Linear Algebraic Formulation of Edge-centric K-truss Algorithms with Adjacency Matrices

Tze Meng Low, Daniele G. Spampinato, Anurag Kutuluru, Upasana Sridhar, Doru Thom Popovici, Franz Franchetti
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, USA
{lowt, spampinato}@cmu.edu, {akutulur,upasanas}@andrew.cmu.edu, {dpopovic, franzf}@cmu.edu

Scott McMillan
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
smcmillan@sei.cmu.edu

***Abstract*—Edge-centric algorithms using the linear algebraic approach typically require the use of both the incidence and adjacency matrices. Due to the two different data formats, the information contained in the graph is replicated, thereby incurring both time and space for the replication. Using K-truss as an example, we demonstrate that an edge-centric K-truss algorithm, the Eager K-truss algorithm, can be identified from a linear algebraic formulation using only the adjacency matrix. In addition, we demonstrate that our implementation of the linear algebraic edge-centric K-truss algorithm out-performs a Galois' K-truss implementation by an average (over 53 graphs) of more than 13 times, and up to 71 times.**

***Index Terms*—Graph Algorithms, Edge-centric Algorithms, K-truss, Linear Algebra, High Performance**

I. Introduction

The linear algebraic approach to graph algorithms is commonly associated with vertex-centric graph algorithms and "Think-like-a-vertex" graph frameworks [1], [2]. Under the vertex-centric paradigm, algorithms are described in terms of operations on vertices, while edges are viewed as conduits for passing input and output values between vertices. As such, it is conventional wisdom that edge-centric operations such as K-truss [3], an important graph operation for identifying cohesive groups, either 1) are inefficient when formulated as linear algebra operations [4], or 2) require the use of the incidence matrix which has to be instantiated prior to the start of the algorithm [5].

We make the observation that all graph formats contain the same information. This means that the information required for computing the K-truss of a graph must already be present in the adjacency matrix. This implies that identifying edge-centric K-truss algorithms can be distilled down to finding out how edges are stored in the adjacency matrix, and iterating over the storage format in an appropriate manner to compute the support value of all edges.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM18-0852]. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

In this paper, we introduce the Eager K-truss algorithm, a linear algebraic formulation of an edge-centric K-truss algorithm without the use of the incidence matrix. We demonstrate that, over 16 cores, the parallel Eager K-truss algorithm attains performance that, on average, is more than four orders of magnitute faster than the sequential reference implementation provided by the Graph Challenge [5], and, on average, more than 13 times faster than the 2017 Graph Challenge Champions' implementation [4].

II. Linear Algebraic K-truss

The K-trusses of a simple undirected graph $\mathcal{G}$ are subgraphs $\mathcal{G}_k$, where every edge in $\mathcal{G}_k$ is a part of $k-2$ triangles in $\mathcal{G}_k$. The number of triangles plus two is defined as the support of an edge. A common approach for computing the K-trusses of $\mathcal{G}$ is to use a two-step process where 1) the support of every edge is computed, and 2) edges with insufficient support are removed. This two steps are repeated until either no edges are removed, or all edges have been removed.

A. Computing edges support in an undirected simple graph

Consider an undirected edge $e = (u, v)$ that connects vertices u and v. Since u and v are the two end-points of e, the number of triangles containing e must be the number of common vertices in the neighborhoods of u and v, i.e.,

$$sup(e) = |N(u) \cup N(v)|, \tag{1}$$

where $sup(e)$ is the support of e and $N(x)$ is the neighborhood, or the set of vertices connected to x. A linear algebraic formulation of Equation 1 is

$$\begin{aligned} sup(e) &= (A_0 u)^T (A_0 v) \\ &= u^T A_0^T A_0 v, \end{aligned}$$

where u and v are basis vectors, i.e., vectors of zeros with a single one in the u^{th} or v^{th} position, respectively; and A_0 is the adjacency matrix of the graph $\mathcal{G}$. Notice that pre and post multiplying $A_0^T A_0$ with basis vectors u and v simply picks out the element in the u^{th} row and v^{th} column of the matrix $A_0^T A_0$. To compute the support of all edges in the graph represented by the adjacency matrix A_0, one simply

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

reads off the elements in the appropriate location of $A_0^T A_0$. In other words, the support of all edges is given by the linear algebraic equation:

$$S = A_0^T A_0 \circ A_0, \tag{2}$$

where $\circ$ is the Hadamard product (i.e., element-wise multiplication). Notice that in the this case, the adjacency matrix A_0 is used as a mask so that only support values corresponding to actual edges in the graph are computed by Equation 2.

B. Eliminating edges with insufficient support

The next step in computing the K-trusses of $\mathcal{G}$ is to remove edges with insufficient support such that only edges with support greater or equal to $k-2$ are left. This can be implemented by first identifying a mask M that only keeps edges with support greater or equal to $k-2$, and then applying that mask on A_0. These two operations can be represented with following two linear algebraic formulas:

$$M = S \geq (k-2) \tag{3}$$
$$A_1 = A_0 \circ M. \tag{4}$$

Using the newly computed subgraph represented by A_1, we repeat the process of computing support, and eliminating edges by repeated computation of Equations 2-4. When no more edges can be removed,

$$G_k = S,$$

where G_k is the adjacency matrix of of the K-truss $\mathcal{G}_k$ of $\mathcal{G}$.

C. Introducing Symmetry

As we are only considering undirected edges, this means that A_0 is a symmetric matrix. As such, only the upper (or lower) triangular part of A_0 needs to be stored. Similarly, the output S is also symmetric and should be stored in the same way. Mathematically, we can capture the symmetry of A_0 and S by partitioning both A_0 and S into submatrices as follows:

$$S \rightarrow \left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline * & S_{BR} \end{array}\right) \text{ and } A_0 \rightarrow \left(\begin{array}{c|c} A_{0_{TL}} & A_{0_{TR}} \\ \hline * & A_{0}BR \end{array}\right),$$

where S_{TL}, S_{BR}, $A_{0}TL$, and $A_{0}BR$ are symmetric submatrices, S_{TR} and $A_{0_{TR}}$ are general submatrices, and $*$ represents submatrices of values in S and A that are not stored.

Substituting these quadrants into Equation 2, we obtain the following expressions that describe how values in different quadrants of the S can be computed using submatrices of A_0,

$$S_{TL} \equiv (A_{0_{TL}}^T A_{0_{TL}} + A_{0_{TR}} A_{0_{TR}}^T) \circ A_{0_{TL}} \tag{5}$$
$$S_{TR} \equiv (A_{0_{TL}}^T A_{0_{TR}} + A_{0_{TR}} A_{0_{BR}}) \circ A_{0_{TR}} \tag{6}$$
$$S_{BR} \equiv (A_{0_{TR}}^T A_{0_{TR}} + A_{0_{BR}}^T A_{0_{BR}}) \circ A_{0_{BR}}. \tag{7}$$

III. Edge-centric Eager K-truss Algorithm

The quintessential edge-centric K-truss algorithm iterates over the edges, computes the support of each edge, and determines if that edge should be eliminated. While it is possible to compute the edges in any order, it is always prudent to compute the edges in the order they are stored so as to improve memory accesses.

Without loss of generality, let us assume that the adjacency matrix is stored in compressed sparse row format (CSR), and only the upper triangular matrix is stored. The use of CSR implies that the support values of the edges are stored in a row-wise manner. Iterating over the edges, from the first row down, and in a left-to-right manner, one would compute the edges in a sequential manner.

For the remaining of the paper, we will use upper-case, lower-case and greek letters to represent matrices, column vectors, and scalar elements, respectively.

A. Eager Computation

We make the observation that computing the support values in a row-wise manner from top to bottom would mean that the partitions S_{TL} and S_{TR} of S have been computed. This in turn requires access to values in $A_{0_{TL}}$, $A_{0_{TR}}$, and $A_{0_{BR}}$. Furthermore, these partitions of A_0 are also used to update S_{BR} (as shown in Equation 7). Therefore, we want to use $A_{0_{TR}}$ to compute as much of S_{BR} as possible to reduce redundant memory accesses. As such, we are eagerly performing as much computation as possible; and hence the name of our proposed Eagar K-truss algorithm.

B. Deriving the updates

Let us assume that at the start of any given iteration, the values in S_{TL} and S_{TR} have been computed. In addition, S_{BR} has been partially updated such that

$$S_{BR} \equiv A_{0_{TR}}^T A_{0_{TR}} \circ A_{0_{BR}}. \tag{8}$$

To make progress towards computing S, we first identify the next row of S to be updated from S_{BR} by splitting S_{BR} into quadrants and isolating the diagonal element σ_{11} and the row vector s_{12}^T, i.e.,

$$\left(\begin{array}{c|c} S_{TL} & S_{TR} \\ \hline * & S_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c} S_{00} & \left(\begin{array}{c|c} s_{01} & S_{02} \end{array}\right) \\ \hline \left(\begin{array}{c} * \\ \hline * \end{array}\right) & \left(\begin{array}{c|c} \sigma_{11} & s_{12}^T \\ \hline * & S_{22} \end{array}\right) \end{array}\right).$$

By partitionining A_0 in a similar fashion as S, i.e.,

$$\left(\begin{array}{c|c} A_{0_{TL}} & A_{0_{TR}} \\ \hline * & A_{0_{BR}} \end{array}\right) \rightarrow \left(\begin{array}{c|c} A_{0_{00}} & \left(\begin{array}{c|c} a_{0_{01}} & A_{0_{02}} \end{array}\right) \\ \hline \left(\begin{array}{c} * \\ \hline * \end{array}\right) & \left(\begin{array}{c|c} \alpha_{0_{11}} & a_{0_{12}}^T \\ \hline * & A_{0_{22}} \end{array}\right) \end{array}\right),$$

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

and subsituting the appropriate submatrices and subvectors of A_0 into Equation 8, we obtain the following expressions for the values of σ_{11}, s_{12}^T, and S_{22}:

$$\begin{aligned} S_{BR} &\equiv A_{0_{TR}}^T A_{0_{TR}} \circ A_{0_{BR}} \\ \left(\begin{array}{c|c} \sigma_{11} & s_{12}^T \\ \hline * & S_{22} \end{array}\right) &\equiv \left(\begin{array}{c|c} a_{0_{01}} & A_{0_{02}} \end{array}\right)^T \left(\begin{array}{c|c} a_{0_{01}} & A_{0_{02}} \end{array}\right) \\ &\quad \circ \left(\begin{array}{c|c} 0 & a_{0_{12}}^T \\ \hline * & A_{0_{22}} \end{array}\right) \\ &\equiv \left(\begin{array}{c|c} 0 & a_{0_{01}}^T A_{0_{02}} \circ a_{0_{12}}^T \\ \hline * & A_{0_{02}}^T A_{0_{02}} \circ A_{0_{22}} \end{array}\right). \end{aligned} \tag{9}$$

At the end of the iteration, we know that we will have computed both σ_{11} and s_{12}^T, and can move them above the thick lines such that we maintain our initial assumptions about the values in the different submatrices of S. This means that at the end of the iteration, the different partitions of S are in the following states:

$$\left(\begin{array}{c|c} \left(\begin{array}{c|c} S_{00} & s_{01}^T \\ \hline * & \sigma_{11} \end{array}\right) & \left(\begin{array}{c} S_{02} \\ \hline s_{12}^T \end{array}\right) \\ \hline \left(\begin{array}{c|c} * & * \end{array}\right) & S_{22} \end{array}\right)$$

Again, partitioning A_0 conformally into

$$\left(\begin{array}{c|c} \left(\begin{array}{c|c} A_{0_{00}} & a_{0_{01}}^T \\ \hline * & 0 \end{array}\right) & \left(\begin{array}{c} A_{0_{02}} \\ \hline a_{0_{12}}^T \end{array}\right) \\ \hline \left(\begin{array}{c|c} * & * \end{array}\right) & A_{0_{22}} \end{array}\right)$$

and substituting the appropriate partitions of A_0 into Equations 6 and 7, we obtain the following expressions for S_{TR} and S_{BR} at the end of an iteration:

$$\begin{aligned} S_{TR} &\equiv (A_{0_{TL}}^T A_{0_{TR}} + A_{0_{TR}} A_{0_{BR}}) \circ A_{0_{TR}} \\ \left(\begin{array}{c} S_{02} \\ \hline s_{12}^T \end{array}\right) &\equiv \left(\left(\begin{array}{c|c} A_{0_{00}} & a_{0_{01}} \\ \hline * & 0 \end{array}\right)^T \left(\begin{array}{c} A_{0_{02}} \\ \hline a_{0_{12}}^T \end{array}\right) + \right. \\ &\quad \left. \left(\begin{array}{c} A_{0_{02}} \\ \hline a_{0_{12}}^T \end{array}\right) A_{0_{22}}\right) \circ \left(\begin{array}{c} A_{0_{02}} \\ \hline a_{0_{12}}^T \end{array}\right) \\ &\equiv \left(\begin{array}{c} (A_{0_{00}}^T A_{0_{02}} + a_{0_{01}} a_{0_{12}}^T + A_{0_{02}} A_{0_{22}}) \circ A_{0_{02}} \\ \hline (a_{0_{01}}^T A_{0_{02}} + a_{0_{12}}^T A_{0_{22}}) \circ a_{0_{12}}^T \end{array}\right) \end{aligned} \tag{10}$$

$$\begin{aligned} S_{BR} &\equiv A_{0_{TR}}^T A_{0_{TR}} \circ A_{0_{BR}} \\ S_{22} &\equiv \left(\left(\begin{array}{c} A_{0_{02}} \\ \hline a_{0_{12}}^T \end{array}\right)^T \left(\begin{array}{c} A_{0_{02}} \\ \hline a_{0_{12}}^T \end{array}\right)\right) \circ A_{0_{22}} \\ &\equiv \left(A_{0_{02}}^T A_{0_{02}} + a_{0_{12}} a_{0_{12}}^T\right) \circ A_{0_{22}}. \end{aligned} \tag{11}$$

Using Equations 9, 10 and 11, we can identify the differences between the expressions for s_{12}^T and S_{22} at the start and end of the iteration. This tells us that updates in the body of the loop have to be

$$\begin{aligned} s_{12}^T &= s_{12}^T + a_{0_{12}}^T A_{0_{22}} \circ a_{0_{12}}^T \\ S_{22} &= S_{22} + a_{0_{12}} a_{0_{12}}^T \circ A_{0_{22}}. \end{aligned}$$

IV. Implementation

Our C implementation of the Eager K-truss algorithm is a direct translation of the algorithm described in the previous section, and is shown in Figure 1.

A. Optimizations

We highlight a number of optimizations that were performed in our implementation.

1) *Operation Fusion.* The updates to S_{22} and s_{12}^T both require the use of the subvector $a_{0_{12}}^T$, and the submatrix $A_{0_{22}}$. As such, we fuse the loops that compute the two updates to reduce redundant data movement.
2) *Size of support array.* Due to the partial update nature of the algorithm, an array is required to store the intermediate support values. As the maximum support is one less than the maximum degree of the vertices, the size of an element of the array is set to the smallest possible datatype. This increases data per byte of memory moved. For all experiments, `uint16_t` suffices.
3) *Packing the intermediate adjacency matrices A_i.* In each iteration of the K-truss algorithm, edges are filtered out from A_i to obtain a subgraph A_{i+1}. To avoid iterating over the edges that no longer contribute to the K-truss computation, we pack A_i to eliminate the removed edges. Packing is performed within each row, where only values in the column index array (`JA`) are packed. This leaves gaps between the end of one row and the start of another. The same memory location allocated for the initial adjacency matrix is reused so no additional memory is required for the packing.

B. Parallelism

Parallelism is introduced in our implementation using the OpenMP [6] `parallel for` construct. As there is a dependency between iterations of the loop that computes S, i.e., the support values, the `atomic` clause is used to ensure that all updates are performed correctly. Dynamic scheduling was used to reduce the effects of variation in the amount of work in each iteration of the loop that computes S. In addition, to avoid excessive cache conflicts, chunk sizes of 128 and 1024 were utilized.

V. Results

In this section, we compare the performance of our Eager K-truss algorithm implemented in C (compiled with `gcc` 4.8.5) against the serial Julia implementation[1] provided on the Graph Challenge website [7] run using Julia 0.6 [8], and the multi-threaded K-truss implementation in Galois [9], [10]. Datasets from the Stanford Network Analysis Project (SNAP) [11], and Measurement and Analysis on the WIDE Internet (MAWI) [12] were downloaded from the Graph Challenge website in tab-separated value (TSV) format.

A. Experimental Setup

We performed our experiments on a dual-socket machine with 256 GB DDR4 memory. Each CPU is an Intel i7 E5-2667 v3 (Haswell) with a frequency of 3.2GHz, 20 MB LLC cache, and eight physical cores. Based on the implementation, we performed the following preprocessing of the input dataset:

[1]The implementation was edited to avoid using deprecated features, and higher performance was attained.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

```
1  void eager_ktruss(uint32_t *IA, uint32_t *JA,            //input matrix in CSR format
2                    uint16_t *M,                           //array of support values
3                    uint32_t NUM_VERTICES, uint32_t K) {
4
5    bool notEqual=true;
6
7    while ( notEqual ){    //repeat until no edges are removed
8
9      notEqual = false;
10
11     #pragma omp parallel for num_threads(NUM_THREADS) schedule(dynamic, CHUNK)
12     for (uint32_t i = 0; i < NUM_VERTICES; ++i) {                    //iterate over every row
13
14       uint32_t a12_start = *(IA + i);
15       uint32_t a12_end = *(IA + i+1);
16       register uint32_t *JAL = JA + a12_start;
17
18       for (uint32_t l = a12_start; *JAL != 0 &&  l != a12_end; ++l) {  //and non-zero columns
19
20         uint32_t A22_start = *(IA + *(JAL));
21         uint32_t A22_end = *(IA + *(JAL) + 1);
22
23         JAL++ ;
24
25         uint16_t ML  = 0;
26         uint32_t *JAK = JAL;
27         uint32_t *JAJ = JA + A22_start;
28         uint16_t *MJ  = M + A22_start;
29         uint16_t *MK  = M + l + 1;
30
31         while (*JAK!=0 && *JAJ != 0 &&                             //check early termination
32                JAK != JA + a12_end && JAJ != JA + A22_end){
33
34           register uint32_t JAj_val = *JAJ;
35           register int update_val=(JAj_val == *JAK);
36
37           if ( update_val ) {
38             #pragma omp atomic
39             ++(*MK);
40           }
41
42           ML += update_val;
43
44           uint32_t tmp = *JAK;
45           uint32_t advanceK = (tmp <= JAj_val );
46           uint32_t advanceJ = (JAj_val <= tmp );
47           JAK += advanceK;
48           MK  += advanceK;
49           JAJ += advanceJ;
50
51           if ( update_val ) {
52             #pragma omp atomic
53             ++(*MJ);
54           }
55           MJ += advanceJ;
56         }
57         #pragma omp atomic
58         *(M+1)+=ML;
59       }
60     }
61
62     #pragma omp parallel for num_threads(NUM_THREADS) schedule(dynamic, CHUNK)
63     for(uint32_t n = 0; n < NUM_VERTICES; ++n) {
64
65       uint32_t st   = *(IA + n);
66       uint32_t end  = *(IA + n+1);
67       uint32_t *J   = JA + st;
68       uint32_t *Jk  = JA + st;
69       uint16_t *Mst = M + st;
70
71       for ( ;  *J != 0 && J != JA + end ; ++Mst,++J) {
72         if ( *Mst >= K - 2 ) {       //check if edge needs to be filtered
73           *Jk = *J;                  //keep it in packed format
74           Jk++;
75         }
76         *Mst = 0;                   //reset support value for next iteration
77       }
78       if ( Jk < J ) {               //some edges in this row has been deleted
79         notEqual = 1;               //no locking needed since always set to 1
80         *Jk = 0;
81       }
82     }
83
84   }
85
86 }
```

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

Fig. 1. Parallel implementation of the Eager K-truss algorithm

- *Julia.* No preprocessing was perform. All experiments were ran using the original TSV input files.
- *Galois.* Data was converted from the TSV files into Galois' proprietary binary format using the graph converter provided with Galois with the smallest (i.e., 32-bit integer) edge type option provided.
- *Eager K-truss.* The adjacency matrix in the TSV file was first converted into an upper triangular matrix and then stored into a compressed-sparse-row (CSR) binary format.

B. Correctness

We verified that our implementation reported a non-empty graph for maximum K-truss values (i.e., $k_{\max}$) reported in existing literature [13], [14]. In addition, we also checked for the the absence of $(k_{\max}+1)$-trusses. Finally, where possible, we checked that our implementation returns the same K-truss returned by the Galois implementation.

C. Performance Results

In Table I, we report execution time for all three K-truss implementations with $k = k_{\max}$. Execution times were measured after data has been loaded into memory. For Julia, we report timing in seconds (s), while execution times for Galois and our Eager K-truss implementation in milliseconds (ms). Time fields marked with a dash (-) indicate Julia and Galois executions that ran for more than an hour. Parallel implementations are run using 16 threads. As a number of options were available for the Galois implementation, we ran the default bulk synchronous algorithm, and also the bulk synchoronus Jacobi algorithm under a variety of do_all flags and reported the fastest timing obtained. In addition, we also report performance speedup attained by our implementation over both Julia and Galois.

On average, our sequential Eager K-truss outperforms the Julia baseline by three orders of magnitude on average. With 16 threads, the parallel Eager K-truss outperforms the Julia baseline by an average factor of 22,500 and peaks at more than 194,000 times faster.

With the exception of two out of 54 datasets, our parallel Eager K-truss implementation is consistently faster than the parallel Galois implementation by an average of 17 times. While the maximum speedup of 71.26 times was obtained on a relatively small graph, our parallel eager algorithm outperforms Galois by a factor between 1.79 and 5.94 times on large graphs (defined as graph with more than a million edges).

In Figure 2, we showed Millions of Edges Traversed per sec (METPS) for the different implementations of K-truss. On all graphs, the parallel Eager K-truss implementation with 16 threads has a METPS that is approximately an order of magnitute higher than the sequential implementation. For most graphs, the Eager K-truss attains a higher METPS than the Galois implementation. It is interesting to note that the overall trend in performance across multiple graphs is similar for the sequential and parallel Eager K-truss, and to a lesser extent the sequential Julia implementation. This suggests that the structure of the graphs affects the performance of the implementations significantly.

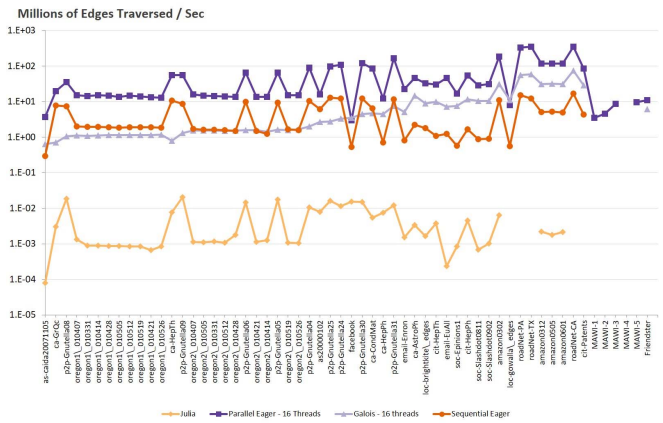

Fig. 2. Comparison of millions of edges traversed / sec (METPS) for different implementations.

VI. Conclusion & Future Directions

In this paper, we introduced the edge-centric Eager K-truss algorithm that was derived and formulated with linear algebra operations on the adjacency matrix. We demonstrated that the performance attained by an algorithm derived using a linear algebraic approach can be significantly higher than the performance based on other approaches. Specifically, we demonstrate that our parallel Eager K-truss implementation outperformed the Galois' K-truss implementation by up to 71 times.

Even better performance could be achieved by improving our memory consumption and introducing more appropriate NUMA-aware memory allocation schemes to prevent excessive cache coherency side effects and improve scalability.

In particular, while we have highlighted a number of edge-centric K-truss algorithms when the adjacency matrix is stored in the CSR format, we believe that a similar approach for other adjacency matrix storage format can yield different sets of edge-centric algorithms. This could include algorithmic variants more suitable for reducing thread divergence and minimizing communication, allowing for an extension of our approach to distributed-memory and highly parallel GPU- and FPGA-accelerated systems.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[3] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," 2008.

[4] C. Voegele, Y. S. Lu, S. Pai, and K. Pingali, "Parallel triangle counting and k-truss identification using graph-centric methods," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2017, pp. 1–7.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

TABLE I

PERFORMANCE COMPARISON OF PARALLEL EAGER K-TRUSS ALGORITHM ON 2-SOCKET 16-CORE INTEL HASWELL AGAINST A SEQUENTIAL JULIA AND PARALLEL GALOIS IMPLEMENTATIONS. DATASET IS SORTED BY INCREASING NUMBER OF EDGES IN THE GRAPH.

Graph	**Edges** (in thousands)	$k_{\max}$	**Sequential Julia** Time (s)	**Galois** Time (ms) 16 threads	**Eager** Time (ms) Sequential	**Eager** Time (ms) 16 threads	**Speed-up** over Julia Seq. vs seq.	**Speed-up** over Julia Par. vs seq.	**Speed-up** over Galois Par. vs par.
as-caida20071105	12.6	16	157.2	19.97	43.24	3.45	3,636	45,639	5.80
ca-GrQc	14.5	44	4.8	20.27	1.90	0.74	2,530	6,522	27.58
p2p-Gnutella08	20.8	5	1.1	19.82	2.83	0.59	399	1,906	33.48
oregon1_010407	22.0	14	16.7	19.96	11.03	1.46	1,518	11,508	13.72
oregon1_010331	22.0	16	24.8	20.22	11.20	1.53	2,216	16,201	13.20
oregon1_010414	22.5	15	25.6	20.13	11.69	1.49	2,190	17,227	13.55
oregon1_010428	22.5	15	26.0	19.80	11.92	1.55	2,178	16,705	12.74
oregon1_010505	22.6	14	26.4	19.82	12.22	1.65	2,161	16,029	12.03
oregon1_010512	22.7	15	26.6	19.96	11.87	1.55	2,242	17,143	12.86
oregon1_010519	22.7	15	27.0	20.18	12.17	1.62	2,216	16,603	12.43
oregon1_010421	22.7	15	34.6	19.94	12.05	1.73	2,869	19,969	11.52
oregon1_010526	23.4	14	27.7	20.18	12.84	1.84	2,154	15,012	10.96
ca-HepTh	26.0	32	3.4	33.06	2.46	0.46	1,369	7,256	71.26
p2p-Gnutella09	26.0	5	1.3	19.91	3.06	0.47	419	2,732	42.45
oregon2_010407	30.9	24	27.4	20.01	18.36	1.93	1,494	14,234	10.39
oregon2_010505	30.9	21	28.5	20.13	19.27	2.10	1,479	13,571	9.58
oregon2_010331	31.2	25	27.1	20.51	19.32	2.20	1,405	12,351	9.34
oregon2_010512	31.3	21	29.4	21.28	20.02	2.26	1,469	13,002	9.41
oregon2_010428	31.4	21	17.6	20.32	21.01	2.31	839	7,631	8.80
p2p-Gnutella06	31.5	4	2.2	19.84	3.22	0.48	674	4,517	41.33
oregon2_010421	31.5	22	28.2	20.34	20.97	2.30	1,345	12,241	8.83
oregon2_010414	31.8	24	25.0	22.79	25.40	2.37	984	10,562	9.63
p2p-Gnutella05	31.8	4	1.8	19.95	3.45	0.49	533	3,786	41.14
oregon2_010519	32.3	24	30.3	20.34	19.70	2.13	1,537	14,227	9.56
oregon2_010526	32.7	25	31.5	19.96	20.80	2.17	1,515	14,542	9.22
p2p-Gnutella04	40.0	4	3.7	20.19	3.85	0.45	973	8,352	45.07
as20000102	53.4	10	6.8	19.74	8.71	3.34	783	2,046	5.92
p2p-Gnutella25	54.7	4	3.4	19.70	4.27	0.56	794	6,002	34.92
p2p-Gnutella24	65.4	4	5.7	19.66	5.38	0.61	1,065	9,452	32.43
facebook	88.2	97	5.7	25.64	168.78	29.88	34	192	0.86
p2p-Gnutella30	88.3	4	5.9	19.88	7.31	0.73	813	8,151	27.27
ca-CondMat	93.4	26	17.4	20.18	14.51	1.09	1,198	15,957	18.53
ca-HepPh	118.5	239	15.9	26.54	166.19	9.84	95	1,612	2.70
p2p-Gnutella31	147.9	4	12.3	19.81	12.98	0.89	948	13,814	22.23
email-Enron	183.8	22	120.2	36.60	229.12	8.28	524	14,518	4.42
ca-AstroPh	198.1	57	58.3	13.44	90.46	4.36	644	13,378	3.08
loc-brightkite_edges	214.1	43	129.3	24.03	118.61	6.59	1,090	19,622	3.65
cit-HepTh	352.3	30	94.1	35.84	324.12	11.73	290	8,025	3.05
email-EuAll	364.5	20	1,551.0	51.03	294.11	7.98	5,274	194,390	6.40
soc-Epinions1	405.7	33	479.3	53.64	702.46	23.98	682	19,987	2.24
cit-HepPh	420.9	25	93.8	36.01	253.12	7.85	371	11,956	4.59
soc-Slashdot0811	469.2	35	684.7	45.58	539.35	16.25	1,269	42,129	2.80
soc-Slashdot0902	504.2	36	495.8	49.08	566.18	16.54	876	29,970	2.97
amazon0302	899.8	7	140.9	29.46	83.10	4.95	1,696	28,494	5.96
loc-gowalla_edges	950.3	29	-	86.14	1732.51	120.52	-	-	0.71
roadNet-PA	1,541.9	4	-	27.53	101.38	4.74	-	-	5.81
roadNet-TX	1,921.7	4	-	32.65	159.47	5.49	-	-	5.94
amazon0312	2,349.9	11	-	76.92	465.10	20.31	-	-	3.79
amazon0505	2,439.4	11	-	76.88	470.23	20.90	-	-	3.68
amazon0601	2,443.4	11	-	79.36	498.07	21.02	-	-	3.77
roadNet-CA	2,766.6	4	-	37.27	165.23	7.92	-	-	4.71
cit-Patents	16,518.9	36	-	572.58	3843.52	194.28	-	-	2.95
MAWI-1	19,020.2	3	-	-	-	5,484	-	-	-
MAWI-2	37,242.7	3	-	-	-	8,216	-	-	-
MAWI-3	71,707.5	3	-	-	-	8,286	-	-	-
MAWI-4	135,117.4	Data was corrupted. Unable to convert to CSR							
MAWI-5	0.24M	3	-	-	-	24,756	-	-	-
Friendster	1.80M	129	-	298,333	-	166,450	-	-	1.79

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

[5] S. Samsi, V. Gadepally, M. B. Hurley, M. Jones, E. K. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. S. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," *CoRR*, vol. abs/1708.06866, 2017. [Online]. Available: arxiv.org/abs/1708.06866

[6] OpenMP Architecture Review Board, "OpenMP application program interface," November 2015. [Online]. Available: www.openmp.org

[7] "Graph Challenge," graphchallenge.mit.edu, 2017.

[8] "The Julia programming language," 2018. [Online]. Available: julialang.org/downloads

[9] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Programming Language Design and Implementation (PLDI)*, 2011, pp. 12–25.

[10] "Galois: C++ library for multi-core and multi-node parallelization," github.com/IntelligentSoftwareSystems/Galois/, 2018.

[11] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," snap.stanford.edu/data/, June 2014.

[12] M. W. Group, "Measurement and analysis on the wide internet," mawi.wide.ad.jp/mawi/, June 2014.

[13] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," *CoRR*, vol. abs/1707.02000, 2017. [Online]. Available: arxiv.org/abs/1707.02000

[14] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE