Analyzing Graph Structure in Streaming Data with STINGER

Jason Riedy, Robert C. McColl, David Ediger, and David A. Bader with Anita Zakrzewska and Oded Green

Georgia Institute of Technology

28 February 2013
Outline

Background
Where graphs appear... (hint: everywhere)
Data volumes and rates of change
Why analyze data streams?

Technical Bits
How to analyze streams of graph-structured data
STING status and immediate plans
Community detection and monitoring

Observations
Exascale Data Analysis

Health care  Finding outbreaks, population epidemiology
Social networks  Advertising, searching, grouping
Intelligence Decisions at scale, regulating algorithms
Systems biology Understanding interactions, drug design
Power grid Disruptions, conservation
Simulation Discrete events, cracking meshes

The data is full of semantically rich relationships.

Graphs! Graphs! Graphs!
Full of structures: not simple ones.

Yifan Hu's (AT&T) visualization of the in-2004 data set
http://www2.research.att.com/~yifanhu/gallery.html
Full of structures: not well-defined ones.

LinkedIn Labs Map of my network
http://inmaps.linkedinlabs.com
No shortage of data...

Existing (some out-of-date) data volumes

NYSE 1.5 TB generated daily into a maintained 8 PB archive

Google “Several dozen” 1PB data sets (CACM, Jan 2010)

LHC 15 PB per year (avg. 21 TB daily)

Wal-Mart 536 TB, 1B entries daily (2006)

EBay 2 PB, traditional DB, and 6.5PB streaming, 17 trillion records, 1.5B records/day, each web click is 50-150 details. http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/

Facebook > 1B monthly users...

• All data is rich and semantic (graphs!) and changing.
• Base data rates include items and not relationships.
General approaches

- **High-performance static graph analysis**
  - Develop techniques that apply to unchanging massive graphs.
  - Provides useful after-the-fact information, starting points.
  - Serves many existing applications well: market research, much bioinformatics, ...
  - Needs to be $O(|E|)$.

- **High-performance streaming graph analysis**
  - Focus on the dynamic changes within massive graphs.
  - Find trends or new information as they appear.
  - Serves upcoming applications: fault or threat detection, trend analysis, ...
  - Can be $O(|\Delta E|)$? $O(\text{Vol}(\Delta V))$? Less data $\Rightarrow$ faster, cheaper.

Both very important to different areas.
Remaining focus is on streaming.

Note: Not CS theory streaming, but analysis of streaming data.
Why analyze data streams?

Data volumes
- NYSE 1.5TB daily
- LHC 41TB daily
- Facebook Who knows?

Data transfer
- 1 Gb Ethernet: 8.7TB daily at 100%, 5-6TB daily realistic
- Multi-TB storage on 10GE: 300TB daily read, 90TB daily write
- CPU ↔ Memory: QPI,HT: 2PB/day@100%

Data growth
- Facebook: > 2×/yr
- Twitter: > 10×/yr
- Growing sources: Bioinformatics, µsensors, security

Speed growth
- Ethernet/IB/etc.: 4× in next 2 years. Maybe.
- Flash storage, direct: 10× write, 4× read. Relatively huge cost.
STING: Spatio-Temporal Interaction Networks and Graphs

- Shared-memory framework (in progress) for
  - inserting and removing typed edges and vertices in batches,
  - running analysis kernels on pre- and post-combinations of
    - the batch of changes,
    - the graph,
    - other kernels’ results,
    - ...

- STINGER data structure maintains the graph.

- Free software available through [http://www.cc.gatech.edu/stinger/](http://www.cc.gatech.edu/stinger/), supports OpenMP and Cray XMT.

- Generally, *STING* is the framework and *STINGER* is the data structure, but we often say STINGER.
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.
- Gather data into a typed graph structure, STINGER.
Rule #1: No explicit locking.
  • Rely on atomic operations, or
  • tolerate data (but not data structure) inconsistency.

Rule #2: In memory...

Massive graph: Scattered updates, scattered reads rarely conflict. Be optimistic whenever possible.

Use time stamps for some view of time.
STING status and immediate plans

Current distribution: http://www.cc.gatech.edu/stinger/

- Oriented towards research and demonstration.
- Included kernels:
  - insertion and removal demo / benchmark [1],
  - updating clustering coefficients [2, 3],
  - updating connected components [4, 3],
  - community re-agglomeration,
  - static benchmarks: BFS, components.
- Available:
  - incremental betweenness centrality* [5].
- Uses:
  - Multiple internal projects.
  - External collaborators at Intel, CMU, SAIC, Nobilis, and others.

Development: rad branch

- Framework supporting communicating, run-time work-flow.
What do we mean?

- **Partition** a graph’s vertices into disjoint communities / clusters.
- A community locally maximizes some metric...
- No single accepted hard definition.
- Try to capture that vertices are *more similar* within one community than between communities.
Community detection

Assumptions

- **Disjoint** partitioning of vertices.
- There is no one unique answer.
  - Many metrics are NP-complete to optimize or just plain ill-defined.
  - Graph is **lossy** representation.
- Want an adaptable detection method.
A common method (e.g. Clauset, et al. [6]) agglomerates vertices into communities.

Each vertex begins in its own community.

An edge is chosen to contract.
- Merging maximally increases modularity.
- *Priority queue.*

Known often to fall into an $O(n^2)$ performance trap with modularity (Wakita & Tsurumi [7]).
Sequential agglomerative method

- A common method (e.g. Clauset, et al. [6]) agglomerates vertices into communities.
- Each vertex begins in its own community.
- An edge is chosen to contract.
  - Merging maximally increases modularity.
  - *Priority queue.*
- Known often to fall into an $O(n^2)$ performance trap with modularity (Wakita & Tsurumi [7]).
Sequential agglomerative method

- A common method (e.g. Clauset, et al. [6]) agglomerates vertices into communities.
- Each vertex begins in its own community.
- An edge is chosen to contract.
  - Merging maximally increases modularity.
  - **Priority queue.**
- Known often to fall into an $O(n^2)$ performance trap with modularity (Wakita & Tsurumi [7]).
Sequential agglomerative method

- A common method (e.g. Clauset, *et al.* [6]) *agglomerates* vertices into communities.
- Each vertex begins in its own community.
- An edge is chosen to contract.
  - Merging maximally increases modularity.
  - *Priority queue.*
- Known often to fall into an $O(n^2)$ performance trap with modularity (Wakita & Tsurumi [7]).
Parallel agglomerative method

- We use a matching to avoid the queue. [8, 9, 10]
- Compute a heavy weight, large matching.
  - Simple greedy algorithm.
  - Maximal matching.
  - Within factor of 2 in weight.
- Merge all matched communities.
- Maintains some balance.
- Produces different results. Fast.
- Agnostic to weighting, matching...
  - Can maximize modularity, minimize conductance.
- Think AMG, ParMetis, ... Effectively $\tilde{O}(|E|)$. 
Parallel agglomerative method

- We use a **matching** to avoid the queue. [8, 9, 10]
- Compute a heavy weight, large matching.
  - Simple greedy algorithm.
  - Maximal matching.
  - Within factor of 2 in weight.
- Merge all matched communities.
- Maintains some balance.
- *Produces different results.* Fast.
- Agnostic to weighting, matching...
  - Can maximize modularity, minimize conductance.
- Think AMG, ParMetis, ... Effectively $\tilde{O}(|E|)$.
Parallel agglomerative method

- We use a **matching** to avoid the queue. [8, 9, 10]
- Compute a heavy weight, large matching.
  - Simple greedy algorithm.
  - Maximal matching.
  - Within factor of 2 in weight.
- Merge all matched communities.
- Maintains some balance.
- **Produces different results.** Fast.
- Agnostic to weighting, matching...
  - Can maximize modularity, minimize conductance.
- Think AMG, ParMetis, ... Effectively $\tilde{O}(|E|)$. 
Adapting for streaming data

Static implementation notes

- Uses a simple binned edge list.
- Only stores undirected edges, not pairs of directed edges.
- Very un-STINGER.
- Example of adapting an existing fast code into STING.

Simplest thing that could work.

- Parallel agglomeration produces a contracted community graph.
- Community graph vertex: set of original graph vertices
- Batch of edge insertions and removals touches a subset of original graph vertices.
- Extract using STINGER, then re-start agglomeration.
- Consistent with some contraction ordering starting from scratch, possibly not locally maximal.
Extracting vertices, edges

Before changing STINGER graph

1. Collect unique vertices from edge change batch.
   - (common operation across kernels)
2. Extract affected vertices into new communities.
   - Except the last one... Use community volume to check.

Given changed STINGER graph

1. Append corrective edges to community graph edge list.
   - Append edge with negative weight to cancel old edge between old communities, positive weight to link new communities.
   - (Edge removals handled similarly.)

- Operations $\propto$ to volume of affected vertices.
- Re-agglomeration $\propto$ edges in community graph plus batch size.
Performance: Setup

Test cases

• Three graphs from DIMACS10, initial communities:

| Name            | |V| | E| |C| | E_C |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| caidaRouterLevel| 192 244         | 1 218 132       | 18 343          | 30 776          |
| coPapersDBLP    | 540 486         | 30 866 466      | 1 401           | 205 856         |
| eu-2005         | 862 664         | 16 138 468      | 55 624          | 194 971         |

• Edge action generation: 15/16 insertion, 1/16 removal
  • Insertion: Between two communities (∝ density), select endpoints with prob. inversely proportional to degree.
  • Removal: Randomly sample from existing edges inside communities, then from inserted edges if exhausted.

• Only for execution performance, not quality. Using a real stream for quality is in progress.

Platform: Quad 6-core, 2.0GHz Intel Westmere-based system with 1 TiB RAM (courtesy or Oracle).
Community graph expansion

![Graph Expansion Diagram]

- **Edges in C' relative to G**
- **caidaRouterLevel**
- **coPapersDBLP**
- **eu-2005**

- **Batch size**
  - 1
  - 100
  - 1000
  - 10000
  - 100000

**Axes:**
- **Y-axis:** Logarithmic scale ranging from $10^{-5}$ to $10^0$
- **X-axis:** Logarithmic scale ranging from $10^0$ to $10^5$

**Legend:**
- Different markers represent different batch sizes.
Updates per second, w/STINGER

<table>
<thead>
<tr>
<th>Batch size</th>
<th>1</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>caidaRouterLevel</td>
<td>5 10 15 20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coPapersDBLP</td>
<td></td>
<td>5 10 15 20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eu-2005</td>
<td></td>
<td></td>
<td>5 10 15 20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Threads (x86)

Updates per second
Updates per second, only re-agglom.
Latency, w/STINGER

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Batch size</th>
<th>Threads (x86)</th>
</tr>
</thead>
<tbody>
<tr>
<td>caidaRouterLevel</td>
<td>1</td>
<td>5, 10, 15, 20</td>
</tr>
<tr>
<td>coPapersDBLP</td>
<td>100</td>
<td>5, 10, 15, 20</td>
</tr>
<tr>
<td>eu-2005</td>
<td>1000</td>
<td>5, 10, 15, 20</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>5, 10, 15, 20</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>5, 10, 15, 20</td>
</tr>
</tbody>
</table>

Latency: seconds per batch

Batch size:
- 1
- 100
- 1000
- 10000
- 100000

Static
Speed-up over static

<table>
<thead>
<tr>
<th>caidaRouterLevel</th>
<th>coPapersDBLP</th>
<th>eu−2005</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td></td>
</tr>
</tbody>
</table>

Threads (x86)

Speed−up over static

caidaRouterLevel

coPapersDBLP

eu−2005
Observations

- Straight-forward to adapt some surprising graph algorithms to streaming data.
- **STING** provides a useful platform for quick development.
  - Good base-line performance on general (OpenMP) and specialized (XMT) architectures.
  - Tunable latency v. raw throughput (updates/sec).
  - Developing a work-flow and blackboard model.
  - External users finding issues, e.g. lower performance on high-degree vertices.
    - Good and bad.

http://www.cc.gatech.edu/stinger
Acknowledgment of support


