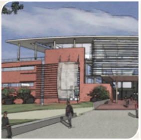
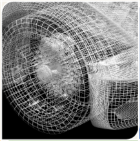
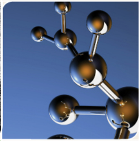
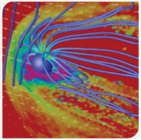
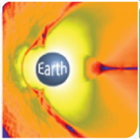










Scalable Algorithms for Analysis of Massive, Streaming Graphs

Jason Riedy, Georgia Institute of Technology;
Henning Meyerhenke, Karlsruhe Inst. of Technology;
with David Bader, David Ediger, and others at GT

15 February, 2012

Georgia Tech College of Computing
Computational Science and Engineering

Outline

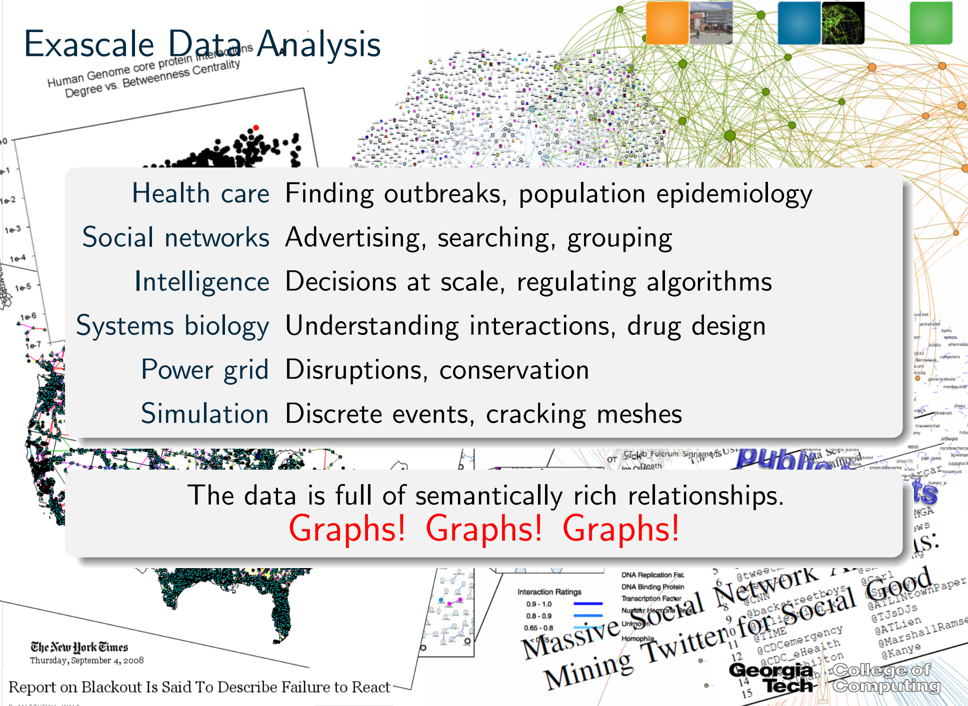
Motivation

Technical

- Why analyze data streams?
- Overall streaming approach
- Clustering coefficients
- Connected components
- Common aspects and questions

Session

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!

Graphs are pervasive

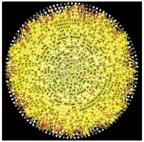
- Sources of massive data: petascale simulations, experimental devices, the Internet, scientific applications.
- New challenges for analysis: data sizes, heterogeneity, uncertainty, data quality.

Astrophysics

Problem Outlier detection
Challenges Massive data sets, temporal variation
Graph problems Matching, clustering

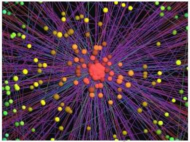
Bioinformatics

Problem Identifying target proteins
Challenges Data heterogeneity, quality
Graph problems Centrality, clustering



Social Informatics

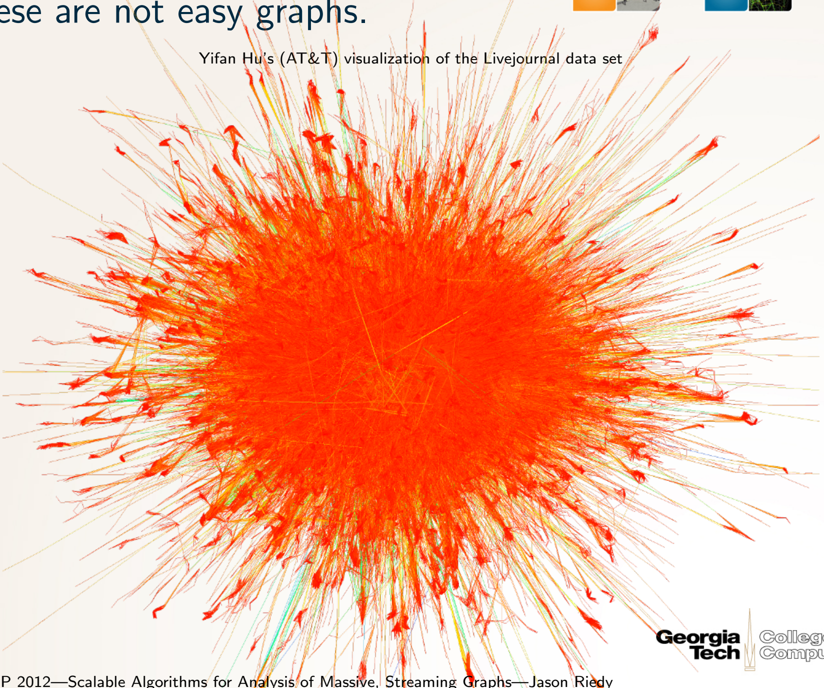
Problem Emergent behavior, information spread
Challenges *New* analysis, data uncertainty
Graph problems Clustering, flows, shortest paths



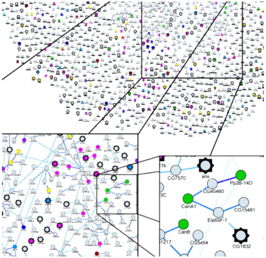
Georgia Tech College of Computing

These are not easy graphs.

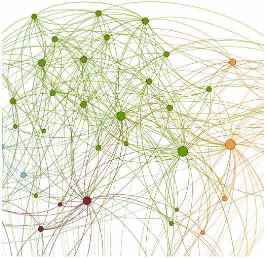
Yifan Hu's (AT&T) visualization of the Livejournal data set



But no shortage of structure...



Protein interactions, Giot *et al.*, "A Protein Interaction Map of Drosophila melanogaster", Science 302, 1722-1736, 2003.



Jason's network via LinkedIn Labs

- Globally, there rarely are good, balanced separators in the scientific computing sense.
- Locally, there are clusters or communities and many levels of detail.

Also 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) `http://public.web.cern.ch/public/en/lhc/Computing-en.html`
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/`
Faceboot	845 M users... and growing.

- 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, ...
- 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, ...

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 $\leftrightarrow$ Memory: QPI,HT: 2PB/day@100%

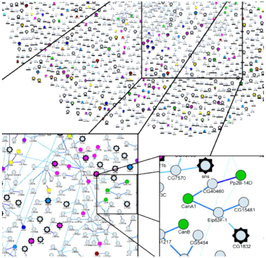
Data growth

- Facebook: $> 2\times$/yr
- Twitter: $> 10\times$/yr
- Growing sources: Bioinformatics, μsensors, security

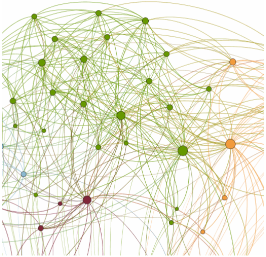
Speed growth

- Ethernet/IB/*etc.*: $4\times$ in next 2 years. Maybe.
- Flash storage, direct: $10\times$ write, $4\times$ read. Relatively huge cost.

Overall streaming approach



Protein interactions, Giot *et al.*, "A Protein Interaction Map of Drosophila melanogaster", Science 302, 1722-1736, 2003.



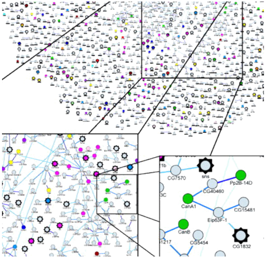
Jason's network via LinkedIn Labs

Assumptions

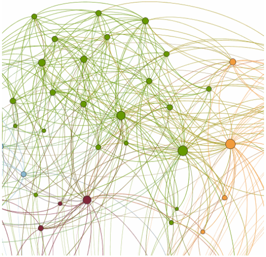
- A graph represents some real-world phenomenon.
  - But **not** necessarily exactly!
  - Noise comes from lost updates, partial information, ...

Georgia Tech College of Computing

Overall streaming approach



Protein interactions, Giot *et al.*, “A Protein Interaction Map of Drosophila melanogaster”, Science 302, 1722-1736, 2003.

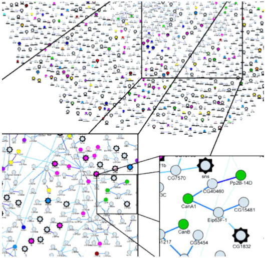


Jason’s network via LinkedIn Labs

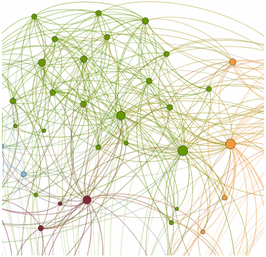
Assumptions

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

Overall streaming approach



Protein interactions, Giot *et al.*, "A Protein Interaction Map of Drosophila melanogaster", Science 302, 1722-1736, 2003.



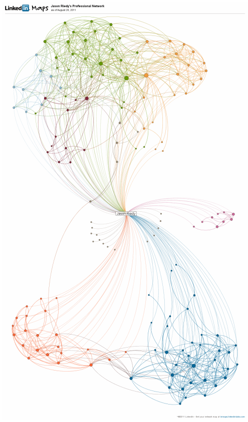
Jason's network via LinkedIn Labs

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.

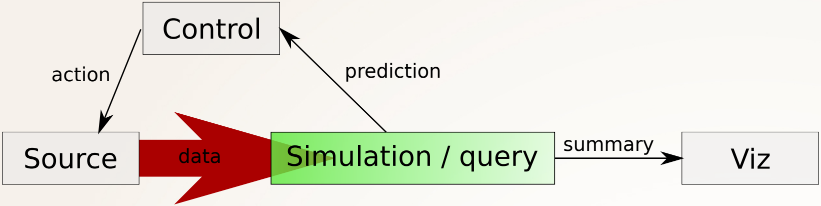
Difficulties for performance

- 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.



Jason's network via LinkedIn Labs

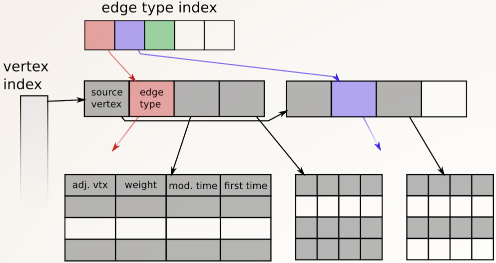
STING's focus



- 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.

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/stinger/

Experimental setup

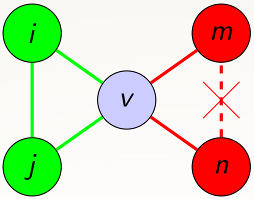
Unless otherwise noted

Line	Model	Speed (GHz)	Sockets	Cores
Nehalem	X5570	2.93	2	4
Westmere	E7-8870	2.40	4	10

- Westmere loaned by Intel (*thank you!*)
- All memory: 1067MHz DDR3, installed appropriately
- Implementations: OpenMP, gcc 4.6.1, Linux $\approx$ 3.0 kernel
- Artificial graph and edge stream generated by R-MAT [Chakrabarti, Zhan, & Faloutsos].
  - 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 & 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:

  # of closed triplets / total # of triplets
- Locally around v or globally for entire graph.

Updating triangle counts

Given Edge $\{u, v\}$ 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

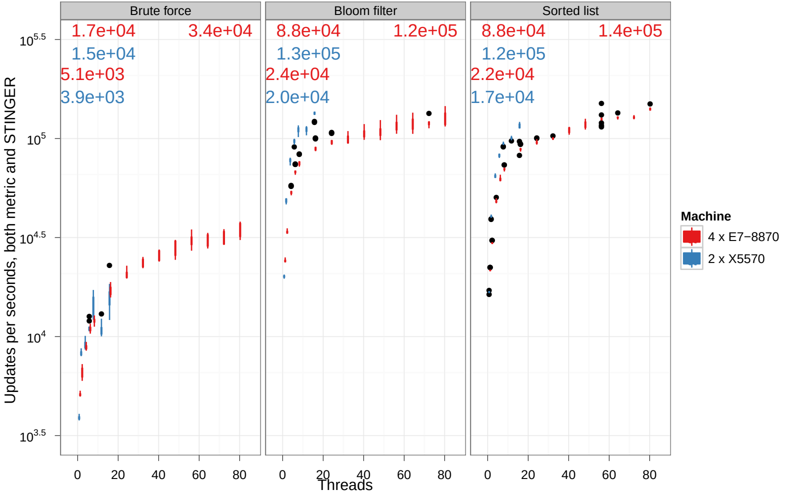
Brute force Intersect neighbors of u and v by iterating over each, $O(d_u d_v)$ time.

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]

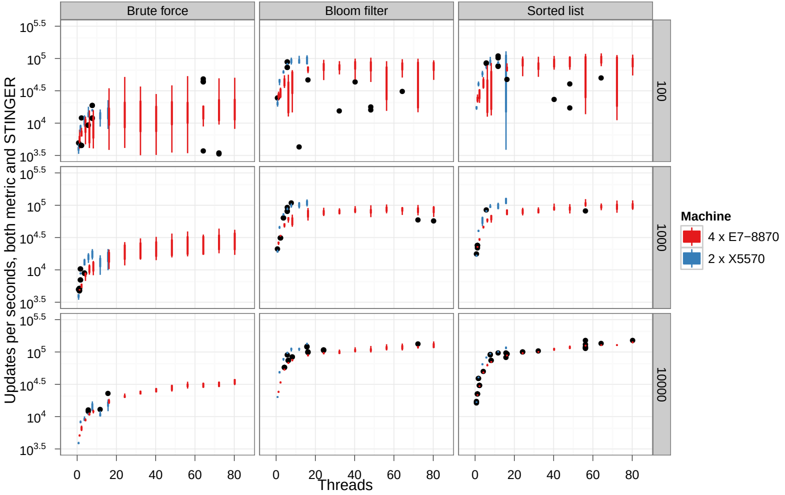
All rely on atomic addition.

Batches of 10k actions



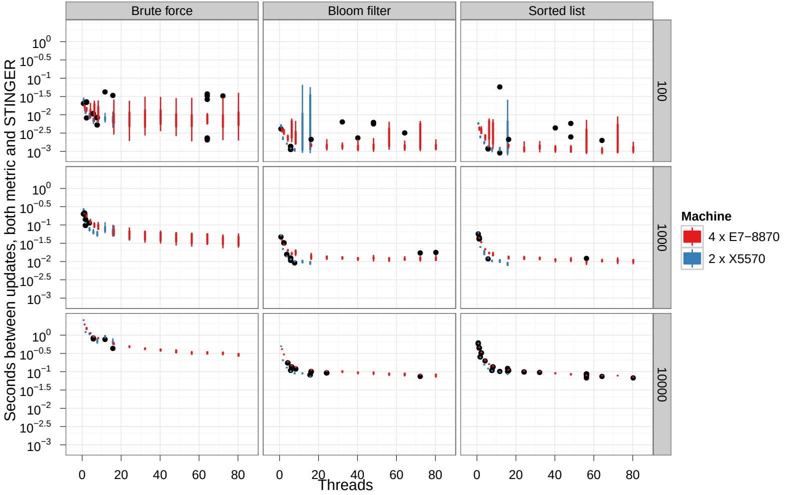
Graph size: scale 22, edge factor 16

Different batch sizes



Graph size: scale 22, edge factor 16

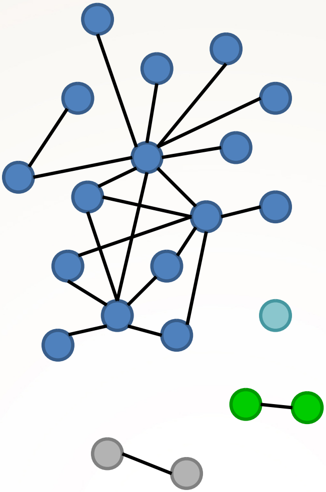
Different batch sizes: Reactivity



Graph size: scale 22, edge factor 16

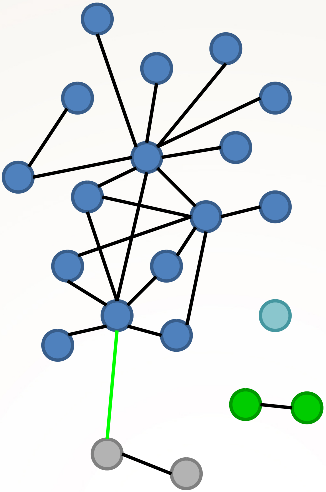
Connected components

- 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

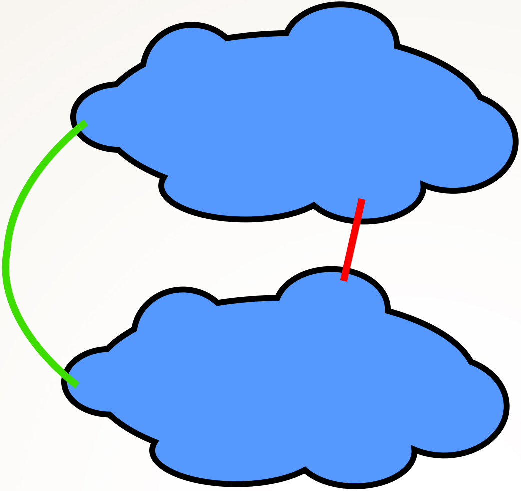
- Maintain a mapping from vertex to component.
- *Global* property, unlike triangle counts
- In “scale free” social networks:
  - Often one big component, and
  - many tiny ones.
- Edge changes often sit *within* components.
- Remaining insertions merge components.
- Deletions are more difficult...



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.



Georgia Tech College of Computing

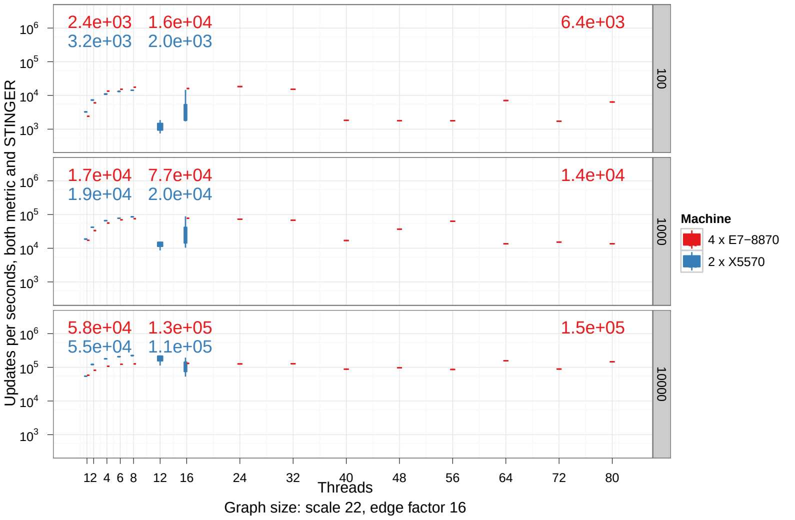
Deletion heuristics

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



Common aspects

- Each parallelizes sufficiently well over the affected vertices V', those touched by new or removed edges.
- Total amount of work is $O(\mathrm{Vol}(V')) = O(\sum_{v \in V'} \deg(v))$.
- Our in-progress work on refining or re-agglomerating communities with updates also is $O(\mathrm{Vol}(V'))$.

- How many interesting graph properties can be updated with $O(\mathrm{Vol}(V'))$ work?
- Do these parallelize well?
- The hidden constant and how quickly performance becomes asymptotic determines the metric update rate. What implementation techniques bash down the constant?
- How sensitive are these metrics to noise and error?
- How quickly can we "forget" data and still maintain metrics?

Session outline

Emergent Behavior Detection in Massive Graphs :
Nadya Bliss and Benjamin Miller, Massachusetts Institute of Technology, USA

Scalable Graph Clustering and Analysis with KDT :
John R. Gilbert and Adam Lugowski, University of California, Santa Barbara, USA; Steve Reinhardt, Cray, USA

Multiscale Approach for Network Compression-friendly Ordering :
Ilya Safro, Argonne National Laboratory, USA; Boris Temkin, Weizmann Institute of Science, Israel

Acknowledgment of support