ANALYSIS OF STREAMING SOCIAL NETWORKS AND GRAPHS ON MULTICORE ARCHITECTURES

Jason Riedy[1], *Henning Meyerhenke*[1,2], *David A. Bader*[1], *David Ediger*[1], *and Timothy G. Mattson*[3]

[1] Georgia Institute of Technology, Atlanta, GA 30332
[2] Karlsruhe Institute of Technology, Institute for Theoretical Informatics, D-76128 Karlsruhe, Germany
[3] Intel Corporation, Microprocessor and Programming Research Lab, DuPont, WA 97124

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. We introduce a framework, STING (Spatio-Temporal Interaction Networks and Graphs), and evaluate its performance on multicore, multisocket Intel®-based platforms. STING achieves rates of around 100 000 edge updates per second on large, dynamic graphs with a single, general data structure. We achieve speedups of up to 1000× over parallel static computation, improve monitoring a dynamic graph's connected components, and show an exact algorithm for maintaining local clustering coefficients performs better on Intel-based platforms than our earlier approximate algorithm.

***Index Terms*—** social network analysis, streaming data, graph analysis, parallel processing

1. INTRODUCTION

Applications ranging from business intelligence and finance to computational biology and computer security are generating data at a massive rate. Social networks such as those from Facebook and Twitter boast hundreds of millions of users posting billions of interactions per month. The NYSE processes over four billion traded shares per day. The data generated are not the dense arrays of signal processing's traditional focus but data connecting multiple entities with multiple attributes. This graph-structured data already challenges high-performance analysis.

The graph representing the data often is *scale-free* [1]. A scale-free graph has low diameter, so connecting paths between any two vertices are short. Many vertices have a small number of neighbors, while a few vertices are connected with a large part of the graph; the degrees follow a power-law distribution. Scale-free graphs lack small separators and present unique challenges for parallel algorithms. The degree distribution also creates imbalance in workload when scheduling vertices among processors. Incorporating dynamic information itself poses new challenges to algorithm design and implementation.

Current large graph analysis tools like Pajek [2] are designed primarily for static graphs. For dynamic inputs these tools assume the properties to change slowly relative to execution time. This assumption does not apply to emerging applications, driving a need for more dynamic analysis. We address these challenges with new algorithmic approaches and new data structures targeting readily available Intel-based platforms. Computing incremental updates to the dynamic graph with batches of updates from the streaming data provides opportunities to improve parallel algorithm performance. We use a new data structure for analyzing complex graphs and networks with possibly billions of vertices that accumulates as much of the recent graph data as possible in main memory. Once the reserved memory is full, older or uninteresting edges are aged off and removed. We update analytical kernels after each batch of edge insertions or deletions and attempt to detect significant changes in the corresponding metrics. We refer to this new approach as *massive streaming data analytics*.

Our system, STING (Spatio-Temporal Interaction Networks and Graphs), achieves real-world rates of 100 000 edge updates per second for monitoring a vertex-local property, clustering coefficients, and 70 000 edge updates per second for monitoring a global property, the connected components, on artificial graphs with 4 million vertices and 67 million edges on Intel®-based platforms.

2. FRAMEWORK FOR STREAMING GRAPH ANALYSIS

Our STING framework consists of a graph data structure, STINGER (STING Extensible Representation) [3], that supports rapid updates and parallel queries as well as a general algorithmic structure for applying analysis kernels to the dynamic data stream. STING maintains a single, large graph image in memory to be used by multiple analysis kernels. Changes accumulate within the single image; individual analysis kernels maintain history and summary information when necessary.

STING collects edge insertions and deletions into batches. These batches amortize parallel overhead across many individual updates and hence improve parallel efficiency. The trade-off lies in responsiveness. Large batch sizes will update analysis metrics less frequently even while supporting more aggregate updates per second. Interactions within a batch, as when an edge is both added and removed, are reconciled before applying the changes to the graph representation to take advantage of any locality within the batch itself.

The STINGER data structure [3] maintains a sparse adjacency matrix representation of the graph. The neighbors of a vertex are stored in a linked list of dense arrays permitting both dynamic growth and fast iteration. STINGER maintains the graph structure (neighbors, weights) as well as meta-data like edge semantic types and time stamps. Analysis kernels access both the batch and the STINGER graph structure without explicit locking. The edges are indexed to permit fast iteration across neighbor lists and edge types.

3. CLUSTERING COEFFICIENTS AND COMPONENTS

We investigate two analysis kernels on undirected, unweighted graphs: local clustering coefficients and the global component labeling. Local clustering coefficients monitor the density of triangles surrounding each vertex and are related to the "small-world" property of social networks [4]. Larger clustering coefficients suggest formation of communities. Monitoring the global component labeling provides information to other kernels like path searching and sampling methods.

3.1. Clustering coefficients

We adopt the terminology of [4]. A triplet is an ordered set of three vertices, (i, v, j), where v is considered the focal point and there are undirected edges $\langle i, v\rangle$ and $\langle v, j\rangle$. An open triplet is defined as three vertices in which only the required two are connected. A closed triplet is defined as three vertices in which there are three edges. A triangle is made up of three closed triplets, one for each vertex of the triangle.

The local clustering coefficient of vertex v is

$$\begin{aligned} C_v &= \frac{\text{number of closed triplets centered around } v}{\text{number of triplets centered around } v} \\ &= \frac{T_v}{d_v(d_v-1)}, \end{aligned}$$

where T_v is the closed triplet count around v and d_v is the degree of v (number of adjacent vertices).

The degrees d_v are maintained in the STINGER data structure. In [5], the authors present three algorithms for maintaining the triangle count T_v. Given a modified edge $\langle u, v\rangle$, the *brute force* algorithm iterates over the neighbor lists of u and v and checks for an intersection in $O(d_u d_v)$ time. An improved algorithm, *sorted list*, sorts the shortest neighboring edge list and searches for an intersection with bisection in $O((d_u + d_v)\log d_u)$ time. An approximate *Bloom filter* algorithm summarizes one edge list using a lossy bit array, reducing the operation complexity to $O(d_u + d_v)$ in exchange for possibly over-estimating the number of triangles.

The counts for each affected vertex in a batch of edge changes are updated in parallel. There is a limited amount of multi-level parallelism available within the brute force algorithm on high-degree vertices, but we do not exploit that here. On the Intel-based platforms discussed in Section 4, the exact sorted list algorithm out-performed the other algorithms overall.

3.2. Component labeling

In an undirected graph, there exists a path between any two vertices within the same connected component and no paths between between vertices in different connected components. Knowing the connected components containing each vertex is vital for search algorithms, sampling and approximation algorithms, and many other applications. Maintaining the array that labels each vertex with the connected component containing that vertex may require global information. Whether a single deletion splits a connected component depends on existence of any other path connecting the deleted edge endpoints.

In scale-free graphs such as social networks, however, many edge insertions and deletions lie entirely within a single, large component. The authors' updating algorithm in [6] resolves edge insertions immediately, rules out some edge deletions through a limited search, and delays the remaining deletions for multiple batches before running a parallel static connected components algorithm [7] on the accumulated graph. An edge insertion looks up the component of each endpoint. If the edge straddles two components, the smaller component is relabeled and merged into the larger. This does not require checking anything within the original graph, only the component labels. Edges cannot cross components, so deletions only occur within a single component. Deletions may cleave the component into two pieces but rarely do. After removing the deleted edges from the STINGER representation, the affected edge endpoints are checked in the same manner as when counting triangles for clustering coefficients. If the vertices remain connected, the deletions have no effect. Otherwise, the component is marked and queued for later testing by the static algorithm. The static algorithm is applied only when the pending deletion queue becomes so large as to affect other results. This local search from [6] marks almost 90% of deletions as having no effect in our tests.

We now discuss an improved heuristic that rules out far more deletions with far less memory traffic. The static connected components algorithm [7] forms a spanning tree for each connected component as a by-product. A deletion can cleave a component only if the deleted edge is an edge in that spanning tree. If the deleted edge is in the tree, the endpoint separated from the root checks its neighbors and tries to repair the spanning tree locally. Only when all these tests fail are

Model	μ-arch	Clock	L3 size	Sockets	Cores
X5570	Nehalem-EP	2.93GHz	8 MiB	2	4
E7-8870	Westmere-EX	2.40GHz	30 MiB	4	10

Table 1. Intel® Xeon® Processor X5570 and E7-8870 platforms both of which support Intel HyperThreading technology with two threads

neighbors checked for connectivity. These tighter tests rule out 99.7% of deletions as having no effect in our test data, drastically decreasing the frequency of static checks while also improving performance through reduced memory traffic.

4. EXPERIMENTS

Our experiments measure performance on an artificial graph generated by the widely-used R-MAT[8] model derived by sampling from a Kronecker product. The R-MAT generator produces scale-free graphs similar to social networks. The vertices' degrees follow a power-law distribution with a few very high-degree vertices and many vertices with small degree. We generate an initial edge list of $16 \times 2^{22} \approx 67$ million edges connecting $2^{22} \approx 4$ million vertices. We use the R-MAT probability parameters $a = 0.55$, $b = 0.1$, $c = 0.1$, and $d = 0.25$ and perturb the parameters by $\pm 5\%$ at each recursion.

We generate an input stream of edge actions (both insertions and deletions) by the same R-MAT distribution and divide this stream into batches. Each generated edge action is an insertion, and insertions are selected with probability $1/16$ to be entered into the deletion queue. Before generating the actions, we apply the same selection to initial edges and enter them into the deletion queue with probability $1/16$. We only delete edges that exist; the framework ignores deletions that do not correspond to edges in the STINGER structure. The batches are constructed by selecting the next insertion with probability $15/16$ or a deletion from the queue with probability $1/16$. Each experiment is run over ten batches of changes on the same initial graph.

Table 1 lists the Intel-based test platform processor characteristics. All are running Red Hat® Enterprise GNU/Linux® 6.1 and all codes are built with gcc 4.6.1. Each platform's DDR3 memory is fully banked and running at 1066MHz. The memory is distributed across sockets, providing non-uniform access (NUMA). Using only memory through one socket added a 5%–100% penalty over striping pages across sockets. Future work will investigate more advanced placement than striping, but all results presented here use the `numactl` utility to stripe allocated pages uniformly across sockets.

Ultimately, we are interested in maximizing the supported edge updates per second while maintaining responsiveness. Batches of many millions of edge actions may reach a million updates per second, but not all applications can wait a second between metric updates. We consider batch sizes of 100, 1000, and 10 000. Figure 1 shows that clustering coefficient performance reaches a point of diminishing returns between batches of 1000 and 10 000 edge actions.

Also, we consider two kinds of speed-up. One is from parallelization throughout our implementations. Another is the speed-up from dynamic updates over static recomputation on snapshots. For a batch size of 1000, Table 2 shows both the achieved edge updates per second for our dynamic methods and the speed-up of that rate over the edges per second achieved by re-running static analysis on the graph snapshot.

Architecture	Algorithm	Min.	Median	Max
E78870	Brute force	17062	31038	41716
		77×	141×	190×
	Bloom filter	57005	84418	97181
		257×	379×	442×
	Sorted list	84963	97079	118913
		370×	437×	541×
	Components	73650	74430	75050
		79×	80×	82×
X5570	Brute force	9881	16509	21057
		153×	256×	326×
	Bloom filter	95755	113835	123203
		1482×	1762×	1907×
	Sorted list	104669	125667	129627
		1620×	1945×	2007×
	Components	10360	39580	88610
		28×	52×	233×

Table 2. Achieved edge updates per second and speed-up over parallel static recomputation for each clustering coefficient algorithm and also the connected components for a batch size of 1000 actions and the maximum number of hardware-supported threads. The speed-ups over parallel static recomputation are variable but substantial. Note that the additional connected component heuristics reduce memory access and greatly improve performance on the dual-socket Intel® Xeon® processor X5570.

5. CONCLUSIONS

Using Intel-based platforms, our STING system supports rates of updates expected with actual applications over existing social networks for both vertex-local and global graph properties. STING can track clustering coefficients at rates exceeding 100 000 updates per second with batches small enough to respond 100 times per second. STING tracks component labels at over 70 000 updates per second and updates the component labels 70 times per second.

Overall, batching edge updates provides useful parallel computational opportunities on Intel-based platforms even with small enough batches to react to changes quickly. Reducing graph searches and memory accesses with better heuris-

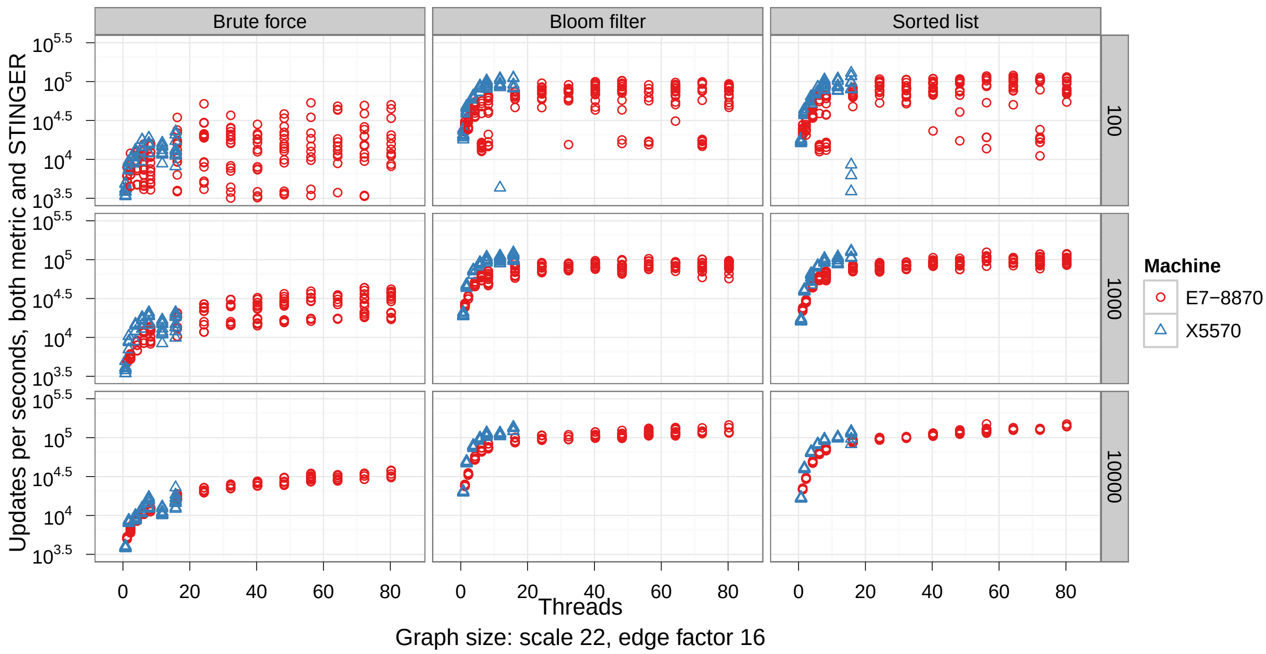

Fig. 1. Updates per second for updating the metric, local clustering coefficients, and the STINGER representation for different batch sizes and platforms. The four 2.4 GHz Intel® Xeon® processor E7-8870's larger number of memory interfaces brings performance beyond the dual 2.93 GHz Intel® Xeon® processor X5570 with a sufficiently large batch size.

tics while monitoring connected components increases performance on systems with fewer pathways to memory. Also, performing the exact sorted-list intersection on Intel-based platforms performs better than the approximate Bloom filter.

6. ACKNOWLEDGMENTS

This work was supported in part by the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program. We thank Uzi Vishkin and the anonymous reviewers for the very helpful comments.

7. REFERENCES

[1] M.E.J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.

[2] V. Batagelj and A. Mrvar, "Pajek – program for large network analysis," *Connections*, vol. 21, no. 2, pp. 47–57, 1998.

[3] D. A. Bader, J. Berry, A. Amos-Binks, D. Chavarría-Miranda, C. Hastings, K. Madduri, and S. C. Poulos, "STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation," Tech. Rep., Georgia Institute of Technology, 2009.

[4] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, Jun 1998.

[5] D. Ediger, K. Jiang, E. J. Riedy, and D. A. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Proceedings of the Workshop on Multithreaded Architectures and Applications (MTAAP'10)*, Apr. 2010.

[6] D. Ediger, E. J. Riedy, and D. A. Bader, "Tracking structure of streaming social networks," in *Proceedings of the Workshop on Multithreaded Architectures and Applications (MTAAP'11)*, May 2011.

[7] Y. Shiloach and U. Vishkin, "An $O(\log n)$ parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.

[8] 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.