Massive Streaming Data Analytics: A Case Study with Clustering Coefficients

David Ediger* Karl Jiang† Jason Riedy†
David A. Bader†
Georgia Institute of Technology

Abstract

We present a new approach for parallel massive graph analysis of streaming, temporal data with a dynamic and extensible representation. Handling the constant stream of new data from health care, security, business, and social network applications requires new algorithms and data structures. We examine data structure and algorithm trade-offs that extract the parallelism and high performance necessary for rapidly updating analysis of massive graphs. Static implementations of analysis kernels often rely on specific structure on the input data, maintaining the specific structures for each possible kernel with high data rates imposes too great a performance price. A case study with clustering coefficients demonstrates incremental updates can be 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 and our Bloom filter approaches 200 000 updates per second.

1 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 keep pace with the data, these tools must analyze the resulting interaction networks and graphs as data arrives in high-volume streams rather than in static snapshots. With the Facebook user base containing over 350 million people [13], Twitter boasting more than four billion tweets [22], and an estimated hundreds of millions of blogs on the Internet [24], the massive graph data sets must be analyzed faster than ever

*School of Electrical and Computer Engineering, Georgia Institute of Technology.
†College of Computing, Georgia Institute of Technology

before. These sources' streaming data differs fundamentally from traditional static graph data sets. Data sets from the literature often are constructed statically from web crawls of a particular domain [3], email correspondence between colleagues [20], patent and literature citations [16], and biological networks [17]. Analysis is carried out a single time on these static sets. The streaming data from social networks and other applications is too large to permit constructing streaming analysis from static snapshots. Here we investigate another approach, computing incremental updates, along with the data structures necessary and assumptions useful for achieving high-performance analysis.

Current massive graph analysis tools like Pajek [5] contain three phases. First, the tools preprocess data into appropriate data structures. Second, a graph kernel analyzes the data. Finally, the tools post-process and store analysis results for later presentation. These tools calculate static graph properties; any dynamic use assumes the properties change slowly relative to execution time. However, there is increasing interest in the temporal properties of the dynamic data set. With millions of users and billions of messages, even the preprocessing time is much larger than the time between potentially large graph changes. Repeatedly processing graphs as snapshots of a process cannot keep up with the data rates.

In this paper, we tackle these problems with a new computational approach for the analysis of complex graphs and networks with billions of vertices based on streaming input of spatio-temporal data. Our approach 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. After each new edge or block of edges, we update one or more analytical kernels and attempt to detect significant changes in these metrics. We refer to this new approach as *massive streaming data analytics*.

To accommodate a stream of edge data, we present a new, extensible data structure for massive graphs: STINGER (Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation) [1]. This data structure provides a compromise between list- and array-based graph representations to support both efficient updates and efficient analysis. STINGER provides the tools necessary for our analysis approach.

As a case study, we demonstrate the effectiveness of the streaming approach to computation and the STINGER dynamic graph representation using an important social network metric called clustering coefficients. Global and local (per-vertex) clustering coefficients give the user an idea of the "small world-ness" of the graph [26]. The metric is derived from the lower technique of triangle counting in the graph. We present efficient multithreaded algorithms to calculate and update the clustering coefficients in an undirected, unweighted graph of 17 million vertices and 537 million edges.

The terms *streaming* [2] and *semi-streaming* [15] appear in related literature to describe a 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 logarithmically many passes over the data. The metric of interest must be maintained or approximated in this fashion without significant access to data other than the edge being observed at any given point in time. Streaming models have been applied to the approximation of local clustering coefficients [6,

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. On these platforms, the streaming model's restrictions are overly conservative. Our massive streaming data approach leverages the continued growth in available memory.

This paper presents data structures to support our approach and a case study with clustering coefficients. Section 2 describes STINGER, an extensible representation that accumulates the dynamic, streaming graph and supports efficient analysis kernels. We outline assumptions and methods for extracting parallelism in Section 3. As an example of our approach, Section 4 considers updating clustering coefficients with streaming data. Section 5 details the implementation of STINGER and streaming clustering coefficients on the Cray XMT, a massively multithreaded architecture for high performance graph analysis. Using our framework, Section 6 compares our three methods for updating local clustering coefficients.

2 STINGER: A General-Purpose Data Structure for Dynamic Graphs

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, were d_v is the degree of vertex v. Modifying the graph, however, can require $O(n+m)$ time. STINGER [1] is a general-purpose graph data structure that aims to support efficient, multithreaded traversal concurrently with efficient insertion and deletion of edges.

Data structures that focus on traversal store edges or end 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.

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 [25]. However, list traversal is expensive, and many graph analysis kernels spend most of their time traversing the edge lists.

We developed the STINGER data structure to support efficient edge insertion and deletion with concurrent readers. STINGER 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. This is a common mechanism for representing

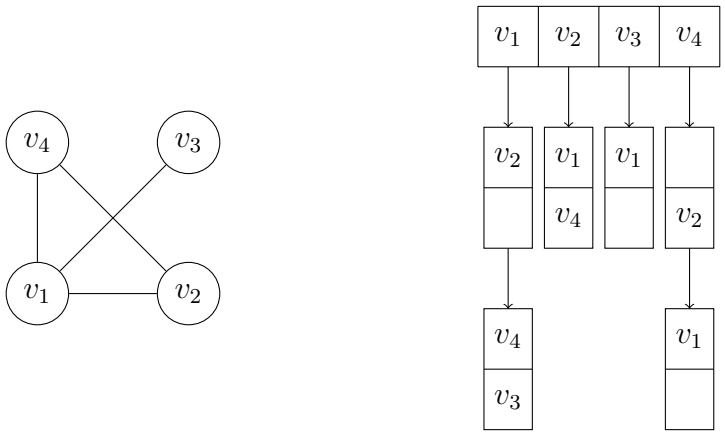


Figure 1: An undirected graph and a representation in STINGER with edge blocks holding two edges. The blanks are holes storing negative numbers rather than end vertex indices.

dynamically sized lists or arrays while supporting rapid traversal.

The arrays are permitted to have holes or blanks represented by negative entries. To delete an edge, the end vertex is found and replaced atomically 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 conflict with readers very rarely. For our applications, the graph already is assumed to be an approximate model of some real-world phenomenon. We are investigating sequence locks [9], which do not actually lock, and other light-weight techniques that allow certain consistency levels without forcing them on all users.

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 need to handle extra or missing edges when walking the edge list.

Alternative graph data structures include forms of binary trees [19]. Trees pay an extra cost in keeping some order on the edges. On our target platform, the Cray XMT, the maintenance cost is substantial and prohibitive. STINGER's linked array structure permits simple multithreaded traversal. Similar work in cache-oblivious algorithms often uses trees where the leaves are ordered arrays with blank entries [7]. 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 [12]. Our target is to support massive graphs, so we must support a wide variety of algorithms 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.

3 Finding Parallelism in Streams and Analytics

For now, 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 dig 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. The changes will not be too scattered in a low-diameter graph with high degree vertices like many social networks, but there is potential for updating separate STINGER edge lists simultaneously. Considering batches from the stream loses some 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 4's local clustering coefficients naturally parallelize over batches of affected vertices. On massive graphs, the number of changes to the vertex scores will be relatively small, only slightly more than the batch size.

Large-support kernels like k-betweenness centrality [18] 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 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 5 and 6 investigate steps 2–5 for a simple analytic, local clustering coefficients.

4 Algorithm for Updating Clustering Coefficients

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 [26]. We adopt the terminology of [26] 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 e_k be the set of neighbors of vertex k, and let $|e|$ be the size of set e. Also let d_v be the degree of v, or $d_v = |e_v|$. We show how to compute C_v by expressing it as

$$C_v = \frac{\sum_{i \in e_v} |e_i \cap (e_v \setminus \{v\})|}{d_v(d_v - 1)} = \frac{T_v}{d_v(d_v - 1)}. \tag{3}$$

To update C_v as edges are inserted and deleted, we maintain the degrees and the triangle count T_v separately.

For the remainder of the paper, 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

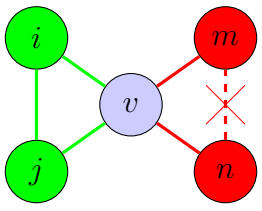


Figure 2: There are two triplets around v in this unweighted, undirected graph. The triplet (m, v, n) is open, there is no edge $\langle m, n\rangle$. The triplet (i, v, j) is closed.

may affect the triangle counts of all neighbors. With some and atomic increment operations available on most high-performance platforms, all of Algorithm 1's can be parallelized fully.

The search in line 5 can be implemented many different ways. A brute-force method simply iterates over every element in e_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 problematical when two high-degree vertices are affected.

Algorithm 1 An algorithmic framework for updating local clustering coefficients. All 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** $x \in e_v$ **do**
5: **if** $x \in e_u$ **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, that buries the update cost in the data structure and incurs too great a penalty in our dynamic structure. 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 searching loop can be run in parallel given atomic addition / subtraction operations. By sorting both edge lists forgoes exploiting fine-grained parallelism in running multiple bisection searches.

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. We summarize neighbor lists with Bloom filters [8], a probabilistic data structure that gives false positives (but never false negatives) with some known probability. We then query against this Bloom filter to determine if the intersection exists.

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 $|e_u \cap e_v|$ can be computed in $O(min\{d_u, d_v\})$ time by iterating over the shorter edge list and checking the bit array. However, the $O(n^2)$ storage is infeasible for massive graphs.

Instead, we approximate an edge list by inserting its vertices into a Bloom filter. A Bloom filter is also a bit array but uses an arbitrary, smaller number of bits. Each edge list e_v is summarized with a Bloom filter for v. A hash function maps a vertex $w \in e_v$ 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 filters' theory is beyond this paper's scope, 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 $\ln 2 \times m/d$ [14], 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 5×10^{-11} and the worst-case rate will be 6×2^{-3}.

Modifications to Algorithm 1 for supporting a Bloom filter are straight-forward. After line 3, initialize the Bloom filter using vertices in e_u:

```
1: for all y ∈ e_u do
2:   for i = 1 → k do
3:     Set bit H_i(y) in B_x to 1
```

Then implement the search in line 5 as follows:

```
1: for i = 1 → k do
2:   if bit H_i(x) = 0 then
3:     Skip to next x
```

5 Multithreaded Platforms and Implementations

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 compiler (version 6.3.1) and its pragmas for the massively multithreaded Cray XMT, and we use OpenMP [21] via the GNU C compiler (version 4.4.1) for a comparison on an Intel Nehalem E5530-based commodity platform.

The Cray XMT provides an ideal platform for massively multithreaded massive graph analysis. Each Threadstorm processor 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 not waiting on memory.

In this architecture, multithreading is used to hide some or all of the long latency of memory accesses. There is no cache in the processors; all latency is handled by threading. The XMT features a large, globally shared memory that is hashed to break up locality and alleviate hot-spotting. Synchronization takes place at the level of 64-bit words, and lightweight primitives like 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 contains 64 Threadstorm processors running at 500 MHz. The globally addressable shared memory totals 512 GiB and can hold graph data structures containing more than 2 billion vertices and 17 billion edges. Because of the Cray XMT is a shared resource, only 32 processors and around 300 GiB of memory were available for our tests.

The Intel Nehalem E5530 is a 2.4GHz quad-core processor with "hyperthreading" [4]. Each physical core holds two thread contexts and switches when one thread stalls while waiting for memory. The context switches are not as frequent as on the Cray XMT, and there are only two contexts available for hiding memory latencies. However, 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.

Code's threading implementation is straight-forward. 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. There is no explicit locking 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. However, to simplify the code, we use the GCC/Intel intrinsic functions similar to the Cray intrinsics.

The algorithm to update the triangle counts T above use 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 [23]. 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 re-used across the inner loop. The sorted update and Bloom filter update compute their summary data using the source vertex rather than choosing the larger list.

6 Performance

Our test data is generated by the RMAT 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 RMAT scale k, the number of vertices $n = 2^k$, and an edge factor f, we generate $f \cdot n$ unique edges for our initial graph. We then select a fraction ρe of those edges and add them to a deletion queue. For these experiments, $\rho = 0.0625$.

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 ρ, a new action is a deletion popped from the deletion queue. Otherwise an action is an insertion generated with the same RMAT generator and parameters. The edge to be inserted may already exist in the graph. Inserted edges are appended to the deletion queue with probability ρ. There are no self-loops in our generated edges, but the algorithm implementations do handle self-loop cases by ignoring edges $\langle v, v \rangle$.

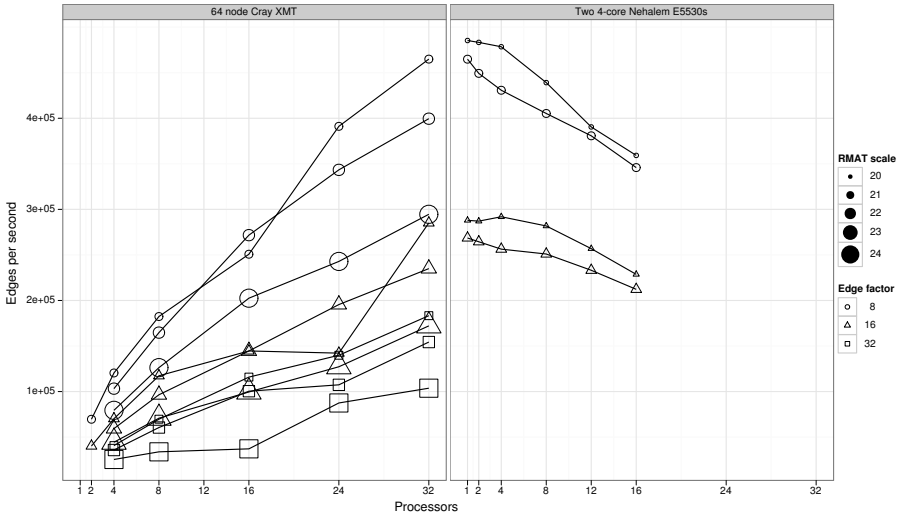
Because of shared usage, our runs on the Cray XMT are limited to 32 processors and around 300 GiB of memory. Clearly the Cray XMT applies to far larger problems than the Intel-based platform. The latter is limited to scale $k = 21$ and edge factor $f = 16$ with our current test harness. On the Cray XMT, our largest experiments were sixteen times larger, with $k = 24$ and $f = 32$, and are limited more by our testing code's structure than the XMT's architecture.

6.1 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 is a straight-forward computation by counting all triangles. For each edge $\langle u, v \rangle$, we count the size of the intersection $|e_u \cap e_v|$. 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

Figure 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 memory system leads to decreasing performance.



parallelize over the vertices. With $2^{21} \approx 2$ million vertices in the smallest case, this is a sufficient amount of 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. The Intel Nehalem's performance decreases, possibly because of memory transaction bottlenecks. We are investigating this performance decrease.

6.2 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 1 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 the speed up of locally recomputing the metric relative to recomputing the entire graph's clustering coefficients. The boxes in Figures 4 and 5 span the 25% – 75% quartiles of the update times for each processor count. The bar through the box shows the median. The lines stretch to the farthest non-outlier, those within 1.5× the distance

Figure 4: Updates per second by algorithm.

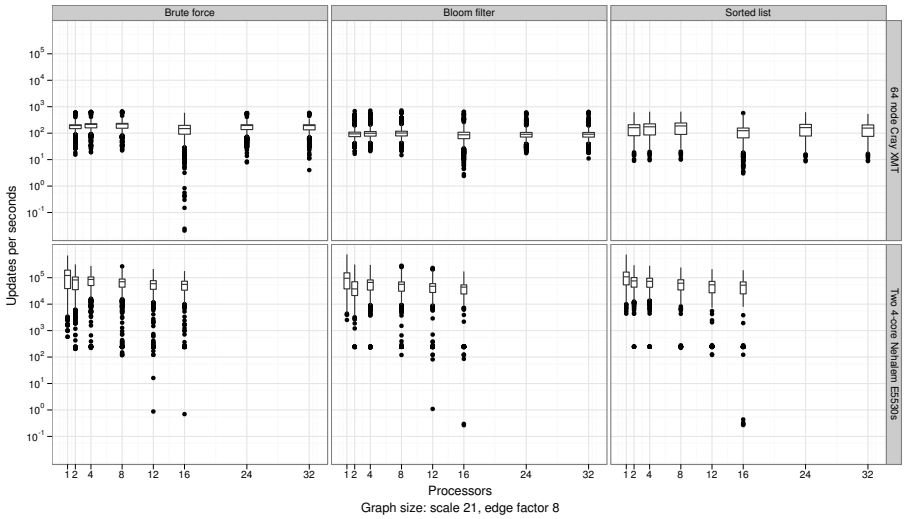
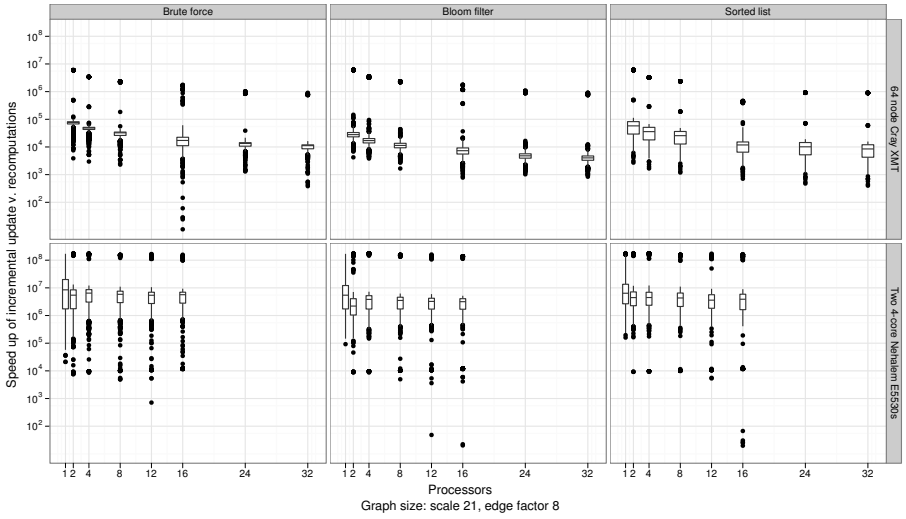


Figure 5: Speed up of incremental, local updates relative to recomputing over the entire graph.



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 1: Summary of update algorithms

Algorithm	Edge by edge	Batched (batch of 1000)	Batched (batch of 4000)
Brute force	90	25,100	50,100
Bloom filter	60	83,700	193,300

Table 2: Comparison of single edge versus batched edge operations on 32 XMT processors, RMAT 24 input, in updates per second

between the median and the closest box side. The 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 4 shows that recomputing only the changed local clustering coefficients speeds up the update rate typically by at least a thousand times.

The Nehalem results degrade with additional processors. The noise at 12 and 16 processors 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 2 million vertices. This Cray XMT is capable of holding a graph in memory up to 135 million vertices.

Table 2 shows performance obtained from batching operations and extracting parallelism. 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. Increasingly the batch size intuitively improves scalability since data parallelism is increased in the update phase.

7 Conclusions and Future Work

We handle individual updates rapidly enough for simple analysis (Figure 4). Update the clustering coefficients after each edge insertion or deletion duplicates setup time, so the sorted update and Bloom filter algorithms perform relatively poorly.

The serial stream contains enough parallelism when batched to exploit the Cray XMT's massively multithreaded architecture. We achieve a speed-up of 550× 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× speed-up over the brute-force method on the Cray XMT.

Acknowledgments

This work was supported in part by the PNNL CASS-MT Center and NSF Grant CNS-0614915. We thank PNNL and Cray for providing access to Cray XMT platforms. We are grateful to 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] David A. Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C. Poulos. STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation. Technical report, Georgia Institute of Technology, 2009.

[2] Ziv Bar-Yossef, Ravi 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)*, pages 623–632, San Francisco, CA, January 2002. Society for Industrial and Applied Mathematics.

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

[4] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang 0003, Scott Pakin, and José Carlos Sancho. A performance evaluation of the Nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, 18(4):453–469, 2008.

[5] V. Batagelj and A. Mrvar. Pajek – program for large network analysis. *Connections*, 21(2):47–57, 1998.

[6] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides 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*, pages 16–24, New York, NY, USA, 2008. ACM.

[7] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 228–237, New York, NY, USA, 2005. ACM.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, 2005.

[10] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262, New York, NY, USA, 2006. ACM.

[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, April 2004. SIAM.

[12] D. Eppstein, Z. Galil, G.F. Italiano, and A. Nissenzweig. Sparsification: a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[13] Facebook. User statistics, 2009. http://www.facebook.com/press/info.php?statistics.

[14] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, pages 254–265, 1998.

[15] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2):207–216, 2005.

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

[17] H. Jeong, S.P. Mason, A.-L. Barabási, and Z.N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41–42, 2001.

[18] K. Madduri, D. Ediger, K. Jiang, D.A. Bader, and D.G. 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'09)*, Rome, Italy, May 2009.

[19] Kamesh Madduri and David 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.

[20] M.E.J. Newman. Scientific collaboration networks: II. shortest paths, weighted networks and centrality. *Phys. Rev. E*, 64:016132, 2001.

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

[22] Nathan Reed. Gigatweet, 2008. http://popacular.com/gigatweet/.

[23] Michael Ringenburg and Sung-Eun Choi. Optimizing loop-level parallelism in Cray XMT™ applications. In *Cray User's Group*, May 2009.

[24] Technorati. State of the blogosphere, 2008. http://technorati.com/blogging/state-of-the-blogosphere/.

[25] J.D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. 14th Ann. ACM Symp. on Principles of Distributed Computing*, pages 214–222, Ottowa, Canada, August 1995.

[26] D.J. Watts and S.H. Strogatz. Collective dynamics of small world networks. *Nature*, 393:440–442, 1998.