STINGER: High Performance Data Structure for Streaming Graphs

David Ediger Rob McColl Jason Riedy David A. Bader
Georgia Institute of Technology
Atlanta, GA, USA

***Abstract*—The current research focus on "big data" problems highlights the scale and complexity of analytics required and the high rate at which data may be changing. Future applications in this space will need to cope with high rates of change at scale. STINGER is a scalable, high performance graph data structure that enables these applications. Key attributes of STINGER are fast insertions, deletions, and updates on graphs with semantic information and skewed degree distributions. We demonstrate a process of algorithmic and architectural optimizations that enable high performance on the Cray XMT family and Intel multicore servers. Our implementation of STINGER processes over 3 million updates per second on a scale-free graph with 537 million edges.**

I. Introduction

The growth of social media, heightened interest in knowledge discovery, and the rise of ubiquitous computing in mobile devices and sensor networks [1] have motivated researchers and domain scientists to ask complex queries about the massive quantity of data being produced. During a recent Champions League match between Barcelona and Chelsea, Twitter processed a record 13,684 Tweets per second [2]. Facebook users posted an average of 37,000 Likes and Comments per second during the first quarter of 2012 [3]. Google's Knowledge Graph for search clustering and optimization contains 500 million objects and 3.5 billion relationships [4].

In the *massive streaming data analytics* model, we view the graph as an infinite stream of edge insertions, deletions, and updates. Keeping complex analytics up to date at these high rates is a challenge that requires new algorithms that exploit opportunities for partial recomputation, new data structures that maximize parallelism and reduce locking, and massive multithreaded compute platforms. In most cases, the new information being received does not affect the entire graph, but only a small subset of vertices. Rather than recomputing an analytic from scratch, it is possible to react faster by only computing on the data that have changed.

Algorithms that take advantage of this framework need a flexible, dynamic data structure that can tolerate the ingest rate of new information. Online social networks, such as Facebook and Twitter, as well as many other human and biological networks, display a "scale-free" property [5]. These graphs typically have low diameters and a power-law distribution in the number of neighbors. To cope with this skewed distribution, a graph data structure must be able to simultaneously accommodate vertices with 10 or 100,000 neighbors. STINGER [6] is a dynamic graph data structure that exposes large amounts of parallelism, supports fast updates, and is well-suited to scale-free graphs. In this paper, we demonstrate the performance of STINGER on commodity Intel multicore servers as well as the Cray XMT family of supercomputers. STINGER is a portable, open source package that can handle graphs with over 2 billion edges and update rates in excess of 3 million updates per second.

In prior work, we offered parallel algorithms for processing edge insertions and deletions while tracking clustering coefficients [7] and connected components [8]. These algorithms exploit the locality of the edge update and avoid full recomputation by updating the metrics appropriately. With clustering coefficients, an new edge insertion or deletion affects only the neighbors of the endpoints. Using a series of set intersections, the triangle counts that make up the clustering coefficient are updated. Connected components can be be tracked for edge insertions using only the mapping between vertices and component IDs. A number of heuristics are proposed (including triangle finding and spanning tree traversal) to avoid recomputation in the case of edge deletions.

In the following section, we will describe STINGER, the data structure for streaming graphs. In Section III, we give an overview of the two different multithreaded platforms being used in our experiments: an Intel multicore server and the Cray XMT family. We describe the microbenchmark used for performance evaluation and the synthetic graph generator that produces our input data sets. Section IV presents a number of optimizations to the STINGER insert and remove procedure to increase performance from 12,000 updates per second to over 1.8 million updates per second on an Intel

multicore system.

II. STINGER

STINGER (Spatio-Temporal Interaction Networks and Graphs Extensible Representation) is a community developed, high performance, extensible data structure for dynamic graph problems [6]. The data structure is based on linked lists of blocks. The number of vertices and edges can grow over time by adding additional vertex and edge blocks. Both vertices and edges have types, and a vertex can have edges of multiple types.

Edges incident on a given vertex are stored in a linked list of edge blocks. An edge is represented as a tuple of neighbor vertex ID, integer weight, first timestamp, and modified timestamp. All edges in a given block have the same edge type. The block contains metadata such as the lowest and highest timestamps and the high-water mark of edges within the block.

Parallelism exists at many levels of the data structure. Each vertex has its own linked list of edge blocks that is accessed from the logical vertex array (LVA). A "for all vertices" loop is parallelized over these lists. Within an edge block, the incident edges can be explored in a parallel loop. The size of the edge block, and therefore the quantity of parallel work to be done, is a user-defined parameter. In our experiments, 32 edges in a block is a good selection.

The edge type array (ETA) is a secondary index that points to all edge blocks of a given type. In an algorithm such as connected components, which is edge parallel, this additional mode of access into the data structure permits all edge blocks to be explored in a parallel for loop.

To assist the programmer in writing a graph traversal, our implementation of STINGER provides parallel edge traversal macros that abstract the complexities of the data structure while still allowing compiler optimization. For example, the STINGER_PARALLEL_FORALL_EDGES_OF_VTX macro takes a STINGER data structure pointer and a vertex ID. The programmer writes the inner loop as if they are looking at a single edge. Edge data is read using macros such as STINGER_EDGE_TYPE and STINGER_EDGE_WEIGHT. More complex traversal macros are also available that limit the edges seen based on timestamp and edge type.

Although most analytic kernels will only read from the data structure, the STINGER must be able to respond to new and updated edges. Functions are provided that insert, remove, increment, and touch edges in parallel. The graph can be queried as to the in-degree and out-degree of a vertex, as well as the total number of vertices and edges in the graph.

STINGER is written in C with OpenMP and Cray MTA pragmas for parallelization. It compiles and runs on both Intel and AMD x86 shared memory servers and workstations and the Cray XMT supercomputing platform. The code is open source and available at http://www.cc.gatech.edu/stinger.

III. Experimental Setup

We will examine STINGER implementations and performance on two multithreaded systems with large-scale memories. The first is a 4-socket Intel multicore system (mirasol) employing the Intel Xeon E7-8870 processor at 2.40 GHz with 30 MiB of L3 cache per processor. Each processor has 10 physical cores and supports HyperThreading for a total of 80 logical cores. The server is equipped with 256 GiB of 1066 MHz DDR3 DRAM.

The second system is the Cray XMT (and the next generation Cray XMT2) [9]. The Cray XMT is a massively multithreaded, shared memory supercomputer designed specifically for graph problems. Each processor contains 128 hardware streams and can execute a different stream at each clock tick. Low-overhead synchronization is provided through atomic fetch-and-add operations and full-empty bit memory semantics. Combined, these features enable applications with large quantities of parallelism to overcome the long latency of irregular memory access. The Cray XMT system at Pacific Northwest National Lab (cougarxmt) has 128 Threadstorm processors with 1 TiB main memory. The Cray XMT2 system at the Swiss National Supercomputing Centre (matterhorn) has 64 processors and 2 TiB main memory.

Due to privacy concerns, it is often difficult to obtain data sets from social media and other sources at the scale of billions of edges. We substitute synthetic graphs to approximate the behavior of our algorithms at scale. For these experiments, we utilize the popular RMAT synthetic graph generator, which produces scale-free graphs with a power-law distribution in the number of neighbors.

Our experiments begin with an initial graph in memory from the RMAT generator (we use RMAT parameters $a = 0.55, b = 0.1, c = 0.1, d = 0.25$). The graph size is given by two parameters: $scale$ and $edgefactor$. The initial graph has 2^{scale} vertices and approximately $2^{scale} * edgefactor$ edges. After generation, we make the graph undirected.

After generating the initial graph, we generate additional edges – using the same generator and parameters – to form a stream of updates. This stream of updates is mostly edge insertions. With a probability of 6.25 percent, we select some of these edge insertions to

be placed in a deletion queue. With the same probability, we take an edge from the deletion queue and add it to the stream as an edge deletion.

The insert/remove microbenchmark builds a STINGER data structure in memory from the generated initial graph on disk. Next, a batch of edge updates is taken from the generated edge stream. The number of edge updates in the batch is variable. We measure the time taken to process each edge update in the data structure. We measure several batches and report the performance in terms of updates per second.

IV. Optimizations

Applications that rely on STINGER are constantly receiving a stream of new edges and edge updates. The ability to react quickly to new edge information is a core feature of STINGER. When an edge update $\langle u, v \rangle$ is received, we must first search all of the edge blocks of vertex u for neighbor v of the given edge type. If the edge is found, the weight and timestamp are updated accordingly. If the edge is not found, an empty space must be located or an empty edge block added to the list.

In an early implementation of STINGER, each new edge was processed in this manner one at a time. This approach maximized our ability to react to a single edge change. On an Intel multicore system with a power law graph containing 270 million edges, inserting or updating one at a time yields a processing rate of about 12,000 updates per second. The Cray XMT performed worse, achieving approximately 950 updates per second.

On systems with many thread contexts and memory banks, there is not enough work or parallelism in the data structure to process a single update at a time. To remedy this problem, we began processing edge updates in batches. A batch amortizes the cost of entering the data structure and provides a larger quantity of independent work to do.

Our implementation of STINGER first sorts the batch (typically 100,000 edge updates at a time) such that all edge updates incident on a particular vertex are grouped together with with deletions separated from insertions. For each unique vertex in the batch, we have at least one work item that can be done in parallel. Deletions are processed prior to insertions to make room for the new edges. Updates on a particular vertex are done sequentially to avoid synchronization costs.

This approach to updates yields a 14x increase on the Intel multicore system. We can process 168,000 updates per second. The Cray XMT implementation reaches 225,000 updates per second, or a 235x increase in performance.

In a scale-free graph, however, a small number of vertices will have many updates to do, while most will only have a single update or no updates at all. This workload imbalance limits the quantity of parallelism we can exploit and forces most threads to wait on a small number of threads to complete.

To solve this problem, we skip sorting the edges and process each edge insertion independently and in parallel. However, processing two edge updates incident on the same vertex introduces race conditions that must be handled with proper synchronization. The Cray XMT is a perfect system for this scenario. The additional parallelism will increase machine utilization and its fine-grained synchronization intrinsics will enable a simple implementation.

There are three possible scenarios when inserting an edge into a vertex's adjacency list in STINGER. If the edge already exists, the insert function should increment the edge weight and update the modified timestamp. If the edge does not exist, a new edge should be inserted in the first empty space in an edge block of the appropriate type. If there are no empty spaces, a new edge block containing the new edge should be allocated and added to the list.

The parallel implementation guarantees these outcomes by following a simple protocol using full-empty semantics on the Cray XMT or using an emulation of full-empty semantics built on atomic compare-and-swap instructions on x86. Since multiple threads reading and writing in the same place in an adjacency list is a relatively uncommon occurrence, locking does not drastically limit performance. When an edge is inserted, the linked list of edge blocks is first checked for an existing edge. If the edge is found, the weight is incremented atomically. Otherwise the function searches the linked list a second time looking for an empty space. If one is found, the edge weight is locked. Locking weights was chosen to allow readers within the first search to continue past the edge without waiting. If the edge space is still empty after acquiring the lock, the new edge is written into the block and the weight is unlocked. If the space is not empty but has been filled with the same destination vertex as the edge being inserted, the weight is incremented and the weight is unlocked. If another edge has been written into the space before the lock was acquired, the weight is unlocked and the search continues as before. If the second search reaches the end of the list having found no spaces, the "next block" pointer on the last edge block must be locked. Once it is locked it is checked to see if the pointer is still null indicating the end of the list. If so, a new block is allocated and the edge is inserted into it before linking the block into the list

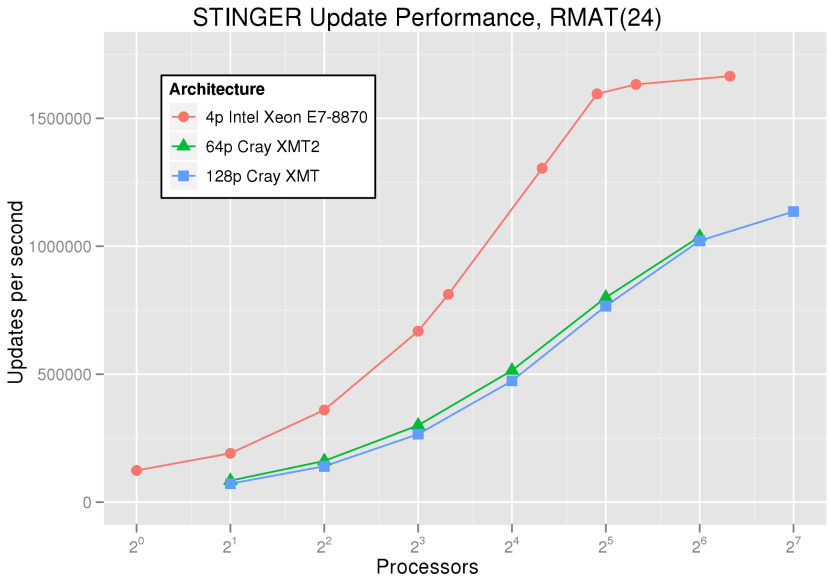

Fig. 1. Updates per second on an RMAT graph with 16 million vertices and 270 million edges with a batch size of 100,000 edge updates.

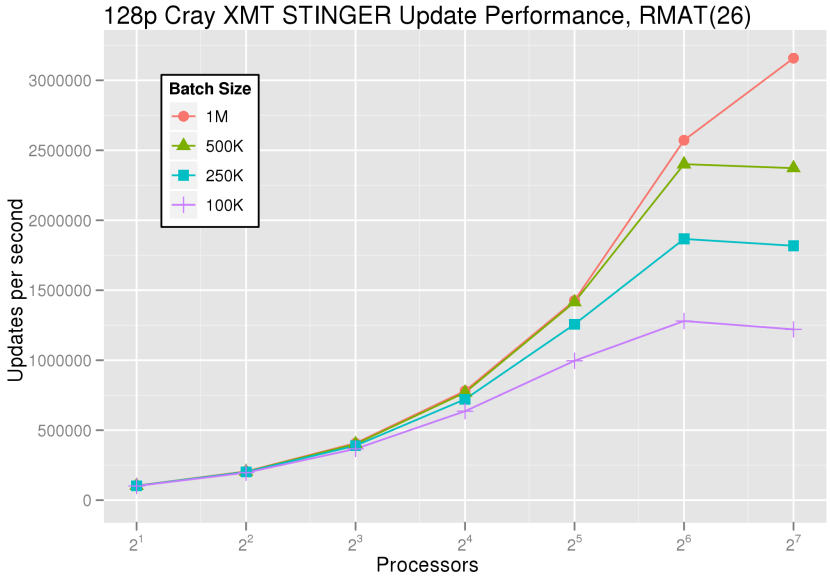

Fig. 2. Increasing batch size results in better performance on the 128-processor Cray XMT. The initial graph is an RMAT graph with 67 million vertices and 537 million edges.

and unlocking the previous "next" pointer. If the pointer is not null, it is unlocked, and the search continues into the next block. In this way, we guarantee that all insertions are successful, that all destination vertices are unique, that no empty space is wasted, and that no new blocks are needlessly allocated. Deletions are handled by a similar algorithm.

Implemented in this way, the Cray XMT reaches 1.14 million updates per second on the scale-free graph with 270 million edges. This rate is 1,200x faster than the single update at a time approach. With this approach, we also have enough parallelism such that the performance scales past 32 processors to 128 processors. Figure 1 compares the performance of the Cray XMT, Cray XMT2, and an Intel multicore system on the same problem.

On the 4-socket Intel multicore system, this method achieves over 1.6 million updates per second on the same graph with a batch size of 100,000 and 1.8 million updates per second with a batch size of 1,000,000. This is 133x faster than the single update at a time approach and nearly 10x faster than the batch sorting approach. The scalability of this approach is linear to 20 threads, but falls off beyond that mark due to limitations imposed by the use of atomics across multiple sockets.

With a larger graph (67 million vertices and 537 million edges), the performance remains flat at 1.22 million updates per second. Increasing the batch size from 100,000 updates to one million updates further increases the available parallelism. In Figure 2, the increased parallelism from increasing batch sizes results in better scalability. The Cray XMT reaches a peak of 3.16 million updates per second on 128 processors for this graph.

The Cray XMT2, which has a 4x higher memory density than the Cray XMT, can process batches of one million updates on a scale-free graph with 268 million vertices and 2.15 billion edges at 2.23 million updates per second. This quantity represents a 44.3x speed-up on 64 processors over a single processor. The graph in memory consumes approximately 313 GiB.

V. Conclusions

Future applications will continue to generate more data and demand faster response to complex analytics. Through algorithmic, compiler, and architectural optimizations and transformations, STINGER is a scalable, high performance graph data structure capable of meeting current and future demand for massive streaming data analytics. As graph sizes swell to billions of vertices and edges, large shared memory systems with many hardware threads and many memory banks will be a practical system solution to these problems. STINGER demonstrates that scalable graph codes can be successfully implemented in a cross-platform manner without loss of performance.

Acknowledgments

This work was supported in part by the Pacific Northwest National Lab (PNNL) Center for Adaptive Supercomputing Software for MultiThreaded Architectures (CASS-MT). We thank PNNL, the Swiss National Supercomputing Centre, Cray, and Intel for access to these systems.

References

[1] C. L. Borgman, J. C. Wallis, M. S. Mayernik, and A. Pepe, "Drowning in data: digital library architecture to support scientific use of embedded sensor networks," in *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, ser. JCDL '07, 2007, pp. 269–277.

[2] Twitter, "#Goal," April 2012, http://blog.uk.twitter.com/2012/04/goal.html.

[3] Facebook, "Key facts," May 2012, http://newsroom.fb.com/content/default.aspx?NewsAreaId=22.

[4] Google, "Introducing the knowledge graph: things, not strings," May 2012, http://insidesearch.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html.

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

[6] 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," Georgia Institute of Technology, Tech. Rep., 2009.

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

[8] D. Ediger, E. J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *5th Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2011.

[9] P. Konecny, "Introducing the Cray XMT," in *Proc. Cray User Group meeting (CUG 2007)*. Seattle, WA: CUG Proceedings, May 2007.