STING: Spatio-Temporal Interaction Networks and Graphs for Intel Platforms
David Bader, Jason Riedy, Henning Meyerhenke, David Ediger

29 August 2011
Outline

Motivation

Technical

Overall streaming approach
Clustering coefficients
Connected components
Community detection (in progress)

Related

Pasqual, a scalable de novo sequence assembler

Plans
## Exascale Data Analysis

<table>
<thead>
<tr>
<th>Domain</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Health care</td>
<td>Finding outbreaks, population epidemiology</td>
</tr>
<tr>
<td>Social networks</td>
<td>Advertising, searching, grouping</td>
</tr>
<tr>
<td>Intelligence</td>
<td>Decisions at scale, regulating algorithms</td>
</tr>
<tr>
<td>Systems biology</td>
<td>Understanding interactions, drug design</td>
</tr>
<tr>
<td>Power grid</td>
<td>Disruptions, conservation</td>
</tr>
<tr>
<td>Simulation</td>
<td>Discrete events, cracking meshes</td>
</tr>
</tbody>
</table>
Graphs are pervasive

- Sources of massive data: petascale simulations, experimental devices, the Internet, scientific applications.
- New challenges for analysis: data sizes, heterogeneity, uncertainty, data quality.

**Astrophysics**
- **Problem**: Outlier detection
- **Challenges**: Massive data sets, temporal variation
- **Graph problems**: Matching, clustering

**Bioinformatics**
- **Problem**: Identifying target proteins
- **Challenges**: Data heterogeneity, quality
- **Graph problems**: Centrality, clustering

**Social Informatics**
- **Problem**: Emergent behavior, information spread
- **Challenges**: New analysis, data uncertainty
- **Graph problems**: Clustering, flows, shortest paths
These are not easy graphs.

Yifan Hu's (AT&T) visualization of the Livejournal data set
Overall streaming approach

**Assumptions**

- A graph represents some real-world phenomenon.
  - But **not** necessarily exactly!
  - Noise comes from lost updates, partial information, ...
Overall streaming approach


Assumptions

- We target massive, “social network” graphs.
  - Small diameter, power-law degrees
  - Small changes in massive graphs often are unrelated.
Overall streaming approach


**Assumptions**

- The graph changes but we don’t need a continuous view.
  - We can accumulate changes into batches...
  - But not so many that it impedes responsiveness.
Difficulties for performance

- **What partitioning methods apply?**
  - Geometric? Nope.
  - Balanced? Nope.
  - Is there a single, useful decomposition? Not likely.

- **Some partitions exist,** but they don’t often help with balanced bisection or memory locality.

- **Performance needs new approaches,** not just standard scientific computing methods.
STING’s focus

- STING manages queries against changing graph data.
  - Visualization and control often are application specific.
- Ideal: Maintain many persistent graph analysis kernels.
  - Keep one current snapshot of the graph resident.
  - Let kernels maintain smaller histories.
  - Also (a harder goal), coordinate the kernels’ cooperation.
STING and STINGER

- Batches provide dual-level parallelism.
  - Busy loci of change: Know to share the busy points.
  - Scattered changes: Parallel across (likely) independent changes.
- The massive graph is maintained in a data structure named STINGER.
STINGER

STING Extensible Representation:

- Rule #1: No explicit locking.
  - Rely on atomic operations.
- Massive graph: Scattered updates, scattered reads rarely conflict.
- Use time stamps for some view of time.
Initial results

Prototype STING and STINGER
Monitoring the following properties:

1. clustering coefficients,
2. connected components, and
3. community structure (in progress).

High-level

- Support high rates of change, over 10k updates per second.
- Performance scales somewhat with available processing.
- Gut feeling: Scales as much with sockets as cores.

http://www.cc.gatech.edu/~bader/code.html
Experimental setup

Unless otherwise noted

<table>
<thead>
<tr>
<th>Line</th>
<th>Model</th>
<th>Speed (GHz)</th>
<th>Sockets</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>X5570</td>
<td>2.93</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Westmere</td>
<td>E7-8870</td>
<td>2.40</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

- Westmere loaned by Intel (thank you!)
- All memory: 1067MHz DDR3, installed appropriately
- Implementations: OpenMP, gcc 4.6.1, Linux ≈ 3.0 kernel
- Artificial graph and edge stream generated by R-MAT[Chakrabarti, et al.].
  - Scale $x$, edge factor $f \Rightarrow 2^x$ vertices, $\approx f \cdot 2^x$ edges.
  - Edge actions: 7/8th insertions, 1/8th deletions
  - Results over five batches of edge actions.
- Caveat: No vector instructions, low-level optimizations yet.
Clustering coefficients

- Used to measure “small-world-ness” [Watts and Strogatz] and potential community structure
- Larger clustering coefficient $\Rightarrow$ more inter-connected
- Roughly the ratio of the number of actual to potential triangles

- Defined in terms of triplets.
  - $i - v - j$ is a closed triplet (triangle).
  - $m - v - n$ is an open triplet.
- Clustering coefficient:
  $$\frac{\text{# of closed triplets}}{\text{total # of triplets}}$$
- Locally around $v$ or globally for entire graph.
Updating triangle counts

Given \( \{u, v\} \) to be inserted (+) or deleted (-)

Approach Search for vertices adjacent to both \( u \) and \( v \), update counts on those and \( u \) and \( v \)

Three methods

Brute force Intersect neighbors of \( u \) and \( v \) by iterating over each, \( O(d_u d_v) \) time.

Sorted list Sort \( u \)'s neighbors. For each neighbor of \( v \), check if in the sorted list.

Compressed bits Summarize \( u \)'s neighbors in a bit array. Reduces check for \( v \)'s neighbors to \( O(1) \) time each. Approximate with Bloom filters. [MTAAP10]

All rely on atomic addition.
Batches of 10k actions

Graph size: scale 22, edge factor 16
Updates per seconds, both metric and STINGER

<table>
<thead>
<tr>
<th>Threads</th>
<th>Brute force</th>
<th>Bloom filter</th>
<th>Sorted list</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.7e+04</td>
<td>8.8e+04</td>
<td>8.8e+04</td>
</tr>
<tr>
<td>20</td>
<td>1.5e+04</td>
<td>1.3e+04</td>
<td>1.2e+05</td>
</tr>
<tr>
<td>40</td>
<td>5.1e+03</td>
<td>2.4e+04</td>
<td>2.2e+04</td>
</tr>
<tr>
<td>60</td>
<td>3.9e+03</td>
<td>2.0e+04</td>
<td>1.7e+04</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Machine
- 4 x E7–8870
- 2 x X5570
Different batch sizes

Graph size: scale 22, edge factor 16

Updates per seconds, both metric and STINGER

<table>
<thead>
<tr>
<th>Machine</th>
<th>Brute force</th>
<th>Bloom filter</th>
<th>Sorted list</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 x E7−8870</td>
<td><img src="image1" alt="Graph" /></td>
<td><img src="image2" alt="Graph" /></td>
<td><img src="image3" alt="Graph" /></td>
</tr>
<tr>
<td>2 x X5570</td>
<td><img src="image4" alt="Graph" /></td>
<td><img src="image5" alt="Graph" /></td>
<td><img src="image6" alt="Graph" /></td>
</tr>
</tbody>
</table>

Threads

Graph size: scale 22, edge factor 16
Connected components

- Maintain a mapping from vertex to component.
- *Global* property, unlike triangle counts
- In “scale free” social networks:
  - Often one big component, and
  - many tiny ones.
- Edge changes often sit *within* components.
- Remaining insertions merge components.
- Deletions are more difficult...
Connected components

- Maintain a mapping from vertex to component.
- *Global* property, unlike triangle counts
- In “scale free” social networks:
  - Often one big component, and
  - many tiny ones.
- Edge changes often sit *within* components.
- Remaining insertions merge components.
- Deletions are more difficult...
Connected components: Deleted edges

The difficult case

- Very few deletions matter.
- Determining *which* matter may require a large graph search.
  - Re-running static component detection.
  - (Long history, see related work in [MTAAP11].)
- Coping mechanisms:
  - *Heuristics.*
  - Second level of batching.
Deletion heuristics

Rule out effect-less deletions

- Use the *spanning tree* by-product of static connected component algorithms.
- Ignore deletions when one of the following occur:
  1. The deleted edge is not in the spanning tree.
  2. If the endpoints share a common neighbor*.
  3. If the loose endpoint can reach the root*.
- In the last two (*), also fix the spanning tree.

Rules out 99.7% of deletions.
Connected components: Performance

Graph size: scale 22, edge factor 16

Updates per second, both metric and STINGER

Machine
- 4 x E7–8870
- 2 x X5570
Community detection (work in progress)

Greedy, agglomerative partitioning

- Partition to maximize modularity, minimize conductance, ... 

Seed set expansion

- Grow an optimal / “relevant” community around selection.
- (Work with Jonny Dimond of KIT.)
Agglomerative community detection

Parallel greedy, agglomerative partitoning [PPAM11]

- Score edges by optimization criteria.
- Chose a maximal, heavy-weight matching.
  - Negate edge scores if minimizing conductance.
- Contract those edges.
- Mimics sequential optimizers, but produces different results.
Performance

- R-MAT on right.
- Livejournal
  - 15M vertex, 184M edge
  - 6-12 hours on E7-8870
- Highly variable performance.
- Algorithm under development.
A scalable *de novo* assembler

Work by Henning Meyerhenke, Xing Liu, Pushkar Pande.

- Next-generation sequencers produce mountains of small gene sequences.
- Assembling into a genome: Yet another large graph problem.
- Pasqual forms a compressed *overlap graph* and traces paths.
- **Only scalable and correct shared-memory assembler.**
  - Faster *and* uses less memory than other existing systems.
  - Evaluation against the few distributed assemblers is ongoing.

http://www.cc.gatech.edu/pasqual/
## Human genome, 33.5Mbp (cov 30)

### Similar speed, better results

<table>
<thead>
<tr>
<th>Length</th>
<th>Code</th>
<th>Time (min)</th>
<th>N50 (bp)</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Velvet</td>
<td>&gt;12h</td>
<td>1355</td>
<td>683</td>
</tr>
<tr>
<td></td>
<td>Edena</td>
<td>451.15</td>
<td>1375</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>ABySS</td>
<td>56.52</td>
<td>1412</td>
<td>521</td>
</tr>
<tr>
<td></td>
<td>SOAPdenovo</td>
<td><strong>15.62</strong></td>
<td><strong>1470</strong></td>
<td><strong>485</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Pasqual</strong></td>
<td>15.50</td>
<td>1451</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>Velvet</td>
<td>132.68</td>
<td>6635</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Edena</td>
<td>466.12</td>
<td>6545</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>ABySS</td>
<td>92.92</td>
<td>6229</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>SOAPdenovo</td>
<td><strong>18.05</strong></td>
<td>6879</td>
<td><strong>142</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Pasqual</strong></td>
<td>19.15</td>
<td><strong>7712</strong></td>
<td>6</td>
</tr>
</tbody>
</table>
Zebrafish, 61Mbp (cov 30)

### Far better speed *and* results

<table>
<thead>
<tr>
<th>Length</th>
<th>Code</th>
<th>Time (min)</th>
<th>N50 (bp)</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Velvet</td>
<td>256.02</td>
<td>4045</td>
<td>799</td>
</tr>
<tr>
<td></td>
<td>Edena</td>
<td>&gt;12h</td>
<td>2661</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>ABySS</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>SOAPdenovo</td>
<td><strong>31.83</strong></td>
<td>3998</td>
<td>725</td>
</tr>
<tr>
<td></td>
<td><strong>Pasqual</strong></td>
<td>57.27</td>
<td><strong>4535</strong></td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>Velvet</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>Edena</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>ABySS</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>SOAPdenovo</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td><strong>Pasqual</strong></td>
<td><strong>38.07</strong></td>
<td><strong>7911</strong></td>
<td>0</td>
</tr>
</tbody>
</table>
Performance v. SOAPdenovo

The graph illustrates the performance comparison between SOAPdenovo and Pasqual across different thread counts. The y-axis represents time in milliseconds, while the x-axis represents the number of threads. Coverage is indicated at certain points on the graph, with 30 and 50 coverage levels marked.

- **Time**: The time taken increases with fewer threads, indicating higher efficiency for a larger number of threads.
- **Coverage**: The coverage markers show the percentage of coverage achieved at different time points.
- **Code**: The graph differentiates between SoapDenovo (circle) and Pasqual (triangle) with distinct colors for each coverage level.
Speed-up

- Speedup vs. Threads
- Coverage: 30, 50
- Code: SoapDenovo, Pasqual

Graph showing speedup on the y-axis and threads on the x-axis, with lines indicating speedup for different coverage levels and code.
Community detection Improving the algorithm, pushing into streaming by de-agglomerating and restarting.

Seed set expansion Maintaining not only one expanded set, but multiple for high-throughput monitoring.

Microbenchmarks Expand on initial promising work on characterizing performance by peak number of memory operations achieved, find bottlenecks by comparing with microbenchmarks.

Distributed/PGAS STINGER fits a PGAS model well (think SCC). Interested in exploring distributed algorithms.

Packaging Wrap STING into an easily downloaded and installed tool.


D. Chakrabarti, Y. Zhan, and C. Faloutsos.
R-MAT: A recursive model for graph mining.
In Proc. 4th SIAM Intl. Conf. on Data Mining (SDM), Orlando, FL, Apr. 2004. SIAM.

D. J. Watts and S. H. Strogatz.
Collective dynamics of ‘small-world’ networks.