Vectorising k-Truss Decomposition for Simple Multi-Core and SIMD Acceleration

Amir Mehrafsa
Department of Computer Science
University of Victoria
Victoria, BC
mehrafsa@uvic.ca

Sean Chester
Department of Computer Science
University of Victoria
Victoria, BC
schester@uvic.ca

Alex Thomo
Department of Computer Science
University of Victoria
Victoria, BC
thomo@uvic.ca

***Abstract*—**
In this paper we tackle truss decomposition of large graphs, which is one of the popular tools for discovering dense hierarchical subgraphs in social and web networks; such subgraphs form the basis of community discovery, one of the cornerstones of modern graph analytics. Our goal is to offer a simple vectorisation approach which can be easily implemented in widely popular Python vector libraries, such as NumPy. This way has two advantages: (1) non-experts with basic knowledge of Python can implement our algorithm, and (2) they can obtain multi-threaded and SIMD parallelism "for free" without them needing to know about computer architecture or sophisticated C++ libraries for multi-threaded processing. We believe this is an important paradigm setting approach that opens the way for applying similar techniques to other problems that might seem at first remote to vectorisation and/or parallelisation.

***Index Terms*—graph analytics, k-truss decomposition, parallel algorithms, vectorization, SIMD, Python, Numpy**

I. Introduction

One of the most important graph analytics tasks for social and web networks is identifying dense subgraphs. Such subgraphs can reveal important information about networks, for example communities of tightly connected users or sites, important influential groups of nodes, and robustness of the network. There are several notions of dense subgraphs and among those, the notion of *truss* is particularly popular for extracting a hierarchical structure of dense subgraphs.

Specifically, the k-truss of a graph G is defined as the largest subgraph H_k such that each edge is part of at least k triangles. A triangle is clique on three vertices. If an edge (a, b) is part of a triangle $\langle a, b, c\rangle$, we can think of node c as "endorsing" the connection between a and b. The highest value of k for each an edge is part of the k-truss is called truss value of that edge. The collection of of all k-trusses for different k comprises the truss decomposition of the graph. Truss decomposition enables a hierarchical structure the truss subgraphs for different k because $H_k \subset H_{k-1}$ for all $k > 1$. Truss decomposition has been used in several important applications, such as community discovery, network visualization, clique computation, etc.

The standard approach to computing truss decomposition is the edge peeling process, which continuously removes edges with less than k triangles. Deleted edges can cause other edges to have less than k triangles, so peeling for a given k goes in waves until all remaining edges have k or more triangles. At this point, the edges with precisely k triangles have a truss value of k. This process is repeated after incrementing k until no edges remain, which results in finding the truss values of each edge. Then the k-truss can be formed by collecting all the edges with truss value equal or greater than k.

The complexity of this process is $O(E^{1.5})$, which is challenging for large graphs. Notably, there have been several works proposing to parallelize the computation of truss decomposition (cf. [1]–[3]). They are based on C/C++ libraries for multithreading, such as OpenMP.

In this work we follow a different approach. We take the classical peeling framework and vectorise it by expressing the whole process as a series of vector operations. By doing so we can re-purpose well-know matrix libraries to compute truss decomposition. We note that NumPy primitive operations are *vectorised* and *parallel* [4]–[6]. These primitive APIs are implemented efficiently in compiled languages like C, C++, and Fortran with SIMD (Single Instruction/Multiple Data) and multi-threading enabled [4], [7]. This way we obtain SIMD and multi-threading parallelism "for free" without the need to program in lower-level languages such as C/C++. Such an approach allows non-experts in parallel computing to benefit from multi-core and SIMD modern computer architectures. This is important because it is a well-known fact that people's experience with parallel programming is often quite frustrating to the point they want to avoid completely low-level multi-threaded programming and not have to deal with race conditions and other challenges associated with parallel programming. Their conclusion is that parallel programming is one of the most difficult and frustrating forms of programming they know of [7].

In order to make the implementation barrier as low as possible, we decided to use the most basic matrix Python library, NumPy, which is ubiquitous in most Python installations. Of course, there are other matrix libraries in Python, such as Numba and PyTorch, on which we can run our vectorised algorithm "as is" and benefit more from parallelisation, especially if a GPU is available (see our previous work [8] on a related problem). However, here we wanted to show that even using the most basic matrix library in Python we are still able to achieve substantial parallelism using a multi-core machine.

II. Background

We represent networks using undirected graphs. We denote an undirected graph by $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We set n and m to be $|V|$ and $|E|$, respectively.

For a vertex $u \in V$, the set of its neighbors is $N_G(u) = \{v : (u, v) \in E\}$. A *triangle*, Δ_{uvw}, in G is a set of three vertices $\{u, v, w\} \subseteq V$ such that all three edges (u, v), (v, w) and (u, w) exist in E. The *support* of an edge $e = (u, v)$ in G, denoted by $\sup_G(e)$, is defined as the number of triangles in G containing e. Formally, $\sup_G(e) = |N_G(u) \cap N_G(v)|$.

The *k-truss* of G is defined as the largest subgraph H of G in which each edge e has $\mathsf{sup}_H(e) \geq k$. The set of all k-trusses forms the truss decomposition of G, where $0 \leq k \leq k_{\max}$, and $k_{\max}$ is the largest support of any edge in G.

The *truss value* of an edge e, $\kappa(e)$, is the largest integer k for which e belongs to a k-truss.

Proposition 1. *The k-truss of G is the subgraph H of G containing all and only the edges e in G with $\kappa(e) \geq k$.*

Based on the above proposition, in this paper, we focus on finding the truss values of all the edges in an input graph. Then the k-truss for any k is constructed by collecting all edges e with $\kappa(e) \geq k$.

III. Existing Approaches

There are two independent steps to perform truss decomposition; triangle enumeration or support initialization for edges and then converging to truss values by methods like peeling. There are several approaches for each step, and numerous works are available based on the approach selected for each. Most of the state-of-the-art approaches start by initial counting of triangles with set intersection operator and then use an iterative approach to converge to the truss values [9].

Parallel processing via shared-memory is often chosen by researchers to speedup the truss decomposition [1], [3]. Authors of [1] showed that their PKT approach outperforms other approaches in a 24-core platform. Recently, Saryice et al. [2] generalized the shared memory approach parallelism for extracting hierarchical dense subgraphs. They used an effective approach for extracting triangles [10], [11] and then generalize the method of iterative h-index computation (cf. [12]) to find the truss value of each edge. Che et. al. in [13] introduced optimizations for accelerating truss decomposition in a CPU-GPU heterogeneous platforms. All these methods have C/C++ implementations that are not directly amenable to vectorisation.

Support initialization or triangle enumeration has a very high work efficiency and therefore there are two general categories of approaches for this step. In the first category, the graph is converted to an oriented graph by replacing the undirected edges with directed ones (typically from lower degree to higher degree node) and counting each triangle only once [14], [15]. The second category is merge-based or hash-based set intersection counting. The hash-based approaches materialize a hash table then they scan for the common neighbors in the hash table [16].

Algorithm 1: Vectorized K-Truss algorithm.

input : $S, E, D, O, P, P1, H, I$
output: K

$P10 = cuckoo_0(P1)$; $P11 = cuckoo_1(P1)$
// Cuckoo hash functions applied on every element of P1 in vectorized way

$C0 = H[0][P10] - P1$; $C1 = H[1][P11] - P1$
// Zeros in $C0$ and $C1$ indicate the intersection of $P1$ and P and hence give the triangles in the graph

$T0 = I[0][P10][C0 == 0]$; $T1 = I[1][P11][C1 == 0]$
$T = merge(T0, T1)$
// Edge indices of the triangles we find.

$G0, G1, G2 = extract\_edges(T)$
// Edge indices of the triangles in the graph. $G0, G1, G2$ have all lengths equal to the number of triangles

$K = bincount(G0)$
// Initial support count of edges

while *No triangle is left* **do**
  // Peeling process, see Section IV-D

IV. Proposed Approach

Our algorithm uses hash values of pairs of nodes in the graph to extract triangles. We generate two flat vectors of hash values. The first hash vector is simply computed by connected node pairs that form the edges in graph. In other words, each edge is going to have a hash value in the vector. The second hash vector is generated by values of node pairs that are connected to each other with one hop. A hash value that exists in both vectors implies an existing triangle in our graph. We store some metadata about the index of hashed values next to hash keys (see Section IV-B), so that we can compute the triangle count for each edge.

For the second step of k-truss computation, we follow the iterative peeling paradigm by removing edges and updating the support value of edges in each iteration to the number of the remaining support triangles. We continue the iterations until they converge to the actual support value. The main task of the proposed algorithm is to enumerate the number of triangles for edges and update the support values. Peeling off edges

translates into deleting hash values from our flat hash vectors and then re-scanning them to find matches on both vectors.

We used the available Numpy array API to apply parallelism and employ SIMD techniques when deleting and looking up in hash vectors. Algorithm 1 describes the overall procedure, but we break down the whole procedure into five subsections to simplify the explanation.

The algorithm requires six vectors as input, of which, four of them represent a flattened adjacency list. Vectors S and E are vectors of Origin (Source) and Destination nodes and are of length m.

Vector D is of length n and contains the degree of each vertex. Vector O (Offset Indices) is also of length n and provides the index in E where each vertex's neighbours begin. Vector P contains the Cantor function values for the edges in S and E. Cantor function maps two values into one and it will be explained in Section IV-A. Vector H contains the initial values of Cuckoo hashing we do on vector P and this will also be explained later, in Section IV-B.

A. Pre-processing and Creating Input Vectors

The pseudo-code for pre-processing is given in Algorithm 2.

Algorithm 2: Pre-processing step.

input : $S, E, h = 1.2$
output: $O, D, P, H, I, P1$

1 $S, E = remove\_duplicates(S, E)$
2 $S, E = remove\_selfloops(S, E)$
3 $O, D = unique(S)$
4 $S, E, D = oriented(S, E, D)$
5 $P = cantor(S, E)$
6 $H[], I[] = cuckoo(P, h)$
7 $S1, E1 = neighbors\_of\_one\_hop(S, E)$
8 $P1 = cantor(S1, E1)$ `// P1 is subset of P`

We assume there is no duplicate edge as well as self loops in the graph. Line 1 and 2 remove duplicates and self-loops from the given graph. Line 3 creates the Offset Indices O and Degree D vectors. Line 4 converts the dataset to an oriented graph. Recall that the graph is converted to an oriented graph by replacing the undirected edges with directed ones from lower degree to higher degree nodes.

In our algorithm we need to map a pair nodes to a unique number. We use for this Cantor Pairing Function [17]. We store the results of the Cantor function over vectors S and E in vector P. This is done in Line 5. Other hashing functions like [18] could be used in this step as an alternative.

The problem with the Cantor function is that it produces very large numbers. Using these generated large numbers in array primitive operations, for example, using them as indices, will have a significantly bad performance effect. As such, in our algorithm we used Cuckoo hashing in order to store the values produced by the Cantor pairing function in a much smaller vector. In Line 6, as the last step of pre-processing, we initialize the Cuckoo hashing object. The initialization of Cuckoo hashing object creates two hash vectors of size h. Value h is a factor of n and in our proposed algorithm we choose this factor to be 1.2. In case of collisions during the pre-processing step, a greater value can resolve the collision issue. However the greater the value of this factor, the more space the data will take, and the greater the runtime will be. Please see Section IV-B for more details on Cuckoo hashing we use.

B. Cuckoo hashing

As we described above, we use the Cantor pairing function in order to hash two integer values into one number and then employ Cuckoo hashing to map these numbers into smaller range.

Cuckoo hashing exhibits worst-case constant lookup time. Its name derives from the Cuckoo bird behavior, where the cuckoo chick kicks its young out of the nest when a younger bird is born; analogously, inserting a new key may kick out an older key to a different hash location.

Cuckoo hashing requires two hash functions and two tables. In Pseudocode 1, these functions are noted by $cuckoo_0()$ and $cuckoo_1()$. The size of the tables will directly affect the runtime of the algorithm, since it affects the locality and proximity of the values to the CPU. On the other hand, bigger tables will have less chance of hash collisions. The smaller the table sizes, the higher the risk of having collisions in our hashing scheme. In case of collisions, we can consider a greater size for the two vectors storing the hash values. In our experiments, for our datasets, we were able to apply Cuckoo hashing without collisions when the table sizes were just 1.2 times the actual edge size, i.e. $hash\_table\_size = 1.2 \times |E|$.

When inserting a new key key in the hash structure, we first try to insert it into the first table using the first hash function. If the slot is available, we put it there. If not, the key occupying the slot, key', is removed and key takes its place. Now we attempt to insert key' into the other table using the other hash function employing the same procedure, which by nature is recursive and continues until an empty slot is found for the dislocated keys or a predetermined recursion depth is reached.

We choose a lazy hash function as our first hashing function in order to speedup the hashing. This function takes the modulo of the input value to the size of the hash table. Since we used a lazy hash function for the first hash, the second hash function needs to have a very good statistical distribution. We choose an improved version of MurmurHash 3 as our second hash function.

Pseudocode 3 and 4 show the process of cuckoo hashing. In our algorithm, we use twice as many vectors as in the original Cuckoo hashing. We used the extra vectors to store the values for each key, which in our case are the indices of values from vector S and E that represent edges; this provides us an efficient way to retrieve those indices without extra computation.

C. Finding triangles of each edge

Figure 1 shows a simple graph. In this graph the edge connecting nodes 0 and 1 participates in forming two triangles

Algorithm 3: Cuckoo hashing *Place* function

input : $key, idx, tableId, nRecursiveCall$
output: $H[], I[]$

```
if nRecursiveCall > maxAllowed then
    throwError
    // Failed to handle collision,
       select higher h and rehash
       entries
pos = hash(key)
if available(pos, tableId) then
    H[tableId][pos] = key
    I[tableId][pos] = idx
    // idx is the value we store for key
else
    key' = H[tableId][pos]
    idx' = I[tableId][pos]
    H[tableId][pos] = key
    I[tableId][pos] = idx
    place(key', idx', (tableId +
      1)%2, nRecursiveCall + 1)
```

Algorithm 4: Cuckoo hashing initialization

input : P
output: $H[], I[]$

```
   // H and I contain two flat vectors
   each
idx = 0
for key in P do
    Place(key, idx, 0, 0)
    idx = idx + 1
```

$(0, 1, 3)$ and $(0, 1, 2)$. When an edge is part of a triangle it means that there is an edge directly connecting two nodes, stored in the edge list file (or in other words in the S and E vectors) and there are two other edges connecting these nodes with one hop only. So we need to find all the one-hop neighbors. The steps for this section can be done as follows:

1) Find and store one-hop pairs in vector $P1$ (see Line 8 in Algorithm 2).
2) Compute the intersection of $P1$ and P. In fact we perform this intersection using our Cuckoo hashing tables.

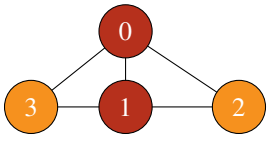

Fig. 1. Example graph

In Figure 1, the original direct pairs are:

$$(0,1),(0,2),(0,3),(1,0),(1,2),\\(1,3),(2,0),(2,1),(3,0),(3,1) \tag{1}$$

and the one-hop pairs for nodes 1 and 2 are:
$(0, 2, 1), (0, 3, 1), (1, 2, 0), (1, 3, 0)$
which will give intersection pairs:
$(0, 1), (0, 1), (1, 0), (1, 0)$.

In the main algorithm, the above intersection pairs will be stored in vector C. The support count for edges $(1, 2)$ and $(2, 1)$ should equal to 2. In our simple example:

$$\begin{gathered} S = [0\ 0\ 0\ 1\ 1\ 1\ 2\ 2\ 3\ 3] \\ E = [1\ 2\ 3\ 0\ 2\ 3\ 0\ 1\ 0\ 1] \\ D = [3\ 3\ 2\ 2] \\ O = [0\ 3\ 6\ 8] \\ D[E] = [3\ 2\ 2\ 3\ 2\ 2\ 3\ 3\ 3\ 3] \\ O[E] = [3\ 6\ 8\ 0\ 6\ 8\ 0\ 3\ 0\ 3] \\ P = [2\ 5\ 9\ 1\ 8\ 13\ 3\ 7\ 6\ 11] \end{gathered} \tag{2}$$

In order to achieve this in a vectorised way, we use the multi-arange operation defined and implemented in [8]. Namely, the *multi-arange* operation, $\diamond$, is a binary operation that transforms two equal-length vectors, S and C, into an output vector of length $\Sigma_{c \in C} c$. Vector S denotes a set of *start* indices and vector C denotes a set of *counts*. For each $(s_i, c_i) \in (S, C)$, it generates the series $s_i, s_i+1, \ldots, s_i+c_i-1$. For example, $[2\ 4\ 1] \diamond [2\ 1\ 3] = [2\ 3\ 4\ 1\ 2\ 3]$. This function is used in our algorithm to generate the indices of one hop neighbors and store those nodes in vector $E1$. We generate these multiple series with the start index of destination nodes, retrieved from $O[E]$ and degree of destination nodes $D[E]$, line 6 in Algorithm 1. We store the generated pairs in vectors $S1$ (Source Tiling) and $E1$ (Destination Neighbors). Vector m is materialized while calling $neighbors\_of\_one\_hope(S1, E1)$ function and is representing the connecting node of vectors $S1$ and $E1$.

$$\begin{gathered} S1 = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 2\ 2\ 3\ 3\ 3\ 3\ 3\ 3] \\ m = [3\ 4\ 5\ 6\ 7\ 8\ 9\ 0\ 1\ 2\ 6\ 7\ 8\ 9\ 0\ 1\ 2\ 3\ 4\ 5\ 0\ 1\ 2\ 3\ 4\ 5] \\ E1 = [0\ 2\ 3\ 0\ 1\ 0\ 1\ 1\ 2\ 3\ 0\ 1\ 0\ 1\ 1\ 2\ 3\ 0\ 2\ 3\ 1\ 2\ 3\ 0\ 2\ 3] \\ P1 = \begin{bmatrix} 0\ 5\ 9\ 0\ 2\ 0\ 2\ 4\ 8\ 13\ 1\ 4\ 1\ 4 \\ 7\ 12\ 18\ 3\ 12\ 18\ 11\ 17\ 24\ 6\ 17\ 24 \end{bmatrix} \end{gathered} \tag{3}$$

D. Peeling Step

Pseudocode 5 shows the peeling step with detailed comments. We count the number of triangles in line 3 and then

Algorithm 5: Peeling Step

input : $S, E, G0, G1, G2, T$
output: K

1 $k = 1$
2 **while** $T.size() > 0$ **do**
3 $T_u, T_c = unique(T)$ // unique triangle indices and their counts
4 **while** $(T_c == k).any()$ **do**
5 // If there is a triangle with count equal to k, its edges have support of k
6 $T_k = T_u[T_c == k]$ // obtain positions in E of edges in triangles with a count of k
7 $T_h = T_u[T_c > k]$ // obtain positions in E of edges in triangles with a count $> k$
8 $K[T_k] = k$
9 $K[T_h] = K[T_h] + 1$
10 // update support vector
11 $T = update(T, G0, G1, G2, T_K)$
12 // Remove T_k instances from T and recompute T
13 $k++$

extract the edges with support count of k in line 8. We set the support value of edges with k count to k in line 10 and 11. In line 11 we update the edge list by removing the edges that are peeled.

Observe that we stored the edge support numbers in vector K, so this vector will contain the truss values of the edges at the end of computation.

V. Experiments

A. Experimental setup

1) Implementation: Our vectorized algorithm is implemented using the NumPy library, version 1.19.5 with multithreading enabled, using Python 3.10. Our source code is publicly available.[1]

2) Datasets: The datasets are real data and taken from the Laboratory for Web Algorithmics [2] and Standford SNAP collection [3]. Isolated vertices have been removed and the directed graphs transformed to be undirected. Table I summarizes the statistical properties of the datasets.

- **Amazon-2008** (AM) is a dataset of books, where a bidirectional edge between books indicates that they are similar.
- **Blog-Catalog** (BC) is a social blog directory.
- **Digg** (DG) is a social news website that stores all links between users.

[1]https://github.com/mexuaz/ktrussvector
[2]http://law.di.unimi.it/datasets.php
[3]https://snap.stanford.edu

dataset	vertices	edges	triangles	k-max	k-avg
amazon-2008	735 K	7 M	4 M	8	3.03
blog-catalog	89 K	4 M	51 M	71	33.18
digg	283 K	9 M	20 M	46	16.29
four-square	639 K	6 M	22 M	31	17.89
cit-patent	4 M	33 M	8 M	18	1.13

TABLE I
STATISTICAL PROPERTIES OF THE DATASETS

dataset	1t	32t	speedup factor
amazon-2008	543	232	2.34
blog-catalog	23291	4391	5.30
four-square	48911	9233	5.29
digg	40125	29962	1.33
cit-patent	4911	1053	4.66

TABLE II
EXECUTION TIMES (SEC) FOR OUR ALGORITHM OVER SELECTED DATASETS, USING 1 (1T) AND 32 CORES (32T)

- **Four-square** (FS) is a list of user-to-user links for location-based online social network.
- **Cit-Patent** (CP) is Category of Citation Networks.

B. Results and Discussion

Table II shows the execution times in seconds for our algorithm on each dataset and each applicable core count, starting from the point that the graph has been loaded into memory and a common, flattened adjacency list has been created.

What we observe is that our simple Python implementation based on our vectorisation approach of k-truss decomposition can handle moderately large datasets and furthermore there is substantial speedup obtained by running in multiple cores. For example, the speedups for *blog-catalog* and *four-square* exceed 5x.

The proposed approach is not dependent on any hardware or operating system and the whole process is simply implemented by less than 5% lines of code, compared to C/C++ implementations of [1], [2], [13] which can take a lot of development time. Furthermore, we achieved parallelism "for free" by being able to vectorise the classical peeling approach as opposed to other approaches who needed to come up with a completely new algorithm and employ complex C++ libraries for multithreading. The latter approach can generally be implemented by experts in the field, whereas our vectorised approach is simple to the point that a non-expert with basic Python and NumPy experience can implement it. As such our approach serves as an example that vectorisation can take us a long way toward achieving parallelism for the masses who only possess basic computing skills and could be unable to use more complex tools and languages.

References

[1] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. Jaipur, India: IEEE, 2017, pp. 13–22.

[2] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *Proc. VLDB Endow.*, vol. 12, no. 1, pp. 43–56, Septemeber 2018. [Online]. Available: https://doi.org/10.14778/3275536.3275540

[3] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2017, pp. 1–6.

[4] "Parallel programming with numpy and scipy," https://scipy.github.io/old-wiki/pages/ParallelProgramming, accessed: 2022-05-13.

[5] "Numpyseveralcpus," https://roman-kh.github.io/numpy-multicore/, accessed: 2022-06-06.

[6] "Numpy simd optimizations," https://numpy.org/doc/stable/reference/simd/simd-optimizations.html, accessed: 2022-06-06.

[7] "Higher-order parallel programming," https://futhark-lang.org/blog/2020-05-03-higher-order-parallel-programming.html, accessed: 2022-05-11.

[8] A. Mehrafsa, S. Chester, and A. Thomo, "Vectorising k-core decomposition for gpu acceleration," in *32nd International Conference on Scientific and Statistical Database Management*, ser. SSDBM 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3400903.3400931

[9] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 812–823, may 2012. [Online]. Available: https://doi.org/10.14778/2311906.2311909

[10] M. Jha, C. Seshadhri, and A. Pinar, "Path sampling: A fast and provable method for estimating 4-vertex subgraph counts," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2015, pp. 495–505. [Online]. Available: https://doi.org/10.1145/2736277.2741101

[11] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield, "Efficient graphlet counting for large networks," in *2015 IEEE International Conference on Data Mining*. Atlantic City, NJ, USA: IEEE, 2015, pp. 1–10.

[12] L. Lü, T. Zhou, Q.-M. Zhang, and H. Stanley, "The h-index of a network node and its relation to degree and coreness," *Nature Communications*, vol. 7, p. 10168, 01 2016.

[13] Y. Che, Z. Lai, S. Sun, Y. Wang, and Q. Luo, "Accelerating truss decomposition on heterogeneous processors," *Proc. VLDB Endow.*, vol. 13, no. 10, pp. 1751–1764, June 2020. [Online]. Available: https://doi.org/10.14778/3401960.3401971

[14] R. Pearce, "Triangle counting for scale-free graphs at scale in distributed memory," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2017, pp. 1–4.

[15] R. A. Rossi, "Fast triangle core decomposition for mining large graphs," in *Advances in Knowledge Discovery and Data Mining*, V. S. Tseng, T. B. Ho, Z.-H. Zhou, A. L. P. Chen, and H.-Y. Kao, Eds. Cham: Springer International Publishing, 2014, pp. 310–322.

[16] A. S. Tom, N. Sundaram, N. K. Ahmed, S. Smith, S. Eyerman, M. Kodiyath, I. Hur, F. Petrini, and G. Karypis, "Exploring optimizations on shared-memory platforms for parallel triangle counting algorithms," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, 2017, pp. 1–7.

[17] M. P. Szudzik, "The rosenberg-strong pairing function," *CoRR*, vol. abs/1706.04129, pp. 1–5, 2017. [Online]. Available: http://arxiv.org/abs/1706.04129

[18] F. C. Botelho and N. Ziviani, "External perfect hashing for very large key sets," in *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, ser. CIKM '07, vol. 1, no. 1. New York, NY, USA: Association for Computing Machinery, 11 2007, pp. 653–662. [Online]. Available: https://doi.org/10.1145/1321440.1321532