Delaunay Triangulation in Parallel

Adarsh Prakash
CSE 633 : Parallel Algorithms
(Spring 2017)

Instructor: Dr. Russ Miller
Definition

Triangle $\Delta ABC$ is a **Delaunay Triangle**, if no other points lie in the circumcircle formed by $\Delta ABC$. 

[Diagram showing invalid and valid Delaunay triangles]
Delaunay – Voronoi: Duality

A Voronoi diagram is constructed by connecting centers of all the circumcircles formed by the Delaunay Triangles in a graph.
Direct Applications

• Nearest Neighbor
• Graph Locality / Point Location
• Surface Mapping / Reconstruction
• Game Development
• Motion Capture
• Path Planning (Autonomous Navigation)
• Physics – studying forces..
• Chemistry – atomic charges..
• Biology, Astrophysics and so on.
Applications
Algorithm

- **Divide-and-conquer** algorithm proposed by Leonidas Guibas and Jorge Stolfi [1].
- Follows closely the Voronoi construction algorithm from Shamos and Hoey [2].
- Difference is it clearly describes how to make use of quad-edge data structure to avoid computation of complete hull.

**Properties:**
- A quad-edge knows its direction (origin-destination NOT point-point)
- A quad-edge maintains pointers to all edges leaving from and terminating at their origin and destination. (4-8 pointers depending on implementation)

**Objective is to parallelize this algorithm.**
Algorithm: Merge Step
Algorithm: Merge Step
Algorithm: Merge Step
Algorithm: Parallel Overview

Processor $i$
- Compute Delaunay Triangulation of Local Region
- Merge Incoming Region and Local Region
- Send or Receive from other Processors

Repeat or exit
Domain Decomposition

Input space divided equally by X-Coordinate among Processors
Implementation

- Implementation in C and MPI
- Pseudo code from paper for serial version of merge – made life easier
- Jobs were run on `general-compute` and `largemem` partitions of CCR
- All communications are point-to-point: `MPI_Send` and `MPI_Recv`
- Data send/receive happens in a single block (as many as 31 million edges ~ 700mb)
- Approx. 500 jobs to cluster
Implementation

• **Input:**
  - Randomly generated points – **Bivariate Uniform Distribution** using Python **numpy** package
  - Equal range and density across both the axes
  - No duplicates and **pre-sorted** by X-Coordinate
  - Each coordinate is “**double**” precision $\in [0, 200*n]$

• **Output:**
  - Edge endpoints as indices

<table>
<thead>
<tr>
<th>Sample Input</th>
<th>Sample Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, 162.422299106, 626335123.072</td>
<td>0 1</td>
</tr>
<tr>
<td>1, 235.609542392, 21674347.1286</td>
<td>0 3</td>
</tr>
<tr>
<td>2, 348.128895741, 545885503.786</td>
<td>0 4</td>
</tr>
<tr>
<td>3, 388.434040826, 160544722.935</td>
<td>0 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Results

• Run-times averaged over 3-5 jobs/runs
• Time spent on reading input file and writing the results to output file is ignored
• Only Computation Time (with communication cost) is used in analysis
• Tried for several core-node combinations:
  • 2 CPUs per node with shm (intranode) and tmi (internode)
  • 1 CPU per node with dapl (internode)
  • 1 CPU per node with tmi (internode)
  • Upto 32 CPUs per node with tcp (intranode) and tcp (internode) – I_MPI_FABRICS and I_MPI_FALBACK to the rescue!
• All results validated against results from standard packages:
  • Python (scipy.spatial.Delaunay) - faster
  • Matlab (triangulation)
Time v/s CPU (1 CPU per node – TMI)
Speedup v/s CPU (1 CPU per node – TMI)
Time v/s CPU (32 CPUs per node – TCP – no shm)

- 1 Million Points
- 2 Million Points
- 4 Million Points

Computation Time (seconds)

Number of CPUs (log scale)
Speedup v/s CPU (32 CPUs per node – TCP – no shm)

Number of CPUs (log scale)

Computation Time (seconds)

1 Million Points
2 Million Points
4 Million Points
Asymptotic Growth (8 CPUs with 1 CPU per node)
Conclusion

• That drop in speedup for 32-cpus-per-node?
  • Communication Cost: Intranode < Internode
  • Difference is significant for TCP and hence the sudden drop

• Hard Merge – High Communication Costs – No Linear Speedup

• But, there is gain

• Data still needs to fit into a single machine!
References

• *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams* – Guibas, L. and Stolfi, J.

• *Closest-Point Problems* – Shamos, M.I. and Hoey, D.

• *On computing Voronoi diagrams by divide-prune-and-conquer* – Amato, N.M. and Ramos, E.A.

• *Chapter 10: Computational Geometry, Algorithms* – Sequential and Parallel – Miller, R. and Boxer, L.
Thank You

adarshpr@buffalo.edu

Binaries, scripts, code and results available at: https://github.com/adrsh18/parallel

Thanks to Dr. M Jones and CCR @ UB
Backup: Analysis

• Sequential runtime: $O(n \times \log n)$ [$T(n) = 2 \times T(n/2) + O(n)$]

• “Heavy” merge step with $O(n)$. Parallelization possible?!!

• Analysis with $p$ processors:
  • Each processor locally and simultaneously computes DT on $\frac{n}{p}$ points → $O\left(\frac{n}{p} \times \log \left(\frac{n}{p}\right)\right)$
  • DTs from each processor is stitched together (happens $\log p$ times) → $O(n \times \log p)$
  • So, total runtime = $O\left(\frac{m}{p} \times \log \left(\frac{n}{p}\right) + n \times \log p\right)$

• If $p = \log n$, runtime = $O(n \log(\log n))$