Fast Triangle Core Decomposition for Mining Large Graphs

Ryan A. Rossi

Purdue University
rrossi@purdue.edu

Abstract. Large triangle cores represent dense subgraphs for which each edge has at least $k-2$ triangles (same as cliques). This paper presents a fast algorithm for computing the triangle core decomposition on big graphs. The proposed triangle core algorithm adapts both the computations and representation based on the properties of the graph. In addition, we develop a fast edge-based parallel triangle counting algorithm, which lies at the heart of the triangle core decomposition. The proposed algorithm is orders of magnitude faster than the currently available approach. We also investigate and propose fast methods for two variants of the triangle core problem: computing only the top-k triangle cores fast and finding the maximum triangle core number of the graph. The experiments demonstrate the scalability and effectiveness of our approach on 150+ networks with up to 1.8 billion-edges. Further, we apply the proposed methods for graph mining tasks including finding dense subgraphs, temporal strong components, and maximum cliques.

Keywords: Triangle-core decomposition, parallel triangle counting, maximum clique, temporal strong components, triangle-core ordering.

1 Introduction

Consider a graph $G = (V, E)$. A k-core of G is a maximal induced subgraph of G where each vertex has degree at least k. The k-core number of a vertex v is the largest k such that v is in a k-core. There is a linear time $O(|E| + |V|)$ algorithm to compute the k-core decomposition [2]. Due to this efficient algorithm, and a simple, but often powerful, interpretation, k-cores are frequently used to study modern networks [6,1,10]. Important k-core related quantities include the size of the 2-core compared with the graph, the distribution of k-core sizes, the largest k-core, and many others. In particular, the maximum value of k such that there is a $(k-1)$-core in G is denoted as $K(G)$ and provides an upper-bound on the largest clique in G, hence $\omega(G) < K(G) + 1$.

An equivalent definition of a k-core is a maximal induced subgraph of G where each vertex is incident on at least k edges. This definition then generalizes to any motif, and in particular, a k triangle core is a maximal induced subgraph of G for which each edge $(u, v) \in E$ participates in at least $k-2$ triangles. There is also a polynomial time algorithm for triangle cores, which is $O(|T|) = O(|E|^{3/2})$ in the worst case.

Triangle cores were first proposed by Cohen [4,3], and a faster algorithm was recently proposed, see [17,16]. In that work, however, they explicitly store an array of triangles in order to check whether or not a triangle has been processed. This storage limits scalability as even graphs of a few thousand vertices may have billions of triangles (see Table 3). In our algorithms, we accomplish the same type of check implicitly by carefully ordering and indexing. Further, we develop a data-driven optimizer to adapt the data structures and computations based on the input graph and its properties. We also propose *parallel* edge and vertex-based triangle counting algorithms and use them for the triangle core decomposition. These algorithms significantly speed up the triangle core algorithms. In addition, we find that counting triangles on edges rather than vertices enables better load balancing for graphs where the number of triangles are not uniformly distributed, as is the case for dense and sparse graphs. Finally, unlike the previous approach, we utilize an efficient compressed edge-based representation to perform fast computations over the edges. This reduces the runtime and memory requirements considerably (see comparison in Section 5).

This paper is the first to use triangle cores for graph mining tasks such as finding the largest cliques, temporal strong components, and discovering dense subgraphs (see Section 6). We find that in many cases, the maximum triangle core number gives the exact maximum clique in G, especially for large sparse graphs. We also investigate two useful variants of the triangle core problem: the top-k triangle core problem and the maximum triangle core problem. Both of which we leverage for the above applications. For these problems, our parallel approach uses a fast heuristic clique finder to obtain a lower-bound on the maximum triangle core number, which is surprisingly tight for sparse graphs.

2 Preliminaries

2.1 Wedges, Triangles, and Cliques

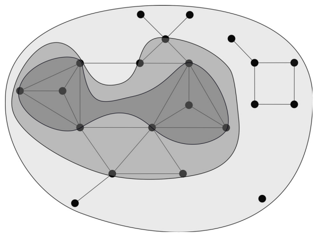
Fig. 1. Triangle cores 2, 3, and 4

A wedge is a 2-length path. The number of wedges W_u centered at u is given by $W_u = d_u(d_u - 1)/2$ where $d_u = |N(u)|$ is the degree. A wedge $\{(u, w), (w, v)\}$ forms a *triangle* if there exists an edge (u, v). Let $tr(u)$ and $tr(u, v)$ be the number of triangles centered at vertex u and edge (u, v), respectively. Further, $tr(G) = \sum_{u \in V} tr(u)$ and $W(G) = \sum_{u \in V} |W_u|$ are the number of triangles and wedges in G, respectively. Using these, we define the density of triangles in G as $\kappa(G) = tr(G)/W(G)$. Observe that if $\kappa(H)$ is close to 1, then H is dense, and a large fraction of vertices must form a clique.

2.2 Triangle Core Decomposition

Definition 1 (Triangle Core): *Consider a graph $G = (V, E)$. A k triangle-core is an edge-induced subgraph of G such that each edge participates in $k - 2$ triangles.*

Hence, a triangle core must contain $k - 2$ triangles, which is the same requirement for a clique of size k, since each edge in the k triangle core has two vertices

(u, v) with at least $k-2$ edges. By the above definition, a triangle is a 3-core, and a clique of size 4 is within the 4-core (See Figure 1).

Definition 2 (**Maximal Triangle Core):** *A subgraph $H_k = (V|E(F))$ induced by the edge-set F is a* maximal triangle core *of order k if $\forall (u, v) \in F : tr_H(u, v) - 2 > k$, and H_k is the maximum subgraph with this property.*

Definition 3 (**Triangle Core Number):** *The* triangle core number *denoted $T(u, v)$ of an edge $(u, v) \in E$ is the largest triangle core containing that edge.*

Definition 4 (**Maximum Triangle Core):** *The* maximum triangle core *of G denoted $T(G)$ is the largest number of $k-2$ triangles for a triangle core of order k to exist.*

2.3 Cliques, Triangle Cores, and K-cores

The relationship between cliques, triangle-cores and k-cores is defined more precisely in this section. Suppose H is a clique of size k, then each edge $(u, v) \in E(H)$ in that clique has two vertices (u, v) that have $k-2$ edges, which also form $k-2$ triangles.

Property* 1*: *Each clique of size k is contained within a k triangle core of G.*

This is a direct consequence of Definition 1 and implies $\omega(G) \leq T(G)$. To understand the relationship between the k-core and the triangle core, we show the relationship between vertex degree and a triangle core number below.

Property* 2*: *Each vertex in the k triangle core has degree $d_v \geq k-1$.*

To see this property, suppose a vertex v has an edge to u in the k triangle core, then v and u by definition, must also share k vertices that form triangles. Let C be the set of at least k common vertices such that $u, v \notin C$. Thus, $|C|$ and u must form at least $k+1$ edges with v.

Property* 3*: *The k triangle core is contained within the $(k-1)$-core.*
To see this property, recall that a T_k triangle core must contain $k-2$ edges and since a k-core is a subgraph where all vertices have at least k degree, then the $k-1$ core must be within the k triangle core.
Property* 4*: *The maximum triangle core number is bounded above by K and below by $\omega(G)$, giving rise to the bounds: $\omega(G) \leq T(G) \leq K(G) + 1 \leq d_{max}$*

This follows directly from the above properties and is used in Section 6.

3 Algorithms

This section describes our exact parallel triangle algorithms and their implementation for large graphs.

Table 1. Parallel triangle algorithms

				time (sec)	
Graph	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert T \rvert$	VERTEX	EDGE
HOLLYWOOD	1.1M	56M	15B	36.2	**23.5**
ORKUT	3.0M	106M	1.6B	40.5	**31.9**
LIVEJOUR	4.0M	28M	251M	3.91	**2.09**
SINAWEIBO	58M	261M	638M	3148	**2194**
TWITTER10	21M	265M	51.8B	5092	**2462**
FRIENDSTER	65M	1.8B	12.5B	43591	***1947**

3.1 Parallel Triangle Counting

Since triangle counting is at the heart of the triangle core computation, we propose a parallel edge-based triangle counting algorithm using shared-memory. In particular, triangle counting is parallelized via the edges (instead of the vertices, see [8]), where jobs are broken up into independent edge computations and distributed dynamically to available workers (see Alg. 1). Our approach offers several benefits. First, the graph can be split up into many smaller *independent edge computations* that can be performed in parallel. We note that the problem with vertex triangle counters is load balancing, see Table 2 where a few vertices may have millions of triangles[1]. Our approach distributes work more evenly along the edges of the graph. For instance, the max number of triangles in sinaweibo (Chinese microblogging site) on any edge is only 17K compared to *8.3 million* for the vertices. As we see later, this approach alleviates many of the problems that arise with vertex-centric parallelization (see Table 1).

Algorithm 1. Parallel Edge Triangle Counting

```
1  Set $p$ to be the number of workers (threads)
2  Init arrays $X_k, 0 \le k \le p$ to be $|V|$ each of all zeros.
3  Set tr to be an array of size $|E|$
4  Set max to be array of length $p$
5  for each $(u, v) \in E$ in parallel do
6      for each $w \in N(v)$ do $X_k(w) = (u, v)$
7      for each $w \in N(u)$ do
8          if $v = w$ then continue
9          if $X_k(w) = (u, v)$ then add 1 to tr$(u, v)$
10     if tr$(u, v) >$ max(k) then max(k) = tr$(u, v)$
11 for each thread $k = 2$ to $p$ do
12     if max(1) > max(k) then max(1) = max(k)
13 return tr and max(1)
```

The parallel edge triangle counting algorithm is given in Alg. 1. The triangle counting computations are performed by dynamically allocating blocks of edges to workers. The workers then compute the number of triangles centered at each edge and store the counts into the edge-indexed array. Locks are avoided by assigning unique ids to the edges, which map to unique positions in the triangle counting array (See Section 3.2 for more details). Each worker also maintains a hash table X_k for $O(1)$ lookups. In particular, line 6 marks the neighbors of v with the unique edge id of (u, v), which is used later in line 9 to determine if a triangle exists. We avoid resetting X_k each time by using the unique edge id. After a worker finishes a job (i.e., block of edges) it requests more work and the process continues as above. Note that each worker also maintains (locally) the total number of triangles processed and the maximum number of triangles at any given edge. Upon completion of the parallel-for loop, the global max is realized in serial (lines 11-12). This takes only $O(p)$ where p is the number of worker nodes. The expected overall time is $O(|E|^{3/2}/p)$ since locks are avoided completely (See Figure 5).

3.2 Triangle Core Decomposition

Edge CSC Format. Edge-based compressed sparse column (ECSC) format adds two arrays to the traditional CSC format to allow for both vertex and

[1] We observe that vertices have highly skewed triangle counts resembling a triangle power-law [7].

edge-based computations. The ECSC graph representation is designed carefully to optimize performance and space while also providing $O(1)$ time access to edges, neighboring edges, and their vertices. In particular, ECSC helps maximize data locality (in disk, ram, cache, core) while minimizing interaction. Both are critical for parallel edge-centric graph algorithms (see [13] for examples). Indeed, the practical importance of ECSC was previously shown in Table 1.

To understand ECSC, we provide an intuitive illustration in Figure 2. The `eid` array is an non-unique edge-indexed array (from the neighbor array) where duplicate edges in the neighbor array are assigned a unique edge id stored in `eid` that acts as a *pointer into the emap array.* This provides two important features. First, observe that `eid` is a surjective function that maps duplicate edges from CSC to their unique position in `emap`. In addition, `eid` also points to the `emap` array where the *vertices* of a specific edge are *consecutively* stored and can be accessed in $O(1)$ time. This function provides the flexibility of using CSC while also giving random access to unique edges and their vertices in $O(1)$. Note that Edge-CSC is efficient to construct taking $O(|E|)$ time (when reading from disk or from CSC).

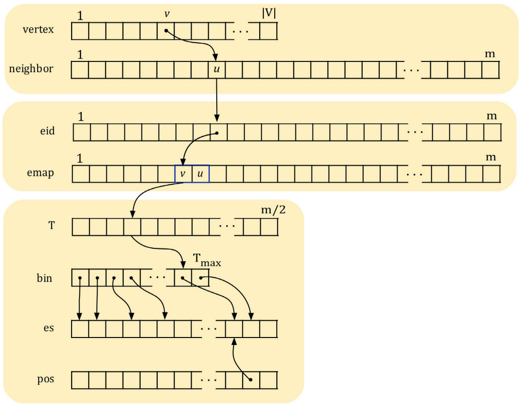

Fig. 2. Illustration of the arrays and pointers used in ECSC for computing the triangle core decomposition. Note $m = |E|$.

Triangle Core Arrays. The triangle core decomposition uses ESCS and four additional smaller arrays. This reduced storage cost is due to `emap` which provides $O(1)$ time access to a unique edge and its vertices (stored consecutively). The edge-indexed T array stores for each edge (u, v) the number of triangles that it participated. An index k of the `emap` array also directly indexes T using $k/2$ since T stores only the unique edges. The `bin` array is indexed by an integer representing triangles, and stores the starting position for each unique number of edge triangles. Hence `y = bin[x]` points directly to the starting position in the `es` array for which the edge given by `es[y]` must have exactly x triangles. The `es` array contains the edge ids sorted by the number of triangles whereas the `pos` array stores the location of a given edge in the sorted `es` array.

The proposed triangle core algorithm is given in Alg. 2. Edges are undirected and the neighbors of each vertex are sorted by degree (to improve caching). The algorithm begins by computing the number of triangles for each edge in parallel. We denote this here by T since it will contain the triangle core numbers upon completion of Alg 2. The lines 2–3 initialize various vars and arrays. Note that the `proc` array is of size m and is used for marking the processed edges. Afterwards, the edges are sorted by their triangle counts in T using bucket sort (lines 4–15). In particular, line 5 counts how many edges will be in each `bin` where a bin is a set of edges with the same triangle count. Thus, bins are numbered from 0 to T_{max}. Given the bin sizes, lines 6–9 setup the starting positions for each bin using the previously computed counts. For each edge, we index into bin with its triangle count and assign it to the position in es returned by bin, then increase that bins position by one (lines 10–13). Finally, lines 14–15 fix the starting positions of the bins.

Now that edges are sorted by their triangle counts, line 17 starts removing each edge in increasing order of triangles (using es). Line 18 retrieves the next edge for processing along with its vertices u and v in $O(1)$ time using emap. Next, line 19 selects v and instead of simply marking the neighbors with a 1 (i.e., $X[w] = 1$), we store the position+1 in which it appeared in the neighbor array. Let us note that the value in X maps directly to a unique edge using eid and emap. For each $w \in N(u)$, we index X and check in $O(1)$ time if w is a neighbor of v (lines 20–21). If so, then the vertices u, v, and w form a triangle, otherwise we continue searching for a match. Using the eid array, we retrieve the unique edge ids for the edges (u,w) and (v,w) in $O(1)$ time from ECSC (lines 22–23) and check if these edges have been processed (line 24). If either of the edges were processed before, then the triangle has been implicitly removed in a previous iteration, and we continue with the next neighboring edge. However, if both edges have not been removed, then we have found a valid triangle to remove. For each of the neighboring edges with a larger triangle core number, we decrease its number of triangles by 1 and move it one bin to the left. All these operations are $O(1)$ time using ECSC. Observe that each of the neighboring edges (u, w) and (v, w) are swapped with the first edge in its bin, respectively. We then swap their positions in the pos array and increase the previous bin and decrease the current bin of the edge by 1. Lines 20–33 are repeated for all neighboring edges of (u,v) that form unprocessed triangles with a triangle count that is currently larger than its own. Finally, the edge is marked as processed and X is reset (line 34–35).

Algorithm 2. Detailed Triangle Core Algorithm

```
1  Set $T$ = ParallelEdgeTriangles($G$)
2  Set pos and proc arrays to be length $|E|$ of all 0.
3  Set the bin array to be of size $T_{max}$ initialized to 0.
4  for e = 0 to $|E|$ do bin[$T(e)$]++
5  Set start = 0
6  for x = 0 to $T_{max}$ do
7      n = bin[x]
8      bin[x] = start
9      start = start + n
10 for e = 0 to $|E|$ do
11     pos[e] = bin[T[e]]
12     tris[pos[e]] = e
13     bin[T[e]]++
14 for t = $T_{max}$ to 0 do bin[t] = bin[t-1]
15 bin[0] = 0
16 Set X to be an array of length $|V|$ containing zeros.
17 for i = 0 to $|E|$ do
18     k = es[i]    v = emap[2k]    u = emap[2k+1]
19     for w ∈ $N$(v) do X[w] = $w_{pos}$ + 1
20     for w ∈ $N$(u) do
21         if X[w] is not marked then continue
22         uw = eid[j]
23         uv = eid[X[w] - 1]
24         if proc[uw] or proc[uv] are marked then
25             continue
26         for adj ∈ {uw, uv} do
27             if T[adj] > T[k] then
28                 tw = T[adj]   and   pw = pos[adj]
29                 ps = bin[tw]   and   xy = es[ps]
30                 if xy ≠ wv then
31                     pos[adj] = ps   and   es[pw] = xy
32                     pos[xy] = pw   and   es[ps] = adj
33                 bin[tw]++   and   T[adj]−−
34     proc[k] = 1
35     for w ∈ $N$(v) do X[w] = 0
```

4 Problem Variants

Two variants are investigated and space-efficient multi-threaded algorithms suitable for CPU and GPU parallelization are proposed. Both variants arise from real-world motivations, e.g., finding maximum cliques and dense subgraphs in big data (See Section 6).

4.1 Top K Triangle Cores

Using property 3 as a basis, we propose a parallel method that leverages a relationship between the k-core and triangle core for computing only the top-k triangle cores. Key to our approach is the heuristic clique finder which allows us to obtain a fast lower-bound.

Algorithm 3 Top-K Triangle Cores

```
1  $K$ = CoreNumbers($G$)
2  max = HeuMaxClique($G$,$K$)
3  for each $u \in$ V in parallel do
4      if $K(u) \geq$ max then add $u$ to $W$
5  Set $H = G(W)$ – the induced graph from $W$.
6  $tr$ =ParallelEdgeTriangles($H$)
7  for each $u, v \in E_H$ in parallel do
8      if $tr(u, v) <$ max then
9          prune $(u, v)$ from $H$
10         for each neighboring edge $((w, v)$or$(w, u))$ do
11             set $tr(w, u)$ to be one less
12             prune $(w, u)$ if $tr(w, u) <$ max
13 $T$ =TriangleCores($H$)
```

Problem 1 (**Top K Triangle Cores**)**:** *Given a graph G and an integer $k > 2$, find the set of edges that have triangle core numbers greater than k.*

The output is a subgraph H such that $u, v \in E(H)$ if $T(u, v) \geq k$ and an edge-indexed array indicating the triangle core numbers of each edge. Note that we use this variant later for the maximum clique problem and finding dense subgraphs (see Section 6.3).

The first step in Alg 3 computes the k-core numbers denoted K of G. This gives us an upper bound on the triangle core and thus if k is larger we set it to be the maximum k-core. Next, for each vertex $v \in V$ we add it to the vertex set W if it satisfies $K(u) + 1 \geq k$. Let H be an explicit vertex-induced subgraph from G using the vertex set W. Triangle counts for the edges are computed in parallel and if an edge $(u, v) \in E(H)$ has less triangles than the integer k given as input, $\mathrm{tr(v, u) \leq k}$, then we can safely prune it, and update the graph, and the triangle count of any edge (w, v) or (w, u) that formed a triangle through (v, u). This smaller graph is then given to the triangle core routine. Instead of selecting k arbitrarily, we use a heuristic clique finder [12] to obtain a tight lower bound denoted $\tilde{\omega}(G)$. Observe that $\tilde{\omega}(G) \leq \omega(G) \leq T(G) \leq K(G) + 1$ where computing $\tilde{\omega}(G)$ is $O(|E| \cdot d_{\max})$ time. See Table 2 for heuristic runtimes.

4.2 Max Triangle Core

In many applications, the cost of computing the full triangle core decomposition is too expensive and/or not needed. For instance, the maximum triangle core may significantly speedup the termination of maximum clique algorithms when used for pruning. Thus, we solve the following problem instead:

Algorithm 4 Dense Triangle Subgraph

```
1  $K$ = CoreNumbers($G$)
2  $k$ = HeuristicClique($G$)
3  for each $u \in$ V in parallel do
4      if $K(u) \geq k$ then add $u$ to $W$
5  Set $H = G(W)$ to be the induced graph from $W$.
6  T = TriangleCores(H)
7  Set $F = \emptyset$
8  for each $u \in$ V(H) in parallel do
9      for each $w \in N(u)$ do
10         if $T(u,w) \geq k$ then
11             add edge $(u,w)$ to $F$
12             add 1 to $d_u$
13     if $d_u \leq k$ then Remove all edges of $u$ from $F$
14 return the edge-induced graph from $F$
```

Problem 2 (Maximum Triangle Core Number): *Given a graph G, find the maximum triangle core number $T(G)$ directly.*

Our approach starts by computing the k-cores, then scans the vertices in order of their removal times from the k-core algorithm. A vertex v is added to W if $K(v) > K - \delta$ where for triangles we set $\delta = 3$. We then compute $H = (W, E[W])$ and give this smaller graph to the triangle core routine. Observe that $\tilde{T}$ is a fast but accurate approximation of the maximum triangle core.

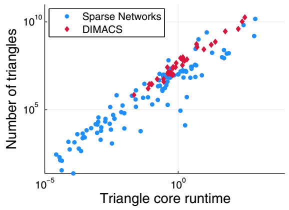

Fig. 3. Runtime vs. number of triangles

5 Experiments

In this section, we systematically investigate the performance and accuracy of our methods on over 150 graphs. The experiments are designed to answer the following questions:

- ▷ **Triangle core decomposition.** How fast is our algorithm for computing the triangle core decomposition? and how does it scale up for both dense and large sparse graphs of up to a billion edges.
- ▷ **Parallel edge triangle counting.** Does our parallelization scheme work? Is the edge triangle counting algorithm faster than vertex triangle counters? and do they have better load balancing?
- ▷ **Performance of variants.** How fast can we solve each variant? and how does the performance compare to the full triangle core decomposition?
- ▷ **Clique upper bound**. How tight are the triangle core upper bounds compared to the k-core? For which types of graphs are they better?

For the experiments, we used a 2 processor, Intel E5-2760 system with 16 cores and 256 GB of memory. Our algorithms never came close to using all the memory.

5.1 Performance of Proposed Algorithm

For the first question, we evaluate the performance of the proposed triangle core algorithm on over 150 graphs of all types. The results in Tables 2 and 3 demonstrate the effectiveness of the proposed methods for both large sparse graphs and dense graphs, respectively.

Table 2. Performance of heuristic, parallel triangle count algorithm, and triangle cores

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	$d_{\max}$	$\bar{\kappa}$	$tr_{\max}$	tr_{avg}	K	T	$\tilde{\omega}$	$\tilde{\omega}$ s.	tr s.	T s.
DBLP-2010	226k	716k	4.8M	238	0.64	5.9k	21	75	75	75	0.02	0.10	0.29
CITESEER	227k	814k	8.1M	1.4k	0.68	5.4k	35	87	87	87	0.05	0.09	0.43
MATHSCI	333k	821k	1.7M	496	0.41	1.6k	5.2	25	25	25	0.11	0.14	0.53
HOLLYWOOD	1.1M	56M	15B	11k	0.77	4.0M	13k	2209	2209	2209	1.57	23.5	727.2
YOUTUBE	496k	1.9M	7.3M	25k	0.11	151k	14	50	19	14	0.42	0.71	15.92
FLICKR	514k	3.2M	176M	4.4k	0.17	525k	343	310	153	49	0.21	1.26	18.67
ORKUT	3.0M	106M	1.6B	27k	0.17	1.3M	525	231	75	45	13.8	31.9	770.5
LIVEJOUR	4.0M	28M	251M	2.7k	0.26	80k	62	214	214	214	3.2	2.09	54.3
TWITTER10	21M	265M	51.8B	698k	0.04	44M	2.4k	1696	1153	174	145.7	2462	39535
FRIENDSTER	65M	1.8B	12.5B	5.2k	0.16	190	158k	305	129*	129	561	1947	45247
TEXAS84	36k	1.6M	34M	6.3k	0.19	141k	922	82	62	48	0.070	0.34	4.80
PENN94	42k	1.4M	22M	4.4k	0.21	68k	520	63	48	44	0.05	0.22	3.27
P2P-GNUT	63k	148k	6.1k	95	0.01	17	0.1	7	4*	4	0.01	0.01	0.02
RL-CAIDA	191k	608k	1.4M	1.1k	0.16	6.0k	7	33	19	15	0.04	0.06	0.33
AS-SKITTER	1.7M	11M	86M	35k	0.26	565k	50	112	68	64	0.37	4.33	70.6
IT-2004	509k	7.2M	1.0B	469	0.82	93k	19k	432	432	432	0.09	1.36	6.88
WIKIPEDIA	1.9M	4.5M	6.7M	2.6k	0.16	12k	3	67	31*	31	0.66	0.54	3.54

Table 3. Performance on dense DIMAC graphs.

graph	$\lvert E\rvert$	$\lvert T\rvert$	$tr_{\max}$	K	T	tr s.	T sec
C500-9	112k	45M	98k	433	373	0.08	0.58
C2000-5	1.0M	500M	288k	941	435	1.65	21.02
C1000-9	450k	365M	385k	875	764	0.58	5.23
C4000-5	4.0M	4.0B	1.1M	1910	899	12.09	177.75
C2000-9	1.8M	2.9B	1.5M	1759	1549	3.02	59.32
P-HAT15K3	569k	274M	372k	505	314	0.66	9.41
P-HAT15K	847k	741M	676k	930	597	1.31	23.83
MAN-81	5.5M	18B	5.5M	3281	3241	8.59	315.84
KELLER6	4.6M	10B	3.6M	2691	2084	7.92	252.37

Notably, the proposed algorithm counts 18B triangles in 8 seconds, while taking 315 seconds for triangle cores. For these graphs, the triangle core algorithm adapts the graph representation and computation to better exploit the structure. This includes using an adj structure for $O(1)$ time lookups, selecting specialized subroutines based on densities, among many other optimizations. Indeed, the triangle core algorithm is shown to be fast and scalable as it easily handles graphs with billions of edges while using a small memory footprint. Fig. 3 compares dense and sparse graphs directly. This indicates that many of the sparse graphs perform better than expected whereas dense graphs deviate much less from the diagonal.

Performance profile plots. We also compare with the implementation of Zhang et al. [17] using performance profiles [5]. Figure 4 evaluates the performance on 24 graphs with up to 1.8 billion edges. In all cases, the proposed algorithm outperforms the recent state-of-the-art algorithm [17]. We note that for a few large graphs, the baseline algorithm runs out of memory and eventually terminates.

5.2 Parallel Edge Triangle Counting

The proposed parallel edge triangle counting algorithm is fast and scalable as shown in Table 2. Moreover, we also find that it outperforms a recent MapReduce triangle counting algorithm [15]. In that work, it takes 319 seconds using a MapReduce cluster of 1636 nodes to compute triangles for LiveJournal whereas it takes us only 2.09 seconds (See Table 1).

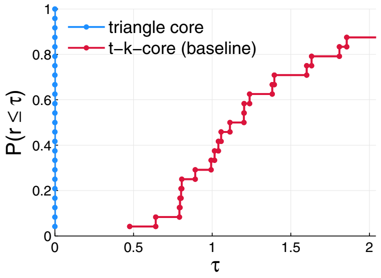

Fig. 4. In all cases, our algorithm is significantly faster than the recently proposed algorithm of Zhang et al. [17]. The vertical line formed from our algorithm indicates it outperformed the state-of-the-art on each of the 24 graphs tested. Further, the other algorithm fails on 3 of the largest graph as illustrated by the right-most point on the curve.

Speedup. Fig 5 shows that our approach scales well, especially for sparse graphs. Observe that for dense graphs, triangles are more uniformly distributed and thus the improved load balancing from our approach does not help as much. We also find that on average, less time is spent per triangle for dense graphs.

Edge vs. vertex triangle counting. We parallelize a vertex triangle counter to evaluate the performance of our parallel edge-based algorithm. The superiority of the parallel edge triangle counting algorithm is clearly shown in Table 1. In one example, the friendster social network of 1.8 billion edges takes only 1,947 seconds whereas the vertex triangle counter takes 43,591 seconds. The edge triangle counter is 22x faster.

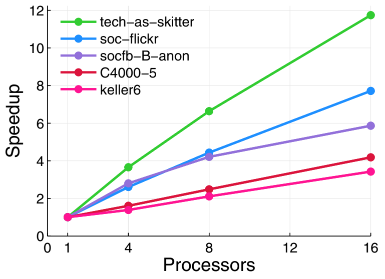

Fig. 5. Speedup of our parallel edge triangle counting algorithm

5.3 Performance of Variants

Results are shown in Table 4. We compare the greedy maximum triangle core to the exact triangle core decomposition on the basis of speed and accuracy. Table 4 clearly demonstrates the effectiveness of the greedy maximum triangle core procedure. In some instances, a speedup of over 650x is observed while also returning the exact maximum triangle core number. For the top-k cores, we find that these are generally 1-20x faster than the full triangle core decomposition while guaranteed to be exact, as previously shown in Property 3. The large table of results are omitted due to brevity, but are later used for finding temporal strong components.

5.4 Bounds

This section evaluates the triangle core clique bound on sparse and dense graphs. In Table 2, we find that the triangle core upper bound is sometimes significantly tighter than the k-core upper bound, especially for sparse graphs that exhibit a weak power-law including social, facebook, and technological networks. Further, Table 3 shows that for dense graphs, the triangle core bound is almost always much tighter than the k-core bound. In addition, we use the triangle core upper bound to validate a fast heuristic clique finder. We find three such cases and mark them with a star in Table 2. Note this excludes the cases where the k-core upper bound also verifies the large clique as optimal. Besides the maximum triangle core number, we also found the full distribution of triangle core numbers interesting.

Table 4. Performance and accuracy of greedy maximum triangle core

	density		**triangle core**		**time (sec)**	
graph	ρ	greedy	T	greedy	exact	greedy
AS-SKITTER	7.7e-06	0.50	68	68	65.39	0.10
FB-ANON-A	4.9e-06	0.06	30	29	83.56	0.31
STANFORD	0.008	0.04	58	58	2.0	0.87
LIVEJOUR.	3.4e-06	0.99	212	212	60.95	0.03
FLICKR	2.4e-05	0.44	151	151	22.55	3.16
HOLLYWOOD	9.8e-05	0.99	2207	2207	918.7	92.5
DBLP12	2.0e-05	0.99	112	112	0.85	0.007

6 Applications

This section demonstrates the effectiveness of the proposed triangle core algorithms for a variety of graph mining applications.

6.1 Maximum Clique Algorithms

In this subsection, we utilize triangle core numbers to prune and order vertices in a state-of-the-art maximum clique algorithm. In particular, we aggressively compute the *triangle core numbers* of each vertex induced neighborhood. This gives rise to the following: $\omega \leq T(N(v)) \leq K(N(v)) \leq d(N(v))$. The approach proceeds similar to pmcx from [12] which uses k-core bounds with greedy coloring applied at each step. After pruning the vertex neighborhoods, we compute the density of the subgraph denoted $\rho(N(v))$ and use *neighborhood triangle cores* only if $\rho(N(v)) > 0.85$, otherwise, we proceed same as pmcx above. Intuitively, if density is large enough, then edges are pruned using the neighborhood cores. We compare trmc against two recent state-of-the-art finders: bbmc [14] and pmcx [12]. Let us note that bbmc uses a bitset encoding for set intersections and was shown to outperform all others tested [11]. In Fig. 6 we find that trmc outperforms bbmc on every graph, but outperforms pmcx on only a few instances.

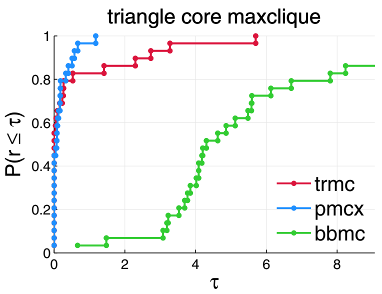

Fig. 6. Performance profile plot. trmc outperforms bbmc on all tested graphs and beats pmcx on a few.

6.2 Temporal SCC

Table 5. Max temporal-SCC via triangle cores

	bounds			time (seconds)			
graph	$\tilde{\omega}$	K	T	$\tilde{\omega}$ s	K s	T s	$\tilde{T}$ s
FB-MESSAGES	707	707	707	0.15	0.01	6.72	1.54
REALITY	1236	1236	1236	0.28	0.03	323	11.21
TWITTER-COP	581	583	581	0.21	0.01	5.85	1.11

We use the algorithms from section 3 to explore the effectiveness of the top-k triangle cores for computing the largest temporal strongly connected component (TSCC), which is a known NP-hard problem [9]). The top-k triangle core algorithm in Alg 4 uses the k-cores and a large clique from a heuristic maximum clique finder [12] to prune the triangle cores without ever computing them. Table 5 shows the performance of our full triangle core algorithm compared with the top-k triangle core algorithm. Strikingly, in all cases, the triangle core upper bound gives the *exact size* of the largest TSCC. Moreover, the top-k triangle core algorithm is significantly faster than the full algorithm. For instance, the REALITY graph in Table 5 takes 323 seconds to compute the full decomposition, whereas the fast top-k algorithm takes only 11 seconds.

6.3 Dense Subgraph Mining

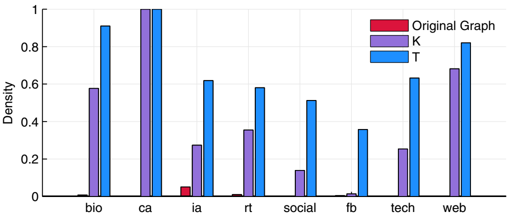

Fig. 7. Networks are grouped by type (bio, social, etc) and densities from each are averaged.

A particularly useful property of the *greedy maximum triangle core* is that it returns a relatively dense subgraph. In Table 4 we find the density of the as-skitter subgraph returned by the greedy maximum triangle core method is 0.50 with 345 vertices with an average degree of 173. Concerning performance, our greedy maximum triangle core procedure is clearly much faster while also accurate. In addition, Figure 7 summarizes the results from Alg 4. The top-k triangle cores are shown to be better at finding dense subgraphs.

7 Conclusion

We have described a fast scalable algorithm for computing the *triangle core decomposition* and systematically investigated its scalability and effectiveness on a variety of large sparse networks. The proposed algorithm is shown to be orders of magnitude faster than the current state-of-the-art while also using less space. In addition, the algorithm was designed for graphs of arbitrary size and density, allowing it to be used for a variety of applications. We also proposed a *parallel edge triangle enumeration algorithm* and showed that it is significantly faster than vertex-based methods. Future work will investigate the proposed family of triangle core ordering techniques for use in greedy coloring methods and clique finding algorithms.

References

1. Alvarez-Hamelin, J.I., Dall'Asta, L., Barrat, A., Vespignani, A.: Large scale networks fingerprinting and visualization using the k-core decomposition. In: NIPS (2005)
2. Batagelj, V., Zaversnik, M.: An o(m) algorithm for cores decomposition of networks. arXiv preprint cs/0310049 (2003)
3. Cohen, J.D.: Graph twiddling in a mapreduce world. Computing in Science & Engineering 11(4), 29–41 (2009)
4. Cohen, J.D.: Trusses: Cohesive subgraphs for social network analysis. National Security Agency Technical Report (2008)
5. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Mathematical Programming 91(2), 201–213 (2002)
6. Dorogovtsev, S.N., Goltsev, A.V., Mendes, J.F.F.: K-core organization of complex networks. Physical Review Letters 96(4), 040601 (2006)
7. Kang, U., Meeder, B., Faloutsos, C.: Spectral analysis for billion-scale graphs: Discoveries and implementation. In: Huang, J.Z., Cao, L., Srivastava, J. (eds.) PAKDD 2011, Part II. LNCS, vol. 6635, pp. 13–25. Springer, Heidelberg (2011)
8. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5(8), 716–727 (2012)
9. Nicosia, V., Tang, J., Musolesi, M., Russo, G., Mascolo, C., Latora, V.: Components in time-varying graphs. Chaos 22(2) (2012)
10. Ahmed, N.K., Neville, J., Kompella, R.: Network sampling: From static to streaming graphs. ACM Transactions on Knowledge Discovery from Data (TKDD), 1–54 (2013)
11. Prosser, P.: Exact algorithms for maximum clique: A computational study. arXiv:1207.4616v1 (2012)
12. Rossi, R.A., Gleich, D.F., Gebremedhin, A.H., Patwary, M.A.: Fast maximum clique algorithms for large graphs. In: WWW Companion (2014)
13. Rossi, R.A., McDowell, L.K., Aha, D.W., Neville, J.: Transforming graph data for statistical relational learning. JAIR 45, 363–441 (2012)
14. San Segundo, P., Rodríguez-Losada, D., Jiménez, A.: An exact bit-parallel algorithm for the maximum clique problem. Comput. Oper. Res. 38, 571–581 (2011)
15. Suri, S., Vassilvitskii, S.: Counting triangles and the curse of the last reducer. In: WWW, pp. 607–614. ACM (2011)
16. Wang, J., Cheng, J.: Truss decomposition in massive networks. Proceedings of the VLDB Endowment 5(9), 812–823 (2012)
17. Zhang, Y., Parthasarathy, S.: Extracting analyzing and visualizing triangle k-core motifs within networks. In: ICDE, pp. 1049–1060 (2012)