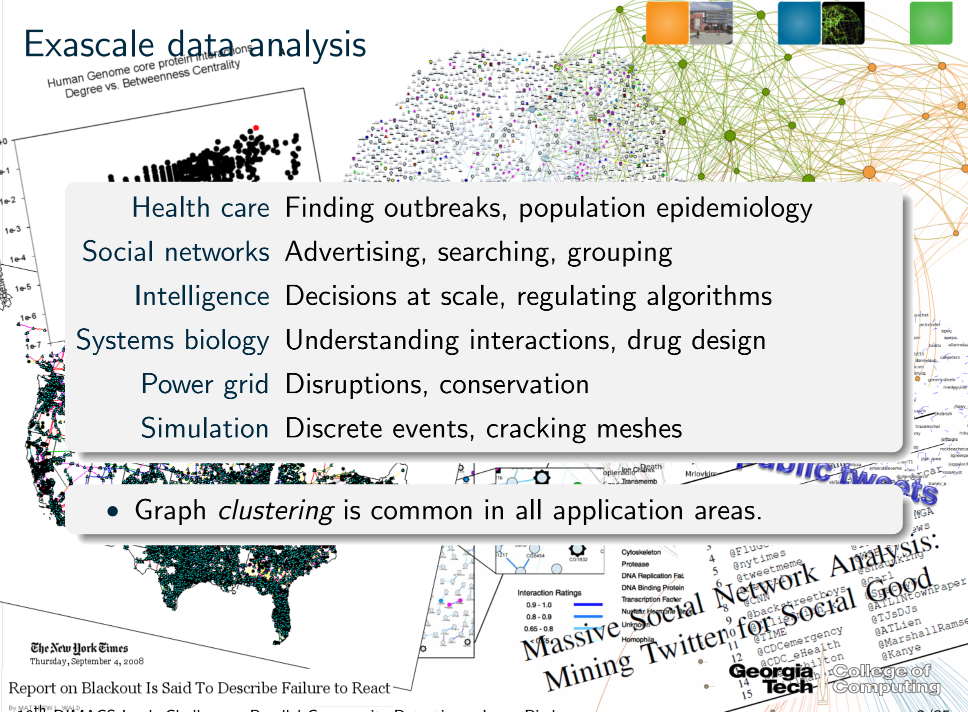
Parallel Community Detection for Massive Graphs

E. Jason Riedy, Henning Meyerhenke, David Ediger, and David A. Bader

14 February 2012

Georgia Tech College of Computing
Computational Science and Engineering

Exascale data analysis

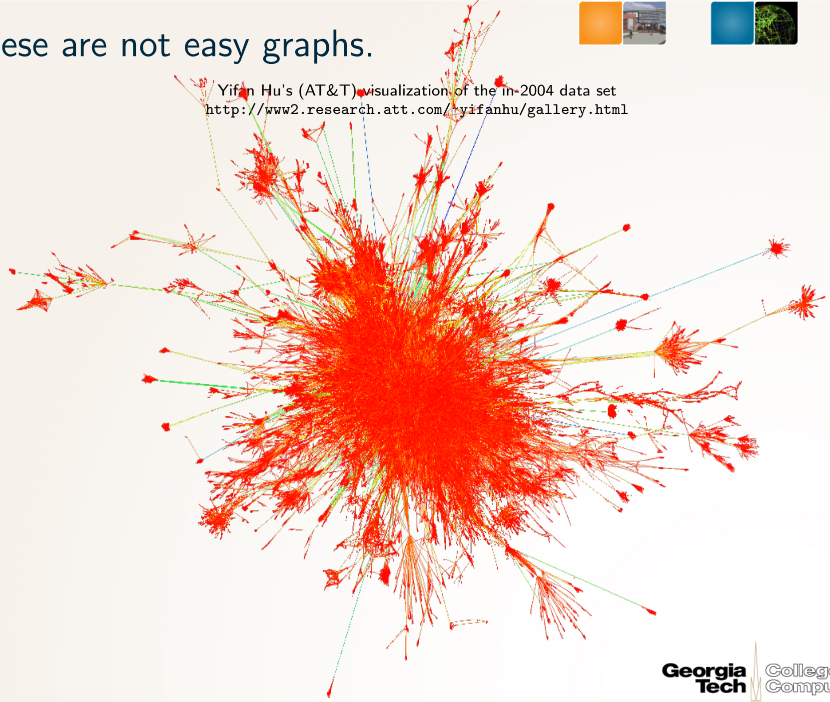


Health care	Finding outbreaks, population epidemiology
Social networks	Advertising, searching, grouping
Intelligence	Decisions at scale, regulating algorithms
Systems biology	Understanding interactions, drug design
Power grid	Disruptions, conservation
Simulation	Discrete events, cracking meshes

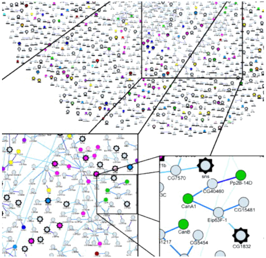
- Graph *clustering* is common in all application areas.

These are not easy graphs.

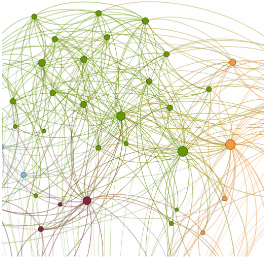
Yifan Hu's (AT&T) visualization of the in-2004 data set
http://www2.research.att.com/~yifanhu/gallery.html



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

- Locally, there are clusters or communities.
- First pass over a massive social graph:
  - Find smaller communities of interest.
  - Analyze / visualize top-ranked communities.
- Our part: *Community detection at massive scale.* (Or kinda large, given available data.)

Outline

Motivation

Shooting for massive graphs

Our parallel method

Implementation and platform details

Performance

Conclusions and plans

Can we tackle massive graphs *now*?

Parallel, of course...

- **Massive** needs distributed memory, right?
- Well... Not really. Can buy a 2 TiB Intel-based Dell server on-line for around $200k USD, a 1.5 TiB from IBM, *etc.*

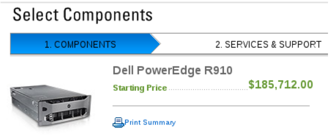


Image: dell.com.

NOT AN ENDORSEMENT, JUST EVIDENCE!

- Publicly available "real-world" data fits...
- Start with shared memory to see what needs done.
- Specialized architectures provide larger shared-memory views over distributed implementations (*e.g.* Cray XMT).

Designing for parallel algorithms

What should we avoid in algorithms?

Rules of thumb:

- "We order the vertices (or edges) by..." unless followed by bisecting searches.
- "We look at a region of size *more than two steps*..." Many target massive graphs have diameter of ≈ 20. More than two steps swallows much of the graph.
- "Our algorithm requires *more than* $\tilde{O}(|E|/\#)$..." Massive means you hit asymptotic bounds, and $|E|$ is plenty of work.
- "For each vertex, we *do something sequential*..." The few high-degree vertices will be large bottlenecks.

Remember: Rules of thumb can be broken *with reason*.

Designing for parallel implementations

What should we avoid in implementations?

Rules of thumb:

- Scattered memory accesses through traditional sparse matrix representations like CSR. *Use your cache lines.*

idx: 32b	idx: 32b	...

val: 64b	val: 64b	...

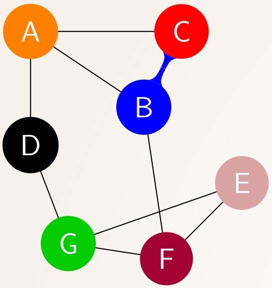
idx1: 32b	idx2: 32b	val1: 64b	val2: 64b	...

- Using too much memory, which is a painful trade-off with parallelism. Think Fortran and workspace...
- Synchronizing too often. There will be work imbalance; try to use the imbalance to reduce "hot-spotting" on locks or cache lines.

Remember: Rules of thumb can be broken *with reason*. Some of these help when extending to PGAS / message-passing.

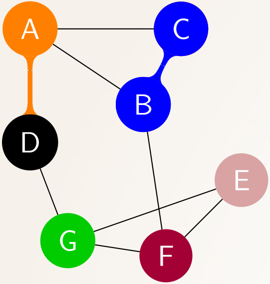
Georgia Tech College of Computing

Sequential agglomerative method



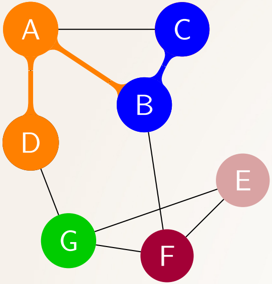
- A common method (*e.g.* Clauset, Newman, & Moore) *agglomerates* vertices into communities.
- Each vertex begins in its own community.
- An edge is chosen to contract.
  - Merging maximally increases modularity.
  - *Priority queue.*
- Known often to fall into an $O(n^2)$ performance trap with modularity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

Sequential agglomerative method



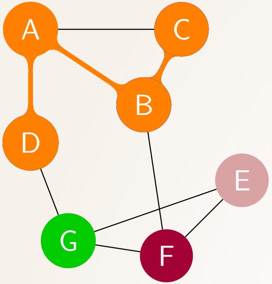
- A common method (*e.g.* Clauset, Newman, & Moore) *agglomerates* vertices into communities.
- Each vertex begins in its own community.
- An edge is chosen to contract.
  - Merging maximally increases modularity.
  - *Priority queue.*
- Known often to fall into an $O(n^2)$ performance trap with modularity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

Sequential agglomerative method



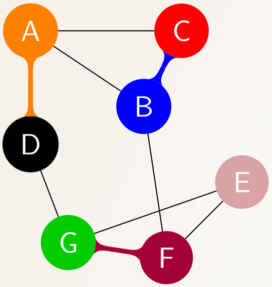
- A common method (*e.g.* Clauset, Newman, & Moore) *agglomerates* vertices into communities.
- Each vertex begins in its own community.
- An edge is chosen to contract.
  - Merging maximally increases modularity.
  - *Priority queue.*
- Known often to fall into an $O(n^2)$ performance trap with modularity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

Sequential agglomerative method



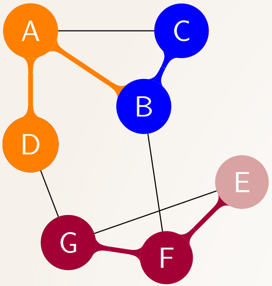
- A common method (*e.g.* Clauset, Newman, & Moore) *agglomerates* vertices into communities.
- Each vertex begins in its own community.
- An edge is chosen to contract.
  - Merging maximally increases modularity.
  - *Priority queue.*
- Known often to fall into an $O(n^2)$ performance trap with modularity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

Parallel agglomerative method



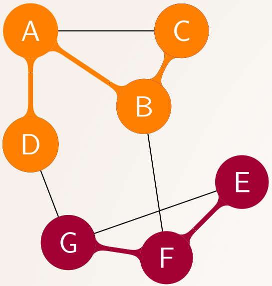
- We use a matching to avoid the queue.
- Compute a heavy weight, large matching.
  - Simple greedy algorithm.
  - Maximal matching.
  - Within factor of 2 in weight.
- Merge all communities at once.
- Maintains some balance.
- *Produces different results.*
- Agnostic to weighting, matching...
  - Can maximize modularity, minimize conductance.
  - Modifying matching permits easy exploration.

Parallel agglomerative method



- We use a matching to avoid the queue.
- Compute a heavy weight, large matching.
  - Simple greedy algorithm.
  - Maximal matching.
  - Within factor of 2 in weight.
- Merge all communities at once.
- Maintains some balance.
- *Produces different results.*
- Agnostic to weighting, matching...
  - Can maximize modularity, minimize conductance.
  - Modifying matching permits easy exploration.

Parallel agglomerative method



- We use a matching to avoid the queue.
- Compute a heavy weight, large matching.
  - Simple greedy algorithm.
  - Maximal matching.
  - Within factor of 2 in weight.
- Merge all communities at once.
- Maintains some balance.
- *Produces different results.*
- Agnostic to weighting, matching...
  - Can maximize modularity, minimize conductance.
  - Modifying matching permits easy exploration.

Platform: Cray XMT2

Tolerates latency by massive multithreading.

- Hardware: 128 threads per processor
  - Context switch on every cycle (500 MHz)
  - Many outstanding memory requests (180/proc)
  - "No" caches...
- Flexibly supports dynamic load balancing
  - Globally hashed address space, no data cache
- Support for fine-grained, word-level synchronization
  - Full/empty bit on with every memory word

- 64 processor XMT2 at CSCS, the Swiss National Supercomputer Centre.
- 500 MHz processors, 8192 threads, 2 TiB of shared memory

Image: cray.com

Platform: Intel® E7-8870-based server

Tolerates some latency by hyperthreading.

- Hardware: 2 threads / core, 10 cores / socket, four sockets.
  - Fast cores (2.4 GHz), fast memory (1 066 MHz).
  - Not so many outstanding memory requests (60/socket), but large caches (30 MiB L3 per socket).
- Good system support
  - Transparent hugepages reduces TLB costs.
  - Fast, user-level locking. (HLE would be better...)
  - OpenMP, although I didn't tune it...

- mirasol, #17 on Graph500 (thanks to UCB)
- Four processors (80 threads), 256 GiB memory
- gcc 4.6.1, Linux kernel 3.2.0-rc5

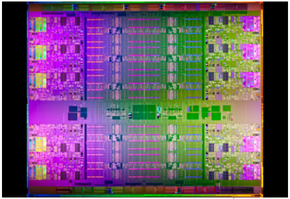


Image: Intel® press kit

Implementation: Data structures

Extremely basic for graph $G = (V, E)$

- An array of $(i, j; w)$ weighted edge pairs, each i, j stored only once and packed, uses $3|E|$ space
- An array to store self-edges, $d(i) = w$, $|V|$
- A temporary floating-point array for scores, $|E|$
- A additional temporary arrays using $4|V| + 2|E|$ to store degrees, matching choices, offsets...

- Weights count number of agglomerated vertices or edges.
- Scoring methods (modularity, conductance) need only vertex-local counts.
- Storing an undirected graph in a symmetric manner reduces memory usage drastically and works with our simple matcher.

Georgia Tech College of Computing

Implementation: Data structures

Extremely basic for graph $G = (V, E)$

- An array of $(i, j; w)$ weighted edge pairs, each i, j stored only once and packed, uses $3|E|$ 32-bit space
- An array to store self-edges, $d(i) = w$, $|V|$
- A temporary floating-point array for scores, $|E|$
- A additional temporary arrays using $2|V| + |E|$ 64-bit, $2|V|$ 32-bit to store degrees, matching choices, offsets...

- Need to fit `uk-2007-05` into 256 GiB.
- Cheat: Use 32-bit integers for indices. Know we won't contract so far to need 64-bit weights.
- Could cheat further and use 32-bit floats for scores.
- (Note: Code didn't bother optimizing workspace size.)

Georgia Tech | College of Computing

Implementation: Data structures

Extremely basic for graph $G = (V, E)$

- An array of $(i, j; w)$ weighted edge pairs, each i, j stored only once and packed, uses $3|E|$ space
- An array to store self-edges, $d(i) = w$, $|V|$
- A temporary floating-point array for scores, $|E|$
- A additional temporary arrays using $2|V| + |E|$ 64-bit, $2|V|$ 32-bit to store degrees, matching choices, offsets...

- Original ignored order in edge array, killed OpenMP.
- New: Roughly bucket edge array by first stored index. Non-adjacent CSR-like structure.
- New: Hash i, j to determine order. Scatter among buckets.
- (New = MTAAP 2012)

Implementation: Routines

Three primitives: Scoring, matching, contracting

Scoring Trivial.

Matching Repeat until no ready, unmatched vertex:

1. For each unmatched vertex in parallel, find the best unmatched neighbor in its bucket.
2. Try to point remote match at that edge (lock, check if best, unlock).
3. If pointing succeeded, try to point self-match at that edge.
4. If both succeeded, yeah! If not and there was some eligible neighbor, re-add self to ready, unmatched list.

(Possibly *too* simple, but...)

Georgia Tech College of Computing

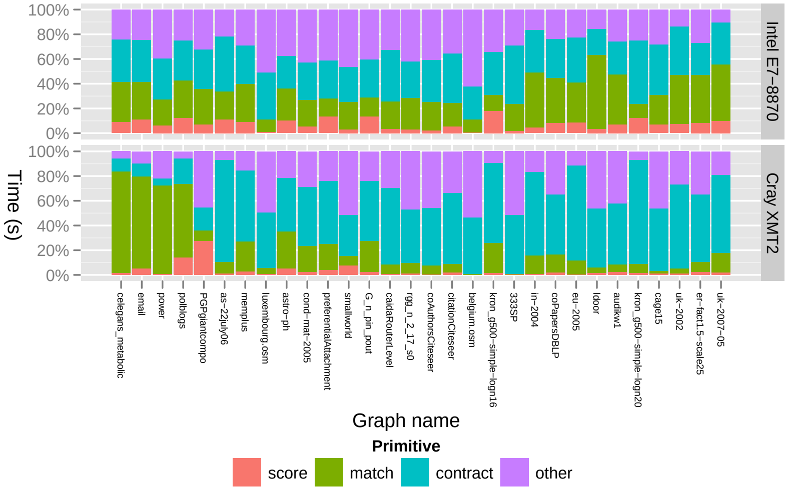
Implementation: Routines

Contracting

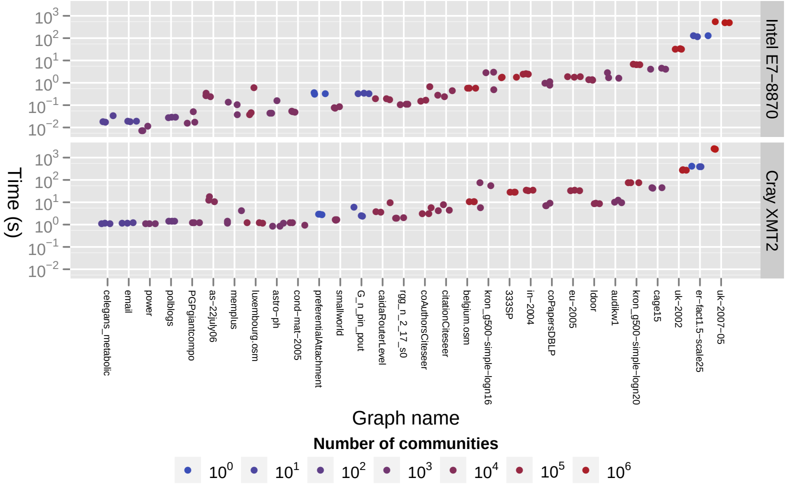
1. Map each i, j to new vertices, re-order by hashing.
2. Accumulate counts for new i' bins, prefix-sum for offset.
3. Copy into new bins.

- Only synchronizing in the prefix-sum. That could be removed if I don't re-order the i', j' pair; haven't timed the difference.
- Actually, the current code copies twice... On short list for fixing.
- Binning as opposed to original list-chasing enabled Intel/OpenMP support with reasonable performance.

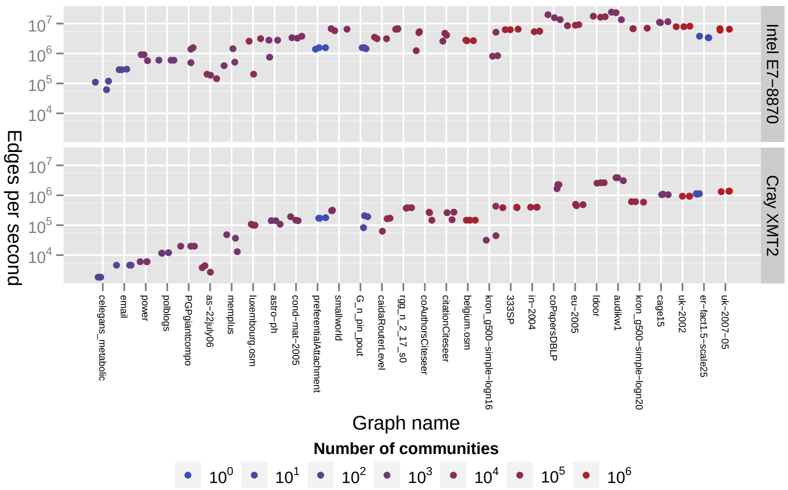
Implementation: Routines



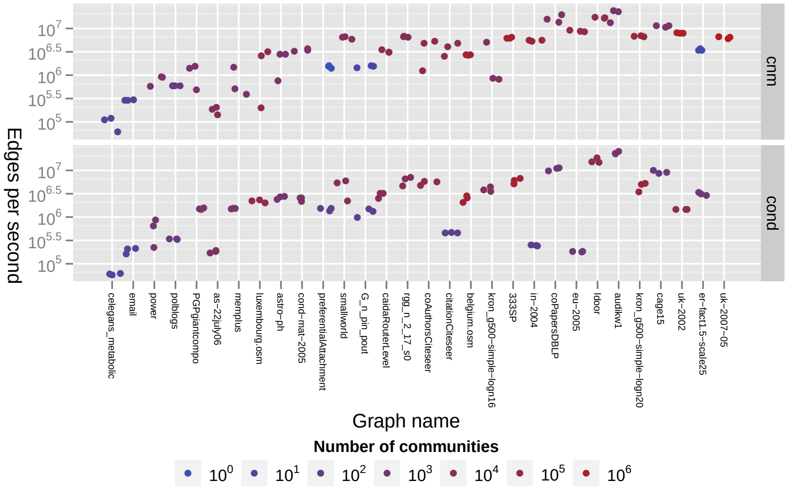
Performance: Time by platform



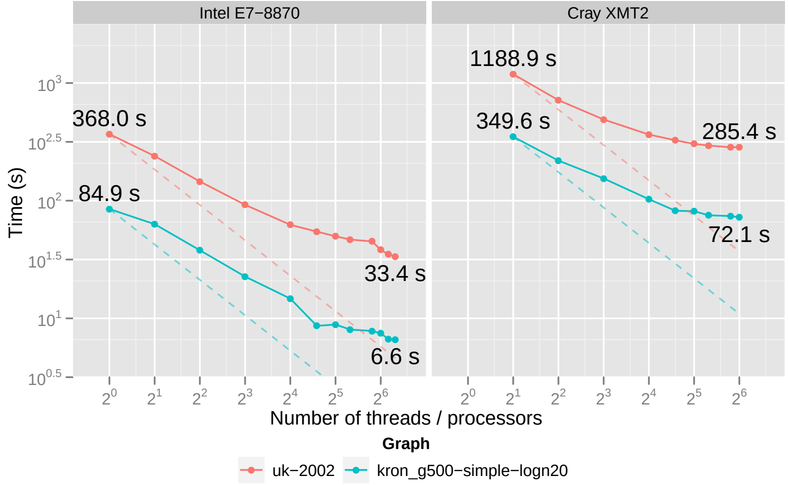
Performance: Rate by platform



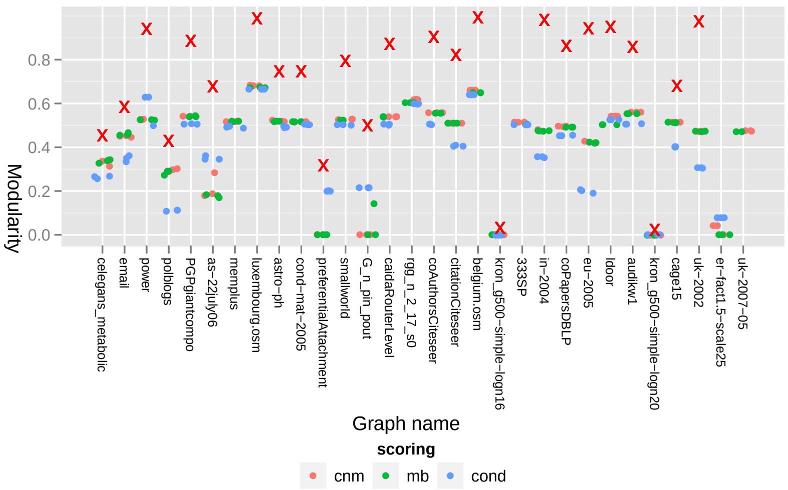
Performance: Rate by metric (on Intel)



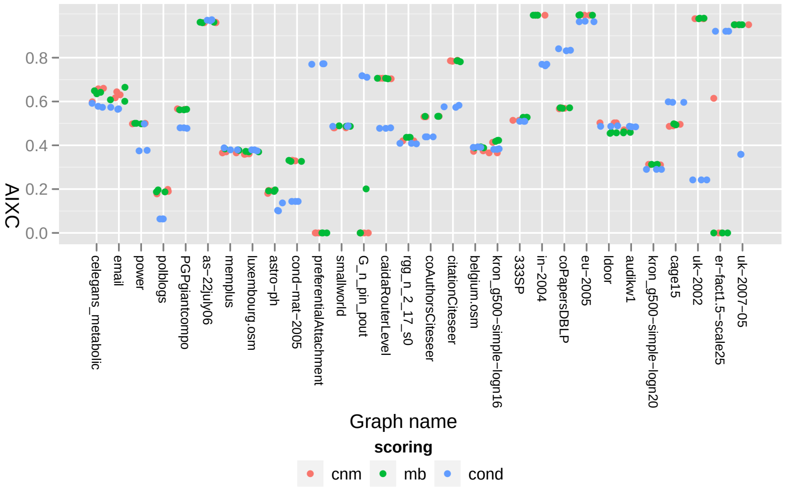
Performance: Scaling



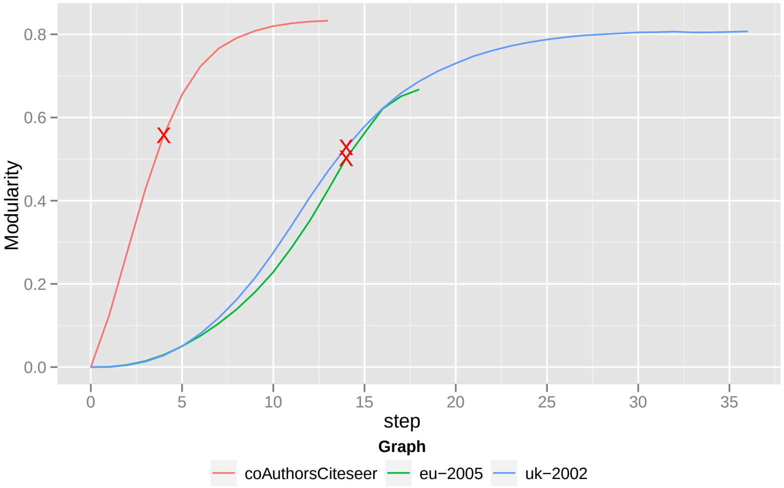
Performance: Modularity at coverage ≈ 0.5



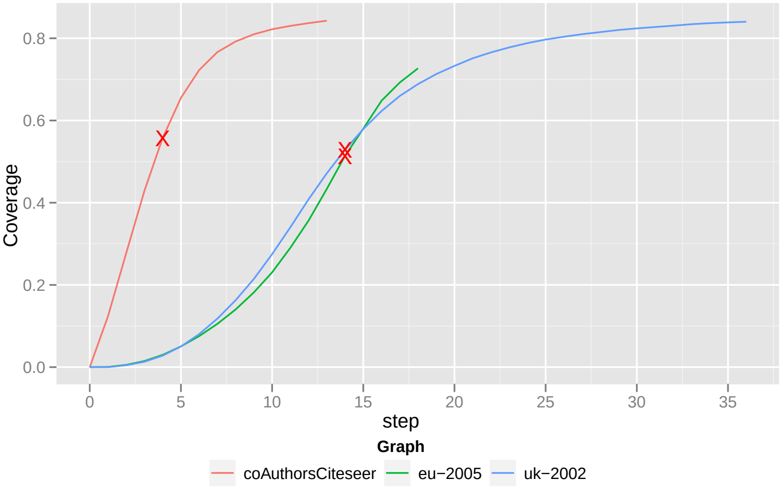
Performance: Avg. conductance at coverage ≈ 0.5



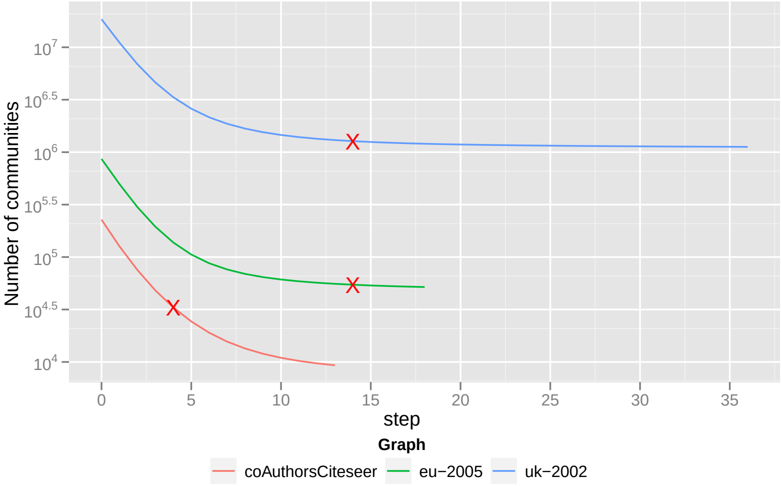
Performance: Modularity by step



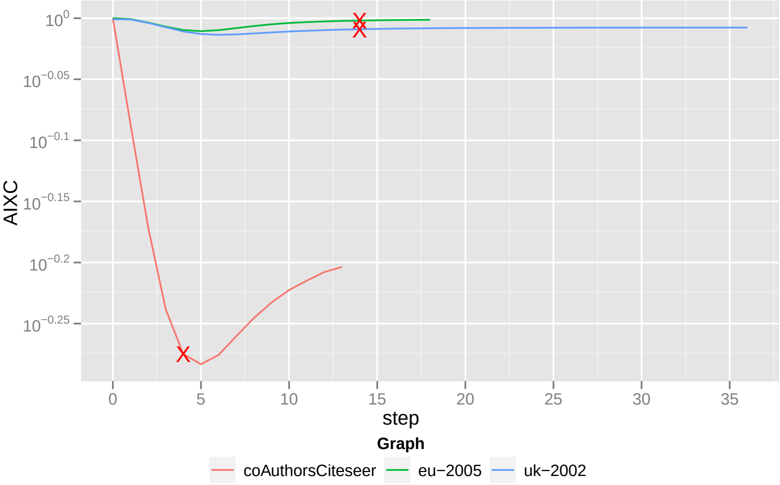
Performance: Coverage by step



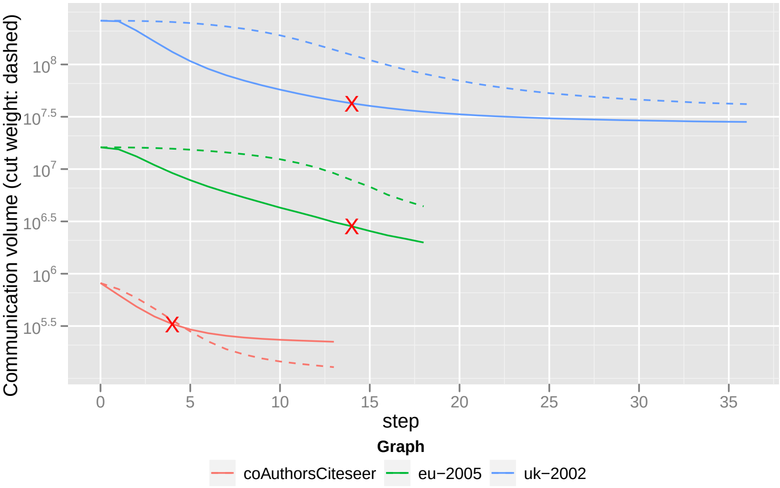
Performance: # of communities



Performance: AIXC by step



Performance: Comm. volume by step



Conclusions and plans

- Code: `http://www.cc.gatech.edu/~jriedy/community-detection/`
- First: Fix the low-hanging fruit.
  - Eliminate a copy during contraction.
  - Deal with stars (next presentation).
- Then... Practical experiments.
  - How volatile are modularity and conductance to perturbations?
  - What matching schemes work well?
  - How do different metrics compare in applications?
- Extending to streaming graph data!
  - Includes developing parallel refinement... (distance 2 matching)
  - And possibly de-clustering or manipulating the dendogram.

Georgia Tech College of Computing

Acknowledgment of support