Multithreaded Community Monitoring for Massive Streaming Graph Data

Jason Riedy David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA

***Abstract*—Analyzing static snapshots of massive, graph-structured data cannot keep pace with the growth of social networks, financial transactions, and other valuable data sources. Current state-of-the-art industrial methods analyze these streaming sources using only simple, aggregate metrics. There are few existing scalable algorithms for monitoring complex global quantities like decomposition into community structure. Using our framework STING, we present the first known parallel algorithm specifically for monitoring communities in this massive, streaming, graph-structured data. Our algorithm performs incremental re-agglomeration rather than starting from scratch after each batch of changes, reducing the problem's size to that of the change rather than the entire graph. We analyze our initial implementation's performance on multithreaded platforms for execution time and latency. On an Intel-based multithreaded platform, our algorithm handles up to 100 million updates per second on social networks with one to 30 million edges, providing a speed-up from 4× to 3700× over statically recomputing the decomposition after each batch of changes. Possibly because of our artificial graph generator, resulting communities' modularity varies little from the initial graph.**

I. Introduction

The world is awash in data of all forms. Highway sensors generate continuous traffic information, high-throughput sequencers produce vast quantities of genetic information, people send text and images constantly, and more. Everyone recognizes the sheer volume of raw data already surpasses our analysis capabilities and keeps growing. Much of this data consists of *relationships*, providing a rich and changing graph structure.

To derive insight from the mass of data requires more than current high-performance, parallel automatic analysis. Applying static analysis to a changing network of a billion Facebook users sharing and communicating content produces information potentially long after the context has changed. Tracking closely linked sets of accounts in Twitter during the Euro 2012 final match between Spain and Italy would have required analyzing a graph representing messages arriving at rates up to fifteen thousand times per second [1]. Analyzing global computer networks for anomalous or suspect behavior requires more rapid turn-around time than static algorithms deliver. Emerging applications require more complex graph analysis that adapts quickly to *changing* graph data by working with the stream of information as it arrives.

Much work on analysis of streaming, graph-structured data focuses primarily on aggregate information like counts and averages. These sufficient statistics assist in diffusion models for accurate trend prediction in power-law networks [2] and other applications, but do not help with more complex analysis like guiding sampling for approximate centrality measures. Here we build on our previous work in both complex analysis of streaming graph data [3], [4], [5] and community detection in static graphs [6], [7], [8] to provide the first algorithm for maintaining a community decomposition of a changing, undirected social network. On an Intel-based multithreaded platform, our algorithm handles up to 100 million updates per second on social networks with one to 30 million edges, providing a speed-up from $4\times$ to $3700\times$ over statically recomputing the decomposition after

each batch of changes. On a Cray XMT2, our algorithm provides provides a speed-up from $1.3\times$ to almost $3200\times$ over static recomputation. In all cases, the static recomputation uses a highly scalable multithreaded algorithm[7], [8], [6].

Note that our use, *analysis of streaming data*, differs from *streaming algorithms* and *dynamic algorithms* in computer science. We optimize analysis performance by using batches of streaming data to reduce the problem size. Streaming algorithms consider using a minimal amount of memory while passing over a data set once or possibly many times. Dynamic algorithms specialize data structures and algorithms for completely dynamic operations. Our approach maintains a single large data structure, STINGER, and incrementally updates analysis results using smaller, analysis-kernel-specific data structures, adopting ideas from both bodies of existing work.

A. STING, a Framework for Streaming, Graph-Structured Data Analysis

To tackle analysis in these new situations, we are developing the free STING (Spatio-Temporal Interaction Networks and Graphs) framework[1] [3], [5], [4]. STING balances portability, productivity, and performance for research and development. Our young framework accumulates batches of edge changes into a semantic graph data structure, STINGER (STING Extensible Representation), and runs analysis kernels to monitor graph properties. STING is a C framework portable across POSIX and OpenMP platforms as well as the Cray XMT. STING includes kernels maintaining vertex-local quantities like clustering coefficients [5] and global information like connected component labeling [4].

Here we add maintaining a global community decomposition through edge insertions and removals. Edge insertions add new edges or increment the weight of existing edges. Edge removals entirely delete an edge from the graph if the edge exists.

B. Outline

This paper introduces the first community monitoring analysis kernel for STING. To our knowledge, this is not only the first parallel algorithm but the first algorithm at all to update communities rather than recompute them from scratch. Our algorithm runs in time proportional to the amount of the graph affected by the batch of changes plus the size of a graph representing the community structure, a much smaller graph than the entire network. Section II describes the algorithm and important implementation details. Section III specifies our test case generator. We begin with a real-world graph and add artificial edges to measure our implementation's performance in Section IV.

[1]Available at http://www.cc.gatech.edu/stinger/.

II. Re-agglomeration Algorithm

We adapt our static parallel agglomerative community detection algorithm for streaming data by *re-agglomeration*. The algorithm adapts the community mapping by extracting affected vertices from a contracted graph representing the communities and then re-applying the static algorithm to this much smaller graph. We briefly outline our static parallel agglomerative community detection algorithm and data structure [6], [7], [8], highlighting adaptations to streaming data. Then we describe the re-agglomeration in more detail, particularly the process of modifying the community graph. Our implementation's source code is available in the STING development repository.

A. Static Agglomerative Community Detection

The agglomerative algorithm takes an initial graph as input and returns a contracted *community graph* as well as a mapping from the initial graph's vertices to the community labels. The community graph has a vertex for each community. An edge between two communities exists when any vertex in one community has a neighbor in the other. The community edge weight is the sum of all edges between the two communities.

Our algorithm for streaming data applies our prior static agglomerative algorithm to an updated community graph. If the community graph is denoted by the pair $G_C = (C, E_C)$, where C is the set of communities and E_C are the weighted edges between communities, then each agglomeration step requires $O(|E_C|)$ operations, both arithmetic and memory [7].

The static agglomerative algorithm works on a simple edge list data structure. Each graph edge $\{i, j; w\}$ between vertices $i \neq j$ with weight w is stored exactly once in an array of all edges. Self-edge weights are accumulated in a separate array; a self-edge weight of zero implies there are no self edges. This structure permits low-overhead parallelization across the entire edge list.

The static algorithm repeats three phases until the community metric stops improving: scoring, matching, and contraction. Scoring assigns to each edge the change in the community metric. Here we use modularity for the community metric. Scoring details are available in prior work; the important feature is that the score of an edge relies only on information local to the edge and its endpoints.

In matching, our algorithm computes a maximal matching that greedily maximizes the sum of the matching's edges' scores. A matching with large score suffices for local metric optimization; we do not require the maximum weight maximal matching. Our algorithm provides a matching with total score within a factor of two of the maximum score.

The matching identifies edges to contract. The contraction phase implements this contraction from one community graph to another. Community graph vertices are relabeled with new vertex identifiers, the edges are binned in parallel by the first stored vertex, and each bin is collapsed to accumulate redundant edges weights. To spread load, edges between vertices i and j are stored in a hashed order. If the vertices are both even or both odd, i is stored first, otherwise j is stored first. This seems sufficient to prevent any one vertex bin from growing too large. We re-use the contraction algorithm to incorporate graph changes from the incoming data stream.

B. Agglomeration for Streaming Data

Our streaming re-agglomeration algorithm works by de-agglomerating the community graph and then applying the static agglomeration algorithm to the expanded community graph. We extract vertices affected by edge changes from their communities and re-start the agglomeration process. The expanded community graph is far smaller than the full network. As before, the community graph is $G_C = (C, E_C)$. The full graph is $G = (V, E)$ with vertex set V and edge set E.

The algorithm uses the full graph G represented with STINGER, the maintained community graph G_C represented in an edge list as in Section II-A, and a handful of work arrays. Our current implementation uses $O(|V|)$ storage for work arrays and maintains G_C using storage beyond the minimal $|C|+3|E_C|$, but this could be changed to reallocate and copy as needed with some performance penalty. The community updates occur after applying the batch of changes to the STINGER graph structure.

We assume edges inserted within communities or removed between communities will not cause the communities to split or merge. This is true for edge-local agglomeration using metrics like modularity but may not be true for all metrics. This assumption may reduce the number of changes under consideration drastically.

To implement the reduction, we scan the batch of changes for *active vertices*. We consider a vertex active if it appears as an endpoint for a changed edge that is either inserted between communities or removed within a community. Let ΔV be the set of active vertices represented in a list of size at most $|V|$.

After the batch of actions is applied to the STINGER structure, our algorithm executes the following steps:

1) Collect active vertices as defined above into ΔV using a scatter/gather buffer and atomic compare-and-swap operations.
2) Extract the vertices in ΔV from their existing communities, appending edges to G_C to account for the edge changes (see Algorithm 1).
3) Collapse G_C to accumulate the edge changes using a self-contraction.
4) Re-run agglomeration on G_C.

Note that our algorithm will not necessarily detect when a community is split into separate components. Using a component tracking kernel [4] as input is future work.

C. Modifying the Edge List

De-agglomerating the vertices in ΔV requires extracting graph vertices into new community vertices

within G_C, modifying existing edge weights, and inserting new edges touching the new community vertices. This uses at small multiple of $|V|$ workspace. The final contraction and re-agglomeration require $5|C'| + 3|E'_C|$ space [7], where $G' = (C', E'_C)$ is the de-agglomerated graph. The edge list modification takes the community graph G_C with community vertex counts as input along with the global STINGER graph G, the batch of edge changes, and an $|V|$-long array $\mathrm{cmap}[]$ mapping vertices in G to community labels. The workspace holds the list ΔV of active vertices and an array $\mathrm{mark}[]$ set to extracted vertex community ids or -1 for vertices not extracted into new communities.

The pseudo-code for extracting vertices and adjusting edge weights is in Algorithm 1. Loops are parallel across vertices in ΔV. With additional atomic operations, loops across the adjacency lists could be parallel, which may improve performance on architectures with fine-grained threading.

After the edge list is updated with adjustment edges, we commit the new community labels in $\mathrm{mark}[]$ to $\mathrm{cmap}[]$ in parallel. We then call our existing edge list contraction routine from prior work to collapse duplicate edges, correcting their weights for the extraction operations. Our implementation slightly optimizes the case of self-contraction, but otherwise the contraction algorithm is the same as in [7], [8].

Data: community graph G_C, array cmap mapping vertices to old community ids, an array with each old community size, and array mark that, when non-negative, maps vertices to new community ids
Result: G_C with appended edges adjusting for the batch changes
foreach vertex $i \in \Delta V$ (in parallel) **do**
  Atomically subtract one from i's community's size;
  if the result is not zero **then**
    Atomically obtain a new community id (≥ 0) by incrementing the number of communities;
    Set $\mathrm{mark}[i]$ to the new community id;
    Set the community size of $\mathrm{mark}[i]$ to one;
  else the last vertex stays in its old community
    Restore the community size to one.;
    Set $\mathrm{mark}[i]$ to -1.;
foreach vertex $i \in \Delta V$ with $\mathrm{mark}[i] \geq 0$ (in parallel) **do**
  foreach neighbor j with weight w from STINGER **do**
    begin remove old edges
      if $\mathrm{cmap}[i] \neq \mathrm{cmap}[j]$ **then** append $\{\mathrm{cmap}[i], \mathrm{cmap}[j]; -w\}$ to G_C (only once by requiring $\mathrm{mark}[j] < 0$ or $\mathrm{cmap}[i] < comm[j]$);
      else atomically subtract w from i's community weight (sum of all internal edges);
    begin append new edges
      if $\mathrm{mark}[i] = \mathrm{mark}[j]$ **then** i and j are in the same new community so accumulate w into the new community weight;
      else if $\mathrm{mark}[j] < 0$ **then** append $\{\mathrm{mark}[i], \mathrm{cmap}[j]; w\}$ to G_C;
      else append $\{\mathrm{mark}[i], \mathrm{mark}[j]; w\}$ to G_C when $\mathrm{mark}[i] < \mathrm{mark}[j]$;

Algorithm 1: Extracting individual vertices to new communities (containing only those vertices) and appending edges to account for the weight changes.

III. Generating Test Cases

To test performance of the Section II's algorithm, we rely on artificial edge action streams applied to topical real-world graphs. The generated test cases are not intended to model real-world changes perfectly but only well enough to verify and debug our algorithm performance. Real-world data sets like public actions at GitHub[2] or Stack Exchange[3] require significant data extraction and model curation. We test difficult-case performance using artificial data.

Testing against artificial data suffices to demonstrate rough algorithm performance. Our test cases are not entirely artificial. Each starts with a real-

[2]http://www.githubarchive.org/

[3]http://www.clearbits.net/creators/146-stack-exchange-data-dump

world graph from the 10th DIMACS Implementation Challenge[4] [9] on graph partitioning and clustering. We then generate a stream of edge insertions between the initial graph's vertices and removals from initial or inserted graph edges. Each action is an insertion with probability 15/16 and a removal with probability 1/16.

Given a graph, the generator computes an initial community decomposition. The decomposition is both the starting community used in Section IV's experiments and also the source of edge actions. The edge actions are generated based only on this initial decomposition. The generator is memory-less and does not update the graph based on previously generated edges.

To generate an edge insertion, the generator chooses two distinct communities without replacement with probability proportional to their average weighted volume, the average weight of all edges adjacent to the community's vertices. An endpoint is chosen from each community with probability inversely proportional to its degree. This adds edges between lightly connected vertices in large communities.

Generating edge removals emphasizes existing edges within the graph, but occasional removals of previously removed edges occur. We first fill a queue of initial removals by sampling $2|V|/|E|$ edges randomly from the original graph. Generated removals are extracted from that queue until it is empty. Afterwards, removals are sampled randomly previously inserted edges. Neither case protects against removing the same edge twice. Multiple removals test correctness and performance in the face of somewhat noisy data.

IV. Experiments and Results

We apply our re-agglomeration algorithm to data sets generated from three different real-world graphs and consider total performance, parallel scalability, and speed-up over static recomputation. In each case, the modularity appears roughly similar across all the changes. Further analysis of the dynamic and static result communities is necessary for detailed community quality comparisons.

[4]http://www.cc.gatech.edu/dimacs10/

A. Graphs and Generated Actions

We use the three graphs in Table I for our initial experiments. These are drawn from the 10th DIMACS Implementation Challenge repository. The graph caidaRouterLevel is a graph depicting a router-level view of the Internet collected by the Cooperative Association for Internet Data Analysis (CAIDA) in 2003. The graph coPapersDBLP connects papers in the Digital Bibliography and Library Project by co-authorship [10]. And graph eu-2005 is a small web crawl of the .eu domain [11]. Table I provides both the initial graph sizes and the sizes of the contracted initial community graph.

Each experiment begins from the same initial community graph. For each graph, we generate five million edge actions as described in Section III. All experiments for different batch sizes start from the same initial community graph and use the same edge actions. Experiments apply five consecutive batches of actions individually to the initial community graph, and each experiment is repeated five times to capture system variability. Our experiments use batch sizes of 1, 3, 10, 30, ..., up to one million. Plots do not show all sizes to reduce visual noise. Plots also are limited to the Intel-based platform for space.

B. Multithreaded Platforms

We evaluate parallel performance on two different threaded hardware architectures, an Intel-based server and the Cray XMT2.

The Intel-based server platform is located at Georgia Tech. It has four eight-core Intel Xeon E7-4820 processors running at 2 GHz with 18 MiB of L3 cache per processor. The processors support HyperThreading, so the 32 physical cores appear as 64 logical cores. This server is equipped with 1 TiB of 1 067 MHz DDR3 RAM.

The next generation Cray XMT2 is located at the Swiss National Supercomputing Centre (CSCS). Its 64 processors run at 500 MHz and support four times the memory density of the Cray XMT for a total of 2 TiB. These 64 processors support over 6 400 hardware thread contexts. The improvements over the XMT also include additional memory bandwidth within a node, but exact specifications are not yet officially available.

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

TABLE I
TEST GRAPHS. ALL GRAPHS ARE UNDIRECTED AND COUNTS IGNORE SELF-LOOPS. $|C|$ IS THE NUMBER OF COMMUNITIES, AND $|E_C|$ IS THE NUMBER OF EDGES BETWEEN COMMUNITIES.

The Cray XMT allocates entire processors to applications, each with at least 100 threads, while the OpenMP platforms allocate individual threads which are mapped to cores. Results are shown per-Cray-XMT processor and per-OpenMP-thread. We run up to the number of physical Cray XMT processors or logical Intel cores. Intel cores are allocated in a round-robin fashion across sockets, then across physical cores, and finally logical cores. The Intel-based system allocates 2 MiB large memory pages interleaved across sockets.

C. Update Rates and Latencies

Different users may require different measures of performance. Non-interactive uses may prefer a large aggregate update rate, while interactive queries may require rapid response. Rapid response translates to a low latency between community updates. Measured times include both updating the STINGER data structure as well as the community decomposition.

Table II provides the peak update rates achieved on our three test cases. The peak update rates require large batch sizes with relatively large latency. Table III considers the lowest latency. The lowest latencies are in microseconds but achieve a peak update rates three to five orders of magnitude below the peak. For both cases, the speed-ups over repeating static global community detection ranges from $4\times$ to $3500\times$, showing the benefit of incremental updates. The lowest speed-ups occur with caidaRouterLevel and batch sizes nearly equal to the number of edges in the original graph.

Figures 1 and 2 show the multithreaded scaling of our update algorithm on a subset of batch sizes. The incremental updates work on relatively little data, limiting the total scalability. Small batch sizes reduce the incremental work to moving a handful of vertices with little room for parallelization. For large updates, our algorithm effectively uses up to 16 threads. This provides fast updates while leaving additional resources for applying other analysis kernels to the same data. Figure 3 shows the speed-up over static global re-computation by threads. The speed-up over recomputation decreases as the batch size and thread size increase; the static algorithm scales very well with increasing thread count on these graphs [8].

V. RELATED WORK

There is little work targeting incremental clustering for community metrics like modularity in a massive, changing graph. Existing work like Nguyen, *et al*. [12] applies similar incremental techniques but sequentially. Other work like Bourqui, *et al*. [13] applies repeated static analysis instead of updating the communities incrementally. A recent survey by Fortunato [14] covers many aspects of community detection with an emphasis on modularity maximization. Fortunato covers on dynamic communities from the perspective of community characteristics. One notable earlier work, Hopcraft, *et al*. [15], finds that agglomerative methods are very sensitive to the order of agglomeration. They apply static clustering to large, randomly selected subsets to identify stable "natural communities." The implications on our method may become more clear as we investigate the tracked community quality.

Graph partitioning, graph clustering, and community detection are tightly related topics. There is a vast literature on adapting graph partitions for finite-element, finite-volume and other physical applications. These established methods are incorporated in state-of-the-art software packages like Zoltan [16] and Trilinos [17]. They focus on equal-work or -communication partitions for load balancing and not on optimizing community clustering metrics.

Graph	Platform	# threads	Batch Size	Updates/Sec	Speed-up	Latency (s)
caidaRouterLevel	IA32-64	56	100000	1.20e+07	4.01e+01	8.34e-03
	XMT	56	1000000	2.49e+06	4.28e+00	4.01e-01
coPapersDBLP	IA32-64	20	1000000	2.89e+06	1.08e+01	3.46e-01
	XMT	48	300000	2.23e+06	2.09e+01	1.35e-01
eu-2005	IA32-64	40	100000	4.79e+06	3.27e+02	2.09e-02
	XMT	64	1000000	2.05e+06	4.26e+01	4.88e01

TABLE II

PEAK UPDATES PER SECOND. SPEED-UP MEASURES THE SPEED-UP OVER STATIC RECOMPUTATION AND NOT THE PARALLEL SPEED-UP.

Graph	Platform	# threads	Batch Size	Updates/Sec	Speed-up	Latency (s)
caidaRouterLevel	IA32-64	2	1	4.64e+02	4.28e+02	2.16e-03
caidaRouterLevel	XMT	4	30	5.15e+03	2.29e+02	5.83e03
coPapersDBLP	IA32-64	4	30	7.01e+03	1.85e+03	4.28e-03
coPapersDBLP	XMT	40	1	1.40e+02	3.80e+02	7.12e03
eu-2005	IA32-64	12	1	4.17e+02	3.50e+03	2.40e-03
eu2005	XMT	20	10	1.53e+03	3.18e+03	6.54e03

TABLE III

LEAST LATENCY BETWEEN COMMUNITY UPDATES. SPEED-UP MEASURES THE SPEED-UP OVER STATIC RECOMPUTATION AND NOT THE PARALLEL SPEED-UP.

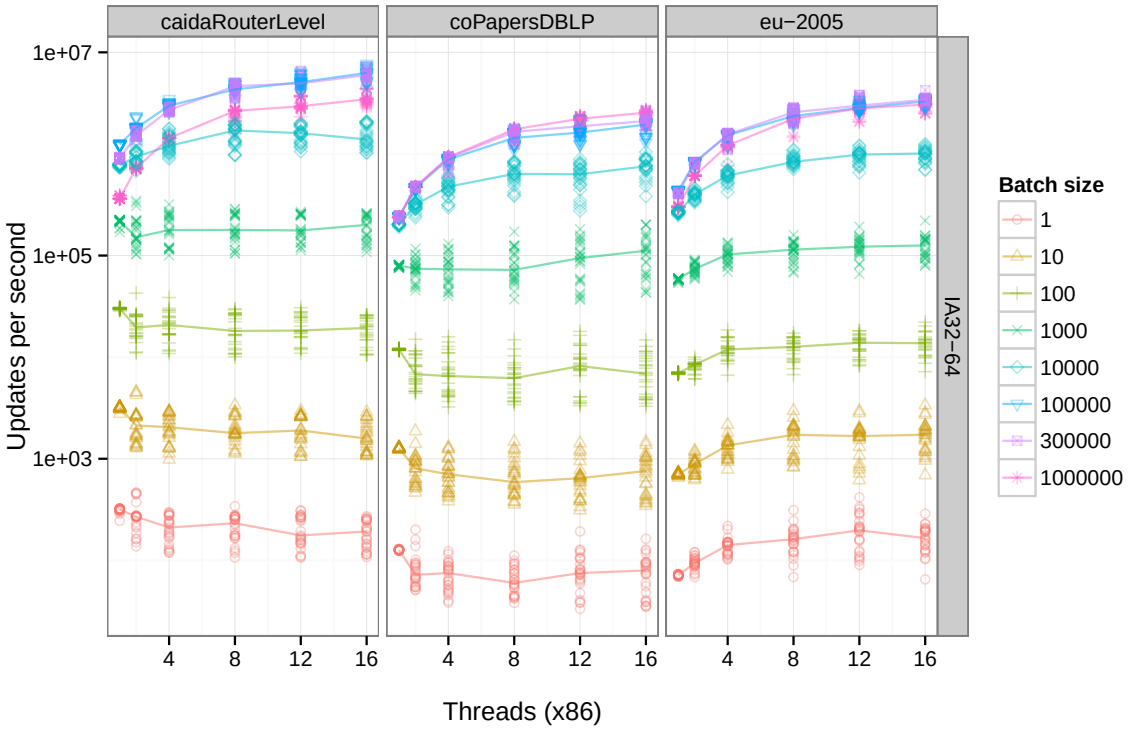

Fig. 1. Updates per second by threads for each test graph and batch size per platform. The solid line connects the median points.

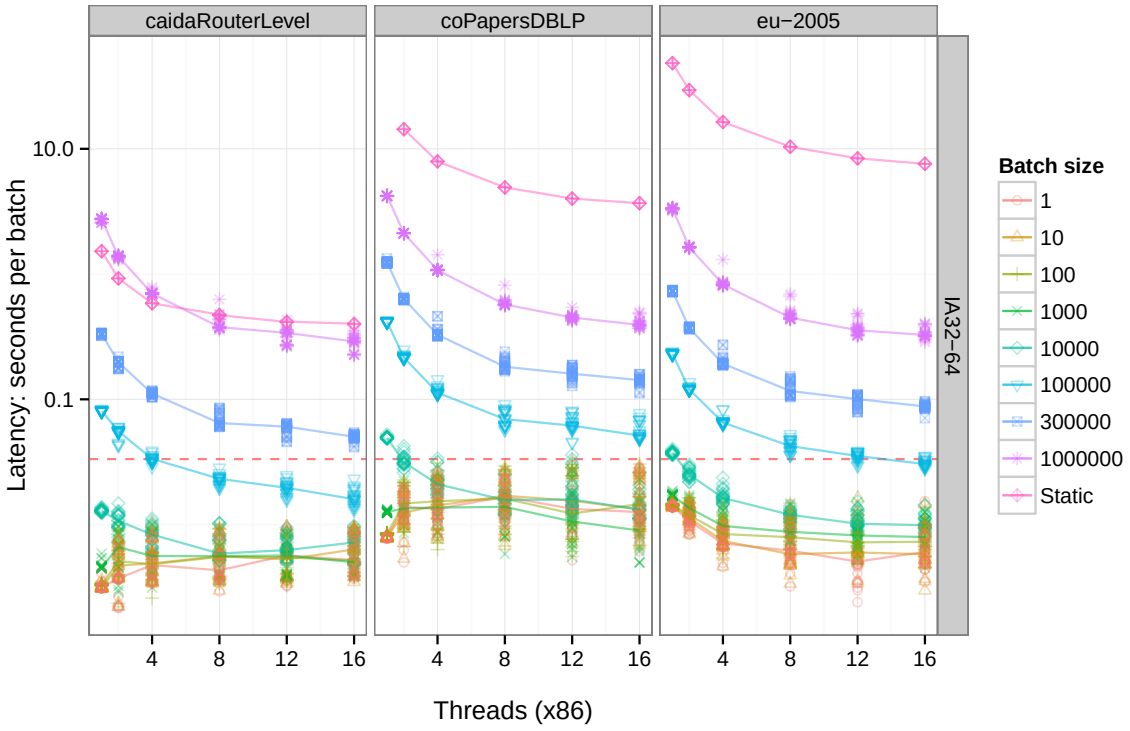

Fig. 2. Latency (seconds per batch) by threads for each test graph and batch size per platform. The solid line connects the median points. The best static algorithm execution time on the original graph is shown for comparison. The horizontal dashed line represents 30 frames per second, or real-time animation speed. Note that the large batch size rewrites almost all of caidaRouterLevel.

Many techniques are similar, but their physically-based graphs often have sufficiently different structure from social networks that different performance optimizations and parallel methods apply.

Our previous work [6], [7], [8] established and extended the first parallel agglomerative algorithm for community detection and provided results on the Cray XMT. Prior modularity-maximizing algorithms sequentially maintain and update priority queues [18], and we replace the queue with a weighted graph matching. Separately, Fagginger Auer and Bisseling developed a similar modularity-optimizing clustering algorithm [19] targeting GPUs.

Gehweiler and Meyerhenke [20] proposed a distributed diffusive heuristic for implicit modularity-based graph clustering. A diffusive heuristic could be adapted for refining a community mapping. Applying refinement after each batch would adapt any such algorithm to streaming scenarios. Refinement needs targeted to the graph changes for high performance, however. Work on sequential multilevel agglomerative algorithms like [21] with a focus on edge scoring and local refinement also could be adapted to streaming settings.

VI. Observations and Directions

Our streaming community re-agglomeration algorithm achieves high aggregate performance and low-latency updates (although not simultaneously) by working on a far smaller problem than global community detection. The reduced problem size limits parallel scalability of re-agglomeration but still performs better than recomputing with a state-of-the-art scalable static community detection code.

Further work needs to incorporate community quality into the trade-off between aggregate performance and latency. On initial inspection, community quality as measured by modularity

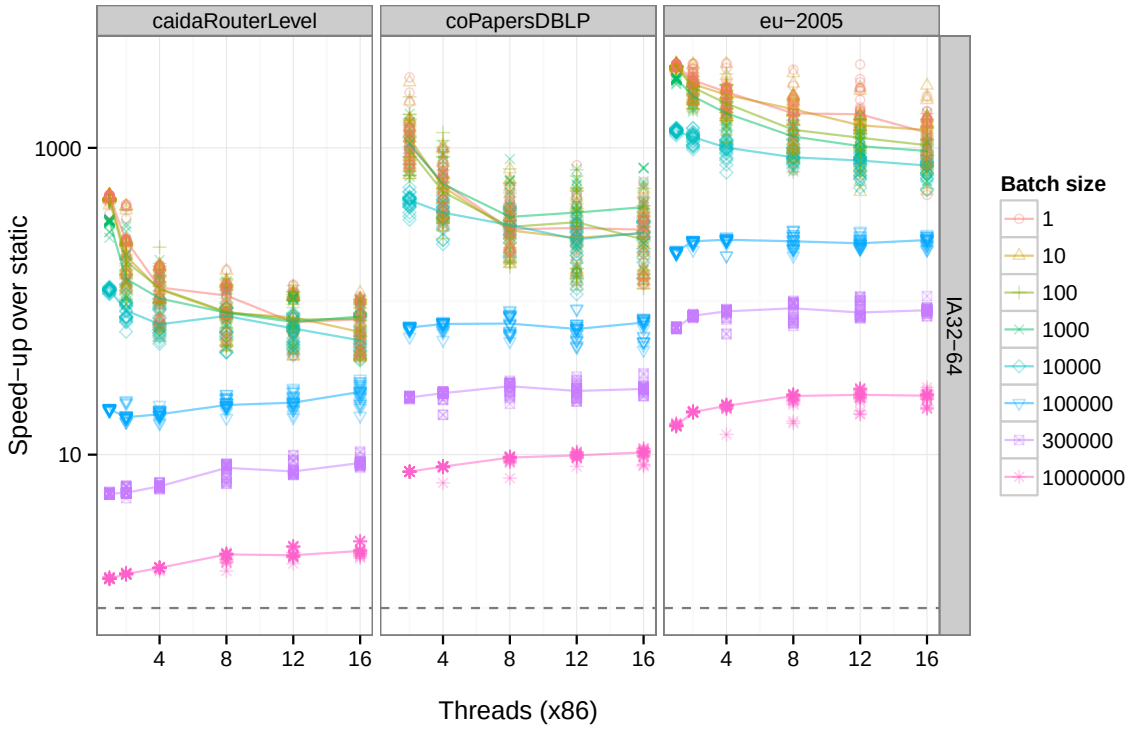

Fig. 3. Speed-up over static recomputation by threads for each test graph and batch size per platform. The solid line connects the median points. The static, parallel algorithm scales very well, but the smaller data set with incremental updates remains faster.

changes very little. Longer-scale experiments are needed. Improvements may require repeated re-agglomeration, cascading changes beyond the initial affected vertex set ΔV. Refining communities according to the metric also may prove interesting not only for quality but also to remove the initial global decomposition. However, refinement requires investigating specific community metrics and is not as agnostic as our current approach. Tracking component changes is not necessary for optimizing many community metrics but is important in practice. We are extending STING to support combining our existing component tracker with community monitoring.

Acknowledgments

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT) and the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program. We thank PNNL and the Swiss National Supercomputing Centre for providing access to Cray XMT systems and Oracle for the Intel-based server. We also thank graduate students Rob McColl and David Ediger for continuing maintenance of the STING software framework.

The work depicted in this paper was partially sponsored by Defense Advanced Research Projects Agency (DARPA) under agreement #HR0011-13-2-0001. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred. Distribution Statement A: "Approved for public release; distribution is unlimited."

References

[1] M. Rios, "Euro 2012 recap," 2012, http://blog.twitter.com/2012/07/euro-2012-recap.html.

[2] Y. Altshuler, W. Pan, and A. Pentland, "Trends prediction using social diffusion models," in *Social Computing, Behavioral - Cultural Modeling and Prediction*, ser. Lecture Notes in Computer Science, S. Yang, A. Greenberg, and M. Endsley, Eds. Springer Berlin Heidelberg, 2012, vol. 7227, pp. 97–104. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29047-3_12

[3] D. Ediger, J. Riedy, D. A. Bader, and H. Meyerhenke, "Computational graph analytics for massive streaming data," in *Large Scale Network-Centric Computing Systems*, ser. Parallel and Distributed Computing, H. Sarbaziazad and A. Zomaya, Eds. Wiley, Jul. 2013, ch. 25, (to appear).

[4] D. Ediger, E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*, Anchorage, Alaska, May 2011.

[5] D. Ediger, K. Jiang, E. J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*, Atlanta, Georgia, Apr. 2010.

[6] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," in *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics*, Torun, Poland, Sep. 2011.

[7] E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Scalable multi-threaded community detection in social networks," in *Workshop on Multithreaded Architectures and Applications (MTAAP)*, Shanghai, China, May 2012.

[8] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, "Parallel community detection for massive graphs," 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, Atlanta, GA, Tech. Rep., Feb. 2012. [Online]. Available: http://www.cc.gatech.edu/dimacs10/papers/[15]-dimacs10-community-detection.pdf

[9] D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Competition rules and objective functions for the 10th DIMACS Implementation Challenge on graph partitioning and graph clustering," Sep. 2011, http://www.cc.gatech.edu/dimacs10/data/dimacs10-rules.pdf.

[10] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality," in *10th Workshop on Algorithm Engineering and Experimentation*. San Francisco: SIAM, 2008, pp. 90–108.

[11] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[12] N. Nguyen, T. Dinh, Y. Xuan, and M. Thai, "Adaptive algorithms for detecting community structure in dynamic social networks," in *INFOCOM, 2011 Proceedings IEEE*, april 2011, pp. 2282 –2290.

[13] R. Bourqui, F. Gilbert, P. Simonetto, F. Zaidi, U. Sharan, and F. Jourdan, "Detecting structural changes and command hierarchies in dynamic social networks," in *Social Network Analysis and Mining, 2009. ASONAM '09. International Conference on Advances in*, july 2009, pp. 83 –88.

[14] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75 – 174, 2010.

[15] J. Hopcroft, O. Khan, B. Kulis, and B. Selman, "Tracking evolving communities in large linked networks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 101, no. Suppl 1, pp. 5249–5253, 2004. [Online]. Available: http://www.pnas.org/content/101/suppl.1/5249.abstract

[16] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–97, 2002.

[17] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, "An Overview of Trilinos," Sandia National Laboratories, Tech. Rep. SAND2003-2927, 2003.

[18] A. Clauset, M. Newman, and C. Moore, "Finding community structure in very large networks," *Physical Review E*, vol. 70, no. 6, p. 66111, 2004.

[19] B. O. Fagginger Auer and R. H. Bisseling, "Graph coarsening and clustering on the GPU," 10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering, Atlanta, GA, Tech. Rep., Feb. 2012. [Online]. Available: http://www.cc.gatech.edu/dimacs10/papers/[16]-gpucluster.pdf

[20] J. Gehweiler and H. Meyerhenke, "A distributed diffusive heuristic for clustering a virtual P2P supercomputer," in *Proc. 7th High-Performance Grid Computing Workshop (HGCW'10) in conjunction with 24th Intl. Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE Computer Society, 2010.

[21] A. Noack and R. Rotta, "Multi-level algorithms for modularity clustering," in *Experimental Algorithms*, ser. Lecture Notes in Computer Science, J. Vahrenhold, Ed. Springer Berlin / Heidelberg, 2009, vol. 5526, pp. 257–268.