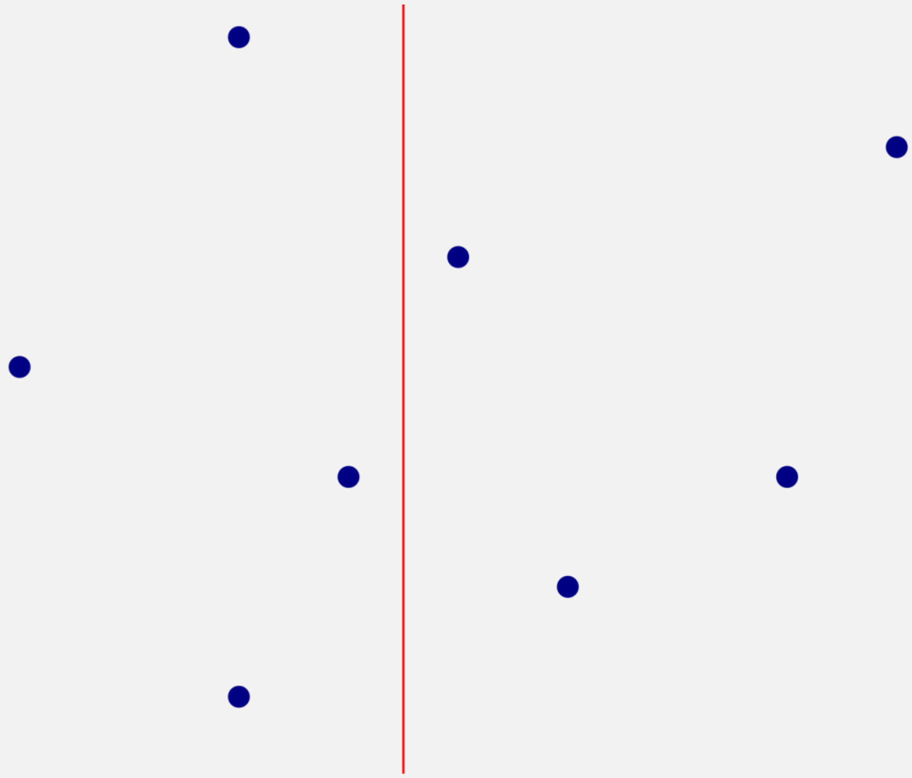
CGAL algorithm



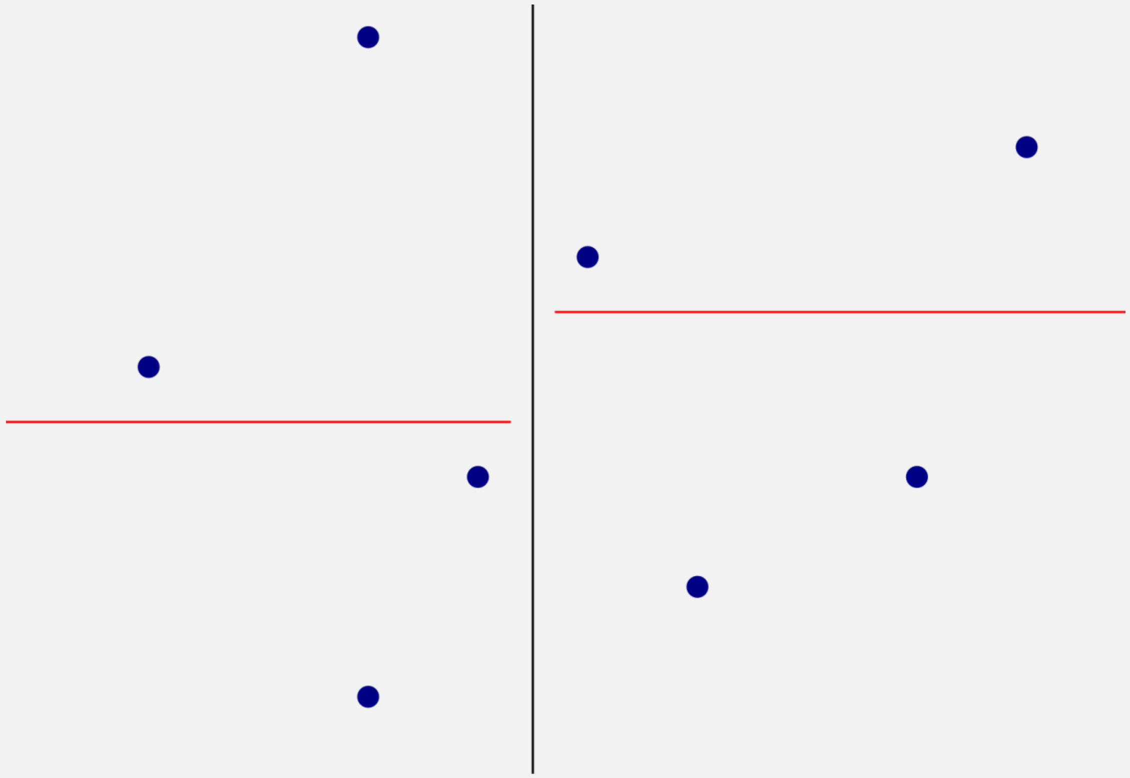
Triangle algorithm



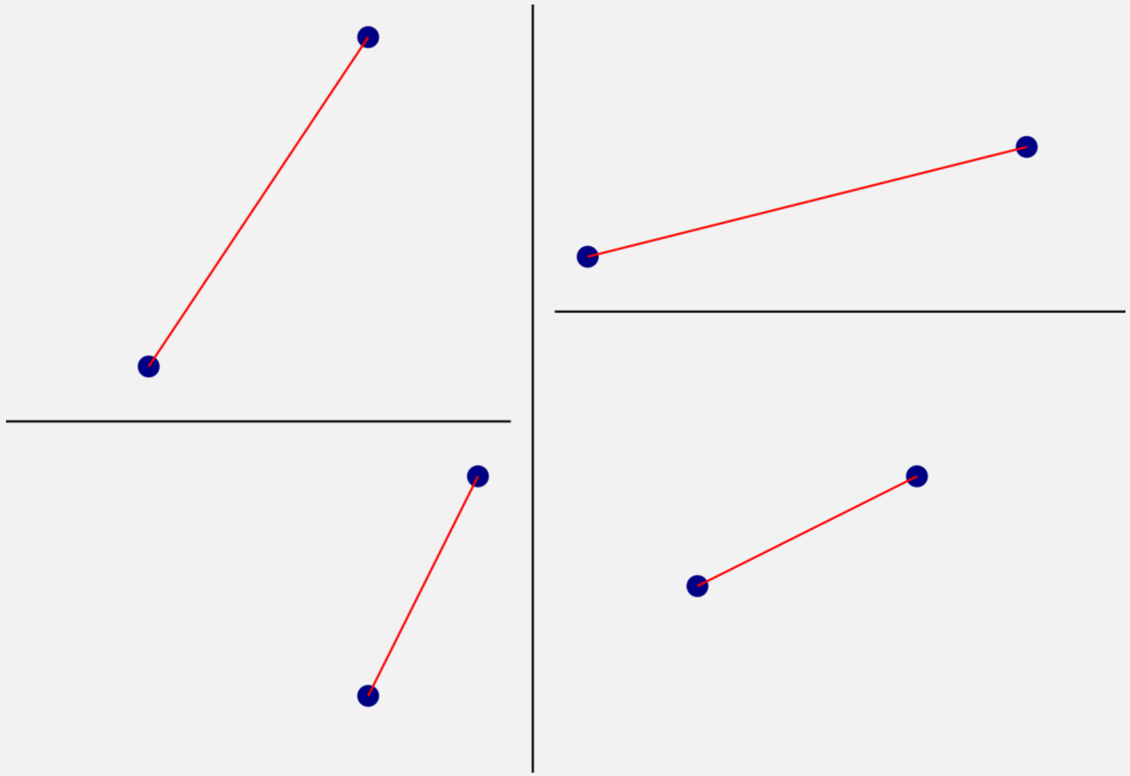
Triangle algorithm



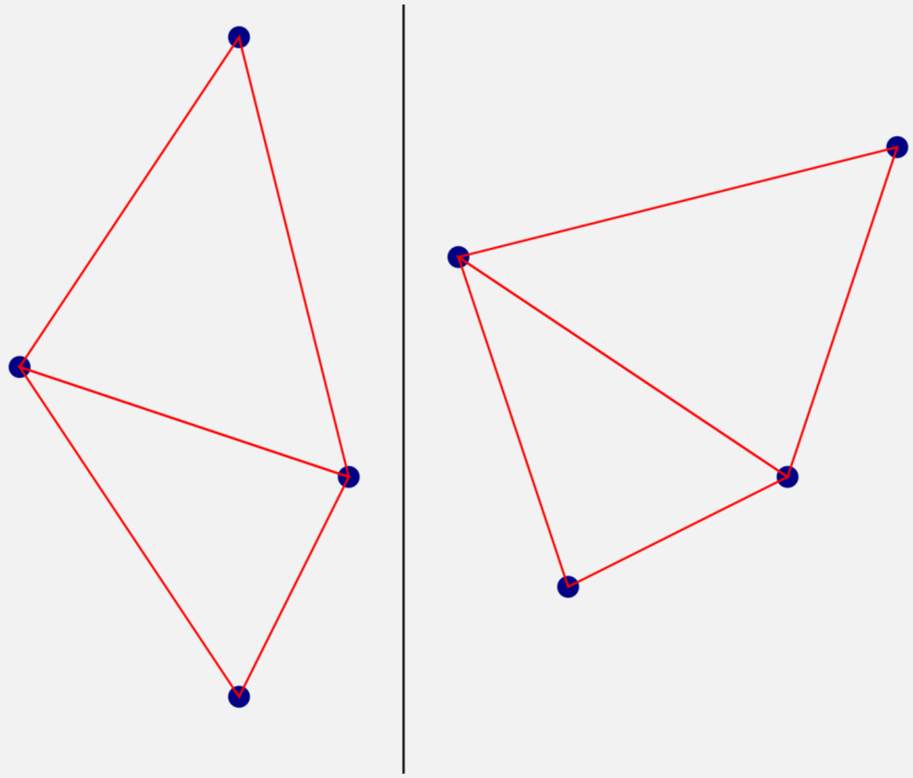
Triangle algorithm



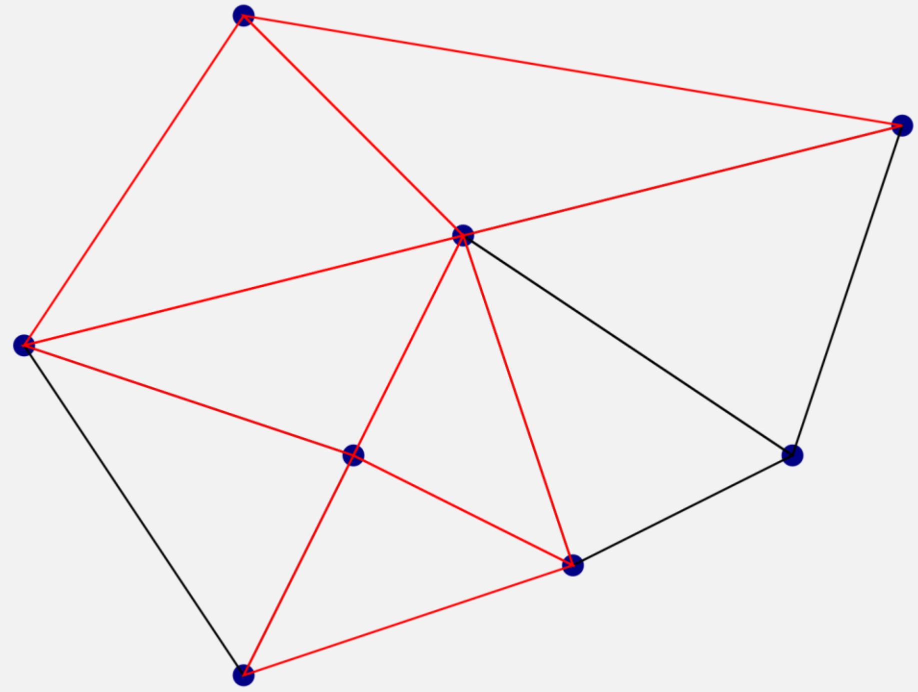
Triangle algorithm



Triangle algorithm



Triangle algorithm



Difficulties with parallelization

- CGAL:
  - Multiple threads need to read/modify a shared data structure
- Triangle:
  - D&C: Limited parallelism at the start
  - Devisive sorting algorithm, problematic for scalability (compare top-down BVH construction)

Difficulties with parallelization

- CGAL:
  - Multiple threads need to read/modify a shared data structure

- Triangle:
  - D&C: Limited parallelism at the start
  - Devisive sorting algorithm, problematic for scalability (compare top-down BVH construction)

Our solution

THE LINEAR QUAD-TREE (WITH A TWIST)

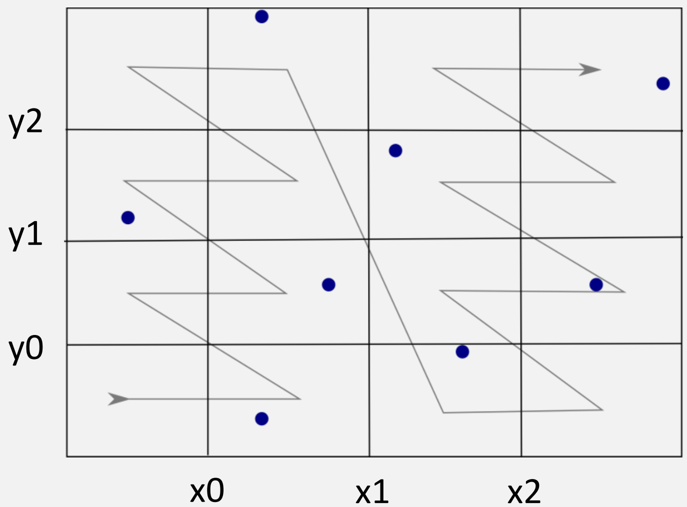
Linear Quad-tree

- Concept known from BVH construction algorithms (linear oct-tree):
  - HLBVH [Pantaleoni, Luebke, 2010]
  - AAC [Gu, He, Fatahaliam, Blelloch, 2013]

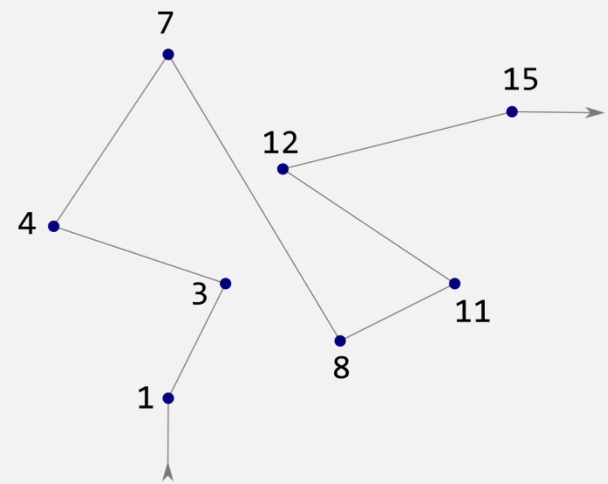
- Basic idea:
  Morton codes + (Radix) sort -> memory layout of points corresponds to depth-first traversal of quad-tree

Linear Quad-tree

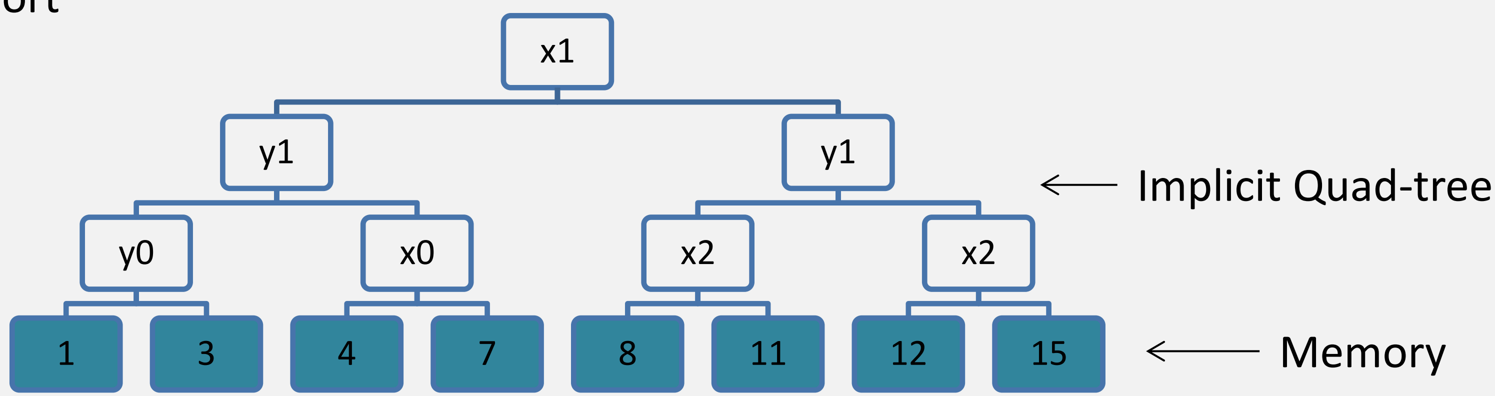
1. Define grid



2. Compute Morton codes

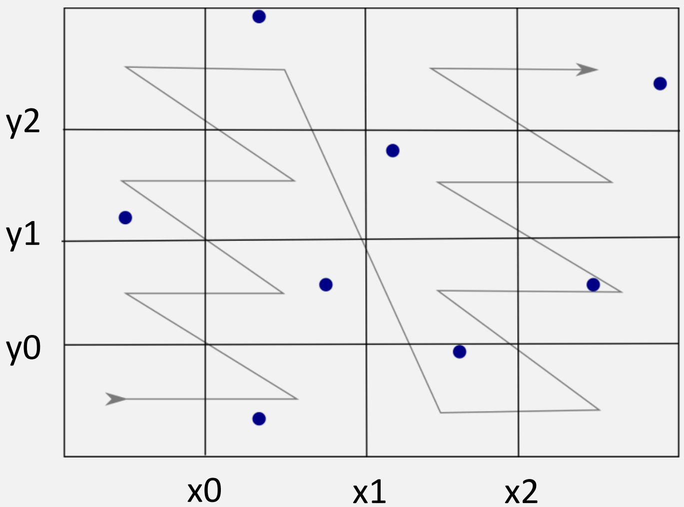


3. (Radix) sort

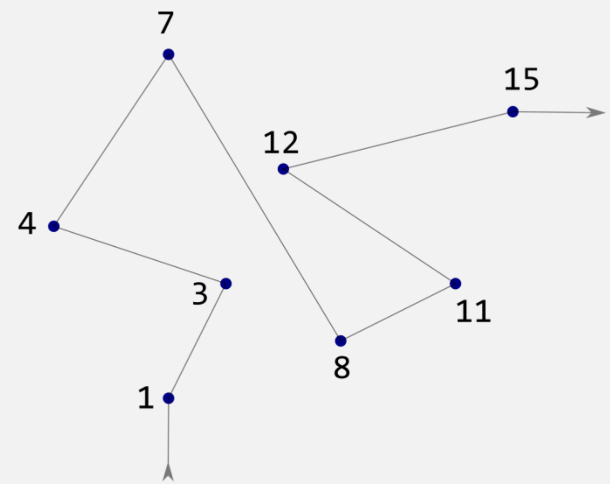


Linear Quad-tree

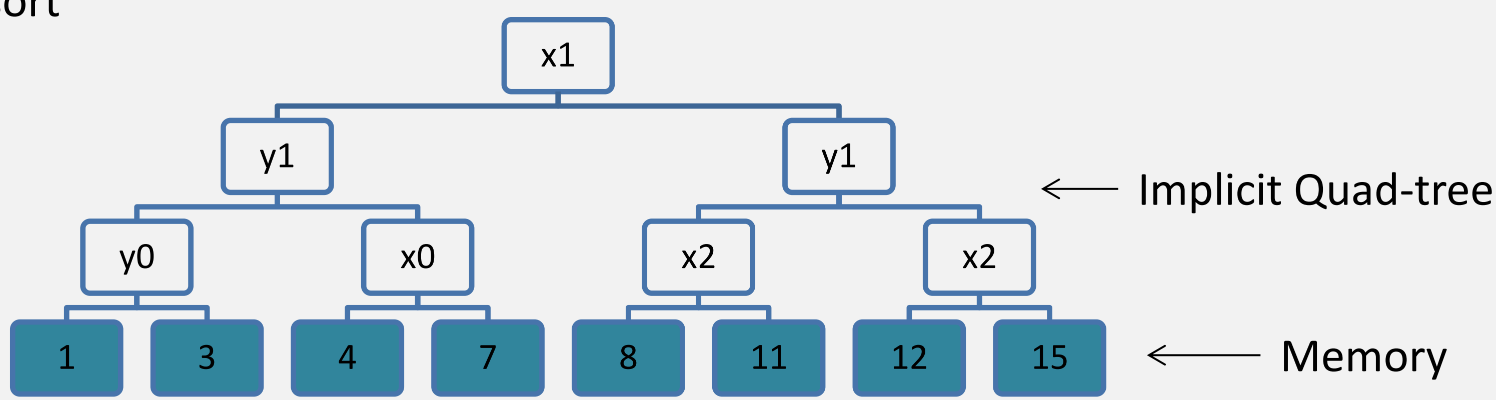
1. Define grid



2. Compute Morton codes

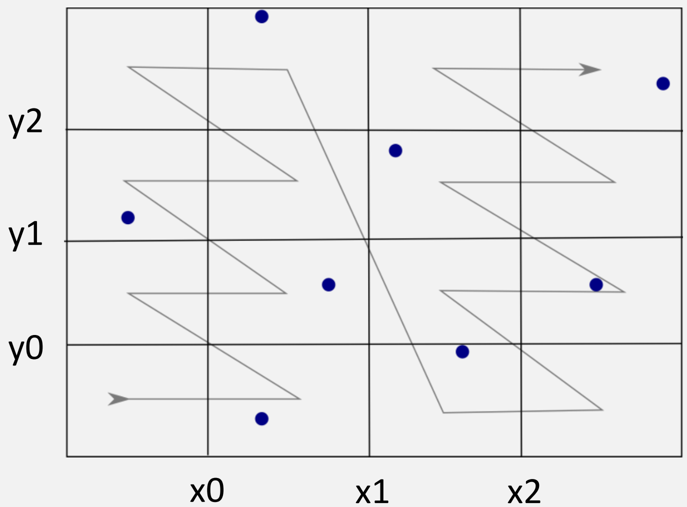


3. (Radix) sort

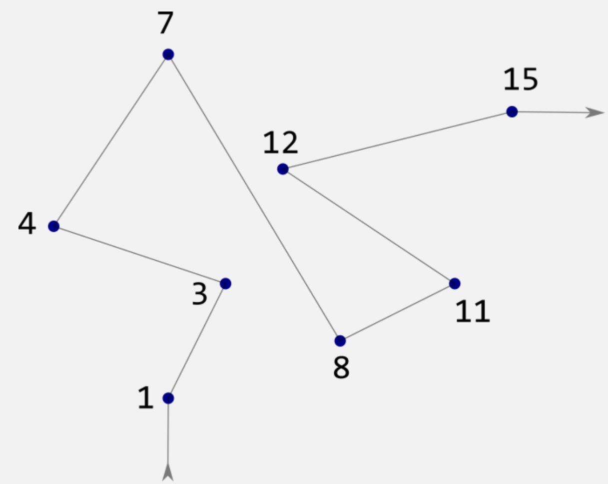


Linear Quad-tree

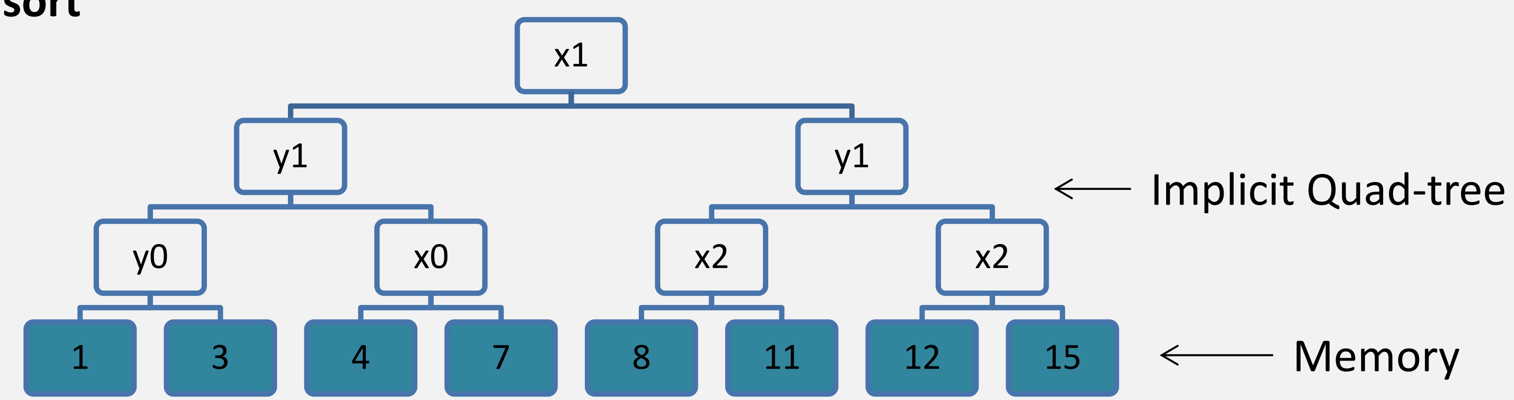
1. Define grid



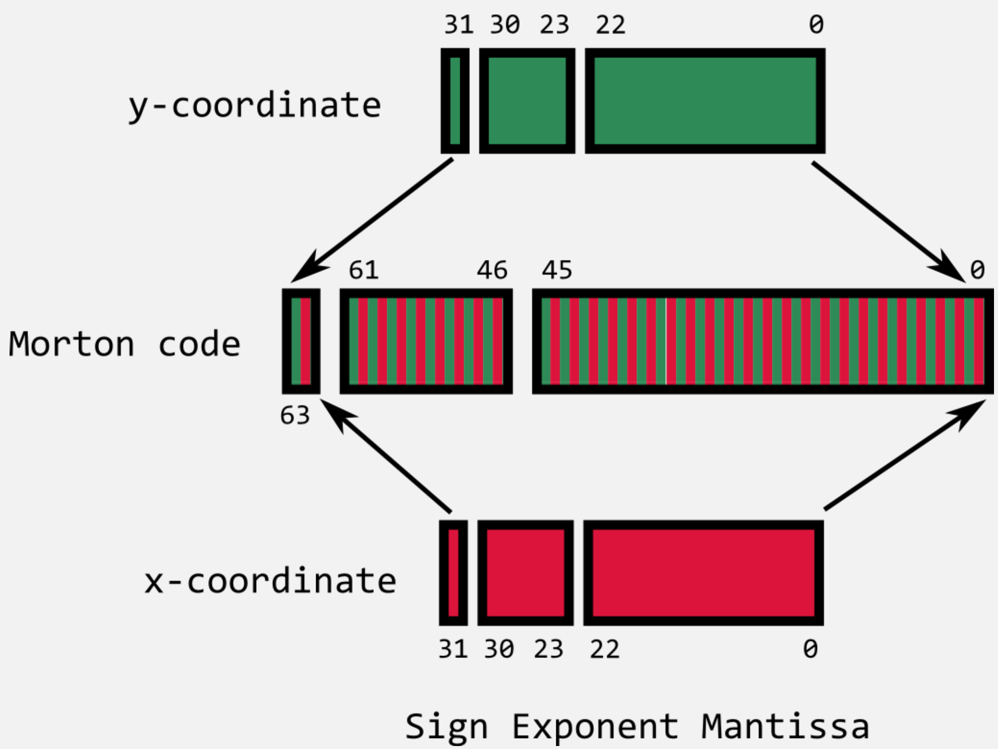
2. Compute Morton codes



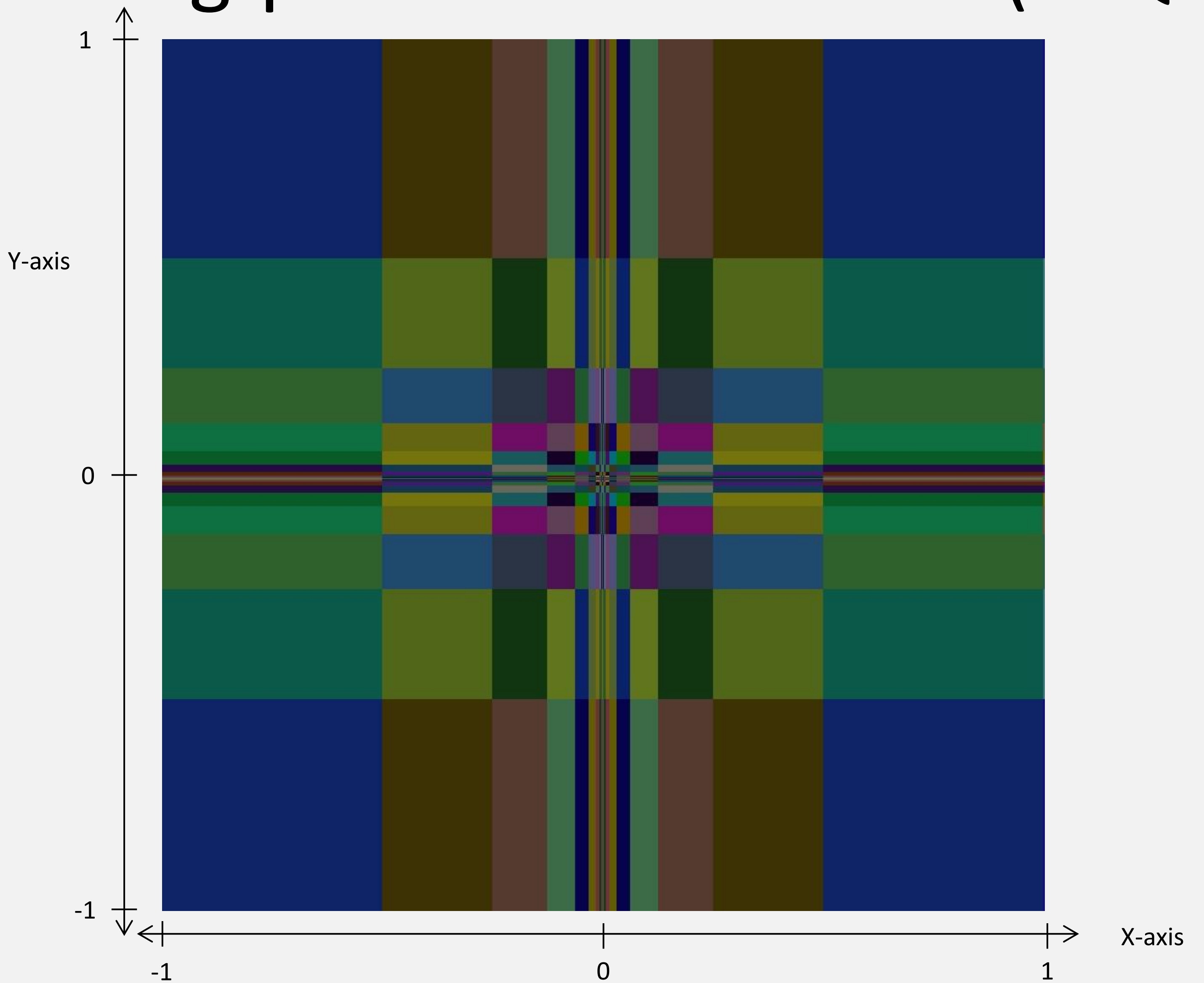
3. (Radix) sort



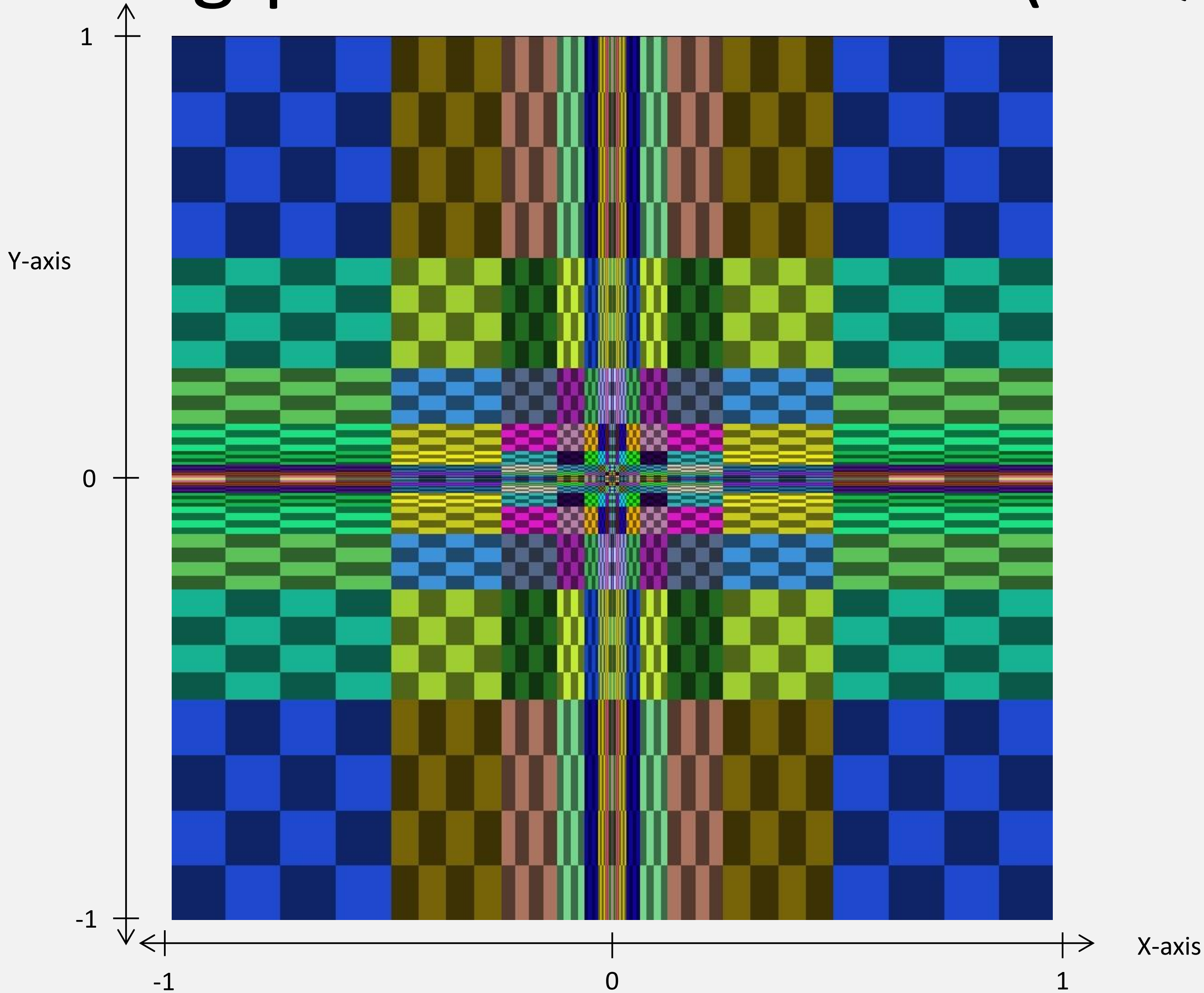
Twist: Morton codes directly from floating-point representation



Quad-tree structure generated by floating-point Morton codes (LFQT)



Quad-tree structure generated by floating-point Morton codes (LFQT)



Advantages of LFQT compared to regular linear Quad-tree

- Bijectivity: code <-> value
  - ‘infinite’ resolution, i.e. one exclusive grid cell for every possible point
  - Reduced memory footprint
- One fixed grid for every possible data set
- Rigorous numerical structure, used for adaptive precision arithmetic -> see paper

The algorithm

AN EFFICIENT, PARALLEL DT IMPLEMENTATION

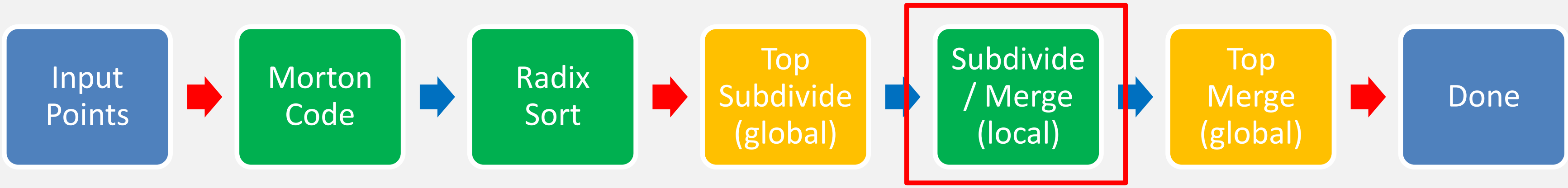
Algorithm phases

Full parallelism

Limited parallelism

Global synchronization

Algorithm phases



Full parallelism

Limited parallelism

Global synchronization

Subdivide method

Input: Array of points from lidx to ridx

Algorithm 1 Subdivision of the floating point quad-tree.

```
Subdivide(lidx, ridx)
if points[lidx] is equal to points[ridx − 1] then
    Decode points lidx to ridx − 1
    return Partition with single point lidx
else if ridx − lidx is equal to 2 then
    Decode points lidx and lidx + 1
    return Partition with points lidx and lidx + 1
else
    l ← lidx
    r ← ridx − 1
    level ← BSR( points[l] ⊕ points[r] )
    mask ← Shift left 1 by level
    while not (points[l + 1] & mask) do
        m ← (l + r)/2
        if points[m] & mask then
            r ← m
        else
            l ← m
        end if
    end while
    left ← Subdivide(lidx, l + 1)
    right ← Subdivide(l + 1, ridx)
    return Merge(left, right)
end if
```

Subdivide method

Single point or only degenerate points left?

Algorithm 1 Subdivision of the floating point quad-tree.

```
Subdivide(lidx, ridx)
if points[lidx] is equal to points[ridx − 1] then
    Decode points lidx to ridx − 1
    return Partition with single point lidx
else if ridx − lidx is equal to 2 then
    Decode points lidx and lidx + 1
    return Partition with points lidx and lidx + 1
else
    l ← lidx
    r ← ridx − 1
    level ← BSR( points[l] ⊕ points[r] )
    mask ← Shift left 1 by level
    while not (points[l + 1] & mask) do
        m ← (l + r)/2
        if points[m] & mask then
            r ← m
        else
            l ← m
        end if
    end while
    left ← Subdivide(lidx, l + 1)
    right ← Subdivide(l + 1, ridx)
    return Merge(left, right)
end if
```

Subdivide method

Two points left?

Algorithm 1 Subdivision of the floating point quad-tree.

```
Subdivide(lidx, ridx)
if points[lidx] is equal to points[ridx − 1] then
    Decode points lidx to ridx − 1
    return Partition with single point lidx
else if ridx − lidx is equal to 2 then
    Decode points lidx and lidx + 1
    return Partition with points lidx and lidx + 1
else
    l ← lidx
    r ← ridx − 1
    level ← BSR( points[l] ⊕ points[r] )
    mask ← Shift left 1 by level
    while not (points[l + 1] & mask) do
        m ← (l + r)/2
        if points[m] & mask then
            r ← m
        else
            l ← m
        end if
    end while
    left ← Subdivide(lidx, l + 1)
    right ← Subdivide(l + 1, ridx)
    return Merge(left, right)
end if
```

Subdivide method

Find most significant bit which is different

Algorithm 1 Subdivision of the floating point quad-tree.

```
Subdivide(lidx, ridx)
if points[lidx] is equal to points[ridx − 1] then
    Decode points lidx to ridx − 1
    return Partition with single point lidx
else if ridx − lidx is equal to 2 then
    Decode points lidx and lidx + 1
    return Partition with points lidx and lidx + 1
else
    l ← lidx
    r ← ridx − 1
    level ← BSR( points[l] ⊕ points[r] )
    mask ← Shift left 1 by level
    while not (points[l + 1] & mask) do
        m ← (l + r)/2
        if points[m] & mask then
            r ← m
        else
            l ← m
        end if
    end while
    left ← Subdivide(lidx, l + 1)
    right ← Subdivide(l + 1, ridx)
    return Merge(left, right)
end if
```

Subdivide method

Find position where the bit changes

Algorithm 1 Subdivision of the floating point quad-tree.

```
Subdivide(lidx, ridx)
if points[lidx] is equal to points[ridx − 1] then
    Decode points lidx to ridx − 1
    return Partition with single point lidx
else if ridx − lidx is equal to 2 then
    Decode points lidx and lidx + 1
    return Partition with points lidx and lidx + 1
else
    l ← lidx
    r ← ridx − 1
    level ← BSR( points[l] ⊕ points[r] )
    mask ← Shift left 1 by level
    while not (points[l + 1] & mask) do
        m ← (l + r)/2
        if points[m] & mask then
            r ← m
        else
            l ← m
        end if
    end while
    left ← Subdivide(lidx, l + 1)
    right ← Subdivide(l + 1, ridx)
    return Merge(left, right)
end if
```

Subdivide method

Algorithm 1 Subdivision of the floating point quad-tree.

Subdivide($lidx$, $ridx$)

if points[$lidx$] is equal to points[$ridx-1$] **then**
    Decode points $lidx$ to $ridx-1$
    return Partition with single point $lidx$
else if $ridx-lidx$ is equal to 2 **then**
    Decode points $lidx$ and $lidx+1$
    return Partition with points $lidx$ and $lidx+1$
else
    $l \leftarrow lidx$
    $r \leftarrow ridx-1$
    $level \leftarrow$ BSR(points[l] $\oplus$ points[r])
    $mask \leftarrow$ Shift left 1 by $level$
    while not (points[$l+1$] & $mask$) **do**
        $m \leftarrow (l+r)/2$
        if points[m] & $mask$ **then**
            $r \leftarrow m$
        else
            $l \leftarrow m$
        end if
    end while
    $left \leftarrow$ Subdivide($lidx$, $l+1$)
    $right \leftarrow$ Subdivide($l+1$, $ridx$)
    return Merge($left$, $right$)
end if

Recurse subdivision and merge triangulations

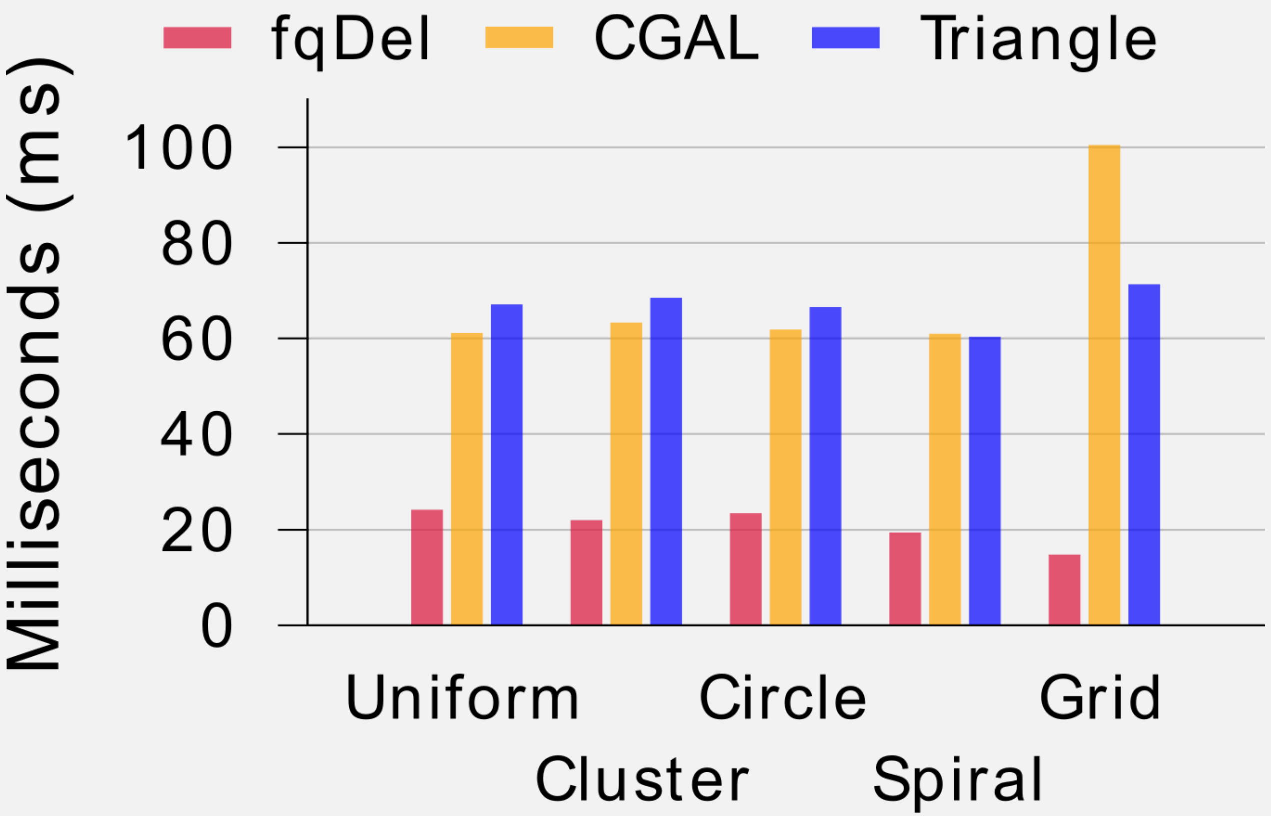
Evaluation

HOW DOES THE <u>LFQT</u> IMPACT <u>DT</u> PERFORMANCE?

Experimental Setup

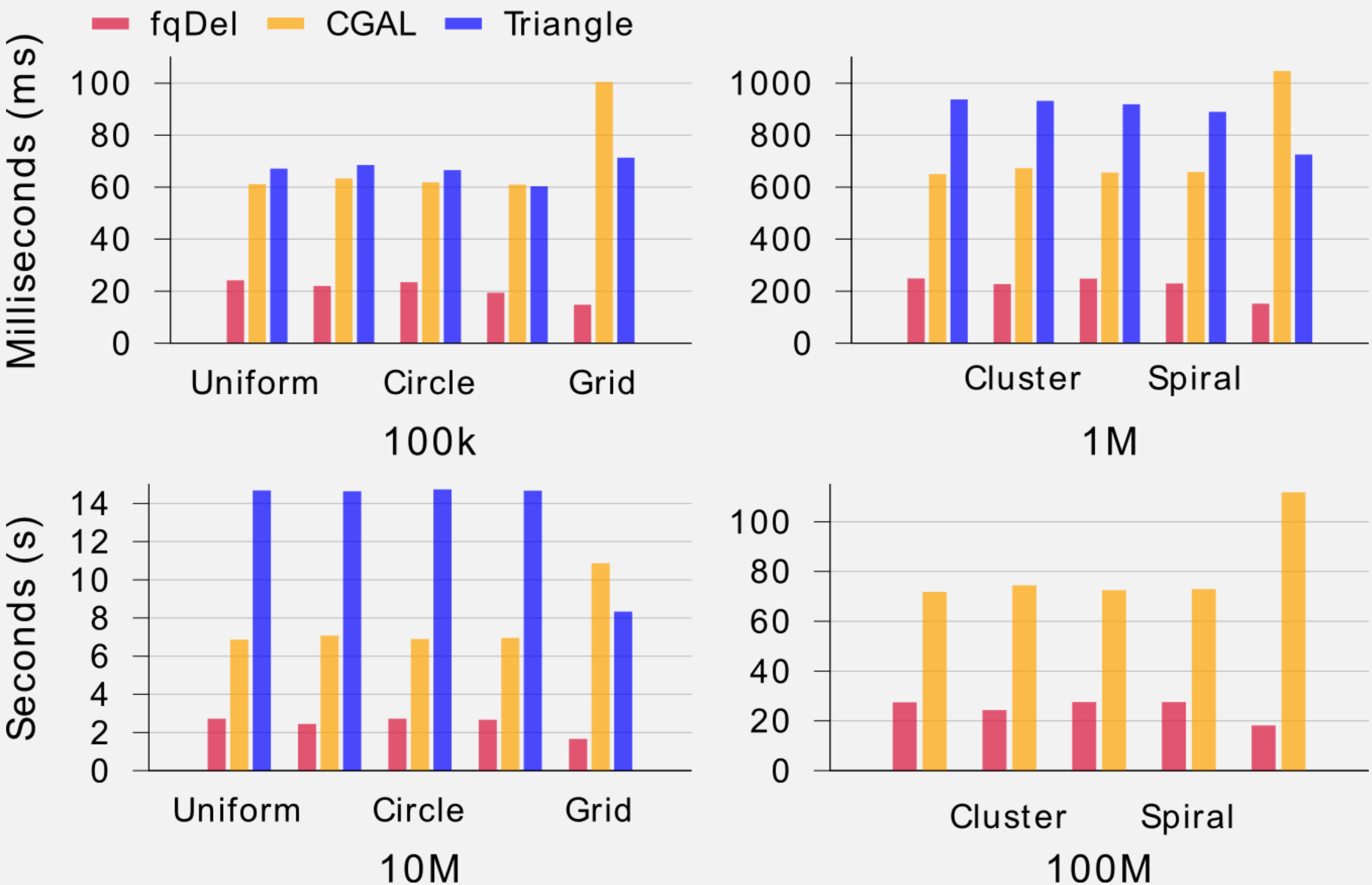
- Dual-socket Intel Xeon E5-2670 @ 3.0 GHz
  - 16 cores / 32 threads, 64 GB DDR3
- fqDel (our implementation)
- Triangle 1.6
- CGAL 4.3
- Random point distributions (fixed seed)
  - Uniform, Cluster, Grid, Circle and Spiral

Single-threaded performance: fqDel vs. CGAL vs. Triangle

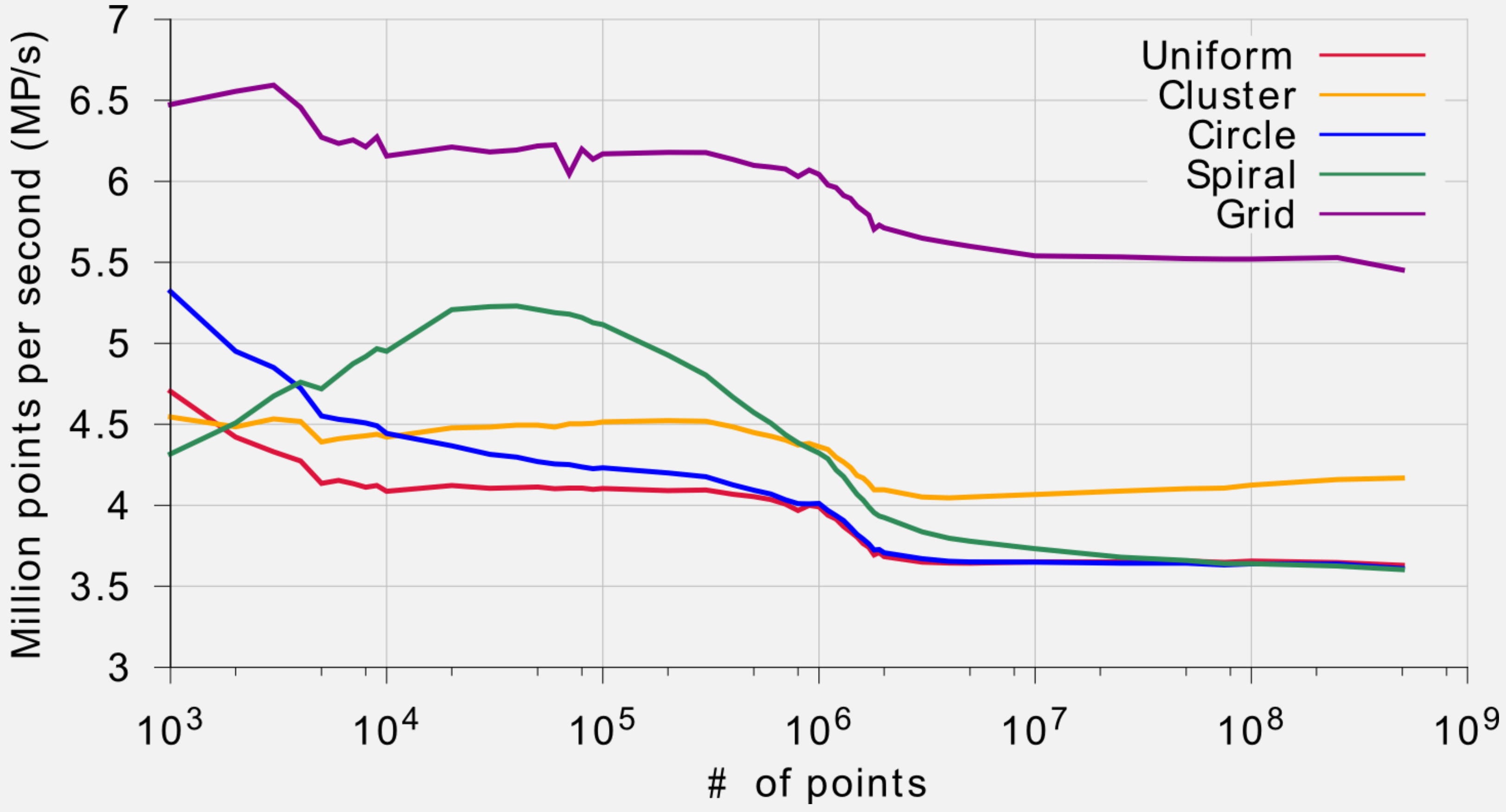


100k

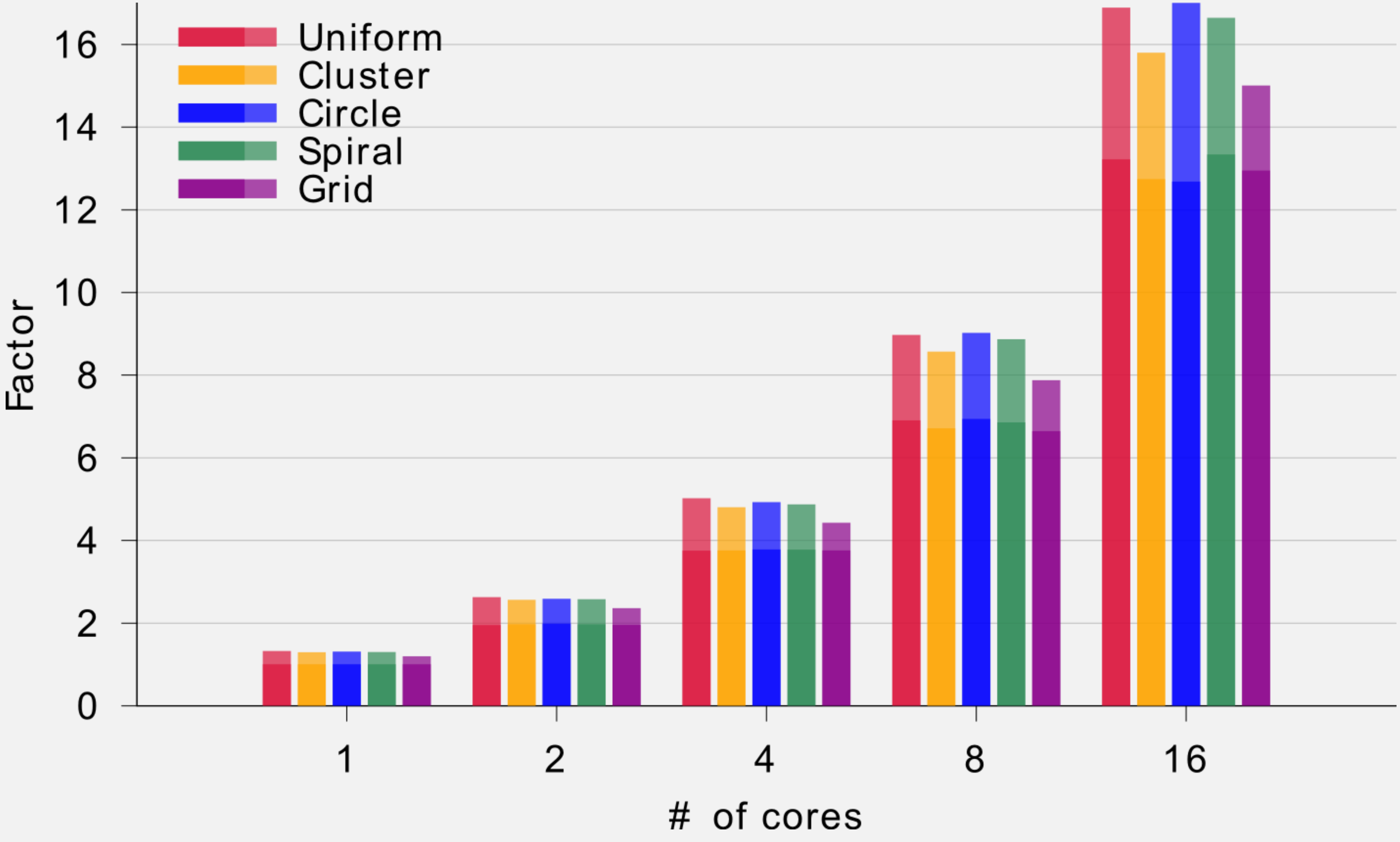
Single-threaded performance: fqDel vs. CGAL vs. Triangle



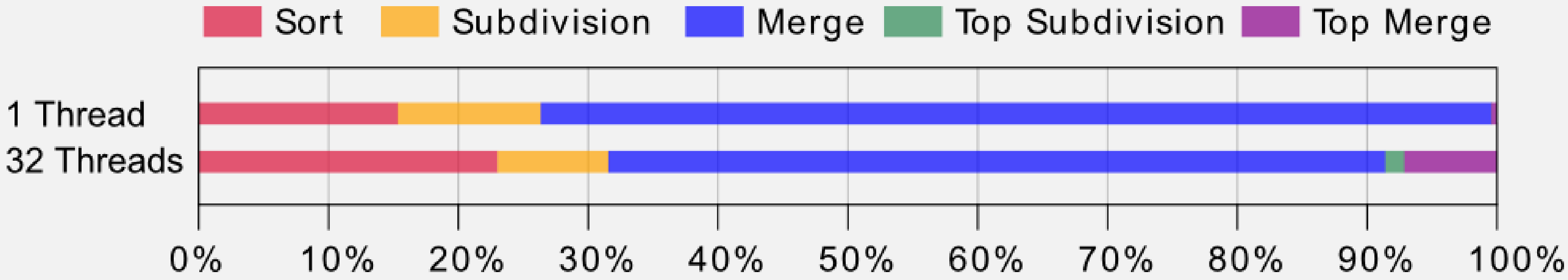
fqDel performance scaling with input size



Multi-threading: fqDel performance scaling with thread count



Run-time distribution: How much time is spent in each part of fqDel



Parallel GPU alternatives (CUDA)

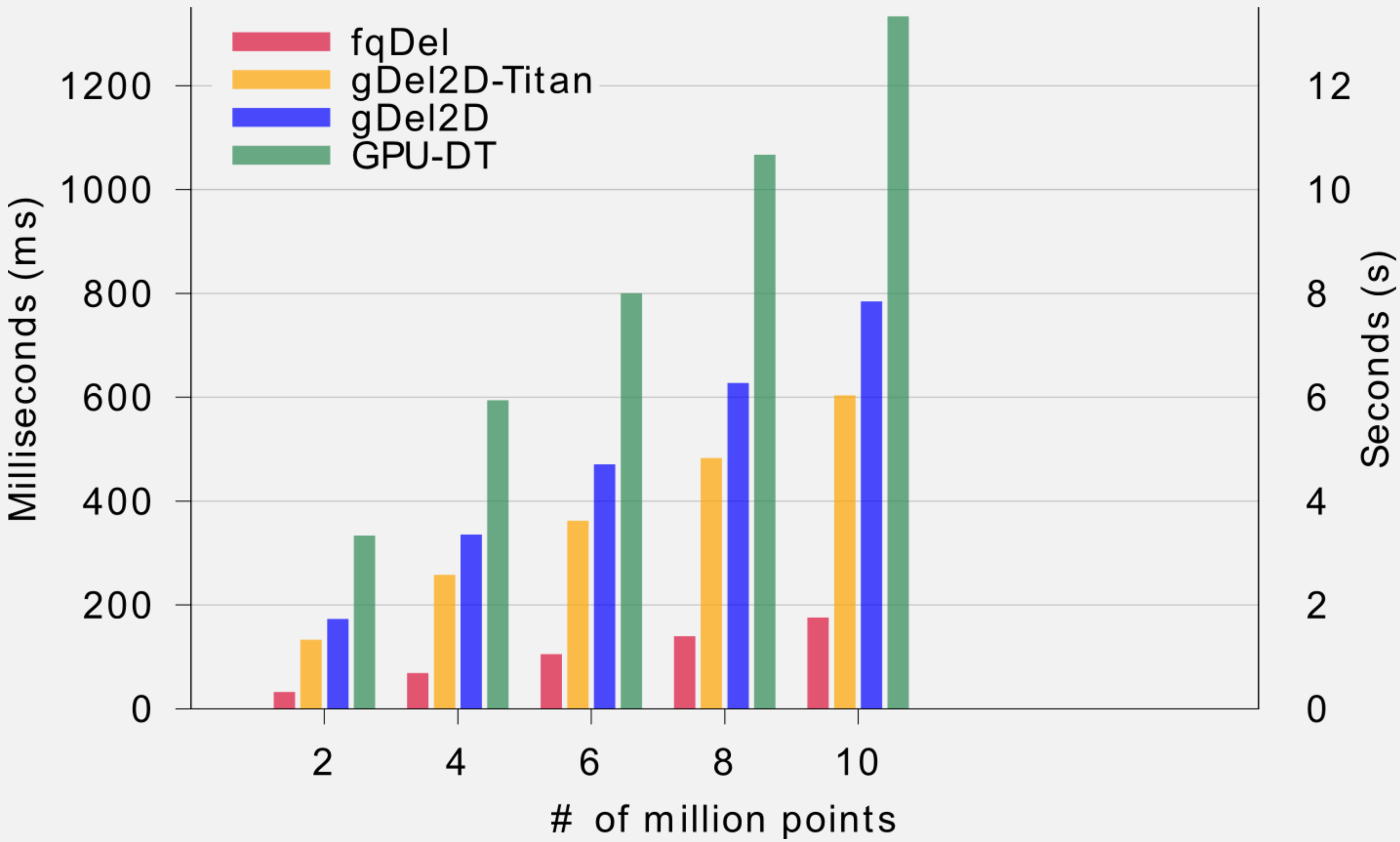
- GPU-DT [Qi, Cao, Tan ‘12]
  - Digital Voronoi diagram + edge flipping

- gDel2D [Cao, Nanjappa, Gao, Tan ‘14]
  - Parallel point-insertion

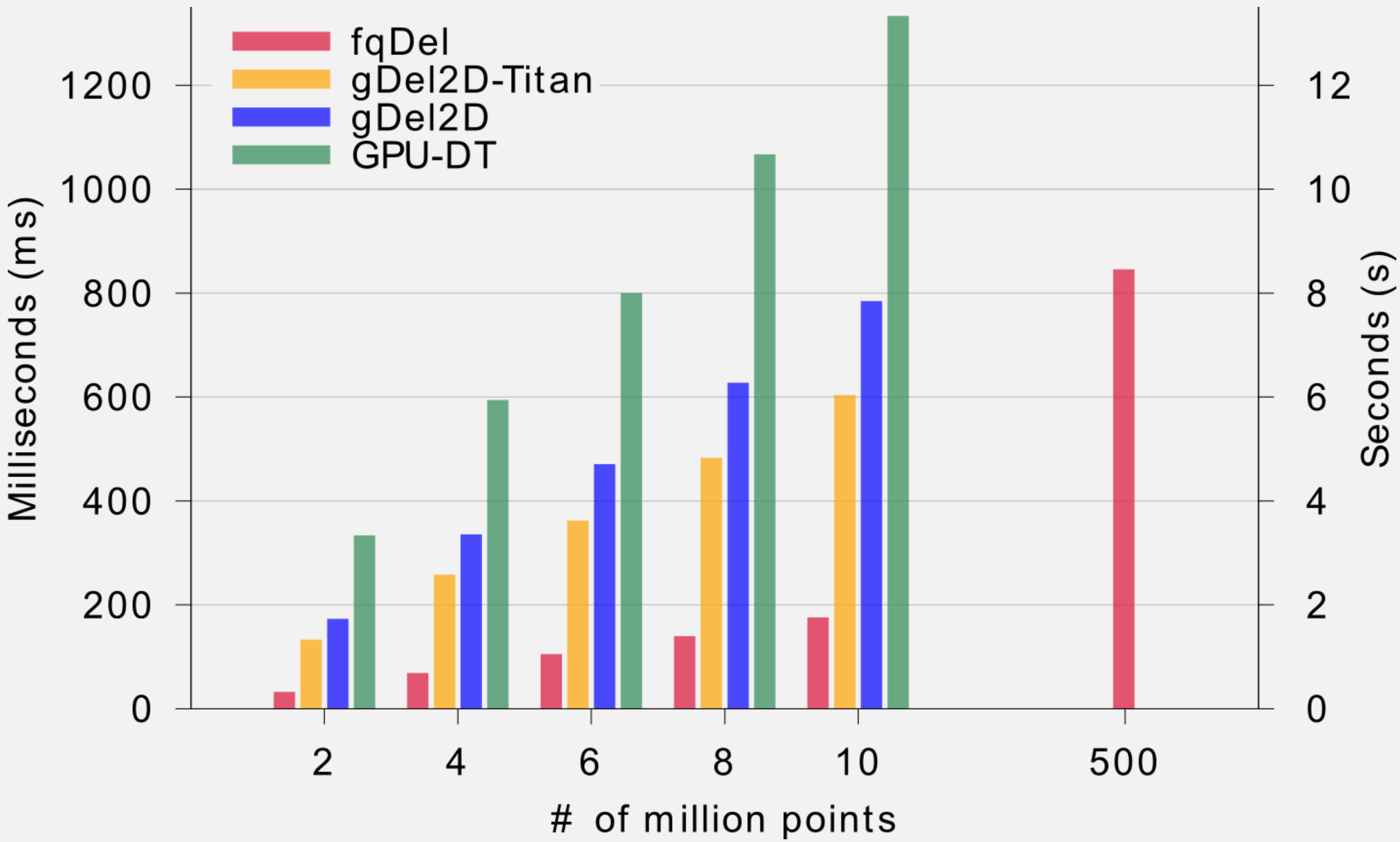
Benchmarks with Geforce GTX 580

Note: both use **double-precision**

fqDel vs. GPU alternatives



fqDel vs. GPU alternatives



Summary

- Efficient DT implementation for 2D point sets

  - Results 40-50x faster than previous CPU implementations
  - Considerably faster than GPU implementations

Thank you!