Efficient Calculation of Triangle Centrality in Big Data Networks

Wali Mohammad Abdullah
Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
w.abdullah@uleth.ca

David Awosoga
Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
odo.awosoga@gmail.com

Shahadat Hossain
Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
shahadat.hossain@uleth.ca

2022 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-9786-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPEC55821.2022.9926324

***Abstract*—The notion of "centrality" within graph analytics has led to the creation of well-known metrics such as Google's PageRank [1], which is an extension of eigenvector centrality [2]. Triangle centrality is a related metric [3] that utilizes the presence of triangles, which play an important role in network analysis, to quantitatively determine the relative "importance" of a node in a network. Efficiently counting and enumerating these triangles are a major backbone to understanding network characteristics, and linear algebraic methods have utilized the correspondence between sparse adjacency matrices and graphs to perform such calculations, with sparse matrix-matrix multiplication as the main computational kernel. In this paper, we use an intersection representation of graph data implemented as a sparse matrix, and engineer an algorithm to compute the triangle centrality of each vertex within a graph. The main computational task of calculating these sparse matrix-vector products is carefully crafted by employing compressed vectors as accumulators. As with other state-of-the-art algorithms [4], our method avoids redundant work by counting and enumerating each triangle exactly once. We present results from extensive computational experiments on large-scale real-world and synthetic graph instances that demonstrate good scalability of our method. We also present a shared memory parallel implementation of our algorithm.**

***Index Terms*—intersection matrix, local triangle count, forward degree cumulative, forward neighbours, sparse graph, triangle centrality**

I. Introduction

A cycle of length 3, colloquially known as a triangle, is the smallest non-trivial clique found in a graph. Counting and enumerating triangles is crucial to gaining insights into the underlying composition and distribution of network data, leading to the creation of many metrics. To properly analyze graph characteristics, its structure must be critically examined and understood, because an efficient representation of network data will dictate analysis capabilities and improve algorithm performance and data visualization potential [5]. Large real-life networks are typically sparse in nature, so efficient computations with these graphs must be able to account for their sparsity and skewed degree distribution [6]. A consistent structure makes linear algebra-based triangle enumeration methods appealing, and most methods use direct or modified matrix-matrix multiplication, with a notable exception being the implementation of Low et al. [7]. The MIT/Amazon/IEEE **Graph Challenge** sponsored graph analytics challenge features the current state-of-the art in triangle counting and enumeration [8].

There is a large number of network topological metrics that involve the application of triangle enumeration, including transitivity ratio - the ratio between the number of triangles and the paths of length two in a graph - and clustering coefficient; the fraction of neighbours for a vertex i of a graph who are neighbours themselves. Other real-life applications of triangle counting include spam detection [9], network motifs in biological pathways [10], and community discovery [11]. This paper extends counting [12] and enumeration [13] algorithms to efficiently compute "node centrality" in large networks. We utilize an "intersection" representation of network data obtained as a list of edges [14] and based on sparse matrix data structures [15]. Our triangle enumeration algorithm derives its simplicity and efficiency by employing matrix-vector product calculations as its main computational kernel. The local triangle count and triangle centrality of each vertex is then acquired from the enumerated triangles resulting from this matrix-vector multiplication.

A. Triangle Centrality

Burkhardt [3] introduced triangle centrality as a new centrality measure that captures the influence of triangles on the importance of vertices. A vertex is designated as "important" if many neighbours of that vertex are cohesive with their own neighbours. In other words, if the neighbours of a vertex are members of many triangles, the influence of the vertex of interest is strengthened. The calculation of triangle centrality finds important vertices that have both direct and indirect endorsements, because though a vertex may not be in triangles with many (or any) of its neighbours, it could still be supported by neighbouring vertices that are involved in many triangles. These important vertices cannot be identified using other centrality measures such as degree centrality or eigenvector centrality, and this gives triangle centrality a noted advantage in identifying, for example, spam nodes.

Let $G = (V, E)$ be a connected and undirected graph without multiple edges and self-loops, where V denotes the set of vertices and E denotes the set of edges. Let $N(v)$ be the neighbourhood set of v (i.e., vertices $\{w|\{v, w\} \in E\}$), $N_\Delta(v) \subset N(v)$ be the set of neighbours that are in triangles

with v, and $N_\Delta^+(v) = N_\Delta(v) \cup \{v\}$ be the closed set that includes v. The number of triangles incident on vertex v is called its *local triangle count*, denoted as $\Delta(v)$. Finally, $\Delta(G) = \sum_{v \in V} \Delta(v)$ denotes the total triangle count of graph G. The *triangle centrality* [3] of vertex v is defined as,

$$TC(v) = \frac{\frac{1}{3}\sum_{u \in N_\Delta^+(v)} \Delta(u) + \sum_{w \in \{N(v) \setminus N_\Delta(v)\}} \Delta(w)}{\Delta(G)}$$

The factor of $\frac{1}{3}$ in the first expression accounts for the triple counting of triangle degree at a vertex. In other words, the triangle centrality of vertex v is the sum of v's and its neighbours' local triangle counts divided by the number of triangles in the graph. By definition, the centrality values indicate the proportion of triangles centered at a vertex and its neighbours bounded in the range $[0, 1]$.

The remainder of the paper is organized as follows. In Section II, we use an intersection representation of network data and other data structures, followed by a brief description of the triangle enumeration [13] and the triangle centrality algorithms. The rest of the section explains the main ideas in our intersection matrix-based triangle enumeration implementation using an illustrative example. Li and Bader [16] presented a rapid implementation of triangle centrality using GraphBLAS [17], an API specification for describing graph algorithms in the language of linear algebra. This paper first implements serial triangle centrality and presents a comparative result in Section IV, which also outlines the computing environment employed to perform numerical experiments on three sets of representative network data. In Section III, we describe a simple scalable shared memory parallel implementation of our intersection-based algorithm, **TC-Intersection**. The paper is summarized in Section V with pointers on future research directions.

II. Intersection Representation of Network Data

Let the vertices in V be labelled $1, 2, \ldots, |V| = n$ in a natural order. Using the labels on the vertices, a unique label can be assigned to each edge $e_k = \{v_i, v_j\}, i < j, k = 1, 2, \ldots, |E| = m$.

The *intersection representation* of graph G is a matrix $X \in \{0,1\}^{k \times n}$ in which for each column j of X there is a vertex $v_j \in V$ and $\{v_i, v_j\} \in E$ whenever there is a row l for which $X(l, i) = 1$ and $X(l, j) = 1$. For $k = m$ we have a canonical form such that the rows of X represent the edge list sorted by vertex labels. The transpose of this canonical intersection form is also known as the incidence matrix in graph literature. Matrix X can be viewed as an assignment to each vertex a subset of m labels such that there is an edge between vertices i and j if and only if the inner product of the columns i and j is 1. Since the input graph is unweighted, the edges are simply ordered pairs, and can be sorted in $O(m)$ time. Unlike the adjacency matrix which is unique (up to a fixed labelling of the vertices) for graph G, there can be more than one *intersection matrix* representation associated with graph G [18]. We exploit this flexibility to store a graph in a structured and space-efficient form.

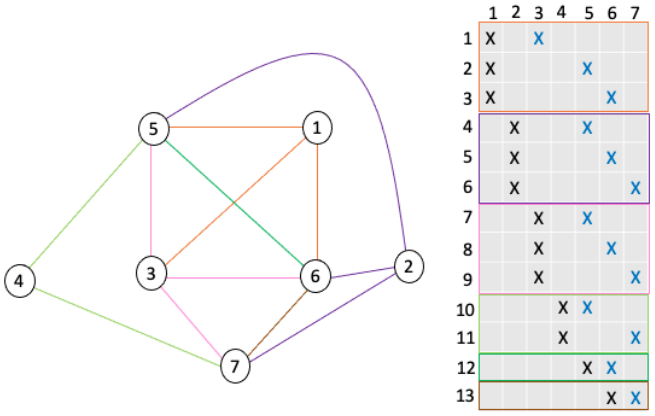

Fig. 1. Intersection Matrix Representation of the Example Input Graph

A. Intersection Matrix-based Triangle Counting

Graph algorithms can be effectively expressed in terms of linear algebra operations [19], and we combine this knowledge with our proposed data representation to count the triangles in a structured three-step method. For a vertex i we first find its neighbours $j > i$ such that $\{i, j\} \in E$ by multiplying the submatrix of X consisting of rows corresponding to edges incident on i (let us call them $(i-j)-$rows) by the transpose of the vector of ones of compatible length. A value of 1 in the vector-matrix product indicates that the corresponding vertex j is a neighbour of vertex i.

Next, we multiply the submatrix of X consisting of columns j identified in the previous step and the rows below the $(i-j)-$rows by a vector of ones of compatible length. A value of 2 in the matrix-vector product indicates a triangle of the form (i, j, j') where j and j' are neighbours of vertex i with $j < j'$. Let l be the row index in matrix X for which the matrix-vector product contains a 2. Then it must be that $X(l, j) = 1$ and $X(l, j') = 1$. Since each row of X contains exactly 2 nonzero entries that are 1, it follows that $\{j, j'\} \in E$. This operation is identical to performing a set intersection on the *forward neighbours* of vertices j and j', where the forward neighbours are defined as the neighbours of a vertex that have a higher label than the vertex of interest.

The number of triangles in the graph is given by the sum of the number of triangles associated with each vertex as described. The sorted representation of the edges in our algorithm ensures that each triangle is counted exactly once. Figure 1 displays the intersection matrix representation of the input graph X. The triangles of the form $(1, j, j')$ where $j, j' \in \{3, 5, 6\}$ are obtained from the product $X(7 : 13, [3\ 5\ 6]) * \mathbf{1}$, where $\mathbf{1}$ denotes the vector of ones. The product has a 2 at locations corresponding to rows 7, 8, and 12 of X and the associated triangles are $(1, 3, 5), (1, 3, 6)$, and $(1, 5, 6)$. Therefore, there are three triangles incident on vertex 1, and it can be easily verified that the graph contains a total of 7 triangles across all of the vertices.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

B. Data Structure

In our preliminary implementation, we use two arrays to store useful information that can be computed after we sort the edges. `FDC` (Forward Degree Cumulative) is an array of size n, with elements corresponding to the total number of "forward neighbours" across the vertices of a graph. With the vertices of the graph labelled in a natural order, finding the forward degree of a vertex j can be calculated as `fd = FDC[j+1] - FDC[j]`. `FN` is an array of size m that stores *which* vertices are the forward neighbours of a vertex j. Using `FN` we can find these forward neighbours of j as `fn(j) = FN[k]`, where k ranges from `FDC[j]` to `FDC[j+1]-1`. The arrays `FDC` and `FN` thus save the vector-matrix products needed to find the forward neighbours. Figure 2 displays the arrays `FDC` and `FN` for the example graph from Figure 1.

FN =	3	5	6	5	6	7	5	6	7	5	7	6	7
FDC =	1		4		7		10		12		13		14

Fig. 2. FN and FDC for the Example Input Graph.

C. fullCount Algorithm

Let j be the column (vertex) of matrix X (graph G) currently being processed in the **fullCount** algorithm given below. For each pair of forward neighbours j' and j'' there is an edge between them if and only if both of the corresponding columns contain a 1 in some row l identifying the triangle (j, j', j''). In terms of the matrix-vector multiplication in line 7 of algorithm **fullCount**, vector T will get updated as $T(k) \leftarrow 2$. Thus the triangle (j, j', j'') can be enumerated immediately. The local triangle count and local edge support (number of triangles incident on an edge) are dynamically updated with this same information and stored in the **vertDeg** and **edgeDeg** arrays respectively, as shown in Figure 3 for the example input graph. The running total of number of triangles in the graph G is updated and stored in *count* as each vertex is processed.

edgeDeg =	2	2	2	1	2	1	2	3	1	0	0	3	2
vertDeg =	3		2		4		0		4		6		2

Fig. 3. vertDeg and edgeDeg for the Example Input Graph.

fullCount (X)
Input: Intersection matrix X
1: Compute FDC ▷ Forward degree cumulative
2: Compute FN ▷ Forward neighbour
3: $count \leftarrow 0$ ▷ Number of triangles in G
4: **for** $j = 1$ to $n - 1$ **do** ▷ $j \in V$, where V is the vertex set
5: $fd \leftarrow FDC[j + 1] - FDC[j]$ ▷ fd: forward degree
6: **if** $fd > 1$ **then**
7: $T = X([FDC[j + 1] : m], fnj) * \mathbf{1}$
8: $S = \{t \mid T[t] = 2\}$
9: **if** $S \neq \emptyset$ **then**
10: $count \leftarrow count + |S|$
11: **for** $t \in S$ **do**
12: update vertDeg ▷ Local triangle count
13: update edgeDeg ▷ Local edge support
14: **return** $count$, $vertDeg$, and $edgeDeg$

D. Triangle Centrality

1) Using GraphBLAS: A calling card of GraphBLAS is the elegance by which graph algorithms can be expressed using linear algebra. Such formulations are easy to understand and give users the ability to quickly develop implementations for various problems [16], [19], [20]. Li and Bader's [16] implementation of triangle centrality uses SuiteSparse:GraphBLAS version 5.1.5 to cast the equation from Section I-A into a linear algebraic formulation. Their algorithm requires two inputs, an adjacency matrix A in sparse matrix representation, and the graph triangle matrix T in sparse matrix representation, where $T = A^2 \circ A$, that is, the Hadamard product of the *graph triangle matrix* T [5] and A. T is computed with a masked matrix multiply in GraphBLAS, which saves space by only computing entries in the pattern of A and not all of A^2. From these inputs, a binary matrix $\hat{T}$ and the remaining components that comprise the notion of triangle centrality vector C are computed to yield

$$C = \frac{(3A - 2\hat{T} + I)T\mathbf{1}}{\mathbf{1}^T T\mathbf{1}}$$

2) Using Intersection Representation: We use our `TC` algorithm to calculate triangle centrality by taking the local triangle count and edge support information from **fullCount** and performing additional computations on each vertex of the network instance. This is done in the function **Neighbour** by creating a binary $neighbours$ array of size $2m$ that is formulated by iterating across each vertex in the graph and examining the local edge support of its neighbours, giving a 1 if the support is nonzero and 0 otherwise. `TC` then iterates over each vertex j in the input graph and uses $neighbours$ to determines whether its neighbour k forms a triangle with it. If k does, a direct endorsement of j's importance is identified and k's local triangle count is added to the *core triangle sum* (CTS) of vertex j's centrality, which is finally divided by three once all of its neighbours have been evaluated to account for the undirectedness of each triangle within the network. This sum is the left side of the numerator from the equation stated in Section I-A. Otherwise, k's local triangle count is added to the reference vertex's *non-core triangle sum*, which signifies an indirect endorsement of importance. These two totals are added together to form the total triangle sum and then divided by the total triangle count in the graph to give vertex j's triangle centrality. Performing the outlined steps across an entire network produces a triangle centrality table with a ranking of each vertex, such as the one for the example input graph shown in Table I.

The full triangle centrality algorithm is given below.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

TABLE I
TRIANGLE CENTRALITY TABLE FOR EXAMPLE INPUT GRAPH

V	CTS/3	NCTS	Centrality	Rank
1	17/3	0	17/21	5
2	14/3	0	14/21	6
3	19/3	0	19/21	2
4	0	6	6/7	4
5	19/3	0	19/21	2
6	21/3	0	21/21	1
7	14/3	0	14/21	6

3) Triangle Centrality Algorithm:

TC (X, $vertDeg$, $count$, $neighbours$)
Input: Intersection matrix X, $vertDeg$, $count$, $neighbours$

for $j = 1$ to n **do** ▷ $j \in V$, where V is the set of vertices
  $CTS \leftarrow vertDeg[j]$ ▷ CTS: $core\_triangle\_sum$
  $NCTS \leftarrow 0$ ▷ NCTS: $non\_core\_triangle\_sum$
  for $k \in neighbours[j]$ **do**
    if k forms a triangle with j **then**
      $CTS \leftarrow CTS + vertDeg[k]$
    else
      $NCTS \leftarrow NCTS + vertDeg[k]$
  $TC[j] \leftarrow \frac{\frac{1}{3} \times CTS + NCTS}{count}$
return TC

III. PARALLEL IMPLEMENTATION OF TRIANGLE CENTRALITY

A. Parallel fullCount Algorithm

We have discussed the serial version of our **fullCount** algorithm in Section II-C, and here we present the parallel version. Each thread counts triangles using the private variable, loc_count in the parallel region, and $loc\_vertDeg$ and $loc\_edgeDeg$ track the local triangle count and edge support for each vertex. After all the threads complete their task locally, the master thread updates all the variables in the critical region to avoid a race condition.

ParallelFullCount (X)
Input: Intersection matrix X

Calculate FDC ▷ Forward degree cumulative
Calculate FN ▷ Forward neighbour
$count \leftarrow 0$ ▷ Number of triangles
parallel region
  $loc\_count \leftarrow 0$ ▷ Local number of triangles
  do in parallel
    for $j = 1$ to $n - 1$ **do**
      $fd \leftarrow FDC[j+1] - FDC[j]$
      if $fd > 1$ **then**
        $T = X([FDC(j+1) : m], fn_j) * \mathbf{1}$
        $S = \{t \mid T[t] = 2\}$
        if $S \neq \emptyset$ **then**
          $loc\_count \leftarrow loc\_count + |S|$
          for $t \in S$ **do**
            update $loc\_vertDeg$
            update $loc\_edgeDeg$
    critical region
      $count \leftarrow count + loc\_count$
      for $j = 1$ to m **do**
        $edgeDeg[j] \leftarrow edgeDeg[j] + loc\_edgeDeg[j]$
        if $j \leq n$ **then**
          $vertDeg[j] \leftarrow vertDeg[j] + loc\_vertDeg[j]$
return $count$, $vertDeg$, and $edgeDeg$

B. Parallel Triangle Centrality Algorithm

The **ParallelTC** algorithm is a straightforward extension of its serial counterpart because the triangle centrality of each vertex $j \in V$ is calculated independently. There is no fear of race conditions occurring in the parallel for loop, greatly streamlining computations.

ParallelTC (X, $vertDeg$, $count$, $neighbours$)
Input: Intersection matrix X, $vertDeg$, $count$, $neighbours$

do in parallel
  for $j = 1$ to n **do**
    $CTS \leftarrow vertDeg[j]$ ▷ CTS: $core\_triangle\_sum$
    $NCTS \leftarrow 0$ ▷ NCTS: $non\_core\_triangle\_sum$
    for $k \in neighbours(j)$ **do**
      if k forms a triangle with j **then**
        $CTS \leftarrow CTS + vertDeg[k]$
      else
        $NCTS \leftarrow NCTS + vertDeg[k]$
    $TC[j] \leftarrow \frac{\frac{1}{3} \times CTS + NCTS}{count}$
return TC

IV. NUMERICAL RESULTS

We refer to the serial implementation by Li and Bader [16] as `TC-GrB` and our algorithm as `TC-Intersection`. We report the running times for **fullCount**, the creation of the $neighbours$ array in **Neighbour**, and the triangle centrality calculation performed in **TC** separately and then sum these times to compare with `TC-GrB`. Our implementation language was C++ and the code was compiled with a g++ version 4.4.7 compiler in an ASUS VivoBook Flip 14 PC with a 7th Gen Intel Core I5-7200U Processor (Dual Core, with 2.5GHz and 5GB RAM), running Linux Mint 19. Both implementations were tested in the same environment so that direct comparisons could be run, and reported times were averaged from 10 trials.

Table II shows the performance comparison between `TC-GrB` and our `TC-Intersection` serial triangle centrality algorithm. The test instances include the graphs from [16] as well as four additional standard benchmark instances [21]. For each test instance, the table lists the number of vertices and the number of edges, as well as the number of triangles. The running time for the algorithms are in seconds. For `TC-Intersection` we give a breakdown of the overall running time: time for `fullCount`, time to compute the neighborhood information `Neighbour`, and time to perform the actual centrality calculation `TC`. As shown by the time breakdown, the majority of the work is done in the *fullCount*

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

TABLE II
PERFORMANCE COMPARISON BETWEEN GRAPHBLAS AND OUR INTERSECTION MATRIX ALGORITHM FOR IMPLEMENTING TRIANGLE CENTRALITY

Graph Characteristics				Time in Seconds				
Name	$\|V\|$	$\|E\|$	$\Delta(G)$	TC-GrB	TC-Intersection			
					fullCount	Neighbour	TC	Total
com-DBLP	317080	1049866	2224385	0.46	0.23	0.08	0.01	**0.32**
roadNet-CA	1971281	2766607	120676	**0.13**	0.06	0.06	0.02	0.14
web-Google	916428	4322051	13391903	4.29	3.20	0.63	0.06	**3.89**
com-Youtube	1134890	2987624	3056386	**3.23**	3.42	1.44	0.03	4.89
as-Skitter	1696415	11095298	28769868	**9.63**	60.82	15.73	0.08	76.62
cit-Patents	3774768	16518947	7515023	11.52	2.65	1.95	0.31	**4.90**
com-LiveJournal	3997962	34681189	177820130	**45.93**	43.13	5.59	0.33	49.05
com-Orkut	3072441	117185083	627584181	645.20	453.31	46.63	1.29	**501.23**

algorithm, as the triangle centrality calculation done in `TC` makes up a small fraction of the total time. As shown in the table, the overall performance of `TC-Intersection` and `TC-GrB` is comparable, the lone exception being *as-Skitter*. On the largest instance `com-Orkut`, `TC-Intersection` is faster. We remark that the numerical experiments as presented reflects the assumption that we perceive triangle counting tasks of computing $\Delta(G)$, *vertDeg*, and *edgeDeg* (as in algorithm **fullCount**) as basic or kernel analytics which form essential ingredients in exploring for further insights into the underlying structural or functional characteristics of the network via calculations such as *triangle centrality* [3], *K-count* [13], [22], *K-Truss* [23], [24] etc. On the other hand, for an efficient implementation of a specific calculation e.g., the triangle centrality, the edge support calculation in algorithm **fullCount** is unnecessary. As edge support calculation is computationally more expensive than the local triangle count calculation, the running time for computing just the local triangle degree vector *vertDeg* is expected to be much smaller than **fullCount**. Moreover, triangle neighbourhood information (shown under label "Neighbour" in Table II) can be obtained directly from *vertDeg* calculations. Comparisons on larger graphs were not performed due to capacity limits within the testing environment and will be the subject of further study.

A. Scalability of the Parallel Implementation

This section contains experimental results from selected test instances where we run our parallel implementation of *TC-Intersection* on the High-Performance Computing system (Graham Cluster) at Compute Canada. The implementation language was C++, and we varied the number of threads between 1 and 8 using OpenMP directives. The selected test instances came from two groups, the first being brain networks from the Network Repository [25] that had between 15 and 268 million edges and up to 42 trillion triangles. The second set was comprised of the larger `Graph-500` synthetic test instances from GraphChallenge [26]. Table III gives the graph statistics for each tested instance, and includes the relative density of the networks, which we calculated as the ratio of edges present in the graph divided by the number of edges that would be present if the graph was complete:

$$Density = \frac{|E|}{\frac{|V|\times(|V|-1)}{2}}$$

Figures 4 and 5 depict the scalability results for brain networks and `Graph-500` instances respectively, when the number of threads are increased from 1 to 8. The shared memory parallel `TC-Intersection` shows good scalability on both sets of test instances with a slight edge with the brain graph instances, likely due to their increased density compared to the sparser `Graph-500` instances. This could affect the load balancing for each thread in the fullCount algorithm, where the more even work for each node within the brain networks could reduce idling time.

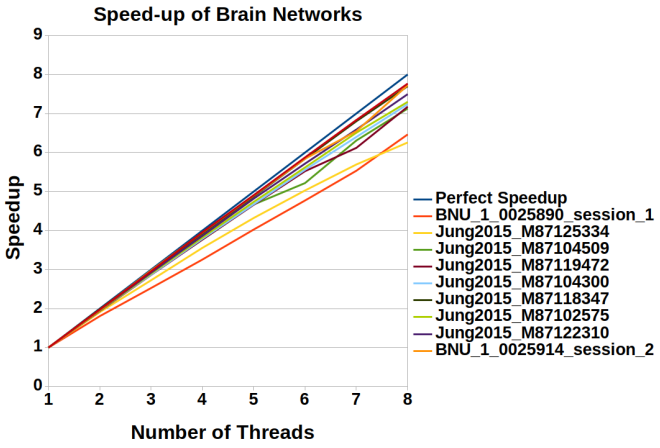

Fig. 4. Speed-up of Brain Networks for Parallel **TC-Intersection**

V. CONCLUSION

Network data is usually input as a list of edges which can be preprocessed into a representation such as an adjacency matrix or adjacency list, suitable for algorithmic processing. We have presented a simple yet flexible scheme based on intersecting edge labels, the intersection matrix, for the representation of and calculation with network data. A novel linear algebra-based method exploits this intersection representation for triangle computation – a kernel operation in big data

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

TABLE III
GRAPH CHARACTERISTICS OF BRAIN AND GRAPH–500 NETWORK INSTANCES

Name (prepended with "bn-human")	$\|V\|$	$\|E\|$	$\Delta(G)$	Density
BNU_1_0025890_session_1	177,584	15669037	662694994	9.94E-04
Jung2015_M87125334	763,149	40258003	1515479025	1.38E-04
Jung2015_M87104509	737,579	50037313	2512591873	1.84E-04
Jung2015_M87119472	835,832	59548327	3023865951	1.70E-04
Jung2015_M87104300	851,113	67658067	3516573387	1.87E-04
Jung2015_M87118347	428,842	79114771	6884218472	8.60E-04
Jung2015_M87102575	935,265	87273967	4732564614	2.00E-04
Jung2015_M87122310	924,284	94370886	5577667716	2.21E-04
BNU_1_0025914_session_2	701,145	103134404	9531928703	4.20E-04
BNU_1_0025916_session_1	714,571	112519748	10147347192	4.41E-04
Graph-500 Instances (prepended with "graph500")				
scale18-ef16	262144	4194304	82287285	1.22E-04
scale19-ef16	524288	8388608	186288972	6.10E-05
scale20-ef16	1048576	16777216	419349784	3.05E-05
scale21-ef16	2097152	33554432	935100883	1.53E-05
scale22-ef16	4194304	67108864	2067392370	7.63E-06

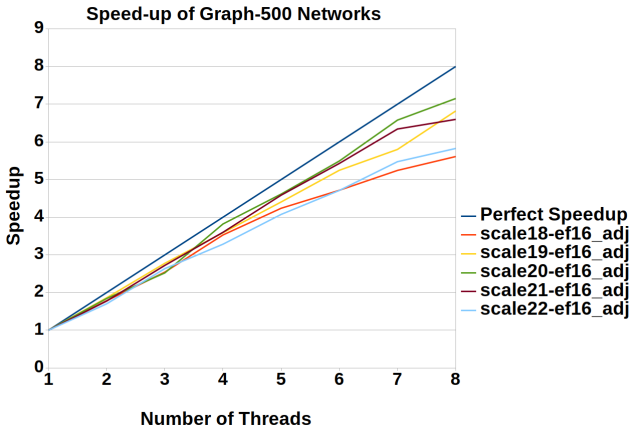

Fig. 5. Speed-up of Graph-500 Networks for Parallel **TC-Intersection**

analytics. The two arrays FDC and FN together constitute a compact representation of the sparsity pattern of network data, requiring only $n+m$ units of storage. This is incredibly useful in the exchange of network data, with the potential to allow for the computation of many intersection-matrix-based network analytics, of which triangle centrality is one application. By drawing from both direct and indirect triangle endorsements of vertices, triangle centrality stands apart from other centrality measures and has the potential to provide better performance in network analysis, for example, in spam node detection. This paper demonstrated the effectiveness of our *TC-Intersection* algorithm for computing the triangle centrality of a vertex and showed competitive comparative results with the implementation of GraphBlAS. We also remark that the *TC-Intersection* algorithm can be made significantly faster by avoiding unnecessary computations as outlined in Section IV. The parallel implementation of *TC-Intersection* showed promising scalability results on large synthetic and real-world instances, and cache efficiency is being studied for additional optimizations within the algorithm, exploring temporal and spatial locality to analyze the memory footprint and provide further improvements.

ACKNOWLEDGMENT

This research was supported in part by NSERC Discovery Grant (Individual). A part of our computations was performed on Compute Canada HPC system (http://www.computecanada.ca), and we gratefully acknowledge their support.

REFERENCES

[1] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 30, pp. 107–117, 1998. [Online]. Available: http://www-db.stanford.edu/ backrub/google.html

[2] P. Bonacich, "Factoring and weighting approaches to status scores and clique identification," *The Journal of Mathematical Sociology*, vol. 2, no. 1, pp. 113–120, 1972. [Online]. Available: https://doi.org/10.1080/0022250X.1972.9989806

[3] P. Burkhardt, "Triangle centrality," *ArXiv*, vol. abs/2105.00110, 2021.

[4] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with kokkoskernels," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.

[5] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, no. 3, pp. 157–166, 2017.

[6] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 2, p. e1226, 2018.

[7] T. M. Low, V. N. Rao, M. Lee, D. Popovici, F. Franchetti, and S. McMillan, "First look: Linear algebra-based triangle counting without matrix multiplication," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–6.

[8] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Graphchallenge.org triangle counting performance," IEEE HPEC, 2020.

[9] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient algorithms for large-scale local triangle counting," in *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 4, no. 3. ACM New York, NY, USA, 2010, pp. 1–28.

[10] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[11] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *nature*, vol. 435, no. 7043, pp. 814–818, 2005.

[12] W. M. Abdullah, D. Awosoga, and S. Hossain, "Intersection representation of big data networks and triangle counting," in *2021 IEEE International Conference on Big Data (Big Data)*, 2021, pp. 5836–5838.
[13] ——, "Intersection representation of big data networks and triangle enumeration," in *International Conference on Computational Science*. Springer, 2022, pp. 413–424.
[14] E. Szpilrajn-Marczewski, "A translation of sur deux propriétés des classes d'ensembles by," *Fund. Math*, vol. 33, pp. 303–307, 1945.
[15] M. Hasan, S. Hossain, A. I. Khan, N. H. Mithila, and A. H. Suny, "DSJM: a software toolkit for direct determination of sparse Jacobian matrices," in *International Congress on Mathematical Software*. Springer, 2016, pp. 275 – 283.
[16] F. Li and D. A. Bader, "A graphblas implementation of triangle centrality," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–2.
[17] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–9.
[18] W. M. Abdullah, S. Hossain, and M. A. Khan, "Covering large complex networks by cliques—a sparse matrix approach," in *Recent Developments in Mathematical, Statistical and Computational Sciences*, D. M. Kilgour, H. Kunze, R. Makarov, R. Melnik, and X. Wang, Eds. Cham: Springer International Publishing, 2021, pp. 117–127.
[19] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.
[20] M. Pelletier, W. Kimmerer, T. A. Davis, and T. G. Mattson, "The graphblas in julia and python: the pagerank and triangle centralities," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.
[21] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014, accessed: 2022-07-08.
[22] M. M. Wolf, J. W. Berry, and D. T. Stark, "A task-based linear algebra building blocks approach for scalable graph analytics," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015, pp. 1–6.
[23] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, no. 3.1, 2008.
[24] P. Burkhardt, V. Faber, and D. G. Harris, "Bounds and algorithms for k-truss," *arXiv preprint arXiv:1806.05523*, 2018.
[25] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Twenty-ninth AAAI conference on artificial intelligence*, 2015. [Online]. Available: https://networkrepository.com
[26] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," http://graphchallenge.mit.edu/data-sets, IEEE HPEC, 2017, accessed: 2021-07-09.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE