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

<table>
<thead>
<tr>
<th>CGAL</th>
<th>Triangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Point-insertion</td>
<td>• Divide-and-conquer (Dwyer’s algorithm)</td>
</tr>
</tbody>
</table>

(www.cgal.org)  (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

Implicit Quad-tree

Memory
Linear Quad-tree

1. Define grid
2. Compute Morton codes
3. (Radix) sort

Implicit Quad-tree
Memory
Linear Quad-tree

1. Define grid

2. Compute Morton codes

3. (Radix) sort

Implicit Quad-tree

Memory
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 $\leftrightarrow$ 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

Input Points → Morton Code → Radix Sort → Top Subdivide (global) → Subdivide / Merge (local) → Top Merge (global) → DT

- Full parallelism
- Limited parallelism
- Global synchronization
Algorithm phases

Input Points => Morton Code => Radix Sort => Top Subdivide (global) => Subdivide / Merge (local) => Top Merge (global) => Done

- Full parallelism
- Limited parallelism
- Global synchronization
Subdivide method

Input: Array of points from $lidx$ to $ridx$

```plaintext
Algorithm 1 Subdivision of the floating point quad-tree.

Subdivide($lidx, ridx$)

1: if points[$lidx$] is equal to points[$ridx - 1$] then
2:     Decode points $lidx$ to $ridx - 1$
3:     return Partition with single point $lidx$
4: else if $ridx - lidx$ is equal to 2 then
5:     Decode points $lidx$ and $lidx + 1$
6:     return Partition with points $lidx$ and $lidx + 1$
7: else
8:     $l ← lidx$
9:     $r ← ridx - 1$
10:    $level ← BSR( points[l] ⊕ points[r] )$
11:    $mask ←$ Shift left 1 by $level$
12:    while not (points[$l + 1$] & $mask$) do
13:        $m ← (l + r)/2$
14:        if points[$m$] & $mask$ then
15:            $r ← m$
16:        else
17:            $l ← m$
18:        end if
19:    end while
20:    $left ←$ Subdivide($lidx, l + 1$)
21:    $right ←$ Subdivide($l + 1, ridx$)
22:    return Merge($left, right$)
23: end if
```
Subdivide method

Algorithm 1 Subdivision of the floating point quad-tree.
Subdivide(lidx, ridx)

1: if points[lidx] is equal to points[ridx - 1] then
2: Decode points lidx to ridx - 1
3: return Partition with single point lidx
4: else if ridx - lidx is equal to 2 then
5: Decode points lidx and lidx + 1
6: return Partition with points lidx and lidx + 1
7: else
8: l ← lidx
9: r ← ridx - 1
10: level ← BSR( points[l] ⊕ points[r] )
11: mask ← Shift left 1 by level
12: while not (points[l + 1] & mask) do
13: m ← (l + r) / 2
14: if points[m] & mask then
15: r ← m
16: else
17: l ← m
18: end if
19: end while
20: left ← Subdivide(lidx, l + 1)
21: right ← Subdivide(l + 1, ridx)
22: return Merge(left, right)
23: end if

Single point or only degenerate points left?
Subdivide method

Algorithm 1 Subdivision of the floating point quad-tree.

Subdivide(lidx, ridx)

1: if points[lidx] is equal to points[ridx − 1] then
2: Decode points idx to ridx − 1
3: return Partition with single point lidx

4: else if ridx − lidx is equal to 2 then
5: Decode points lidx and lidx + 1
6: return Partition with points lidx and lidx + 1

7: else
8: l ← lidx
9: r ← ridx − 1
10: level ← BSR( points[l] ⊕ points[r] )
11: mask ← Shift left 1 by level
12: while not (points[l + 1] & mask) do
13: m ← (l + r)/2
14: if points[m] & mask then
15: r ← m
16: else
17: l ← m
18: end if
19: end while
20: left ← Subdivide(lidx, l + 1)
21: right ← Subdivide(l + 1, ridx)
22: return Merge(left, right)
23: end if

Two points left?
Subdivide method

Algorithm 1 Subdivision of the floating point quad-tree.

Subdivide(lidx, ridx)
1: if points[lidx] is equal to points[ridx - 1] then
2:   Decode points lidx to ridx - 1
3:   return Partition with single point lidx
4: else if ridx - lidx is equal to 2 then
5:   Decode points lidx and lidx + 1
6:   return Partition with points lidx and lidx + 1
7: else
8:   l ← lidx
9:   r ← ridx - 1
10:   level ← BSR(points[l] ⊕ points[r])
11:   mask ← Shift left 1 by level
12:   while not (points[l + 1] & mask) do
13:     m ← (l + r) / 2
14:     if points[m] & mask then
15:       r ← m
16:     else
17:       l ← m
18:   end if
19: end while
20: left ← Subdivide(lidx, l + 1)
21: right ← Subdivide(l + 1, ridx)
22: return Merge(left, right)
23: end if

Find most significant bit which is different
Subdivide method

Algorithm 1 Subdivision of the floating point quad-tree.

Subdivide($lidx$, $ridx$)

1: if points[$lidx$] is equal to points[$ridx$ - 1] then
2: Decode points $lidx$ to $ridx$ - 1
3: return Partition with single point $lidx$
4: else if $ridx$ - $lidx$ is equal to 2 then
5: Decode points $lidx$ and $lidx$ + 1
6: return Partition with points $lidx$ and $lidx$ + 1
7: else
8: $l$ ← $lidx$
9: $r$ ← $ridx$ - 1
10: level ← BSR( points[$l$] ⊕ points[$r$] )
11: mask ← Shift left 1 by level
12: while not (points[$l$ + 1] & mask) do
13: $m$ ← ($l$ + $r$) / 2
14: if points[$m$] & mask then
15: $r$ ← $m$
16: else
17: $l$ ← $m$
18: end if
19: end while
20: $left$ ← Subdivide($lidx$, $l$ + 1)
21: $right$ ← Subdivide($l$ + 1, $ridx$)
22: return Merge($left$, $right$)
23: end if

Find position where the bit changes
Subdivide method

Algorithm 1  Subdivision of the floating point quad-tree.

Subdivide(ldx, ridx)
    1: if points[ldx] is equal to points[ridx − 1] then
    2:     Decode points lidx to ridx − 1
    3:     return Partition with single point lidx
    4: else if ridx − lidx is equal to 2 then
    5:     Decode points lidx and lidx + 1
    6:     return Partition with points lidx and lidx + 1
    7: else
    8:     l ← lidx
    9:     r ← ridx − 1
   10:     level ← BSR( points[l] ⊕ points[r] )
   11:     mask ← Shift left 1 by level
   12:     while not (points[l + 1] & mask) do
   13:         m ← (l + r)/2
   14:         if points[m] & mask then
   15:             r ← m
   16:         else
   17:             l ← m
   18:     end if
   19: end while
   20:     left ← Subdivide(ldx, l + 1)
   21:     right ← Subdivide(l + 1, ridx)
   22:     return Merge(left, right)
   23: end if

Recurse subdivision and merge triangulations
Evaluation

HOW DOES THE LFQT IMPACT DT 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

Milliseconds (ms)

fqDel | CGAL | Triangle
--- | --- | ---
Uniform Cluster | | |
Circle Spiral | | |
Grid | | |

100k
Single-threaded performance: fqDel vs. CGAL vs. Triangle

- **Milliseconds (ms)**
  - Uniform
  - Circle
  - Grid

- **Seconds (s)**
  - Uniform
  - Circle
  - Grid

- **100k**
  - fqDel
  - CGAL
  - Triangle

- **1M**
  - Cluster
  - Spiral

- **10M**
  - Uniform
  - Circle
  - Grid

- **100M**
  - Cluster
  - Spiral
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

![Chart comparing fqDel and GPU alternatives]

- fqDel
- gDel2D-Titan
- gDel2D
- GPU-DT

Milliseconds (ms) vs. Seconds (s) for different numbers of million points.
Summary

- Efficient DT implementation for 2D point sets
  - Results 40-50x faster than previous CPU implementations
  - Considerably faster than GPU implementations
Thank you!