2011 IEEE International Parallel & Distributed Processing Symposium

Tracking Structure of Streaming Social Networks

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

***Abstract*—Current online social networks are massive and still growing. For example, Facebook has over 500 million active users sharing over 30 billion items per month. The scale within these data streams has outstripped traditional graph analysis methods. Real-time monitoring for anomalies may require dynamic analysis rather than repeated static analysis. The massive state behind multiple persistent queries requires shared data structures and flexible representations. We present a framework based on the STINGER data structure that can monitor a global property, connected components, on a graph of 16 million vertices at rates of up to 240 000 updates per second on 32 processors of a Cray XMT. For very large scale-free graphs, our implementation uses novel batching techniques that exploit the scale-free nature of the data and run over three times faster than prior methods. Our framework handles, for the first time, real-world data rates, opening the door to higher-level analytics such as community and anomaly detection.**

I. Introduction

Online social networking has expanded to massive scale in recent years. For example, Facebook has grown three orders of magnitude in just over three years and, as of December 2010, has over 500 million active users [1]. An average degree of 130 translates into 32.5 billion edges in the "friendship" graph. Considering other content like photos, Facebook claims over 30 billion items shared per month.

Automatic analysis of these data supports analysts in various areas, including "friend" recommendations, online advertising, anomaly detection, and information dissemination. Questions of interest include which nodes are most important in a network, how the network's communities are structured, and which nodes change their behavior over time. Analyzing dynamic graphs through static snapshots suffices only for small networks with low rates of change. Moreover, many static graph algorithms are difficult to scale to massive, real-world data and do not use parallel architectures efficiently. Both the size of real-world data and the rate of change present challenges for analysis.

Online social networks like Facebook and Twitter, as well as many other networks observed in nature and human society, have the *scale-free* property [2]. Scale-free graphs have a low diameter, and the number of neighbors follows a power law distribution. Many vertices have a small number of neighbors, while a few vertices are connected with a large part of the graph. Parallelizing algorithms on these graphs is challenging because of the lack of a small separator and the dynamically changing edges. The degree distribution also creates workload imbalance when scheduling vertices among processors.

Answering the questions of interest for massive scale-free networks requires establishing suitable data structures and algorithmic kernels. This paper's example kernel monitors the network's connected components. This problem is a well-studied graph analytical metric representative of both the graph's structure as well as the vertices' connectivity. As edges are inserted into and deleted from the graph, the number of connected components, the mapping between vertices and components, the size distribution of components, and when components merge and separate over time become characteristics of interest.

A. Shortcomings of Previous Work

For general graphs, insertions must only check the component membership of their endpoints to determine if two components have merged. Deletions are much more difficult to handle as the endpoints do not contain information about the topology of the component or any other paths that may reconnect the two vertices. Enumerating all of the possible edge deletions that would separate components, in light of the constant stream of new edges being inserted in the graph, is infeasible. Several algorithms have been proposed that handle deletions using breadth-first searches. Even and Shiloach [3] spawn two breadth-first searches for each edge deletion. One looks to verify that the component

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.326

remains intact while the other looks to determine that the component has been separated. Another algorithm stores the graph as a sequence of graphs that are created with each subsequent insertion [4]. A Union-Find or Least Common Ancestor algorithm establishes connectivity. Henzinger et al. [5] also use a sequence of graphs and a coloring during a connected components computation to determine that a new component has been created after a batch of deletions.

These algorithms do not expose enough parallelism to support the analysis of the massive graphs of interest on massively parallel architectures. In a scale-free graph, a single breadth-first search is an expensive operation that quickly consumes the entire graph. Running one or more traversals for each edge deletion, when a deletion is unlikely to have caused a change in the component structure, will not support the high data rates of today's input streams.

Another approach utilizes partitions of the graph in order to determine components. Henzinger and King [6] create a spectrum of partitions from dense to sparse subgraphs. After a deletion, a search begins within the densest level to re-establish connectivity between the two endpoints. Failure to find a link moves to a sparser level until connectivity is ruled out. Eppstein et al. [7] use partitions according to the average degree and sparsification techniques to maintain a minimum spanning forest of components.

For the social networks of interest, it would be difficult to create a spectrum of subgraphs from dense to sparse. Facebook, with 500 million vertices and average degree 130, is already very sparse. Using the Henzinger and King method, one will only create two levels of subgraphs with the majority of the graph in the first level. Most edge deletions cause what amounts to a static recomputation of connected components. Partitioning according to average degree is difficult because of the power law distribution in the vertex degree. For space and performance reasons, we wish to avoid maintaining extra data structures like a minimum spanning forest in favor of tracking connected components from the common graph data structure.

B. Our New Approach

To motivate our approach, we would like to answer, in a persistent manner, the question "do vertex u and vertex v lie in the same component?" while at the same time supporting the ability to insert and delete edges presented as a batch in parallel. We focus our efforts on scale-free networks like those of social networks and other biological networks and capitalize on their structural properties. Unlike prior approaches, our new framework manufactures large amounts of parallelism by considering the input data stream in batches. All of the edge actions within the batch are processed in parallel. Applying any of the breadth-first search-based algorithms to a batch of deletions would result in thousands of concurrent graph traversals, exhausting the memory and computational resources of the machine.

In Section II we present our algorithmic approach for tracking connected components given a stream of edge insertions and deletions. Optimizations afforded by the scale-free nature of the graph are highlighted there, too. As an example, we take advantage of the low diameter by intersecting neighborhood bit arrays to quickly re-establish connectivity using triangles. Also, our algorithm uses a breadth-first search of the significantly smaller component graph, whose size is limited by number of insertions and deletions in the batch being processed. The scale-free nature of the input data means that most edges in the batch do not span components, so the number of non-self-edges in the component graph for a given batch is very small.

Our techniques fully utilize the fine-grained synchronization primitives of the Cray XMT to look for triangles in the edge deletions and quickly rule out most of the deletions as inconsequential without performing a single breadth-first search. All of the neighbor intersections can be computed concurrently, providing sufficient parallelism for the architecture. More details about the implementation of our algorithm on the Cray XMT are given in Section III.

Experimental results on the Cray XMT are presented in Section IV. Despite the challenges posed by the input data, we show that the scale-free structure of social networks can be exploited to accelerate graph connectivity queries in light of a stream of deletions. On a synthetic social network with over 16 million vertices and 135 million edges, we are able to maintain persistent queries about the connected components of the graph with an input rate of 240 000 updates per second, a three-fold increase over previous methods.

Hence, our new framework for incremental computation, rather than recomputation, enables scaling to very large data sizes on massively parallel, multithreaded supercomputing architectures. Note that in this paper we add the monitoring of a global component, whereas our previous work [8] was concerned with local clustering coefficients, which are easier to handle due to their locality. As a next step, studying massive dynamic graphs with more complex methods based on our algorithmic kernels will lead to insights about community and

anomaly detection unavailable within smaller samples.

To summarize, the main contributions of this paper are:

1) a novel parallel framework for maintaining a global property by batching the input stream,
2) the replacement of breadth-first search with efficient parallel triangle counting, and
3) a faster approach to tracking connected components in massive scale-free graphs.

II. Tracking Connected Components in Scale-Free Graphs

A. Problem Structure

Given an edge to be inserted into a graph and an existing labeling of the connected components, one can quickly determine if it has joined two components. Given a deletion, however, recomputation (through breadth-first search or $s - t$ connectivity) is the only known method with subquadratic space complexity to determine if the deleted edge has cleaved one component into two. If the number of deletions that actually cause structural change is very small compared to the total number of deletions (such as in the case of a scale-free network), our goal will be to quickly rule out those deletions that are "safe" (*i.e.* does not split a component). The framework we propose for this computation will establish a threshold for the number of deletions we have not ruled out between recomputations.

Our approach for tracking components is motivated by several assumptions about the input data stream. First, a very small subset of the vertices are involved in any series of insertions and deletions. Insertions that alter the number of components will usually join a small component to the large component. Likewise, a deletion that affects the structure of the graph typically cleaves off a relatively small number of vertices. We do not anticipate seeing the big component split into two large components. The small diameter implies that connectivity between two vertices can be established in a small number of hops, but the low diameter and power law distribution in the number of neighbors also implies that a breadth-first search quickly consumes all vertices in the component.

Second, we adopt the *massive streaming data analytics* model [8]. We assume that the incoming data stream of edge insertions and deletions is infinite, with no start or end. We store as much of the recent graph topology as can be held in-memory alongside the data structures for the computation. The graph contains inherent error arising from the noisy data of a social network containing false edges as well as missing edges. As a result, we will allow a small amount of error at times during the computation, so long as we can guarantee correct results at specific points. The interval between these points will depend on the tolerance for error. We process the incoming data stream as batches of edge insertions and deletions, but do not go back and replay the data stream.

B. Algorithm

The pseudocode of our algorithm appears in Algorithm 1. The algorithm consists of four phases that are executed for each batch of edge insertions and deletions that is received. These phases can be summarized as follows: First, the batch of edges is sorted by source and destination vertices. Second, the edges are inserted and/or deleted in the STINGER data structure. Third, edge deletions are evaluated for their effect on connectivity. Finally, insertions are processed and the affected components are merged.

We will consider unweighted, undirected graphs, as social networks generally require links to be mutual. The graph data structure, the batch of incoming edges currently being processed, and the metadata used to answer queries fit completely within the memory. We can make this assumption in light of the fact that current high-end computing platforms, like the Cray XMT, provide shared memories on the order of terabytes. We will now examine, in detail, each phase of the algorithm.

In the sort phase, we are given a batch of edges to be inserted and deleted. In our experiments, the size of this batch may range from 1,000 to 1 million edges. We use a negative vertex ID to indicate a deletion. The batch must first be sorted by source vertex and then by destination vertex. On the Cray XMT, we bucket sort by source using atomic fetch-and-add to determine the size of the buckets. Within each bucket, we can sort by destination vertex using an arbitrary sequential sorting algorithm, processing each bucket in parallel. At the end of the sort phase, each vertex's operations are clustered within the batch into a group of deletions and a group of insertions pertaining to that vertex. At this stage, one could carefully reconcile matching insertions with deletions, which is especially important for multigraphs. For our experiments, we will skip reconciliation, processing each inserted and deleted edge, and allowing only the existence or non-existence of a single edge between each pair of vertices.

In Phase 2, the data structure update phase, the STINGER data structure is given the batch of insertions and deletions to be processed. For each vertex, deletions

Algorithm 1 A parallel algorithm for tracking connected components.

Input: Batch B of edges $\langle u, v \rangle$ to be inserted and deleted, component membership M, threshold R_{thresh}, number of relevant deletions R, bit array A, component graph C
Output: Updated component membership M'

```
1:  Sort(B)                                                  ▷ Phase 1: Prepare batch
2:  for all b ∈ B in parallel do
3:    if b is deletion then
4:      Push(Q_del, b)
5:    else
6:      Push(Q_ins, b)
7:  StingerDeleteAndInsertEdges(B)                           ▷ Phase 2: Update data structure
8:  for all b ∈ Q_del in parallel do                         ▷ Phase 3: Process deletions
9:    ⟨u, v⟩ ← b
10:   A_u ← 0⃗                                               ▷ All bits set to zero
11:   for all n ∈ Neighbors(u) in parallel do
12:     Set bit n in A_u to 1
13:   F ← 0
14:   for all n ∈ Neighbors(v) in parallel do
15:     if bit n in A_u = 1 then
16:       F ← 1                                              ▷ Triangle found
17:   if F = 0 then
18:     atomic R ← R + 1                                     ▷ No triangles found
19: if R > R_thresh then
20:   R ← 0
21:   M' ← ConnectedComponents(G)
22: else                                                     ▷ Phase 4: Process insertions
23:   C ← ∅
24:   for all b ∈ Q_ins in parallel do
25:     ⟨u, v⟩ ← b
26:     Add ⟨M[u], M[v]⟩ to C
27:   T ← ConnectedComponents(C)
28:   for all v ∈ V in parallel do
29:     M'[v] ← T[v]
```

are handled first, followed by insertions. This ordering creates space in the data structure before insertions are added, minimizing the number of new blocks that must be allocated in the data structure and thereby reducing overhead.

After updating the graph, edge deletions identified earlier are checked to see if they disrupt connectivity in Phase 3. We create a bit array, in which each bit represents a vertex in the graph, for each unique source vertex in the batch of edge deletions. A bit set to 1 indicates that the vertex represented by that bit is a neighbor. Because of the scale-free nature of the graph, the number of bit arrays required for a batch is much less than the number of vertices. Since vertices can be involved in many edge deletions, the fine-grained synchronization available on the Cray XMT enables parallelism in the creation phase and re-use of the bit arrays in the query phase. We compute the intersection of neighbor sets by querying the neighbors of the sink vertices in the source bit array. Given that a social network is a scale-free graph, the rationale is that this intersection will quickly reveal that most of the edge deletions do not disrupt connectivity. Regarding running time and memory consumption, note that a common case bit array intersection for vertices with small degree can be handled by a quick lookup in the sorted list of neighbors and the bit matrix intrinsics of the Cray XMT.

At this point, we can take the remaining edge deletion

candidates and further process them to rule out or verify component structure change, likely using a breadth-first search. Otherwise, we will store the number of relevant deletions R seen thus far. After this number has reached a given threshold R_{thresh} determined by the tolerance for inconsistency before recomputation, we will re-compute the static connected components to determine the updated structure of the graph given the deletions that have taken place since the last static recomputation.

If we did not exceed R_{thresh} in the previous phase, the insertions must now be processed in Phase 4. For each edge being inserted, we look up the vertex endpoints in the current component mapping and replace them with the component ID to which they belong. In effect, we have taken the batch of insertions and converted it into a component graph. As this is a scale-free network, many of the insertions will now be self-edges in the component graph. The remaining edges will indicate components that have merged. Although it is unlikely, chains of edges, as well as duplicate edges, may exist in the batch. The order of merges is determined by running a static connected components computation on the new component graph[1]. The result is an updated number of components in the graph and an updated vertex-component mapping.

In both the static connected components case and when finding the connected components of the component graph, we use an algorithm similar to Kahan's algorithm [9]. Its first stage, performed from all vertices in the graph, greedily colors neighboring vertices using integers. The second stage repeatedly absorbs higher labeled colors into lower labeled neighbors. Colors are relabeled downward as another series of parallel breadth-first searches. When collisions between colors are no longer produced, the remaining colors specify the components.

III. Implementation on the Cray XMT

The experiments that are presented in Section IV consist of two codes, which we describe in more detail in this section: the tracking connected components application and the STINGER data structure implementation. Each takes advantage of the unique features of the Cray XMT in different ways.

The Cray XMT [10] is a supercomputing platform designed to accelerate massive graph analysis codes. The architecture tolerates high memory latencies using massive multithreading. Fine-grained synchronization constructs are supported through full-empty bits as well as atomic fetch-and-add instructions. A large fully shared memory enables the analysis of graphs on the order of one billion vertices using a well-understood programming model.

Each Threadstorm processor within a Cray XMT contains 128 *hardware streams*. Streams may block temporarily while waiting for a long-latency instruction, such as a memory request, to return. The processor will execute one instruction per cycle from hardware streams that have instructions ready to execute. The Cray XMT does not require locality to obtain good performance. Instead, latency to memory is tolerated entirely by hardware multithreading, making this machine a good candidate for memory-intensive codes like those found in graph analysis.

The Cray XMT used for these experiments is located at Pacific Northwest National Lab and contains 128 Threadstorm processors running at 500 MHz. These 128 processors support over 12 thousand hardware thread contexts. The globally addressable shared memory totals 1 TiB. Memory addresses are hashed globally to break up locality and reduce hot-spotting.

The bit array is a fast data structure that, with the support of fine-grained synchronization, can be built and queried in parallel. To build a bit array for a graph with 2^{24} vertices will require 2 MiB per array. The scale-free nature of the graph means that it suffices to construct relatively few bit arrays. Although there are 16 million vertices, we pre-allocate only 10,000 bit arrays at a cost of about 20 GiB. If the graph were not scale-free, and if the platform lacked sufficient memory, we would likely not have the space to do this for such a large graph. Since high degree vertices will likely be touched by each batch, a learning algorithm could be used to identify these vertices, update their bit arrays, and reuse the bit array from batch to batch, amortizing the creation cost. If the graph were less sparse or memory footprint is a concern, a Bloom filter (or another low-cost insert/delete representation) could be used to conserve space while introducing the probability for error [11]. A Bloom filter, however, could not be reused since it does not support deletions.

The bucket sort implementation takes advantage of the compiler's ability to automatically identify and parallelize reductions and linear recurrences such as parallel prefix sums. These are used to find the size of the buckets in parallel and then reserve space in the output array.

[1]In our implementation we use a compressed sparse row (CSR) representation, rather than creating a new graph data structure, as this only requires a sort of the batch of edges and a prefix sum, both done in parallel.

Prior work on the Cray XMT has shown that batching of the update stream is required to create adequate parallelism for the thousands of user hardware streams [8]. Processing in batches has the additional benefit that it bounds the size of temporary arrays needed for making the calculation. By using these space bounds to preallocate data structures before the computation begins, costly calls to the memory allocator are eliminated.

STINGER is the data structure on which our connected components experiments are built [12]. The data structure must provide the ability to easily and efficiently accept edge insertions and edge deletions from a scale-free graph while also permitting fast querying of neighbor information and other metadata about vertices and edges.

The data structure consists of a hybrid of arrays in memory that are linked through pointers. The size of the blocks can be adjusted based on the degree distribution of the data and the concurrency required by the machine. Deletion consists only of negating a neighbor's vertex ID in the edge block. Insertion requires finding an open space in an existing edge block or allocating a new block. The data structure supports timestamps and other historical metadata as well as vertex and edge types, although none of these are used explicitly for this experiment.

The STINGER specification does not specify consistency; the programmer must assume that the graph can change underneath the application. The programmer is provided routines to extract a snapshot of a vertex's neighbor list, alleviating this concern at the expense of an additional buffer in memory.

Our STINGER implementation 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 making it easy to convert existing static codes that utilize the popular compressed sparse row format to work with STINGER. In the course of developing the connected components code, we observed that this "copy out" strategy resulted in identical or better performance for static codes using STINGER versus the compressed sparse row representation.

IV. Experiments

A. Experimental Settings

The experimental results presented in this paper use synthetic R-MAT [13] input graphs derived by sampling from a Kronecker product. We generate two data files from each R-MAT graph with parameters $a = 0.55$, $b = 0.1$, $c = 0.1$, and $d = 0.25$. The first is an initial graph with $2^{20} \approx 1$ million vertices or $2^{24} \approx 16$ million vertices, both with an edge factor of 8 (meaning the number of edges is 8 times the number of vertices). The second is an input stream of edge actions (both insertions and deletions) that, by construction, favors the same vertices as the initial graph. It is this input stream that we will divide into batches. When creating this input stream, we sample from the distribution to create edge insertions. With a probability of 1/16, we add an edge insertion to a delete queue. We choose an edge from the delete queue to be an edge deletion, rather than choosing a new edge to be inserted, with probability 1/16.

In a real online social network, edge deletions are not independent but are often guided by a kind of pruning based on distance, time, or importance.

On the Cray XMT, we load the initial graph into memory and create our STINGER data structure. We run a static computation of connected components to establish the initial vertex-component mapping, number of components, and size of components. In effect we have entered the infinite data stream at some intermediate point to begin tracking connected components. We measure the time it takes for each batch to be processed, the data structure to be updated, and the new component information to be calculated. We report this performance metric in terms of updates per second.

On the 128 processor Cray XMT, we conduct our experiments using 32 processors for two reasons. First, the input graph of 16 million vertices is small relative to the size of the machine. Second, we imagine tracking connected components alongside of other higher-level analysis kernels that subscribe to the results of this computation to aid in their own computation. This frees up resources for other computations to be taking place at the same time. One example is a kernel that samples vertices to create an ongoing approximation of a metric of interest. The kernel may want to ensure that it is sampling vertices from all components or that it samples in proportion to the size of each component. In this way, it is running at the same time as the connected components kernel and receiving the results into its own computation.

B. Experimental Results

Looking closely at the algorithm, one will note that when handling insertions only, the graph data structure does not need to be accessed. We can track the number of components and their sizes using only the vertex-component mapping. The insertions-only algorithm is very fast as the number of "vertices" in the component

	Batch Size (edges)			
	10,000	100,000	250,000	1,000,000
Insertions Only	21,000	168,000	311,000	931,000
Insertions + STINGER	16,700	113,000	191,000	308,000
Insertions + STINGER + Bit Array	11,800	88,300	147,000	240,000
STINGER + Static Connected Components	1,070	10,200	22,400	78,600

Fig. 1. Updates per second on a graph starting with 16M vertices and approximately 135M edges on 32 processors of a Cray XMT.

graph is small compared to the size of the original graph. Additionally, the number of "edges" in the component graph is bounded by the batch size. We observe insertions-only performance of up to 3 million updates per second on a graph with 1 million vertices and nearly 1 million updates per second on the larger graph with 16 million vertices – see Figure 1 for more data on the larger graph.

STINGER, the data structure, inevitably contributes a small overhead to the processing of updates, but it is scalable with larger batches. The update rate of insertions and the data structure can be viewed as an upper bound on the processing rate once deletions are introduced. One method that has been used for handling temporal updates is to re-compute static connected components after each edge or batch of edges. This update rate can be viewed as a lower bound on processing once deletions are introduced. Any method that will decrease the number or frequency of recomputations will increase the rate at which streaming edges can be processed.

We introduce the bit array intersection method as a means to rule out a large number of deleted edges from the list of deletions that could possibly affect the number of connected components. In the algorithm, we set a threshold R_{thresh} meaning that we will tolerate up to R_{thresh} deletions in which the structure of the graph may have been affected before recomputing static connected components. The performance results for insertions plus STINGER plus the bit array intersection represent the update rate if $R_{\text{thresh}} = \infty$. Choosing R_{thresh} will determine performance between the lower bound and this rate. In practice, reasonable values of R_{thresh} will produce performance closer to the upper rate than the lower bound.

As an example of the effect of the bit array on the number of possibly relevant deletions, our synthetic input graph with 16 million vertices produces approximately 6,000 edge deletions per batch of 100,000 actions. The 12,000 endpoints of these deletions contain only about 7,000 unique vertices. Using a bit array to perform an intersection of neighbors, all but approximately 750 of these vertices are ruled out as having no effect on the graph. Performing a neighbor of neighbors intersection would likely reduce this number considerably again at the cost of increased complexity. As it is, the synthetic graph used for these experiments has an edge factor of 8 making it extremely sparse and a worst-case scenario for our algorithm. A real-world social network like Facebook would have an edge factor of greater than 100. In a scale-free network, this would reduce the diameter considerably making our bit array intersection more effective. In our graph, less than 1 percent of deletions cleave off vertices from the big component, while our bit array intersection algorithm rules out almost 90 percent of deletions at a small cost in performance. Given that we can tolerate R_{thresh} deletions between costly static recomputations, a reduction in the growth rate of R, the number of unsafe deletions, will increase the time between recomputations, increasing throughput accordingly.

The left plot in Figure 2 depicts update rates on a synthetic, scale-free graph with approximately 1 million vertices and 8 million edges. Looking at insertions only, or solely component merges without accessing the data structure, peak performance is in excess of 3 million updates per second. The STINGER implementation incurs a small penalty with small batches that does not scale as well as the insertions-only algorithm. The bit array optimization calculation has a very small cost and its performance tracks that of the data structure. Here we see that the static recomputation, when considering very large batches, is almost as fast as the bit array. In this particular test case, we started with 8 million edges, and we process an additional 1 million edge insertions and/or deletions with each batch.

On Figure 2's right, we consider a similar synthetic, scale-free graph with approximately 16 million vertices and 135 million edges. The insertions only algorithm outpaces the data structure again, but by a smaller factor with the larger graph. The bit array performance again closely tracks that of the data structure. At this size, we can observe that the static recomputation method is no

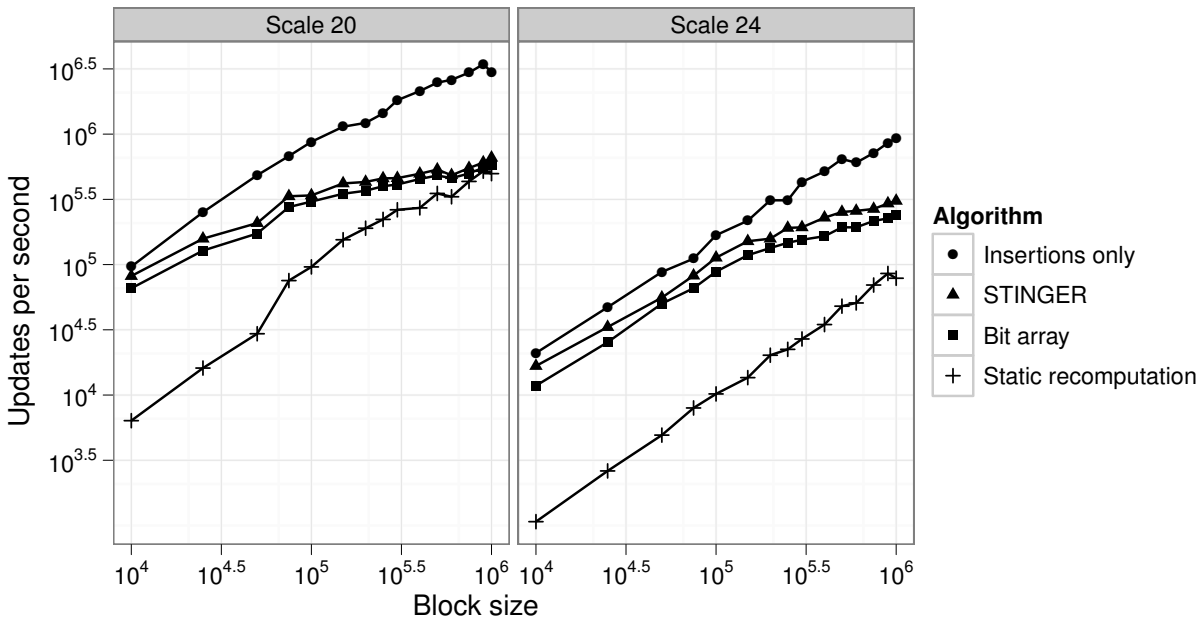

Fig. 2. Update performance for a synthetic, scale-free graph with 1 million vertices (left) and 16 million vertices (right) and edge factor 8 on 32 processors of a 128 processor Cray XMT.

longer feasible, even for large batches. At this size, there is an order of magnitude difference in performance between the static connected components re-computation and the insertions-only algorithm.

We observe a decrease in performance in all four experiments as the size of the graph increases from 1 million to 16 million vertices. There are several properties of the graph that change and affect performance. The larger graph has a larger diameter. Longer paths in the graph increase the running time of breadth-first search. As a result, the static connected computations recomputation time increases. With more vertices to consider, the insertions only algorithm slows down when merging and relabeling components. The larger graph has a higher maximum degree, so walking the edge list of a vertex in STINGER will require additional time. Likewise, although the average degree remains fixed at 8, there are more vertices with degree larger than 8. Retrieving their neighbor list and performing the neighbor intersection will also see a performance penalty. Given that these steps are $O(n)$ in the worst case and are easily parallelized, we would expect good scalability with increasing graph size. In our experiments, the bit array optimization slowed by a factor of 2 when the graph grew by a factor of 16. The connected components recomputation slowed by a factor of 6 for the same increase in graph size.

V. Conclusions and Future Work

We have proposed a parallel algorithm for tracking the connected components of a graph derived from a scale-free social network when the graph is given as an infinite stream of edge insertions and deletions. We process this edge stream as a sequence of batches and we maintain the connected components information from batch to batch. From prior work in the literature, we can see that the goal of any streaming connected components algorithm should be to minimize the number of graph traversals that must be done in order to establish or verify connectivity following a deletion. We have contributed a bit array intersection method as an optimization that can be used to drastically reduce the number of deletions that may cause structural change to the graph.

We have shown through experimentation on synthetic graphs with millions of vertices and hundreds of millions of edges that the use of bit array intersection is done at a small cost relative to the cost of static recomputation. The bit array intersection within our connected components algorithm enables graph processing to operate at speeds comparable to those that do not handle deletions at all. We have demonstrated the performance of these algorithms on a massive graph running on the

Cray XMT and their scalability with increasing batch sizes. By comparing performance for each algorithm on two sizes of graphs, we have established that the running time of our algorithms increases significantly slower than the growth of the graph.

Our connected components algorithm is built atop a framework for analyzing massive streaming graphs when the cost of processing a deletion is high compared to the cost of an insertion. When we tolerate a small amount of error in our analytic, we can handle deletions lazily and proceed at insertion speed. If few deletions actually change the analytic and we can quickly rule out those that do not, we can further extend the time between recomputation.

Although we have presented results focusing on the Cray XMT, our software is portable to other platforms and can be compiled with OpenMP parallelization on multicore systems.

Further work is needed to identify fast data-dependent methods to further isolate only those deletions that cause structural change in the graph. For a given batch of insertions and deletions, the overwhelming majority of new or deleted edges cause no change in the number or size of connected components. Additional heuristics could be developed to approximate component size and distribution without requiring expensive graph traversals that are the predominant cost in establishing connectivity after a deletion.

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 NSF Grant CNS-0708307. We thank PNNL and Cray for providing access to Cray XMT systems.

REFERENCES

[1] Facebook, "User statistics," December 2010, http://www.facebook.com/press/info.php?statistics.

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

[3] Y. Shiloach and S. Even, "An on-line edge-deletion problem," *J. ACM*, vol. 28, no. 1, pp. 1–4, 1981.

[4] L. Roditty and U. Zwick, "A fully dynamic reachability algorithm for directed graphs with an almost linear update time," in *Proc. of ACM Symposium on Theory of Computing*, 2004, pp. 184–191.

[5] M. R. Henzinger, V. King, and T. Warnow, "Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology," in *Algorithmica*, 1999, pp. 333–340.

[6] M. R. Henzinger and V. King, "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *J. ACM*, vol. 46, p. 516, 1999.

[7] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, "Sparsification—a technique for speeding up dynamic graph algorithms," *J. ACM*, vol. 44, no. 5, pp. 669–696, 1997.

[8] 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.

[9] J. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Proc. Workshop on Multithreaded Architectures and Applications*, Long Beach, CA, March 2007.

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

[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[12] 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.

[13] 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: SIAM, Apr. 2004.