# Computational Graph Analytics for Massive Streaming Data

David Ediger Jason Riedy David A. Bader Henning Meyerhenke
School of Computational Science and Engineering
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA

**Abstract**

Handling the constant stream of data from health care, security, business, and social network applications requires new algorithms and data structures. We present a new approach for parallel massive analysis of streaming, temporal, graph-structured data. For this purpose we examine data structure and algorithm trade-offs that extract the parallelism necessary for high-performance updating analysis of massive graphs. As a result of this study, we propose the extensible and flexible data structure for massive graphs called STINGER (Spatio-Temporal Interaction Networks and Graphs Extensible Representation).

Two case studies demonstrate our new approach's effectiveness. The first one computes a dynamic graph's vertices' *clustering coefficients*. We show that incremental updates are far more efficient than global recomputation. Within this kernel, we compare three methods for dynamically updating local clustering coefficients: a brute-force local recalculation, a sorting algorithm, and our new approximation method using a Bloom filter. On 32 processors of a Cray XMT with a synthetic scale-free graph of $2^{24} \approx 16$ million vertices and $2^{29} \approx 537$ million edges, the brute-force method processes a mean of over 50 000 updates per second, while our Bloom filter approaches 200 000 updates per second.

The second case study monitors a global feature, a dynamic graph's connected components. We use similar algorithmic ideas as before to exploit the parallelism in the problem and provided by the hardware architecture. On a 16 million vertex graph, we obtain rates of up to 240 000 updates per second on 32 processors of a Cray XMT. For the large scale-free graphs typical in our applications, 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 new framework is the first to handle real-world data rates, opening the door to higher-level analytics such as community and anomaly detection.

**Keywords:** Graph algorithms, social network analysis, massive multithreading, streaming, Cray XMT.

Parts of this work have been published in a preliminary form in the proceedings of the 2010 Workshop on Multithreaded Architectures and Applications (MTAAP 2010), in conjunction with the 24th IEEE International Parallel & Distributed Processing Symposium [13], and in the proceedings of the 2011 Workshop on Multithreaded Architectures and Applications (MTAAP 2011), in conjunction with the 25th IEEE International Parallel & Distributed Processing Symposium [14].

## I. Introduction

The data deluge from a wide range of application domains from business and finance to computational biology and computer security requires development of new analysis tools and algorithms to retrieve the information stored within the data. With the Facebook user base containing over 500 million people [17], Twitter boasting more than four billion tweets [32], over 150 million tracked blogs on the Internet [8], and the NYSE processing over four billion traded shares per day [27], massive graph data sets must be analyzed faster than ever before. Studying these massive data sets will lead to insights about community structure and anomaly detection unavailable within smaller samples.

Current large graph analysis tools like Pajek [4] are primarily designed for static graphs. For dynamic inputs these tools assume the properties to change slowly relative to execution time. Data sets from the literature like web crawls of a particular domain [2], email correspondence between colleagues [25], patent and literature citations [21], and biological networks [22] often are static and small relative to the current massive data sources. Many static graph algorithms demonstrated on these smaller data sets are difficult to scale to massive, real-world data and do not use parallel architectures efficiently. Also, existing approaches do not address interesting temporal properties of real-world, dynamic data sets.

Social networks like Facebook and Twitter as well as a variety of other networks observed in nature and human society have the *scale-free* property [26]. A scale-free graph has low diameter, and the number of neighbors to each vertex 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. 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.

We address the challenges posed by massive streaming input of spatio-temporal data represented by scale-free dynamic graphs with new algorithmic approaches and new data structures. Computing incremental updates and tolerating temporary non-coherence opens up parallel algorithmic 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 new edge or block of edges and attempt to detect significant changes in the corresponding metrics. We refer to this new approach as *massive streaming data analytics*.

To accommodate a stream of edge data, we present a new, extensible, and flexible data structure for massive graphs called STINGER (Spatio-Temporal Interaction

Networks and Graphs (STING) Extensible Representation) in Section II, where we also outline assumptions and methods for extracting parallelism when dealing with massive streaming graphs. The STINGER data structure provides a compromise between list- and array-based graph representations supporting both efficient updates and efficient analysis.

Two algorithmic case studies demonstrate the effectiveness of the streaming approach to computation and the STINGER dynamic graph representation. Section III presents the first study, computing a widely used network analysis metric called clustering coefficients. Global and local (per-vertex) clustering coefficients quantify the "small world-ness" of the graph [34]. The metric is derived from counting a graph's triangles. We present an efficient multithreaded algorithm and its implementation to calculate and maintain the clustering coefficients in an undirected, unweighted graph with a stream of input edge insertions and removals.

Section IV presents the second study, our approach for tracking connected components given a stream of edge insertions and removals. We highlight optimizations afforded by the scale-free nature of the graph. Computing the set of connected components is a well-studied graph analytical metric that is representative of both the structure of the graph as well as the connectivity between vertices. As edges are inserted into and removed from the graph, several metrics are of interest: the number of connected components, the mapping between vertices and components, the size distribution of components, and the point in time when components merge or separate. In scale-free networks, we often observe that most vertices lie in a single, large component while a large number of small components consist of very few vertices. As edges are inserted and deleted over time, most changes occur inside of a given component joining low degree vertices to high degree vertices.

The multithreaded platforms we use in our tests, both the massively multithreaded Cray XMT as well as the more common platform built on Intel's Nehalem architecture, are described in Section V together with their idiosyncracies and our implementation. Experimental results on the Cray XMT are presented in Section VI, with a comparison to the Intel Nehalem platform for the clustering coefficient problem. Despite the challenges posed by the input data, we show that the scale-free structure of social networks can be exploited to accelerate local and global graph analysis 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. Regarding clustering coefficients, a similar rate of 200 000 updates per second can be maintained on the Cray XMT. Related work is described in Section VII.

Our new framework built on incremental computation rather than recomputation enables scaling to massive data sizes on massively parallel, multithreaded supercom-

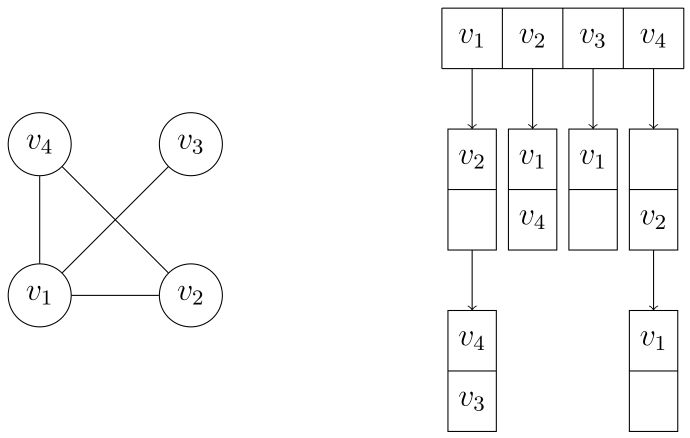


Fig. 1. An undirected graph and an example representation in STINGER with edge blocks holding two edges. The blanks are holes storing negative numbers rather than end vertex indices. Our tests use a block size of 100 and include additional per-vertex and per-edge data.

puting architectures. 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.

## II. STINGER: A General-Purpose Data Structure for Dynamic Graphs

### A. *Related Graph Data Structures*

Traditional graph data structures choose between efficient traversal or efficient modification. For example, a full adjacency matrix permits $O(1)$ edge insertion or removal but requires $O(n^2)$ storage and $O(n)$ time to traverse all edges from any vertex, where $n$ is the number of vertices. Adjacency lists or arrays require only $O(n+m)$ storage, where $m$ is the number of edges, and permit $O(d_v)$ traversal of edges out of vertex $v$, where $d_v$ is the degree of vertex $v$. Modifying the graph, however, can require $O(n+m)$ time for arrays, whereas lists have the drawback of poor locality in memory. The primary traditional approach for representing dynamic graphs uses a linked list for storing end vertices. Insertion and deletion while supporting concurrent readers is well-understood [33]. However, list traversal is expensive, and many graph analysis kernels spend most of their time traversing the edge lists.

Data structures that focus on graph traversal store edges or destination vertices in a packed array. The most common high-performance structure for static graph analysis borrows from sparse matrices and uses a compressed sparse row format (CSR). In CSR form, each edge's end vertex is stored in a single, packed array within a contiguous section corresponding to the edge's source vertex. Inserting or deleting edges requires changing the end vertex array's length and shifting data throughout that array. This not only requires a large amount of data motion but also complicates concurrent access by readers. For more related work see Section VII-B.

## B. The STINGER Data Structure

The STINGER data structure we propose here has been developed to support a wide variety of algorithms and efficient edge insertion and deletion with concurrent readers in massive graphs with a single in-memory structure. STINGER is a compromise that permits dynamic updates while supporting a wide variety of analytical algorithms on a single copy. It takes the efficient element of CSR, storing end vertices in arrays, and loosens other requirements. STINGER also borrows from the list structure and stores edge end vertices as a list of arrays. Each vertex points to a list of fixed-size end vertex arrays; see Figure 1 for a sketch of how STINGER represents a small graph. This is a common mechanism for representing dynamically sized lists or arrays while supporting rapid traversal.

The arrays are permitted to have holes or blanks represented by storing a negative entry in the end vertex slot. To delete an edge, the end vertex is found and replaced by a negative number. Inserting an edge requires replacing an empty slot or possibly adding a new edge block into the linked list. We assume that a single process manages all graph updates and ensures writing does not suffer from race conditions.

Insertion and deletion can occur concurrently with reader access. By default, low-level consistency is not enforced. In a massive sparse graph, graph updates will very rarely conflict with readers. For our applications, the graph is already assumed to be an approximate model of some real-world phenomenon, and analysis kernels must account for or be resilient to some inconsistencies. The inconsistencies do not corrupt the data structure itself. Examples of tolerated inconsistencies include changing adjacency lists and edge or vertex auxiliary information. A kernel that makes multiple passes over a vertex's neighbors may see different information on each pass. None of the inconsistencies cause infinite loops or other structure access faults.

Other information is associated with each edge: a weight and the most recent time stamp. We do not use this information here and do not discuss the relevant consistency issues. STINGER models multi-graphs, graphs with multiple, distinct edges between the same vertices, by associating a numeric type value with each edge. We do not use these edge types in this paper; multiple edges are treated as a single connection. Extra information is stored with the per-vertex index, including current in- and out-degrees. The degrees are updated by atomic operations but are not necessarily consistent with respect to the edge list. Analysis kernels must handle extra or missing edges when walking the edge list.

## C. Finding Parallelism in Streams and Analytics

We consider a single, unified input stream of edge insertions and deletions. This provides a synchronization point for analysis but also a bottleneck. For high performance, we need both to expose parallelism within the analytic kernels and to extract some parallelism from the sequential stream for updating the STINGER structure. We make two primary assumptions that help extract parallelism from streaming data: Changes in the stream are scattered widely enough in the massive graph that batches of them are sufficiently independent to expose parallelism. Analysis kernels have small support and small effect, and so a change to the graph only requires access to local portions and affects only a small portion of the output.

To extract parallelism from the stream, we assume the changes are somewhat scattered in the graph. In a low-diameter graph with high degree vertices, as is the case with many social networks, changes may not be completely independent, but there is potential for updating separate STINGER edge lists simultaneously. Considering the stream as batches of updates loses temporal resolution but exposes more parallelism in data structure and kernel updates. If the graph updates do not interact, then there is little temporal information lost by executing the updates together.

Analytical kernels with small support lend themselves to similar scattering across the graph. For example, per-vertex scores that depend on a fixed radius like Section III's local clustering coefficients naturally parallelize over batches of affected vertices. On graphs with millions or billions of vertices, the number of changes to the vertex scores will be only slightly more than the batch size for a particular set of edge insertions or deletions.

Large-support kernels like $k$-betweenness centrality [23] pose a more difficult challenge. They depend on paths potentially crossing the entire graph and require large-scale recalculation. A small change may update analysis results across the entire graph. Experience with $k$-betweenness centrality performance leads us to limit ourselves currently to kernels with small to medium support.

We expect typical massive graph streaming analytics to fit into the following framework:

```
1: Take a section of the incoming stream as a batch.
2: Split the batch into per-vertex STINGER updates.
3: If necessary, save data (e.g. degrees) to permit incremental computation.
4: Process all the data structure updates.
5: Update analytics on the altered portion of the graph.
6: Transfer changed results to a monitoring process.
```

Sections V-C and VI-A investigate steps 2–5 for a simple analytic: local clustering coefficients. In the second case study, see Sections V-D and VI-B, these steps are considered when monitoring a global graph property, the connected components.

## III. Algorithm for Updating Clustering Coefficients

### A. *Generic Algorithm*

Clustering coefficients measure the density of closed triangles in a network and are one method for determining if a graph is a small-world graph [34]. We adopt the terminology of [34] and limit our focus to *undirected* and unweighted graphs. 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, for example the triplet $(m, v, n)$ in Figure 2. A closed triplet is defined as three vertices in which there are three edges, or Figure 2's triplet $(i, v, j)$. A triangle is made up of three closed triplets, one for each vertex of the triangle.

The global clustering coefficient $C$ is a single number describing the number of closed triplets over the total number of triplets,

$$C = \frac{\text{number of closed triplets}}{\text{number of triplets}} = \frac{3 \times \text{number of triangles}}{\text{number of triplets}}. \tag{1}$$

The local clustering coefficient $C_v$ is defined similarly for each vertex $v$,

$$C_v = \frac{\text{number of closed triplets centered around } v}{\text{number of triplets centered around } v}. \tag{2}$$

Let $N(v)$ be the set of neighbors of vertex $v$, possibly containing a self-loop. Also let $|N(v)|$ be the size of set $N(v)$. The degree of $v$ is denoted by $d_v$, $d_v = |N(v)|$. Using the triangle count $T_v$ (which counts each triangle exactly twice), the local clustering coefficient $C_v$ can be expressed as

$$C_v = \frac{\sum_{u\in N(v)} |N(u) \cap (N(v)\backslash\{v\})|}{d_v(d_v - 1)} = \frac{T_v}{d_v(d_v - 1)}. \tag{3}$$

To update $C_v$ as edges are inserted and deleted, we maintain the values $d_v$ and $T_v$ separately.

For the remainder of this section, we concentrate on the calculation of local clustering coefficients. Computing the global clustering coefficient requires an additional sum reduction over the numerators and denominators.

An inserted edge increments the degree of each adjacent vertex, and a deleted edge decrements the degrees. Updating the triangle count $T_v$ is more complicated. Algorithm 1 provides the general framework. Acting on edge $\langle u, v\rangle$ affects the degrees only of $u$ and $v$ but may affect the triangle counts of all neighbors. With atomic increment operations available on most high-performance platforms, loops in Algorithm 1 can be parallelized fully.

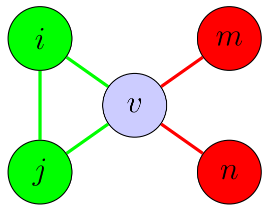


Fig. 2. There are two triplets around $v$ in this unweighted, undirected graph. The triplet $(m, v, n)$ is open, there is no edge $\langle m, n\rangle$. The triplet $(i, v, j)$ is closed.

The search in line 5 can be implemented many different ways. A brute-force method simply iterates over every element in $N(v)$ for each $x$, explicitly searching for all new closed triplets given a new edge $\langle u, v\rangle$. The running time of the algorithm is $O(d_u d_v)$, which may be problematic when two high-degree vertices are affected.

**Algorithm 1** An algorithmic framework for updating local clustering coefficients. Loops can use atomic increment and decrement instructions to decouple iterations.

**Input:** Edge $\langle u, v\rangle$ to be inserted (+) or deleted (−), local clustering coefficient numerators $T$, and degrees $d$

**Output:** Updated local triangle counts $T$ and degrees $d$

```
1: $d_u \leftarrow d_u \pm 1$
2: $d_v \leftarrow d_v \pm 1$
3: $count \leftarrow 0$
4: for all $x \in N(v)$ do
5:   if $x \in N(u)$ then
6:     $T_x \leftarrow T_x \pm 1$
7:     $count \leftarrow count \pm 1$
8: $T_u \leftarrow T_u \pm count$
9: $T_v \leftarrow T_v \pm count$
```

If the edge list is kept sorted as in a static computation, the intersection could be computed more efficiently in $O(d_u + d_v)$ time. However, the cost of keeping our dynamic data structure sorted outweighs the update cost. We can, however, accelerate the method to $O((d_u + d_v)\log d_u)$ by sorting the current edge list of $d_v$ and searching for neighbors with bisection. The sorting routine can employ a parallel sort, and iterations of the search loop can be run in parallel given atomic addition / subtraction operations.

## *B. Approximating Clustering Coefficients using a Bloom Filter*

We present a novel set intersection approximation algorithm with constant-time search and query properties and an extremely high degree of accuracy. In addition to our dynamic data structure described in Section II, we summarize neighbor lists with Bloom filters [9], a probabilistic data structure that gives false positives (but never false negatives) with some known probability.

Edge arrays could be represented as bit arrays. In one extreme, each neighbor list could be an array using one bit per vertex as well as an edge list. Then $|N(u) \cap N(v)|$ can be computed in $O(min\{d_u, d_v\})$ time by iterating over the shorter edge list and checking the bit array. However, maintaining $O(n)$ storage per source vertex is infeasible for massive graphs. Instead, we approximate an edge list by inserting its vertices into a Bloom filter. While a Bloom filter is also a bit array, it uses a smaller number of bits.

Each edge list $N(v)$ is summarized with a Bloom filter for $v$. A hash function maps a vertex $w \in N(v)$ to a specific bit in this much smaller array. With fewer bits, there may be hash collisions where multiple vertices are mapped to the same bit. These will result in an overestimate of the number of intersections.

A Bloom filter attempts to reduce the occurrence of collisions by using $k$ independent hash functions for each entry. When an entry is inserted into the filter, the output of the $k$ hash functions determines $k$ bits to be set in the filter. When querying the filter to determine if an edge exists, the same $k$ hash functions are used and each bit place is checked. If any bit is set to 0, the edge cannot exist. If all bits are set to 1, the edge exists with a high probability.

Bloom filters have several parameters useful to fix a given probability of failure. In-depth description of Bloom filter theory is beyond the scope of this paper, but a few useful features include the following: Bloom filters never yield false negatives where an edge is ignored, only false positives where a non-existent edge is counted. The probability of falsely returning membership is approximately $(1 - e^{-kd_u/m})^k$ where $m$ is the length of the filter. This can be optimized by setting $k$ to an integer near $\ln 2 \cdot m/d$ [18], choosing $d$ according to the expected degrees in the graph. Our initial implementation uses two hash functions, $k = 2$, and a 1 MiB filter. The probability of a false-positive will vary depending on the degree of the vertex. In a scale-free graph with an average degree of 30 and a maximum degree of 200,000, the average false-positive rate will be $5 \cdot 10^{-11}$ and the worst-case rate will be $2 \cdot 10^{-3}$.

When intersecting two high-degree vertices, the Bloom filter holds a slight asymptotic edge over sorting one edge list, but a multithreaded implementation benefits from additional parallelism throughout the Bloom filter's operations. Note that entries cannot be deleted from a Bloom filter. A new filter is constructed for each application of Algorithm 1, so we never need to delete entries.

Modifications to Algorithm 1 for supporting a Bloom filter are straight-forward. After line 3, initialize the Bloom filter using vertices in $N(u)$, the filter's bit array $B$, and hash functions $H_i$, $1 \leq i \leq k$:

```
1: $B \leftarrow (0, \ldots, 0)$
2: **for all** $y \in N(u)$ **do**
3:   **for** $i = 1 \rightarrow k$ **do**
4:     $B[H_i(y)] \leftarrow 1$
```

Then implement the search in line 5 as follows:

```
1: **for** $i = 1 \rightarrow k$ **do**
2:   **if** $B[H_i(x)] = 0$ **then**
3:     Skip to next $x$
```

Our multithreaded implementation of the complete algorithm for the Cray XMT is described in Section V-C. Before that, we present our second case study, the monitoring of connected components in massive streaming graphs with the scale-free property.

## IV. Tracking Connected Components in Scale-Free Graphs

As argued in the introduction, scale-free networks pose particular challenges to parallel analytics software. Despite these challenges, we will show that the scale-free structure of social networks can be exploited to accelerate graph connectivity queries in light of a stream of deletions.

To monitor the connected components of a general graph, 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. As a result, breadth-first search is often used to re-establish or rule out connectivity between two vertices after a deletion.

### 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.* do 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 a 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 [13]. 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. The Algorithm in Detail*

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 to motivate our algorithm.

The pseudocode of our algorithm appears in Algorithm 2. 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 main 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 are handled first, followed by insertions. This ordering creates space in the data structure before

**Algorithm 2** 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 $R_{\text{thresh}}$, number of relevant deletions $R$, bit array $A$, component graph $C$

**Output:** Updated component membership $M'$

```
1:  $\mathrm{Sort}(B)$                                   ▷ Phase 1: Prepare batch
2:  for all $b \in B$ in parallel do
3:    if $b$ is deletion then
4:      $\mathrm{Push}(Q_{\text{del}}, b)$
5:    else
6:      $\mathrm{Push}(Q_{\text{ins}}, b)$
7:  $\mathrm{StingerDeleteAndInsertEdges}(B)$           ▷ Phase 2: Update data structure
8:  for all $b \in Q_{\text{del}}$ in parallel do            ▷ Phase 3: Process deletions
9:    $\langle u, v\rangle \leftarrow b$
10:   $A_u \leftarrow \vec{0}$                                 ▷ All bits set to zero
11:   for all $n \in \mathrm{Neighbors}(u)$ in parallel do
12:     Set bit $n$ in $A_u$ to 1
13:   $F \leftarrow 0$
14:   for all $n \in \mathrm{Neighbors}(v)$ in parallel do
15:     if bit $n$ in $A_u = 1$ then
16:       $F \leftarrow 1$                                     ▷ Triangle found
17:   if $F = 0$ then
18:     atomic $R \leftarrow R + 1$                            ▷ No triangles found
19: if $R > R_{\text{thresh}}$ then
20:   $R \leftarrow 0$
21:   $M' \leftarrow \mathrm{ConnectedComponents}(G)$
22: else                                                ▷ Phase 4: Process insertions
23:   $C \leftarrow \emptyset$
24:   for all $b \in Q_{\text{ins}}$ in parallel do
25:     $\langle u, v\rangle \leftarrow b$
26:     Add $\langle M[u], M[v]\rangle$ to $C$
27:   $T \leftarrow \mathrm{ConnectedComponents}(C)$
28:   for all $v \in V$ in parallel do
29:     $M'[v] \leftarrow T[v]$
```

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 in Phase 3 to see if they disrupt connectivity. 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 $R_{\text{thresh}}$ determined by the tolerance for inconsistency before recomputation, we will recompute 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 $R_{\text{thresh}}$ 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 [7]. 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

[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.

breadth-first searches. When collisions between colors are no longer produced, the remaining colors specify the components.

### *C. Discussion*

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. We avoid running any breadth-first search by first constructing bit arrays to represent neighbor lists for vertices affected by a deletion in a batch. We take advantage of the low diameter nature of scale-free networks by intersecting these bit arrays to quickly re-establish connectivity using triangles. Our algorithm uses a breadth-first search of the smaller component graph rather than the vertex graph, and is limited by the size of 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 novel methods 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. Thus, our proposed approach to tracking connected components minimizes graph traversal and static recomputation.

## V. Implementation

### *A. Multithreaded Platforms*

Our implementation is based on multithreaded, shared-memory parallelism. The single code base uses different compiler directives, or pragmas, to expose the threaded parallelism. We use Cray's compiler (version 6.3.1) and its pragmas for the massively multithreaded Cray XMT, and we use OpenMP [28] via the GNU C compiler (version 4.4.1) for a comparison on an Intel Nehalem E5530-based commodity platform.

The Cray XMT is a supercomputing platform designed to accelerate massive graph analysis codes. The architecture tolerates high memory latencies using massive multithreading. There is no cache in the processors; all latency is handled by threading. Each Threadstorm processor within a Cray XMT contains 128 *hardware streams* that maintain a thread context. Context switches between threads occur every cycle, with a new thread selected from the pool of streams ready to execute.

A large, globally shared memory enables the analysis of graphs on the order of one billion vertices using a well-understood programming model. Hashing is used to break up locality and reduce hot-spotting. Synchronization takes place at the level of 64-bit words, and lightweight primitives such as atomic fetch-and-add are provided to the programmer. The cost of synchronization is amortized over the cost of memory access. Combined, these features enable the algorithm designer to implement highly scalable parallel algorithms for analyzing massive graphs.

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 12,000 hardware thread contexts. The globally addressable shared memory totals 1 TiB.

The Intel Nehalem E5530 is a 2.4GHz quad-core processor with *hyperthreading* [3]. Each physical core holds two thread contexts and switches on memory stalls. The context switches are not as frequent as on the Cray XMT, and only two contexts are available to hide memory latencies. Each core has 256 KiB of level two cache, and each processor module shares 8 MiB of level three cache. The platform tested has two E5530s, a total of eight cores and 16 threads, with 12 GiB of main memory.

The experiments presented in Section VI consist of three cases: the two applications for computing local clustering coefficients and tracking connected components, respectively, and the STINGER data structure implementation. Each takes advantage of the unique features of the Cray XMT in different ways, which we detail in the upcoming sections.

## *B. The STINGER Data Structure*

Our STINGER 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 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 on the Cray XMT.

The block data format of STINGER enables an additional level of parallelism on the Cray XMT. Blocks can be traversed in parallel while using atomic fetch-and-add instructions to update shared variables.

## C. Multithreaded Implementation of Algorithm 1 (Clustering Coefficients)

Our threaded implementation is straightforward. Each undirected edge $\langle u, v \rangle$ is added to or removed from the data structure using two threads, one to work from each end vertex. No explicit locking is involved. The STINGER structure requires only ordered, atomic read/write of 64-bit integers (*e.g.* end vertices, timestamps) and atomic increment/decrement of counters (*e.g.* degrees, offsets). Both the Cray intrinsics and the OpenMP pragmas can express the specific operations we need. For brevity, we use the GCC/Intel intrinsic functions similar to the Cray intrinsics.

The algorithm to update the triangle counts $T$ above uses appropriate pragmas to parallelize the outer loops. Inner loops are not parallelized under OpenMP; the target platform has insufficient threading resources to benefit from that level of parallelism. However, the inner loops are parallelized on the XMT by a loop collapse [29]. An atomic increment updates the count of a shared neighbor $T_w$.

Local clustering coefficients' properties help us batch the input data. Recomputing changed coefficients only at the end of the batch's edge actions frees us to reorder the insertions and deletions. Reordering repeated insertions and removals of the same edge may alter the edge's auxiliary data, however, so we must take some care to resolve those in sequence order. After resolving actions on the same edge, we process all removals before all insertions to open edge slots and delay memory allocation.

The batch algorithm is as follows:

```
1: Transform undirected edges $\langle i, j \rangle$ into pairs of directed edges $i \to j$ and $j \to i$
   because STINGER stores directed edges.
2: Group edges within the batch by their source vertex.
3: Resolve operations on the same edge in sequence order.
4: Apply all removals, then all insertions to the STINGER structure.
5: Recompute the triangle counts and record which vertices are affected.
6: Recompute the local clustering coefficients of the affected vertices.
```

In Step 5, we use slight variations of the previous algorithms. The source vertex's neighboring vertices are gathered only once, and the array is reused across the inner loop. The sorted update and Bloom filter update strategies compute their summary data using the source vertex rather than choosing the larger list.

## D. Multithreaded Implementation of Algorithm 2 (Connected Components)

Fine-grained synchronization on the Cray XMT enables the connected components code to fully parallelize all loops. Histograms and shared queues can be built in parallel with atomic fetch-and-add instructions. Bit arrays can be built in parallel for different vertices at the coarse level, and neighbors can be set within the individual bit array in parallel using word-level synchronization.

The bit array is a fast data structure that, with the support of fine-grained synchronization, can be built and queried in parallel. Building 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. A Bloom filter, however, cannot be reused since it does not support deletions.

Batching of the update stream is required on the Cray XMT to create adequate parallelism for the thousands of user hardware streams. 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, it is not necessary to allocate memory during processing.

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.

## VI. Experimental Results

### A. Clustering Coefficient Experiments

Our test data is generated by the R-MAT recursive matrix generator [11] with probabilities $A = 0.55$, $B = 0.1$, $C = 0.1$, and $D = 0.25$. Each generated matrix has a few vertices of high degree and many vertices of low degree. Given the R-MAT scale $k$, the number of vertices $n = 2^k$, and an edge factor $f$, we generate $e = f \cdot n$ unique edges for our initial graph. We then select a fraction $\rho e$ of those edges and add them to a deletion queue. For these experiments, $\rho = 1/16$.

After generating the initial graph, we generate 1024 *actions* (edge insertions or deletions) for edge-by-edge runs and 1 million actions for batched runs. With probability $\rho$, a deletion is taken from the deletion queue. Otherwise an insertion is generated with the same R-MAT generator and parameters. The edge to be inserted may already exist in the graph. Inserted edges are appended to the deletion queue with probability $\rho$. There are no self-loops in our generated edges, but the algorithm implementations do handle self-loop cases by ignoring edges $\langle v, v\rangle$.

The Cray XMT applies to far larger problems than the Intel-based platform. The latter is limited to scale $k = 21$ and edge factor $f = 16$ with our current test harness. On the Cray XMT, our current experiments are sixteen times larger, with $k = 24$ and $f = 32$, and are limited more by our testing code's structure than the Cray XMT's architecture.

| Algorithm | Update complexity |
|---|---|
| Brute force | $O(d_u d_v)$ |
| Sorted list | $O((d_u + d_v) \log d_u),\ d_u < d_v$ |
| Bloom filter | $O(d_u + d_v)$ |

TABLE I
SUMMARY OF UPDATE ALGORITHMS

*1) Scalability of the Initial Computation:* We begin by computing the correct clustering coefficients for our initial graph. While not the focus of this paper, performance on the initial computation shows interesting behavior on the two test platforms.

The initial clustering coefficients algorithm simply counts all triangles. For each edge $\langle u, v \rangle$, we count the size of the intersection $|N(u) \cap N(v)|$. This is a static computation, so we use a packed representation with sorted edge arrays for efficiency. The algorithm as a whole runs in $O(\sum_v d_v^2)$ time, where $v$ ranges across the vertices and the structure is pre-sorted. The multithreaded implementation also is straight-forward; we parallelize over the vertices. The $2^{21} \approx 2$ million vertices in the smallest case provide sufficient parallelism for both platforms.

The initial computation has not been seriously tuned for performance on either platform. The algorithm itself is somewhat coarse-grained with sufficiently sized chunks of work to amortize run-time overhead. In Figure 3, the Cray XMT's performance improves with increasing processors. Thread creation and management overhead appears responsible for the Intel Nehalem's decreasing performance.

*2) Number of Individual Updates per Second:* Unlike calculating the triangle counts $T$ for the entire graph, updating $T$ for an individual edge insertion or deletion exposes a variable amount of fine-grained parallelism. We present results showing how aggregate performance of a *single* edge insertion or deletion stays relatively constant.

Table I summarizes the sequential complexity of our update algorithms. Figure 4 presents boxplots summarizing the updates per second achieved on our test platforms. Figure 5 shows local recomputation's speed-up of locally relative to globally recomputing the graph's clustering coefficients. Boxes in Figures 4 and 5 span the 25% – 75% quartiles of the update times. The bar through the box shows the median. The lines stretch to the farthest non-outlier, those within 1.5× the distance between the median and the closest box side. Points are outliers.

In Figure 4, we see the Cray XMT keeps a steady update rate on this relatively small problem regardless of the number of processors. The outliers with 16 processors are a result of sharing resources with other users. The Bloom filter shows the least variability in performance. Figure 5 shows that local recomputation accelerates the

Fig. 3. Performance of the initial clustering coefficient computations, normalized for problem size by presenting the number of edges in the graph divided by the total computation time. The Cray XMT scales well as additional processors are added, while the Nehalem platform's threading overheads decrease performance.

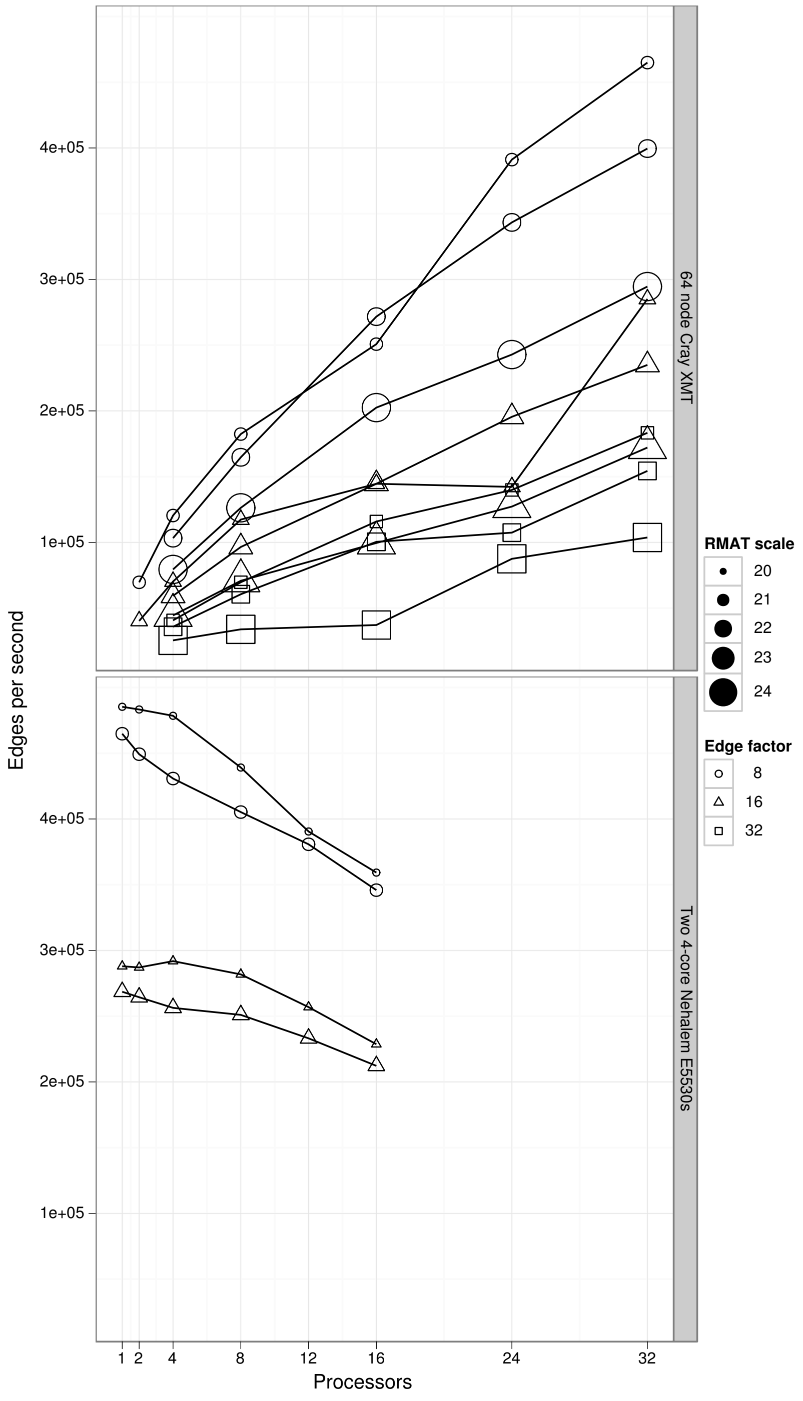


Fig. 4. Updates per second by algorithm.

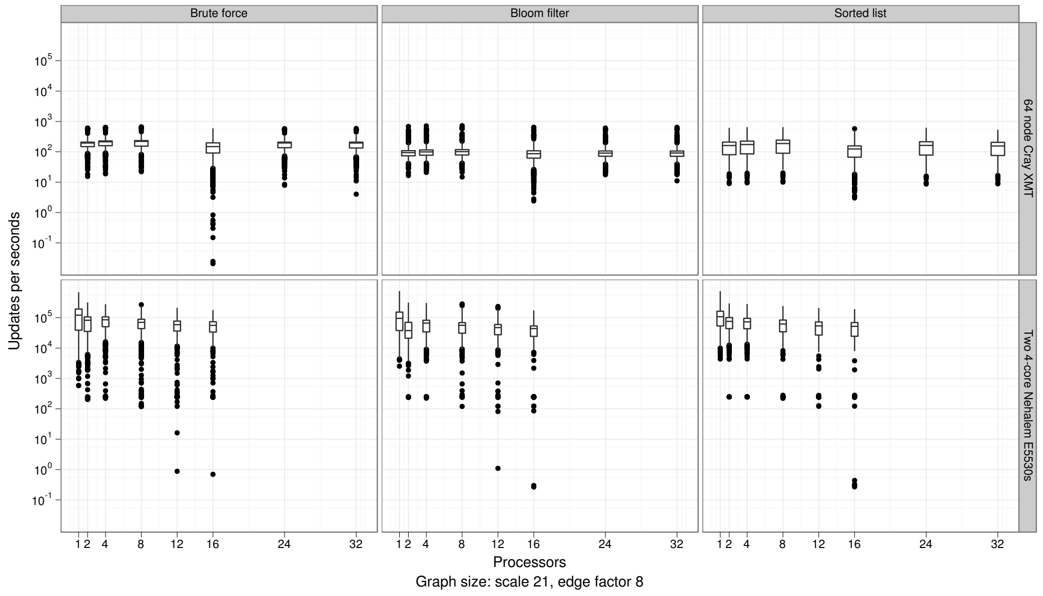
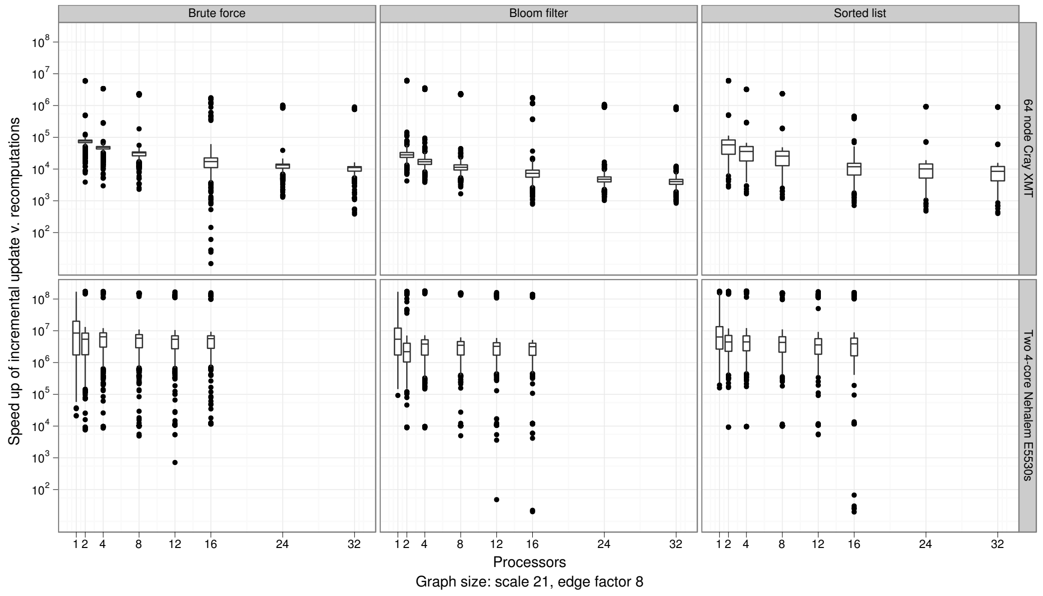


Fig. 5. Speed up of incremental, local updates relative to recomputing over the entire graph.



update rate typically by a factor of over a thousand.

The Nehalem results degrade with additional threads. The noise at 12 and 16 threads results from over-allocation and scheduling from hyperthreading. The Nehalem outperforms the Cray XMT by several orders of magnitude, but can only hold a graph of approximately 8 million vertices and 300 million edges. In comparison, this Cray XMT is capable of holding a graph in memory up to 1 billion vertices and 20 billion

| Batch Size | Brute force | Bloom filter |
|---|---|---|
| Edge by edge | 90 | 60 |
| Batch of 1000 | 25,100 | 83,700 |
| Batch of 4000 | 50,100 | 193,300 |

TABLE II
COMPARISON OF SINGLE EDGE VERSUS BATCHED EDGE OPERATIONS ON 32 XMT PROCESSORS, R-MAT 24 INPUT, IN UPDATES PER SECOND

| | 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 |

TABLE III
UPDATES PER SECOND ON A GRAPH STARTING WITH 16M VERTICES AND APPROXIMATELY 135M EDGES ON 32 PROCESSORS OF A CRAY XMT.

edges.

Table II shows performance obtained from batching operations and extracting parallelism on the Cray XMT. The sorting algorithm was not considered for batching. Notice that increasing the batch size greatly improves performance. For the Bloom filter, this comes at the cost of a proportional increase in memory footprint. A batch size of 4000 required choosing a filter size of 1 MiB to fit within the system's available memory. Even so, we encountered no false positives over 1 million edge actions. Increasing the batch size intuitively improves scalability since data parallelism is increased in the update phase.

### *B. Connected Components*

The experimental results presented in this section use synthetic R-MAT [11] input graphs derived by sampling from a Kronecker product. We generate two graph with $2^{20} \approx 1$ million vertices or $2^{24} \approx 16$ million vertices, both with an edge factor of 8. Again, we use the R-MAT parameters $a = 0.55$, $b = 0.1$, $c = 0.1$, and $d = 0.25$. For each graph we generate additionally 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. Again, with probability $\rho = 1/16$, 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 $\rho = 1/16$.

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. Since data available to researchers do not include deletions, we resort to artificial networks in order to fully demonstrate our ability to track connected components in a streaming fashion. We are unaware of any other edge stream generators for social networks, which is an area ripe for further study.

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.

Looking closely at the algorithm, one will note that when handling insertions only, the graph data structure does not need to be accessed to update the connected components. 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 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. In Figure 6, 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 Table III 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 recompute 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 $R_{\text{thresh}}$ meaning that we will tolerate up to $R_{\text{thresh}}$ 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 $R_{\text{thresh}}$ will determine performance between the lower bound and this rate. In practice, reasonable values of $R_{\text{thresh}}$ 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 component structure of 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 $R_{\text{thresh}}$ 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 6 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 6'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

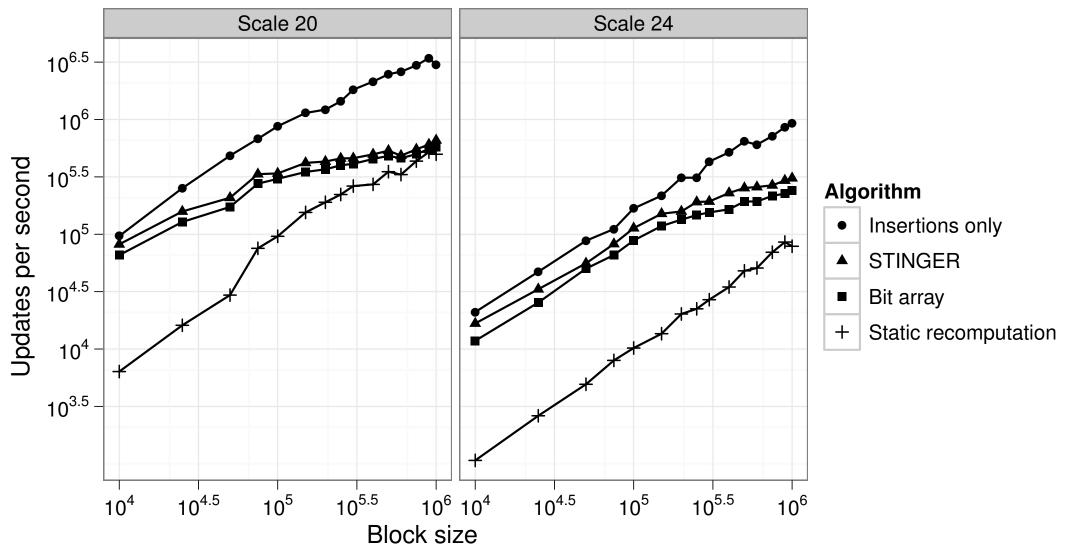


Fig. 6. Update performance for a synthetic, scale-free graph with 1 million vertices (left) and 16 million vertices (right) and edge factor 8 on 32 processors of a 128 processor Cray XMT.

bit array performance again closely tracks that of the data structure. At this size, we can observe that the static recomputation method is no 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 $\mathrm{O}(n)$ 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.

## VII. Related Work

### A. Streaming Data

The terms *streaming* [1] and *semi-streaming* [19] in related literature describe an algorithmic model of computation with very restrictive properties on data accesses. In streaming graph algorithms, the graph edges are read one-by-one in an arbitrary, unknown order. Streaming algorithms typically are limited to storing $O(n)$ or $O(n \operatorname{polylog} n)$ data, where $n$ is the number of vertices, and taking at most $O(\log k)$ many passes over the $k$-long input edge stream. The metric of interest must be maintained or approximated without access to data other than the edge being observed at any given point in time and the relatively small summary. Streaming models have been applied to the approximation of local clustering coefficients [5], [10]. Current high-performance computer platforms like the Cray XMT and IBM Power 595 support enough main memory to store a significant amount of graph data at once and provide fairly uniform, fast access to that memory. On these platforms, the streaming model's restrictions are overly conservative. Our massive streaming data approach leverages the continued growth in available memory. Also, we assume an effectively infinite input stream and do not take multiple passes over the full input data. The algorithmic streaming model does inspire per-batch approximations.

### B. Graph Data Structures

Alternative graph data structures for dynamic graphs beyond the ones described in Section II-A include forms of binary trees [24]. Trees pay an extra cost in keeping some order on the edges. On one target platform, the Cray XMT, the maintenance cost is substantial and prohibitive. STINGER's linked array structure, permits simple multithreaded traversal and maintenance. Similar work in cache-oblivious algorithms often uses trees where the leaves are ordered arrays with blank entries [6]. The blank entries limit data movement when inserting a new edge into the ordered array. We are investigating whether STINGER can take advantage of a similar technique for accelerating intersections of edge lists. More radical alternatives exist, including representations using sparse certificates specific to different analysis kernels [15]. We consider the multi-use STINGER data structure a better compromise between performance and extensibility to new applications.

### C. Tracking Connected Components

Several algorithms using breadth-first searches have been proposed for tracking connected components in case of deletions. Even and Shiloach [31] spawn two breadth-first searches for each edge deletion. One verifies that the component remains intact while the other determines if the component has been separated. Roditty and Zwick [30] stores the graph as a sequence of graphs that are created with each subsequent insertion.

A Union-Find or Least Common Ancestor algorithm establishes connectivity. Henzinger et al. [12] 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. A scale-free graph's small diameter implies that a single breadth-first search is an expensive operation that quickly consumes the entire graph. Running one or more traversals for each edge deletion where deletions are unlikely change the component structure will not support the high data rates of today's input streams.

Another approach partitions the graph to determine components. Henzinger and King [20] 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. [16] use partitions according to the average degree and sparsification techniques to maintain a minimum spanning forest of components.

For the social networks of interest, creating a spectrum of subgraphs from dense to sparse is difficult. Facebook, with 500 million vertices and average degree 130, is already very sparse. The Henzinger and King method would create only 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.

## VIII. Conclusions and Future Work

Our new *massive streaming data analytics* framework for graph analysis and speeds-up two fundamental analysis metrics compared to traditional static recomputation. To store the graph data, we have offered STINGER as a compromise between efficient graph traversal and flexible edge insertions and removals for a wide variety of graph algorithms. We have presented clustering coefficients and connected components as case studies for STINGER and our new analytical framework. Our framework's implementation runs on typical multicore systems like the Intel Nehalem using OpenMP as well as the massively multithreaded Cray XMT and performs well in both environments.

Our algorithms assume that the cost of processing a deletion is high compared to the cost of an insertion. Tolerate a small, *temporary* inconsistency with the static result permits handling deletions lazily and proceeding at insertion speed. When few deletions change the measure of interest, quickly ruling out the inconsequential deletions keeps the temporary inconsistency small and amortizes costly recomputations over many input batches.

Computing clustering coefficients demonstrates our approach's effectiveness. Updating after each edge insertion or deletion duplicates setup time, so the sorted update and

Bloom filter algorithms perform relatively poorly. However, the serial stream contains enough parallelism when batched to exploit the Cray XMT's massively multithreaded architecture. We achieve a speed-up of $550\times$ over edge-by-edge updates. The update rates of nearly $200\,000$ updates per second almost match gigabit Ethernet packet rates.

False positives from Bloom filters may introduce an approximation. A modestly sized filter produces an exact result with no false positives from our sampled scale-free networks. The Bloom filter approach achieved a $4\times$ speed-up over the brute-force method on the Cray XMT for the clustering coefficient case study.

Existing practice shows that the any streaming connected components algorithm should minimize the number of graph traversals necessary to establish or verify connectivity following a deletion. We have contributed a bit array intersection method to reduce drastically the number of deletions that require expensive checks if they cause structural change to the graph. Our experiments on synthetic graphs with millions of vertices and hundreds of millions of edges show 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.

Further work is needed to identify fast data-dependent methods to isolate only those deletions that cause changes in the metric of interest. Regarding connected components, the overwhelming majority of new or deleted edges cause no change in the number or size of connected components for a given batch of insertions and deletions. 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. Also, more complex algorithmic kernels will yield deeper insights about the structure of streaming networks, for example for community and anomaly detection.

## Acknowledgments

This work was supported in part by the PNNL CASS-MT Center and NSF Grants CNS-0614915 and IIP-0934114. We thank PNNL and Cray for providing access to Cray XMT platforms. We are grateful to Karl Jiang, Kamesh Madduri, Daniel Chavarría, Jonathan Berry, Bruce Hendrickson, John Feo, Jeremy Kepner, and John Gilbert, for discussions on large-scale graph analysis and algorithm design.

## References

[1] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proc. 13th Ann. Symp. Discrete Algorithms (SODA-02).* San Francisco, CA: Society for Industrial and Applied Mathematics, Jan. 2002, pp. 623–632.

[2] A.-L. Barabási, "Network databases," 2007, http://www.nd.edu/~networks/resources.htm.

[3] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "A performance evaluation of the Nehalem quad-core processor for scientific computing," *Parallel Processing Letters*, vol. 18, no. 4, pp. 453–469, 2008.

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

[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *KDD '08: Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2008, pp. 16–24.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *SPAA '05: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2005, pp. 228–237.

[7] 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 (MTAAP)*, Long Beach, CA, March 2007.

[8] Blogpulse, "Blogpulse stats," March 2011, http://www.blogpulse.com/.

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

[10] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler, "Counting triangles in data streams," in *PODS '06: Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA: ACM, 2006, pp. 253–262.

[11] 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.

[12] A. T. Computational, M. R. Henzinger, V. K. T, and Y. Warnow, "Constructing a tree from homeomorphic subtrees, with," in *Algorithmica*, 1999, pp. 333–340.

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

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

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

[16] 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.

[17] I. Facebook, "User statistics," March 2011, http://www.facebook.com/press/info.php?statistics.

[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *IEEE/ACM Transactions on Networking*, 1998, pp. 254–265.

[19] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theor. Comput. Sci.*, vol. 348, no. 2, pp. 207–216, 2005.

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

[21] "Infovis databases," 2005, http://iv.slis.indiana.edu/db/index.html.

[22] H. Jeong, S. Mason, A.-L. Barabási, and Z. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, pp. 41–42, 2001.

[23] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarría-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP)*, Rome, Italy, May 2009.

[24] K. Madduri and D. A. Bader, "Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis," in *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 2009.

[25] M. Newman, "Scientific collaboration networks: II. shortest paths, weighted networks and centrality," *Phys. Rev. E*, vol. 64, p. 016132, 2001.

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

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

[28] *OpenMP Application Program Interface; Version 3.0*, OpenMP Architecture Review Board, May 2008.

[29] M. Ringenburg and S.-E. Choi, "Optimizing loop-level parallelism in Cray XMT$^{\text{TM}}$ applications," in *Cray User's Group*, May 2009.

[30] 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.

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

[32] Twitter, Inc., "Happy birthday Twitter!" March 2011, http://blog.twitter.com/2011/03/happy-birthday-twitter.html.

[33] J. Valois, "Lock-free linked lists using compare-and-swap," in *Proc. 14th Ann. ACM Symp. on Principles of Distributed Computing*, Ottowa, Canada, Aug. 1995, pp. 214–222.

[34] D. Watts and S. Strogatz, "Collective dynamics of small world networks," *Nature*, vol. 393, pp. 440–442, 1998.