Graphing trillions of triangles

Paul Burkhardt

Abstract
The increasing size of Big Data is often heralded but how data are transformed and represented is also profoundly important to knowledge discovery, and this is exemplified in Big Graph analytics. Much attention has been placed on the scale of the input graph but the product of a graph algorithm can be many times larger than the input. This is true for many graph problems, such as listing all triangles in a graph. Enabling scalable graph exploration for Big Graphs requires new approaches to algorithms, architectures, and visual analytics. A brief tutorial is given to aid the argument for thoughtful representation of data in the context of graph analysis. Then a new algebraic method to reduce the arithmetic operations in counting and listing triangles in graphs is introduced. Additionally, a scalable triangle listing algorithm in the MapReduce model will be presented followed by a description of the experiments with that algorithm that led to the current largest and fastest triangle listing benchmarks to date. Finally, a method for identifying triangles in new visual graph exploration technologies is proposed.

Keywords
Graph, scalable algorithms, triangle counting, visual analytics, parallel programming, MapReduce

Introduction
Data, data, data, and more data! Obtaining data is just the beginning of analysis, but heaping piles of raw data are rarely useful. How data are structured or represented facilitates analysis. Data must undergo transformation, translation, transmuting and other structuring, categorization, and organization operations to enable intelligent analysis. This includes making data more concise, more relevant, and structured to improve the performance of a query or algorithm. Ultimately, data form the raw material we need to pave the roads to knowledge, and visualization is the interface to insight. The entanglement between data representation and analysis is best exemplified in the graph-theoretic study of complex networks.

With the era of Big Data\(^1\) comes the emergence of Big Graphs\(^2\) and while there has been attention to improving the capacity of graph primitives, such as Breadth-First Search\(^3,4\), it should not be overlooked that the answers to even simple questions, the algorithm output itself, can be tremendously large and complex. Consider the task of identifying all mutually connected triplets in a graph or network of entities, often depicted as a triangle where each vertex of the triangle represents an entity that is connected to the other entities by the edges of the triangle. Finding all such triangles in a graph requires a lot of computation and the number of triangles can be many times more than the number of entities. Despite a triangle being a simple graphic, visualizing all triangles in a graph is very difficult because the triangles overlap and are easily obscured.

Computing triangles in graphs is a fundamental operation in graph theory and has wide application in the analysis of social networks\(^5\), identifying dense subgraphs\(^6\) and network motifs\(^7\), spam detection\(^8\) and uncovering hidden thematic layers in the World Wide Web\(^9\). Yet, given the attention to finding, counting,
and enumerating triangles, there has been very little work done to present a visual representation of triangles that is both scalable and navigable. This may be due in part to the lack of feasible technology that can list out trillions of triangles to provide the specificity needed in a Big Graph visual analytic, where every triangle can be traced by an end-user.

This article will begin first with an exercise to convince the reader of the importance of not just more and bigger data, but also how the underlying representation of data can better enable knowledge discovery in the context of graph analytics. Then a new algebraic method to reduce the number of arithmetic operations for computing all triangles in a graph will be introduced, and additionally a new scalable algorithm that demonstrates the largest triangle enumeration to date will be presented, along with the experimental results. Finally, a method for extending existing algorithms in visual graph exploration tools to include the identification of triangles will be proposed.

An illustration of knowledge manifestation

Consider a network of things, where each thing is identified by a number and these things self-organize into a larger network of a Thing of Things. How do we represent this Thing of Things?

Tabulating

A set of binary relationships, e.g. one thing is associated with another thing, can be simply tabulated in a basic data structure, a list of pairs of things, that is compact and does not require any special management. This is illustrated by the subset of our network of Thing of Things shown in Figures 1(a) and 1(b). Note how 498 connects to 5 and 33—does this suggest a mutual relationship? Does the fact that 17 connects to 7, which connects 33 and in turn connects back to 17 denote transitive closure?

That’s just a tiny sample. Our Thing of Things comprises 500 things, which participate in 25,936 relationships. What can you identify, given all of the data as depicted in Figure 2? Displaying this very small example dataset provides very little insight. Tabulation can obscure non-direct relationships. Will a graph layout manifest information better? Complex relationships naturally form networks, which are ideal for graph treatment, so let’s Graph it!

Graphing

A graph can be displayed in a node-link layout, which appears as the familiar “ball and springs” model. But layout algorithms for graph visualization can have dramatic differences. Using the Graphviz graph visualization package available at www.graphviz.org, to produce Figure 3 our Thing of Things appears more like a ball of yarn with the “force-directed” layout. This doesn’t uncover any more useful information, colorful or not, than the tabulation example in Figure 2. But when the “scalexy” layout is used, as shown in Figure 4, there is a clearly emerging structure! By a simple change in displaying the graph data, three well-defined clusters can be identified in our Thing of Things network. Other visualization approaches can manifest these clusters in more striking fashion, but require another change in the graph representation.

Graph data structures. The following terms and definitions will be used throughout this article. Let $G = (V, E)$ denote a simple, undirected graph with $n = |V|$ vertices and $m = |E|$ edges, where $N(v) = \{u \in V | (v, u) \in E\}$ is the neighborhood of a vertex $v$, $\delta(v) = |N(v)|$ is its degree, and $d_{\text{max}} = \max\{|d(v)| | v \in V\}$ is the maximum degree in $G$. Then $A \in \mathbb{R}^{n \times n}$ is the adjacency matrix of $G$, and so $A^T = A$.

A graph data structure can be a list of pairs, e.g. edges, as noted earlier. In this edge-list representation, there are $2m$ ordered pairs including permutation, e.g. $(1, 2)$ and $(2, 1)$, requiring $4m$ words of storage for all vertices, which is the minimum space needed for a complete representation of the graph dataset. Another graph data structure is the adjacency matrix, where the graph is represented in matrix form instead of as a list of edges. Each row represents an adjacency, i.e. neighborhood, of a vertex. Every non-zero entry indicates an edge, thus $A(i, j) = 1$ identifies an edge between vertices $i$ and $j$. Recall that in matrix algebra zero is an annihilator for multiplication, so it is more efficient to use a sparse matrix representation for sparse graphs that ignores zero entries. Conventionally, $G$ is sparse if $m \in O(n)$. Using a dense matrix for $G$ requires $O(n^2)$
space. In contrast, storing $G$ in a sparse matrix form requires $n + 4m \in O(n + m)$ space, assuming the graph is connected. For a concrete example, consider the Yahoo! Web Graph derived from the Altavista search engine crawl of the World Wide Web circa 2002, available from Yahoo! at www.yahoo.com (https://webscope.sandbox.yahoo.com). This graph consists of $n = 1.4 \times 10^9$ web pages and $m = 6.6 \times 10^9$ URL links. To store this graph in matrix form would require space of the order of order of

$$\text{gigabytes} = \begin{cases} 230,000,000 & \text{dense matrix} \\ 3.2 & \text{sparse matrix} \end{cases}$$

The dense matrix storage for this graph is prohibitive at nearly 230 petabytes whereas in contrast the sparse matrix is over seven orders of magnitude smaller, at just 3.2 gigabytes. A sparse matrix can be plotted and visualized in MATLAB, which can reveal patterns in the distribution of non-zero entries, so let’s Spy it!

**Plotting**

The *Thing of Things* edge list can be converted into a sparse matrix and visualized as a *Spy* plot in MATLAB. First annotate each edge with a 1 for the MATLAB conversion, then use the following commands:

```matlab
load 'edgelist.mat';
A = spconvert(edgelist);
spy(A);
```

The *Spy* plot from this labor is provided in Figure 5. If the reader squints carefully, a pattern, albeit fuzzy, might be apparent, especially along the diagonal, but there is nothing definite enough to justify a purchase in new eyeglasses! However, is it possible to manipulate the graph matrix to reveal the clusters that were manifested in Figure 4?
Spectral partitioning. Let’s try spectral partitioning but this requires another change in representation of the data, specifically a new graph matrix, known as the graph Laplacian. The eigenvectors of this Laplacian can be used to partition the graph. The following definitions will be used in the next passages on spectral partitioning.

\[ D \text{ diagonal matrix with vertex degrees along the diagonal;} \]
\[ L \text{ graph Laplacian, } L = D - A; \]
\[ U \text{ matrix whose columns are the eigenvectors of } L. \]

The graph Laplacian is also symmetric and positive semi-definite, so the eigenvalues of \( L \) must be greater than or equal to zero. The multiplicity of the zero eigenvalue relates to the number of connected components in the graph. The eigenvector of the second smallest eigenvalue of \( L \) is the Fiedler eigenvector, whose namesake discovered its connection to graph connectivity and use in spectral partitioning.\(^{10}\)

Reordering the adjacency matrix \( A \) by the sorted elements in the Fiedler eigenvector helps in envelope reduction to avoid fill-in, increases in non-zeros, during sparse matrix computations,\(^{11}\) and also to reveal small-world characteristics in graphs.\(^{12,13}\) This reordering technique will be helpful in visualizing clusters in the Thing of Things network. The premise in the reordering of \( A \) by a permutation vector is to align the non-zero entries close to the diagonal in \( A \), thereby reducing the geometric distances between vertices, which results in natural partitioning of the graph. The following MATLAB commands will reorder \( A \) by the Fiedler eigenvector and Spy it!

```matlab
dim=size(A);
D=sparse(1:dim(1), 1:dim(2), sum(A,2));
L=D-A;
[U,~]=eigs(L, 2, 'sa');
[~, x]=sort(U(:,2));
spy(A(x,x))
```

Behold, in Figure 6 three well-defined clusters are identifiable. Data representation matters! The choice of representation facilitates knowledge discovery. But it doesn’t end with just the original data. Data begets data. Data analysis output can be many times larger than the input!

**Triangles in graphs**

A triangle in \( G \) is a group of three vertices that are all mutually connected, thus it is a clique, i.e. a graph in...
which every vertex is connected to every other vertex, and a cycle, i.e. a path with identical endpoints. Triangles in graphs are useful for complex network analysis based solely on structural, i.e. adjacency, information. Let $\Delta(v)$ be the local count of triangles involving vertex $v$ and $\Delta(G)$ be the global count of triangles in $G$. There can be many more triangles than vertices and edges, and if the graph is completely connected, where $G$ itself is a clique, the total number of triangles, $\Delta(G)$, is

$$\Delta(G) = \binom{n}{3} \in \Theta(n^3)$$

The triangles in a graph can be easily identified by testing all possible triples, $\{u, v, w\}$, for $\{u, v\}, \{v, w\}, \{u, w\} \in E$, but this takes $O\left(\binom{n}{3}\right)$ time. If most of the triples were triangles and the task was to enumerate all triangles, then the approach is optimal. But generally this brute force method is highly inefficient. We will briefly review the standard method for counting triangles leading to a new matrix approach, then transition into the description of a new algorithm that can potentially enumerate trillions of triangles.

**Counting triangles**

Since cycles of length $n$ are diagonal elements in $A^n$ and a triangle is a cycle of length 3, then it is well known that the local and global triangle counts can be calculated by

$$\Delta(v) = \frac{1}{2} A^3_{vv}$$

$$\Delta(G) = \frac{1}{6} \sum_v A^3_{vv} = \frac{1}{6} \text{Tr}(A^3)$$

where the symbol Tr will refer to the trace operator of a matrix. The $\frac{1}{2}$ factor in $\Delta(v)$ is to account for the cycles from $\{u, w\}$ incident to $v$, and the factor $\frac{1}{6} = \frac{1}{2} \times \frac{1}{3}$ in $\Delta(G)$ is because each triangle is counted thrice.

Computing triangles by $A^3$ by fast matrix multiplication admits the best runtime of $n^\omega + o(1)$, where $\omega \approx 2.3728639$ is the fast matrix product exponent. But the classic matrix product is used in practice, therefore practical improvements would be beneficial. A new matrix approach developed in 2013 by this author reduces the total number of arithmetic operations. Here the element-wise multiplication operator for matrices is denoted by the $\circ$ operator.

**Theorem 1.** Given $G$ and the Hadamard product $(A^2 \circ A)$, then $\Delta(v) = \frac{1}{2} \sum_v (A^2 \circ A)_v$ and $\Delta(G) = \frac{1}{6} \sum_{ij} (A^2 \circ A)_{ij}$.

**Proof.** The paths of length two, i.e. 2-paths, are identified by $A^2$; then element-wise multiplication $A \circ A$ retains only those $\{v, u\}$ endpoints of 2-paths that are also the $\{v, u\} \in E$ endpoints of edges. Then the sum of elements in the $i$th column vector of $A^2 \circ A$ is the number of 3-cycles involving $v$. Using $\Delta(v) = \frac{1}{2} A^3_{vv}$ leads to

$$\Delta(v) = \frac{1}{2} A^3_{vv} = \frac{1}{2} \sum_v (A^2 \circ A)_v \quad (1)$$

Now recall $\text{Tr}(XY^T) = \sum_y (X \circ Y)_y$; then $\text{Tr}(A^2 A^T) = \sum_y (A^2 \circ A)_y$. Finally, since $\Delta(G) = \frac{1}{6} \text{Tr}(A^3)$, then

$$\Delta(G) = \frac{1}{6} \text{Tr}(A^3) = \frac{1}{6} \text{Tr}(A^2 A^T) = \frac{1}{6} \sum_{ij} (A^2 \circ A)_{ij} \quad (2)$$

The corollary to this is that the number of arithmetic operations is cut in half when using classic matrix multiplication.

**Corollary 1.** The number of arithmetic operations for counting all triangles in $G$ by Theorem 1 using sparse matrix representation is $2N(n + 1) - 1 = O(mn)$, where $N = 2m$ is the number of non-zeros, which is approximately half the number of arithmetic operations if counting by $\text{Tr}(A^3)$ using sparse matrix products.

**Proof.** There are a total of $2N$ multiplication and addition operations for a sparse matrix–vector product. Then $A^2$ is possible by direct application of sparse matrix–vector multiplication of $A$ with all $n$ column vectors in $A$ for a total of $2Nn$ arithmetic operations. Then $A \circ A$ takes $N$ operations for the element-wise multiplication and, finally, summing all non-zero elements from this is an additional $(N - 1)$ operations. Thus, $\sum_y (A^2 \circ A)_y$ requires $2N(n + 1) - 1 = O(mn)$ arithmetic operations, since $N = 2m$. In contrast, computing $A^3$ takes $4Nn$ operations and $\text{Tr}(A^3)$ adds an extra $(N - 1)$ operations totaling $4Nn + N - 1$, which is of the order of twice the number of arithmetic operations, as

$$2Nn + 2N - 1 \approx 2Nn \frac{N}{4Nn + N - 1} \approx \frac{1}{2}$$

Using similar logic, it is easy to see that this result also holds for dense graphs and the local triangle count, $\Delta(v)$. 

Burkhardt

161
Computing triangles by Theorem 1 replaces a matrix multiplication requiring $O(n^3)$ operations with a less expensive element-wise multiplication that requires only $O(n^2)$ operations. Moreover, the new approach avoids a potential dense matrix multiplication, which can degrade the efficiency of sparse matrix multiplication. Although triangles can be enumerated by identifying common non-zero entries between $\bar{u}$ and $\bar{v}$ in $A^{2\times A}$ for any $(u, v)$ edge, the next section will describe a new algorithm that is better suited for enumerating trillions of triangles.

**Scalable triangle listing**

Disk-based computation for Big Data problems was popularized by the Google MapReduce paradigm and corresponding open-source Apache Hadoop framework (http://hadoop.apache.org). The theoretical model for MapReduce has been investigated, putting it in firm standing with older computational models, such as Parallel Random Access Machine (PRAM), Bulk Synchronous Parallel (BSP), and Streaming. The largest Breadth-First Search on a graph was accomplished using the Hadoop MapReduce framework. The MapReduce model was also successful for the largest and fastest computation of all Jaccard similarity coefficients in a graph at that time. The following sections will describe the MapReduce algorithm used in the largest and fastest enumeration of triangles in a graph to date.

**MapReduce triangle listing algorithm**

There are relatively few triangle algorithms in the MapReduce framework and these tend to focus on approximating triangles. The most closely related work is that of Suri and Vassilvitskii but that approach does not guarantee a constant amount of working memory needed for massively scalable algorithms. Moreover, the algorithm described here requires half the I/O by not revisiting the original graph edges; rather it uses the edges filtered by degree for subsequent triangle verification. The following is an outline for a new MapReduce triangle listing algorithm:

1. Filter out half the edges so that the end vertex has higher degree (use lexicographic order, in case the degrees are equal).
2. Create labeled, ordered pairs by lexicographic order from the filtered edges:
   (a) Adjacent vertex pairs, i.e. endpoints of edges, labeled as edge;
   (b) Unique neighbor pairs, i.e. endpoints of 2-paths, labeled as non-edge;
3. List 2-paths for an ordered pair if it also has a label for an edge.

Like other algorithms, the 2-paths are emitted only from the lowest-degree vertex in a triangle. This is enabled by the first step, which reduces the neighborhood of each vertex to contain only neighbors with higher degree. Denoting the size of this reduced neighborhood $d[v]$, the work per vertex is $\binom{d[v]}{2}$. Now observe that if a vertex has $\sqrt{m}$ neighbors, each with degree greater than $\sqrt{m}$, it would exceed the number of edges in the graph. Thus, a vertex can only have $O(\sqrt{m})$ neighbors with higher degree. Subsequently, the work per vertex is $\binom{d[v]}{2} \leq \binom{\sqrt{m}}{2} = O(m)$. It then follows that there is $O(m^{3/2}) = O(\sqrt{m})$ work per edge, leading to an overall time-complexity of $O(m \sqrt{m}) = O(m^{3/2})$, which is optimal for sparse graphs, since there could be $\binom{\sqrt{m}}{3} = \Theta(m^{3/2})$ triangles.

Although $d[v]$ can be quite large in highly skewed massive graphs, it is likely to be much less than $\sqrt{m}$. Moreover, the $O(m)$ work to construct all neighbor pairs for a vertex can be load-balanced across parallel tasks, so for sparse graphs the algorithm will be scalable. The next section describes how to distribute the $\binom{d[v]}{2}$ neighbor pair construction in $O(1)$ words of working memory.

**Load balancing**

Unique pairs from a list of elements can be constructed in parallel while ensuring that each task performs a linear amount of work with constant words of memory, which are important for scalability. Consider the straightforward method, where the first element in the list is paired with the remaining $n-1$ elements, then the second element is paired with the remaining $n-2$ elements, and so forth. Now this can be accomplished in parallel by first constructing $n-1$ lists, each containing one element fewer than the previous, and distributing these lists to multiple tasks. Then each task pairs the first element of its list with the remaining elements, simultaneously with all other tasks. The tasks require no more memory than needed to store a pair of elements, since each new element after the first replaces the previous element. It is also easy to generate these lists in $O(1)$ words of working memory by replicating each element by its ordinal in the original list.
Reduction 1: set of key-value pairs for a single vertex

Map 1: Input: \( (u, d(u)), (v, d(v)) \)
if \( d(u) < d(v) \) then
emit \((u, v)\)
else if \( d(u) = d(v) \) then
emit \((u < v, v)\)
end if
Reduce 1: Input: \( \{u, \{v_i \in N(u)\}\} \)
for all \( v_i \in N(u) \) do
emit \((\{u, i\}, v_i)\)
end for
Map 2: Input: \( \langle (u, i), v \rangle \)
for \( j = 1 \ldots i \) do
emit \((\{u, i\}, v)\)
end for
Reduce 2: Input: \( \langle (u, i), (v_i | i = 1, 2, \ldots ) \rangle \)
emit \((\{u, i\}, 0, 0)\)
for remaining \( v_i | i \neq 1 \) do
emit \((\{v_i, v_i, 1\}, u)\)
end for
Map 3: Identity
Reduce 3: Input: \( \langle (vw, $\$), \{u \ldots \} \rangle \)
if \$ = 0 then
emit \((\{vw, u\})\)
end if

Algorithm 1 MapReduce triangle listing

Now to construct all the unique neighbor pairs for a vertex \( u \), each \( v_i \in N(u) \) will be replicated by its \( i \)-th order in \( N(u) \), which is arbitrary but must be consistent. Then pairing \( u \) with each copy of \( v_i \) is simply a matter of copying \( v_i \) from \( 1 \ldots i \) times, which requires only the memory needed to store \( u, v \). To distribute the \( v_i \) copies with \( u \) for parallel, pairwise combination, a set of new \((u, j)\) compound keys is created. The new set of key-value pairs for a single vertex \( u \) and its neighbors in \( N(u) \) is

\[
\{((u, j), v_i)\}_{j = 1 \ldots i} ->
\[
\{(u, 1), v_1\}, \{(u, 2), v_2\}, \{(u, 3), v_3\}, \{(u, 4), v_4\},
\{(u, 1), v_2\}, \{(u, 2), v_3\}, \{(u, 3), v_4\},
\{(u, 1), v_3\}, \{(u, 2), v_4\},
\{(u, 1), v_4\}
\]

Here the pairs are displayed in matrix layout only as a visual aid; in the computation, the pairs are distributed randomly by the MapReduce framework. Then, in a subsequent round, where values are aggregated and sorted for each \((u, i)\) key, the first value is paired with all remaining values. It is important that the values in each key are sorted to avoid duplicate pairing. The pairing is accomplished in \( O(1) \) working memory, since only the first value is kept in memory and paired with the other values in a streaming fashion. The new Triangle Listing algorithm that takes \( O(1) \) working memory and rounds is now given in Algorithm 1.

Table 1. SNAP graphs [http://www.snap.stanford.edu].

| Graph    | \( n \) [vertices] | \( 2m \) [edges] | \( \Delta(G) \) |
|----------|---------------------|------------------|-----------------|
| AstroPh  | 18,772              | 396,160          | 1,351,441       |
| AS-Skitter | 1,696,415           | 22,190,596       | 28,769,868      |
| LiveJournal | 3,997,962          | 69,362,386       | 177,820,130     |
| Orkut    | 3,072,441           | 234,370,174      | 627,584,181     |
| Friendster | 65,608,366         | 3,612,134,278    | 4,173,724,142   |

The first \( 1 \ldots i \) replication of each neighbor vertex in the first step, analogous to the generation of anti-diagonals in a circulant matrix, can also degrade performance when a single task must generate many copies for a large \( i \) value. It is simple to partition and distribute the replication work to other tasks, but this detail is left out for brevity. Also, this method generalizes to creating unique pairs in parallel from any set of elements using \( O(1) \) words of working memory; this can be useful in streaming and other external-memory applications.

Triangle listing benchmarks

The new MapReduce triangle listing algorithm was demonstrated in a two-part experiment. The first part was to test the algorithm on real-world graphs that would be easily repeatable by other practitioners. The second part was a scalability test. The results of the experiment were first presented in the keynote address for the IEEE Exploring Graphs at Scale (EGAS) 2015 workshop.\(^{25}\) The benchmarks were carried out on a cluster of 880 nodes with 24 cores and 64 GB RAM per node. The datasets were modified so that each edge was annotated by degree, e.g. \((u, d(u)), (v, d(v))\). An additional MapReduce job was inserted after the first round to randomize the output, resulting in a total of four rounds to list all triangles.

Part 1

The datasets for this part of the experiment, listed in Table 1, were downloaded from the Stanford Network Analysis Project (SNAP, http://www.snap.stanford.edu). The graphs were undirected and the global triangle count was available for ground truth. The benchmark results are provided in Figure 7 and demonstrate that Algorithm 1 takes less than 5 min on average for these small graphs; this is still comparatively slow because of the overhead needed for parallel processing and reading from disk. The second part of the experiment is more compelling.

Part 2

To test scalability, the Algorithm 1 was benchmarked on synthetically generated datasets using the
Graph500 (www.graph500.org) benchmark parameters. Table 2 lists the sizes of the graphs and the triangle counts computed by the algorithm. The results of this benchmark are given in Figure 8. The top performance was listing 1.6 billion triangles per minute! This was achieved on the RMAT scale 30 graph with $\Delta(G) > 200$ billion, which was 1.8 TB in size; to the best of this author’s knowledge it is the largest and fastest triangle listing experiment. The RMAT 28 and larger graphs have a maximum degree on the order of $10^6$ but because of the aforementioned load-balancing technique, the neighbor pairing for high-degree vertices was distributed across many reduce tasks, demonstrating the efficacy of the approach. Moreover, the number of 2-paths output from the RMAT 30 graph was nearly two trillion and for the RMAT 32 graph exceeded eleven trillion. The experiment on the RMAT 32 graph did not complete owing to time constraints and not failure.

**Visualizing trillions of triangles**

Having established an algorithm capable of listing potentially trillions of triangles in a graph, the next step is to identify how to visualize the triangles. To begin, we need a scalable visual analytics platform but there are very few tools designed for large-scale graph exploration. The GreenHornet visual analytics tool is

---

**Table 2.** Graph500 RMAT graphs ($A = 0.57$, $B = C = 0.19$, $D = 0.05$).

| Graph | $n$ (vertices) | $2m$ (edges) | $\Delta(G)$ |
|-------|----------------|--------------|--------------|
| RMAT 24 | 16,777,216 | 268,435,456 | 2,127,854,845 |
| RMAT 26 | 67,108,864 | 1,073,741,824 | 9,838,965,401 |
| RMAT 28 | 268,435,456 | 4,294,967,296 | 44,970,850,296 |
| RMAT 30 | 1,073,741,824 | 17,179,869,184 | 203,333,933,183 |
| RMAT 32 | 4,294,967,296 | 68,719,476,736 | $< 11,068,947,204,458$ |

---

**Figure 7.** Triangle listing results for SNAP datasets.

**Figure 8.** Triangle listing results for Graph500 datasets.
designed for scalability and utilizes a hierarchical partitioning of the graph, exemplifying the importance of data representation. The data hierarchy in GreenHornet is a sequence of increasingly coarser partitions of the graph, generated by merging vertices of lowest degree to their neighbors. If this merging process also identified common neighbors of the merged vertices, it would identify triangles, performing what is widely-known as an edge iterator algorithm for counting or listing triangles where for every \( \{v, u\} \) edge any \( w \in N(v) \cap N(u) \) completes the \( \{v, u, w\} \) triangle. Moreover, if any \( u, w \) neighbors, to be merged into a vertex \( v \), having degrees at most \( \sqrt{m} \), were instead tested for \( \{u, w\} \in E \), which is possible in \( O(1) \) time, the runtime would be \( O(m^{3/2}) \), which is optimal.  

Recent development is now encroaching on trillion-edge scale graph exploration, which could admit the navigation of trillions of triangles in a graph. The multiscale layout algorithm of Wong et al. is reported to be similar to the merging algorithm in GreenHornet and could therefore be extended in the aforementioned merging proposal of neighboring vertices to identify triangles. This will pave the way forward to Big Graph exploration not only of the input graph but also of the overlay of the graph analytic products with the input.

Acknowledgements

The author thanks David G Harris and Louis W Ibarra for their helpful comments.

Funding

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

1. Burkhardt P. An overview of big data. The Next Wave 2014; 20(4): 4–10.
2. Burkhardt P. Big graphs. The Next Wave 2014; 20(4): 11–20.
3. Burkhardt P and Waring CA. An NSA big graph experiment. Technical Report NSA-RD-2013-056001v1, US National Security Agency, 2013.
4. Burkhardt P and Waring CA. A cloud-based approach to big graphs. In: Proceedings of the 19th annual IEEE conference on high performance extreme computing, Wal- tham, MA, 15–17 September 2015, pp.1–8. Piscataway, NJ: IEEE.
5. Watts DJ and Strogatz SH. Collective dynamics of ‘small-world’ networks. Nature 1998; 393(6684): 440–442.
6. Cohen J. Graph twiddling in a MapReduce world. Comput Sci Eng 2009; 11(4): 29–41.
7. Milo R, Shen-Or S, Itzkovitz S, et al. Network motifs: simple building blocks of complex networks. Science 2002; 298(5594): 824–827.
8. Becchetti L, Boldi P, Castillo C, et al. Efficient algorithms for large-scale local triangle counting. ACM Trans Knowl Discovery Data 2010; 4: 1–28.
9. Eckmann JP and Moses E. Curvature of co-links uncovers hidden thematic layers in the World Wide Web. Proc Natl Acad Sci USA 2002; 99: 5825–5829.
10. Fiedler M. Algebraic connectivity of graphs. Czech Math J 1973; 23: 298–305.
11. Barnard ST, Pothen A and Simon HD. A spectral algorithm for envelope reduction of sparse matrices. In: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, Portland, OR, 15–19 November 1993, pp.493–502. Piscataway, NJ: IEEE.
12. Grindrod P. Range-dependent random graphs and their application to modeling large small-world proteome datasets. Phys Rev E 2002; 66: 066702-1–066702-7.
13. Higham DJ. Unravelling small world networks. J Comput Appl Math 2003; 158: 61–74.
14. Le Gall F. Powers of tensors and fast matrix multiplica- tion. In: Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation, Kobe, Japan, 23 July 2014, pp.296–303. New York, NY: ACM.
15. Burkhardt P. Internal NSA conference presentation, 2013.
16. Dean J and Ghemawat S. MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on symposium on operating systems design and implementation, San Francisco, CA, 2004, pp.137–150. Berkeley, CA: USENIX Association.
17. Karloff H, Suri S and Vassilvitskii S. A model of computation for MapReduce. In: Proceedings of the 21st annual ACM-SIAM symposium on discrete algorithms, Austin, TX, 2010, pp.938–948. Philadelphia, PA, SIAM.
18. Goodrich MT, Sitchinava N and Zhang Q. Sorting, searching, and simulation in the MapReduce framework. In: Asano T, Nakano S, Okamoto Y, et al. (eds) Algorithms and computation. Berlin: Springer, pp.374–383.
19. Pietracaprina A, Pucci G, Riondato M, et al. Space-round tradeoffs for MapReduce computations. In: Proceedings of the 26th ACM international conference on supercomputing, San Servolo Island, Venice, Italy, 2012, pp.235–244. New York, NY: ACM.
20. Burkhardt P. Asking hard graph questions. Technical Report NSA-RD-2014-050001v1, US National Security Agency, 2014.
21. Tsourakakis CE, Kang U, Miller GL, et al. Doulion: counting triangles in massive graphs with a coin. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining, Paris, France, 2009, pp.837–846. New York, NY: ACM.
22. Pagh R and Tsourakakis CE. Colorful triangle counting and a MapReduce implementation. Inf Process Letts 2012; 112: 277–281.
23. Suri S and Vassilvitskii S. Counting triangles and the curse of the last reducer. In: Proceedings of the 20th..
24. Latapy M. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor Comput Sci* 2008; 407: 458–473.

25. Burkhardt P. Graph it! Vis it! Technical report, US National Security Agency, http://www.vacommunity.org/egas2015/papers/IEEEEGAS2015-PaulBurkhardt.pdf (accessed October 26, 2015).

26. Wong PC, Mackey P, Cook KA, et al. A multi-level middle-out cross-zooming approach for large graph analytics. In: *Proceedings of the 2009 IEEE symposium on visual analytics science and technology*, Atlantic City, NJ, 12–13 October 2009, pp.147–154. Piscataway, NJ: IEEE.

27. Wong PC, Haglin D, Gillen D, et al. A visual analytics paradigm enabling trillion-edge graph exploration. In: *Proceedings of the 2015 IEEE Symposium on Large Data Analysis and Visualization*, Chicago, IL, 25–26 October 2015, pp.57–64. Piscataway, NJ, IEEE.