Massive Streaming Data Analytics: A graph-based approach

Analyzing massive streaming graphs efficiently requires new algorithms, data structures, and computing platforms.

By Jason Riedy and David A. Bader

DOI: 10.1145/2425676.2425689

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 that the sheer volume of raw data already surpasses our analysis capabilities and keeps growing.

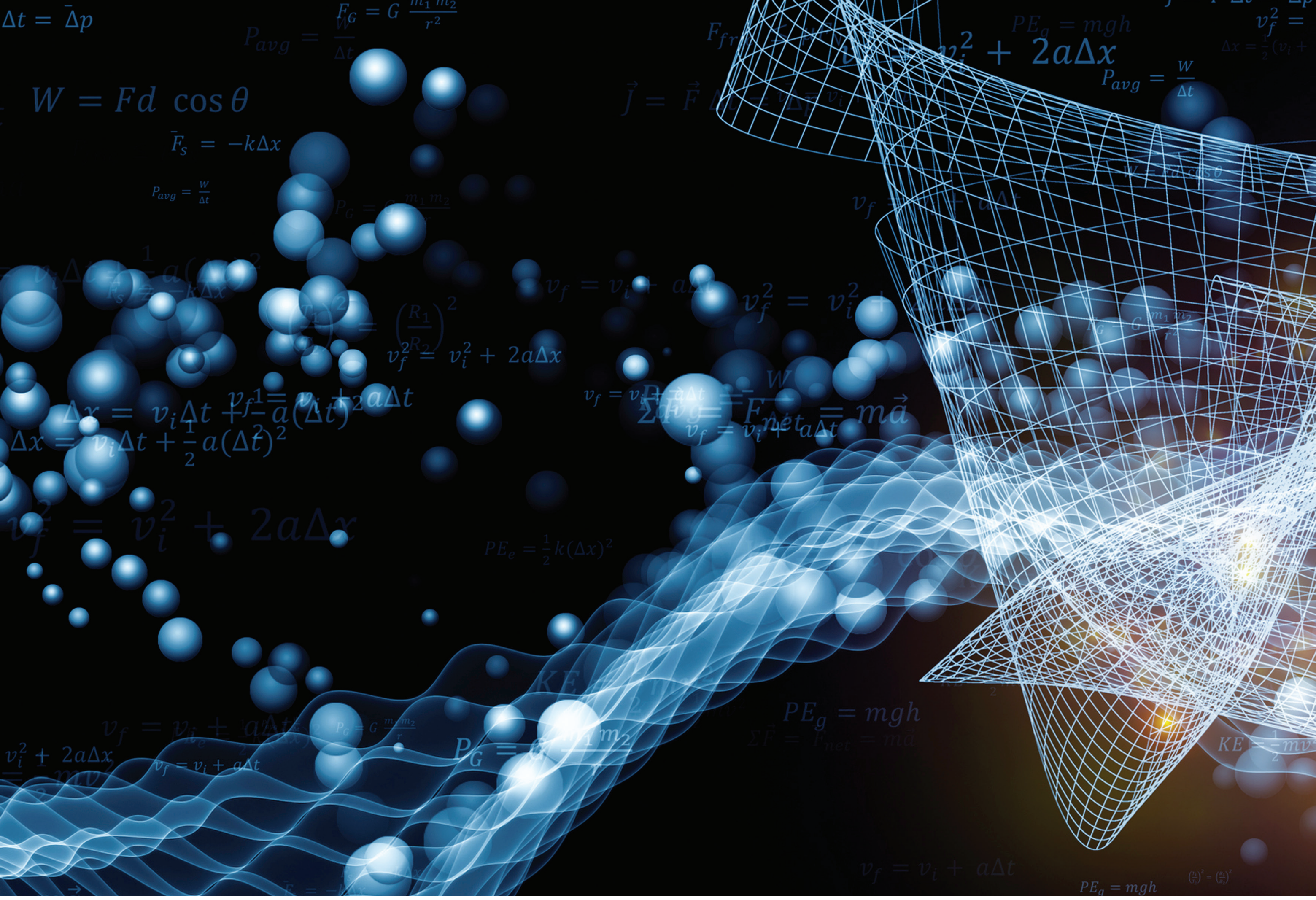
But how do we turn this data into insight? Our goal is to guide decisions with accurate information derived from graph-structured data. Graphs—collections of vertices connected by edges—provide some structure to the mess that is big data. These graphs are semantic, with types given to edges and vertices, although not necessarily the well-curated semantic graphs of the Semantic Web vision. The graphs in our case are more ad-hoc, but still contain vast amounts of useful information.

In many cases, the data can be decomposed into a primary signal, the current trends, plus noise that may hold the next signal. The primary signal often is easy to find. Take a social messaging site like Twitter, for example. Counting words or phrases that appear in tweets allows identifying popular topics and provides insight into trends, as in the Euro 2012 final match between Spain and Italy. At the final goal, Twitter processed more than 15,000 messages per second [1], requiring great scalability of their infrastructure. In some cases, however, the main signal may deceive. After the 2010 Haiti earthquake, initial analysis suggested a correlation between building damage and the number of text messages sent. However, after factoring in the context of the messages, the counts had a slight negative correlation with damage [2].

Making future predictions based on data is even more difficult. Most of the noise observed, like forwarding a short-lived joke shared among a lunch group, does not have an impact. Sometimes, as in computer network security, noise in the data produces effects far beyond its apparent magnitude. Determining what noise is considered meaningful in a fast and efficient way is a challenge. The current state-of-the-art tools for graph analysis are often based on MapReduce-like systems, sometimes with bulk-synchronous parallel (BSP) interfaces like Google's Pregel. These systems extract graphs from large datasets and perform heavy pre-processing work for each analysis.

Often, the rapid growth of data in many applications requires streaming analysis to produce results with sufficiently fast turn-around time, which means that data must be analyzed as it arrives. In the massive streaming data analytics model, the graph itself is viewed as an infinite stream of edge insertions, deletions, and updates. Currently, systems that handle streaming data support only aggregate information like counts or averages. Streaming analysis also opens new avenues to parallel processing.

There are many emerging applications, described in the next section,



which require more complex graph analysis. The computations involved in complex graph analysis (which we refer to as "kernels") are notoriously difficult to parallelize; the kernels require very little computation and often are highly synchronized. Memory latency and bandwidth throughout the memory hierarchy typically limits these kernels' performance. Operations for updating or changing massive graphs are often scattered and require much less synchronization. Working with batches of changes provides more opportunities for high performance. The core of this article outlines our approach and summarizes our framework, STING, for analyzing spatio-temporal interaction networks and graphs.

EMERGING APPLICATIONS

A few important applications of massive streaming graph analysis are emerging; they are discussed as follows.

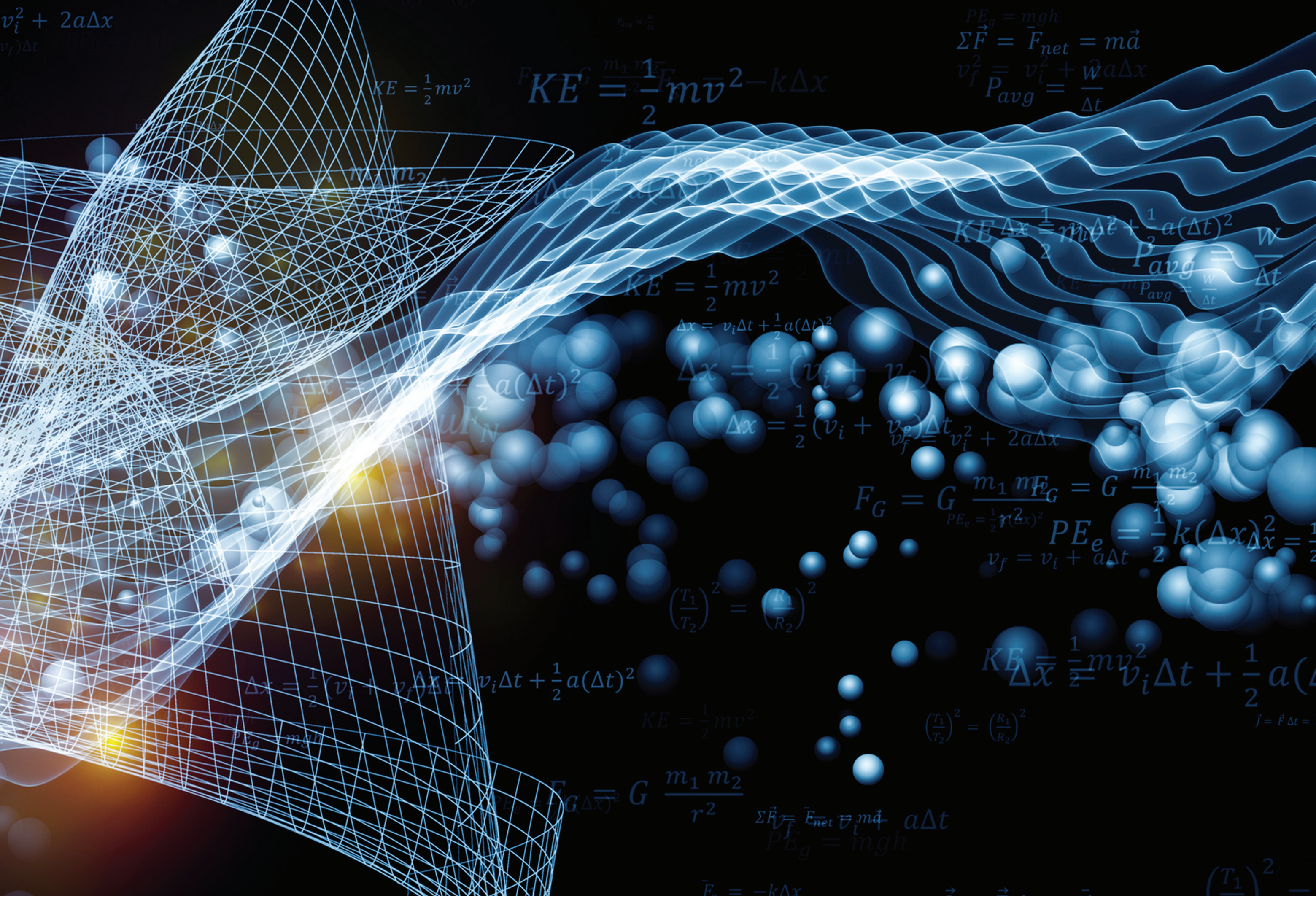
Online social networks. Business intelligence company DOMO combined sources to estimate that in one minute on average, Facebook users share more than 650, 000 pieces of content, Twitter users post more than 100,000 messages, and Tumblr users publish almost 30,000 posts [3]. Some of this content is highly valuable, showing upcoming news or trends. Other pieces are useless spam to be filtered away. Immediate feedback of relative value or "spamminess" to authors, advertisers, and companies could prevent false positives and boost overall quality.

Everyone recognizes that the sheer volume of raw data already surpasses our analysis capabilities and keeps growing. But how do we turn this data into insight?

Public policy and crisis management. Initiatives like the United Nation's Global Pulse and the U.S. government's Data.gov recognize the importance of data and real-time analysis to public policy from planning to crisis response. Spanning across domains including health, energy, and commerce, this data is highly heterogeneous but linked through people, places, and organizations. Researchers are monitoring elections through social media [4], analyzing the effectiveness of public health alerts [5], and helping journalists analyze collections of documents and records [6].

Context is very important throughout this data. As mentioned earlier, initial analysis of simply counting text messages sent within grid cells during the 2010 Haiti earthquake appeared to track building damage. But that wasn't the case [2]. It's clear that insight requires deeper analysis to account for correlations and interactions.

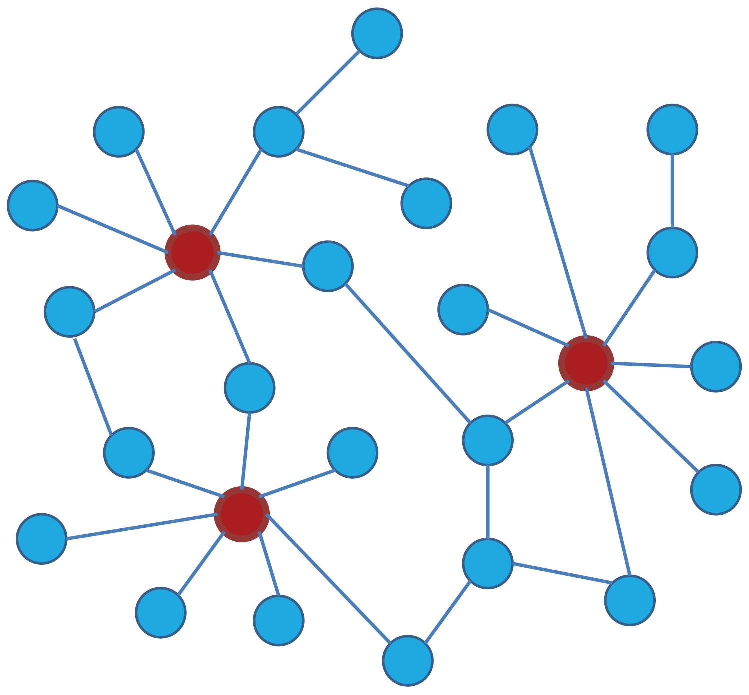
Network security. Many people carry or use multiple devices to connect to



the same online services. They assume a level of security while simultaneously expecting ease of use and adding the confounding factor of multiple access sources. Assuring security requires rapid response to access and posting requests. This is only one variant of the general network security problem facing service providers. Separating authorized access from dangerous access while not alienating users requires rapid response given the context of user identity, access method, and service state. This exists for services ranging from online social networks to distributed file systems. Users need to be authorized rapidly given varied contexts and policies.

Health informatics. The wealth of long-term medical information collected by institutions like the Mayo Clinic is slowly being digitized. This data adds new information about family histories to a network of diagnostic similarities. The medical literature produces more relationships between symptoms, causes, and other factors. Doctors, responders, administrators, and policy makers need different, up-to-date

Figure 1. A small example of a scale-free graph. In larger variants of such graphs, the degree (number of edges) of different vertices can vary by orders of magnitude.



summaries of this disparate data. The information needs to be recent but also respectful of privacy concerns between separate individuals and data stores. The change is relatively slow, but health informatics is leading in combining disparate data across data with strong privacy boundaries.

Financial analysis and monitoring. Computer-driven, high-frequency trading systems trade stocks and commodities at rates far faster than human regulators can monitor. The NYSE processes more than four billion traded shares per day [7]. Investors want rapid, focused analysis and response combining not only the current trades, but also auxiliary information like financial filings and news stories. Regulators require monitoring the overall health of varied, interconnected markets to identify anomalies. Current regulatory controls do not handle interconnections in the "fragmented and fragile" marketplace [8], leading to instability like the flash crash that occurred on May 6, 2012.

NEW ROUTES TO PARALLEL PERFORMANCE

In the multi-everything era, codes performing massive data analysis must expose parallelism wherever possible. Graph codes are notoriously difficult to parallelize efficiently. Working incrementally using a streaming data model offers new opportunities to exploit parallelism.

For now, we consider parallel platforms providing a shared, global address space to programs. Such a space can be implemented over physically distributed memory, as in the Cray XMT, or within a single node, as in Intel-based and other common platforms. We assume a small collection of atomic operations is available to avoid explicit locking. We also assume the processors are highly multi-threaded to tolerate some memory latency. Data structures and algorithms are arranged to permit both the autonomous multi-threading available in most processors and coherent multi-threading as in GPUs.

Graph codes are notoriously difficult to parallelize efficiently. Working incrementally using a streaming data model offers new opportunities to exploit parallelism.

Classic parallel analysis of static graphs that scale to massive data almost always builds trees through some breadth-first search technique. These methods are useful for initial analysis but sacrifice too much performance. Changing graphs are considered in traditional dynamic algorithms work. These algorithms tailor data structures to individual analysis kernels, requiring maintenance of many different massive structures for different kernels. When data size is a problem, multiplying that size often is prohibitive. Other traditional approaches like streaming algorithms assume much less information is maintained per kernel, but that the entire data is available for one or multiple passes. We would prefer focusing only on the data that has changed from one state to the next.

Our approach borrows ideas from each of these but applies them for massive, persistent data. By analyzing batches of the data's changes, we can enhance the parallelism and trade-off between aggregate performance and analysis latency.

STING, A FRAMEWORK FOR STREAMING, GRAPH-STRUCTURED DATA ANALYSIS

To tackle graph analysis in the face of big data, we are developing the STING (Spatio-Temporal Interaction Networks and Graphs) framework [9]. 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.

Kernels currently available in STING include: (1) connected component monitoring, (2) clustering coefficient tracking, (3) STINGER microbenchmarks, and (4) a few example static analysis kernels. Streaming community detection and seed set expansion are scheduled to be included soon.

The STINGER data structure. Our STINGER data structure provides easy and efficient methods to both insert and remove edges from a scale-free graph (a graph whose degree distribution follows a power law), while also permitting fast querying of neighbor information and other metadata about vertices and edges. The STINGER specification does not specify consistency; the programmer must assume that the graph can change underneath the application.

The STINGER data structure resembles an adjacency list graph structure. Neighbors of each vertex are stored in a linked list of "edge blocks." Implementing lists using a list of arrays is common in many systems. The STINGER data structure implements a hybrid between packed arrays and linked lists, allowing support of both rapid updates and multi-threaded iteration. The dense blocks are processed easily with vectors or coherent threads, while the separate blocks provide work for more general but rapidly spawned threads.

Each directed edge stored in STINGER consists of a source vertex, a destination vertex, a weight, a last-modified timestamp, an initial insertion timestamp, and an arbitrary semantic type stored as an integer. Weights within a single data store can be defined either as integers or floating-point data. Undirected graphs are represented with a pair of directed edges. Applications may use the semantic types to distinguish between different kinds of edges (as in an ontology), provide keys into external databases, support permissions, as well as other capabilities. The two timestamps provide a sense of time to analysis kernels without requiring strict time keeping. Application codes can traverse the graph structure directly through C macros or use a safer interface. The safer interface provides a function that gathers the edge list from the various blocks and returns it to the application in an array. This has the double benefit of isolating the application from changes in the data structure as well as simplifying conversion of existing static codes that utilize the popular compressed sparse row format to work with STINGER. On the Cray XMT, we often observe that this safer "copy out" strategy provides identical or better performance for static codes using STINGER compared to compressed sparse row representations.

Often, the rapid growth of data in many applications requires streaming analysis to produce results with sufficiently fast turn-around time, which means that data must be analyzed as it arrives.

The STING pipeline. The currently distributed example STING kernels are self-contained. The two streaming example kernels, which compute cluster coefficients and maintain connected components, work with a straight-forward pipeline. Updates to the graph (edge insertions and removals) are processed in batches, where each batch is pre-processed to reduce redundant work. To update the graph, STING modifies the STINGER data structure and provides the kernels with the batch and the updated data structure.

To pre-process a batch, undirected edges are duplicated into a pair of pseudo-directed edges stored in STINGER. The batch is then sorted and collapsed in parallel, combining multiple actions on each source vertex. In these examples, multiple insertions increment the edge's integer weight but otherwise count as a single edge. Insertions and removals are handled in order of appearance within the input batch of updates. Initial removals only can take effect if the edge is in the graph; this case is handled when modifying the graph after batch pre-processing. In general, different kernels need different access to the graph data. Future versions of STING will support multiple models. One model is not to coordinate with the separately running kernels at all. The kernels continually scan the data structure. This is appropriate for low-performance data auditing tasks. Another model is to notify the kernels of the changes, often only the affected vertices, either before or after updating the data structure depending on the kernel. STINGER maintenance performance is fast enough that this model is sufficient in most practical applications. Kernels needing extreme performance at the cost of more complex programming will be provided the changes simultaneously with the data structure maintenance. This will be useful for approximation algorithms and others that assume a certain level of noise in the data. Although our STINGER data structure remains consistent, the represented graph can suffer from undirected edges only existing in one stored direction.

EXAMPLE ALGORITHM: MONITORING CONNECTED COMPONENTS

Scale-free graphs like social networks often have one massive connected component, where a path exists between all pairs of vertices, and many smaller outlying components. Knowing when the small components merge or when a group splits from the massive component is useful both for general information and to guide other algorithms (e.g., sampling for centrality approximation). The STING distribution includes an analysis kernel that maintains a mapping of vertices to connected components subject to batches of edge insertions and removals.

The analysis kernel is handed the batch and the graph after updating. With this, the algorithm maintains a component label function c where $c(i)$ provides the component label of vertex i. It also maintains a spanning tree for each component; the root of the tree serves as the component label. The spanning tree is used for handling removals and is a natural by-product of the Shiloach-Vishkin parallel static connected components algorithm.

Our algorithm processes insertions before removals. Insertions are straight-forward, but removals require more effort. Given a component label

Figure 2. *STING* Aggregate performance in updates per second when varying thread counts and batch sizes.

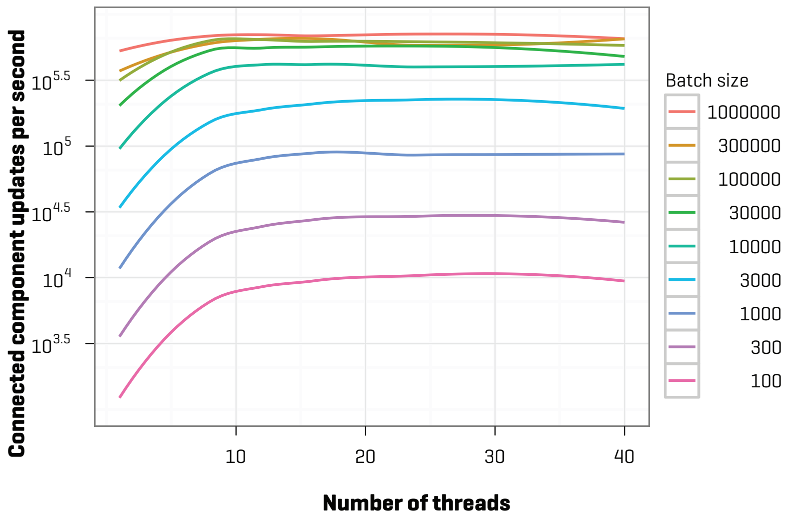
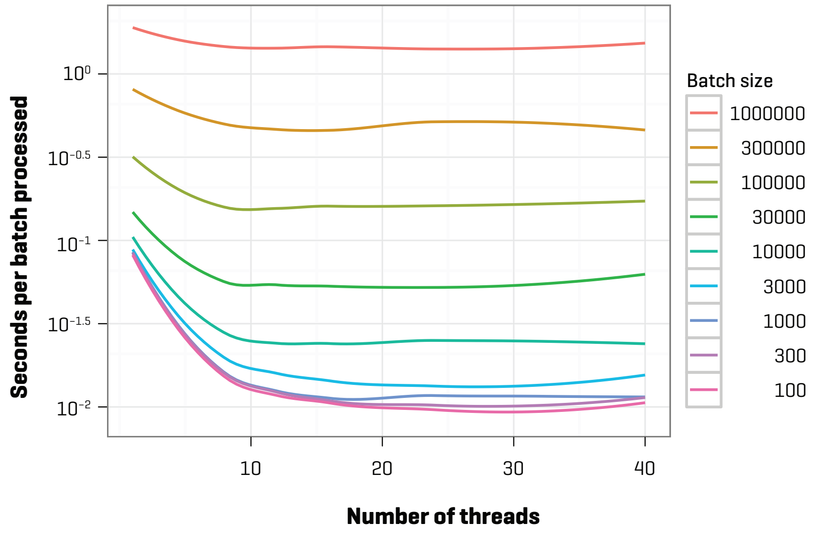


Figure 3. Latency between beginning to analyze the batch and updating the connected components.



function $c(i)$, inserting an undirected edge $\{i,j\}$ into a graph connects components $c(i)$ and $c(j)$. In scale-free graphs, the vast majority of inserted edges are in the same component and $c(i) = c(j)$. Within a batch, these edges produce no component changes and can be ignored. The remaining insertions join components.

We replace the vertices within a batch's inserted edges with their component labels. A parallel counting sort followed by a parallel uniqueness check collapses the batch's insertions into the edges joining previously separate components. We run a parallel, static connected components algorithm on this far smaller graph to identify which components are to be merged, and then we relabel the link the smaller components and link their tree roots to the largest component's root.

Edge removal, as always seems the case, is more complicated. Recomputation through a search is the only known method with sub-quadratic space complexity to determine if the deleted edge has cleaved one component into two. In scale-free graphs, very few deletions will separate components. We rely on heuristics to identify the 99.7 percent of effect-less deletions in our tests.

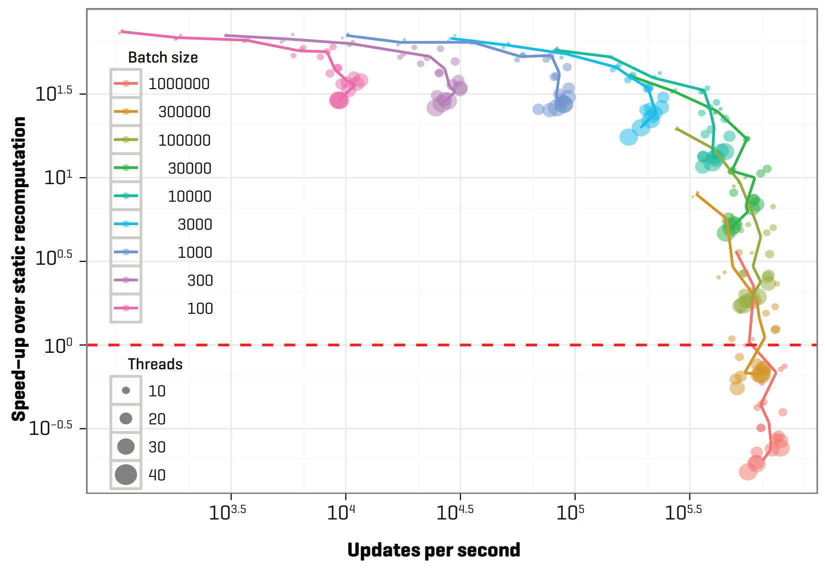
If an edge endpoint has degree one, it is separated into its own component easily. If the deleted edge is not the stored tree edge, the component remains connected. If some neighbor can reach the component's root along existing tree edges, we repair the tree by taking that neighbor as the current vertex's parent. If we cannot reach the root, we save the vertex where the search ended. After all the checks, we attempt to repair the tree by trying a neighbor search from all the unique search-ending vertices. If all these attempts fail, we queue effected vertices until a user-defined threshold and then re-run static connected components. Each of these searches only follows tree edges, limiting memory accesses.

We evaluate performance on an Intel-based server. The server has four 10-core hyperthreading E7-8870 processors at 2.4 GHz with 256 GiB of 1,333 MHz DDR3 memory. We emulate a social network with a relatively small R-MAT [10] generated graph with more than 16.7 million vertices and 88 million edges. R-MAT also generates the edge actions, inserting a possibly duplicate edge with probability of 7/8 or removing a previously inserted edge with a probability of 1/8. Threads are spread across processor sockets first, then cores, and then hyperthreads. Memory is interleaved across sockets by 2 MiB pages using numactl.

For large batches of 30,000 edge actions or more, STING can maintain the connected components at an aggregate rate of over 10^5 updates per second using 20 threads. Figure 2 shows the aggregate performance in updates per second when varying thread counts and batch sizes. Figure 3 translates the performance into the latency between beginning to analyze the batch and updating the connected components. Many applications prefer lower latency to aggregate performance; STING batch size parameters can support a range of latency/performance trade-offs.

The large latency of large batches raises another performance question: How does processing a large batch compare with rerunning the fast static connected components algorithm? Figure 4 provides the speed-up over static recomputation. There is a distinct knee in the curve; batches of 30,000 edge actions provide good speed-up and high aggregate performance with a latency of under 0.32 seconds. Also apparent is that we need not allocate the entire system to a single analysis kernel. The peak performance is achieved with at most 20 threads. We are experimenting to see how the memory and thread resources affect multiple analysis kernels' performance.

Figure 4. Speed-up over static connected components algorithm.



RELATED WORK AND FUTURE DIRECTIONS

Analyzing streaming, graph-structured data combines, and leverages, work in many areas from architecture to systems. The STING approach harnesses current changes in computing architectures. Highly multi-threaded architectures are widely available ranging from the massive yet general multi-threading of the Cray XMT to more modest threading on Intel-based systems and the massive but more coherent threading in general-purpose graphics processing units.

Visualization has long relied on dataflow systems like Data Explorer, Khoros, and other tools to process results as a batched pipeline. Tools like IBM's InfoSphere Streams and Apache's S4 are carrying this model into data analysis as well. The current tools focus on aggregate information like counts but could be extended to support graph structure.

Tools for massive data analysis for graph-structured data, like Google's Pregel and Apache's Hadoop-based Giraph, work on a bulk-synchronous model. Kernels work across the entire data store in large, iterative steps. These systems do not handle streaming data, but do handle important issues like fault-tolerance and truly massive data storage. Other systems like CMU's GraphChi are bringing streaming data into this model.

We see emerging applications using systems combining STING's focus on streaming, semantic data with the massive scale and fault tolerance of these frameworks built as a part of a larger dataflow system on massively multi-threaded architectures.

ACKNOWLEDGMENTS

STING development and porting has been supported in part through the Intel Labs Academic Research Office for the Parallel Algorithms for Non-Numeric Computing Program, the Center for Adaptive Supercomputing Software - Multithreaded Architectures at Pacific Northwest National Laboratory, and the DARPA HPCS program. Work identifying application areas and needs has been supported in part by these programs as well as the National Science Foundation through grant 1216898.

References

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

[2] Ball, P., Klingner, J., and Lum, K. Crowdsourced data is not a substitute for real statistics. 2011. http://benetech.blogspot.com/2011/03/crowdsourced-data-is-not-substitute-for.html

[3] James, J. How Much Data is Created Every Minute? 2012. http://www.domo.com/blog/2012/06/how-much-data-is-created-every-minute

[4] Georgia Tech team uses Twitter, blogs to monitor elections in developing nations. Georgia Institute of Technology. 2011. http://www.gatech.edu/newsroom/release.html?nid=71154

[5] Frias-Martinez, E. and Frias-Martinez, V. Enhancing Public Policy Decision Making using Large-Scale Cell Phone Data. 2012. http://www.unglobalpulse.org/publicpolicyandcellphonedata

[6] Overview; Visualization to Connect the Dots. Knight Foundation. http://overview.ap.org

[7] Consolidated volume in NYSE listed issues, 2010 – current. 2011. NYSE Euronext. http://www.nyxdata.com/nysedata/asp/factbook/viewer_edition.asp?mode=table&key=3139&category=3

[8] Findings Regarding the Market Events of May 6, 2010; Report of The Staffs of the CFTC and SEC to the Joint Advisory Committee on Emerging Regulatory Issues. 2010. U.S. Commodity Futures Trading Commission. http://www.sec.gov/news/studies/2010/marketevents-report.pdf

[9] Bader, D. and Riedy, J., et al. STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. 2012. http://www.cc.gatech.edu/stinger.

[10] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. In the *Proceedings of the 2004 SIAM International Conference on Data Mining* (Nashville, TN, June 13-16). SIAM, Philadelphia, 2004.

Biographies

E. Jason Riedy is faculty in the School of Computational Science and Engineering at Georgia Tech as a Research Scientist II. He primarily develops algorithms and tools parallel analysis of dynamic social networks. His other interests include high-performance and accurate linear algebra, floating-point arithmetic, and parallel combinatorial optimization. His Ph.D. in computer science is from the University of California, Berkeley in 2010 in combinatorial optimization and targeted high-precision arithmetic.

David A. Bader is a full professor in the School of Computational Science and Engineering, College of Computing, at Georgia Institute of Technology, and Executive Director for high performance computing. He received his Ph.D. in 1996 from The University of Maryland. His interests are at the intersection of high-performance computing and real-world applications, including computational biology and genomics and massive-scale data analytics. He is a Fellow of the IEEE and AAAS, a National Science Foundation CAREER Award recipient, a co-founder of the Graph500 List for benchmarking "Big Data" computing platforms. Bader is recognized as a "RockStar" of high performance computing by insideHPC and as *HPCwire*'s People to Watch in 2012.