# Parallel $k$-Truss Decomposition on Multicore Systems

Humayun Kabir Kamesh Madduri
*Computer Science and Engineering*
*The Pennsylvania State University*
*University Park, PA 16802*
*Email: {hzk134, madduri}@cse.psu.edu*

***Abstract*—We discuss our submission to the HPEC 2017 Static Graph Challenge on $k$-truss decomposition and triangle counting. Our results use an algorithm called PKT (Parallel $k$-truss) designed for multicore systems. We are able to process almost all Graph Challenge datasets in under a minute on a 24-core server with 128 GB memory. For a synthetic Graph500 graph with 17 million vertices and 523 million edges, triangle counting takes 16 seconds and truss decomposition takes 29 minutes on the 24-core server.**

## 1. Introduction

The $k$-core [1], [2] and $k$-truss [3] cohesive subgraph formulations are very useful in practice because they can be computed exactly using simple polynomial-time algorithms. Both these formulations can also be used for a hierarchical decomposition of the graph. A $k$-core is a maximal subgraph such that each vertex has degree at least $k$. A $k$-truss [3] is defined as a maximal non-trivial single-component subgraph such that every edge is contained in at least $k-2$ triangles.

In this submission to the HPEC 2017 Static Graph Challenge [4], we use a shared-memory truss decomposition algorithm called PKT [5]. Our implementation source code is available at https://github.com/humayunk1/PKT. The following are the key features of our truss decomposition algorithm PKT:

- We perform a level-synchronous parallelization of a fast sequential algorithm for $k$-truss decomposition [6].
- Our approach is memory-efficient in that the memory requirements of intermediate steps are proportional to the number of edges and not the number of triangles in the graph.
- Unlike other $k$-core and $k$-truss algorithms, we do not use a hash table to check for connectivity. Instead, we use data structures that are amenable to safe and easy concurrent updates.
- For support computation, a key subroutine of many truss decomposition approaches, we use a highly tuned and graph ordering-aware method that performs significantly lower work for graphs with skewed degree distributions.

This paper is organized as follows. We review prior algorithms for $k$-truss decomposition in Section 2. In Section 3, we give a high-level overview of our truss decomposition strategy. For additional details, please refer to [5]. We next analyze performance of our implementation on some of the Graph Challenge instances in Section 4. We conclude by listing future research directions in Section 5.

## 2. Background and Related Work

Let $G = (V, E)$ be an undirected and simple graph with one connected component, $n = |V|$ vertices, and $m = |E|$ edges. We use $N(u)$ to denote the set of neighbors of a vertex $u$, i.e., $N(u) = \{v : \langle u, v\rangle \in E\}$. The degree of a vertex $u$ is denoted by $d(u) = |N(u)|$. A triangle in $G$ is a cycle of length 3. We denote a triangle by the edge triple that forms it: $\langle u, v\rangle$, $\langle v, w\rangle$, and $\langle u, w\rangle$. The order of $u$, $v$, and $w$ does not matter when denoting the triangle. The set of all triangles in $G$ are denoted by $\triangle_G$. Similarly, we define a wedge to be a pair of edges with a common endpoint, e.g., $\langle u, v\rangle$ and $\langle v, w\rangle$. Alternately, a wedge can be defined as a path of length 2 centered at the vertex $v$. Triangles can be viewed as closed wedges, i.e., the edge $\langle u, w\rangle$ is present. Let $\wedge$ denote the set of wedges in the graph. The global clustering coefficient of the graph is given by $\frac{3|\triangle|}{|\wedge|}$. Let $\alpha$ denote the arboricity of graph $G$. Arboricity is defined as "the minimum number of edge-disjoint spanning forests into which $G$ can be decomposed [7]." We define support of an edge $e = \langle u, v\rangle \in G$, $S(e, G)$, as the number of triangles that it is contained in. A $k$-truss [3], [8] is then defined as follows: A $k$-truss $T_k$ ($k \geq 2$) is a maximal non-trivial connected subgraph of $G$ such that for each edge $e \in T_k$, $S(e, T_k) \geq k-2$. An edge is defined to have a trussness [6] of $l$ if it belongs to an $l$-truss of the graph, but not an $(l+1)$-truss. The coreness of a vertex is defined similarly. The problem of $k$-truss decomposition [6] refers to computing the trussness value of every edge in the graph. We use $t_{\max}$ to denote the maximum trussness of any edge in $G$. The $k$-class of $G$ is the set of all edges with trussness $k$. Given edge trussness values, the maximal $k$-truss subgraphs (for a specific $k$) can be determined by executing connected components on the graph after deleting edges with trussness less than $k$. A $k$-truss is (almost) identical to the k-dense [9], triangle $k$-core [10], and the k-community [11] cohesive subgraph formulations independently presented by other authors. Sarıyüce et al. [12] recently presented a new

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE 1. A SUMMARY OF SEQUENTIAL $k$-TRUSS DECOMPOSITION ALGORITHMIC STRATEGIES.

| Algorithm | Time ($T_{\|\triangle\|} + \cdots$) | Memory |
|---|---|---|
| Explicit triangle enum. [10] | $O(\|\triangle\| + m + n)$ | $O(\|\triangle\| + m + n)$ |
| Simple bottom-up alg. [3] | $O(\|\wedge\| + m + n)$ | $O(m + n)$ |
| Wang-Cheng improved alg. [6] | $O(\alpha m + n)$ | $O(m + n)$ |
| Sarıyüce et al. local alg. [13] | $O(n_{\text{iter}}\|\wedge\| + m + n)$ | $O(m + n)$ |
| PKT (our alg.) [5] | $O(\|\wedge\| + t_{\max} m + n)$ | $O(m + n)$ |

formulation for cohesive subgraphs called nucleus decompositions, which subsumes both $k$-core and $k$-truss definitions.

There are several known algorithmic strategies for truss decomposition. The key ones are summarized in Table 1. Since all of these algorithms first compute the support of every edge, and support computation has the same bounds as triangle listing, we refer to the support computation time as $T_{|\triangle|}$. The time for the succeeding steps is listed in the table. Zhang and Parthasarathy [10] introduce the triangle $k$-core formulation, which is almost identical to a $k$-truss. Their algorithm is different in that all triangles are explicitly listed in the support computation step, and the data structure storing them is updated in the subsequent steps. The space requirements of this algorithm scale as $O(|\triangle|)$, but superfluous triangle lookups are avoided and a hash table is not required. We do not consider parallelization of this algorithm because of the considerably higher space requirements.

In the paper introducing the $k$-truss subgraph, Cohen also gives an algorithm for enumerating maximal trusses. After computing edge support, all edges with support less than $k-2$ are removed. When removing an edge, the support of the edges that form a triangle with the removed edge are reduced. For each edge $e = \langle u, v\rangle$, processing it takes time proportional to $d(u)+d(v)$. The total time for this algorithm is thus $\Theta(n+m+\sum_{\langle u,v\rangle \in E}(d(u)+d(v))$, which simplifies to $\Theta(n+m+2\sum_{v\in V} d(v)^2)) = O(m^{1.5})$, since $m = O(n^2)$ for a simple graph. We refer to this as the simple bottom-up algorithm. Cohen also proposed a MapReduce algorithm for computing $k$-trusses [8].

Wang and Cheng [6] present an improvement to Cohen's algorithm. Like Cohen's algorithm, this approach starts by computing the support of each edge. Next, the edges are sorted in ascending order of their support using a linear-time sort such as counting sort. Edges are then processed in increasing order of support. Each edge is processed exactly once, and for an edge $e = \langle u, v\rangle$, a *canonical* edge representation assuming $d(u) \leq d(v)$ is used. For each neighbor $w$ of $u$, the algorithm checks if $u$, $v$, and $w$ form a triangle or not. This is done by using a hash table, where the keys are a pair of vertices. For each edge $e = \langle u, v\rangle$, processing it takes time proportional to $\min(d(u), d(v))$. The total time for this algorithm is thus $\Theta(n + m + \sum_{\langle u,v\rangle \in E}(\min(d(u), d(v)))$, which simplifies to $O(\alpha m+n)$. Sarıyüce et al. [13] recently proposed shared-memory parallelization of their nucleus decomposition formulation based on a $k$-core decomposition algorithm [14]. Since a $k$-truss is a special case of their nucleus decomposition, the Sarıyüce et al. local algorithm can be considered an alternative to the level-synchronous parallelization strategy.

Edge support computation is closely related to the problems of exact triangle counting and listing. Compared to truss decomposition, triangle counting is a very well-studied problem. We refer readers to [7], [15], [16], [17], [18], [19], [20], [21] for a sampling of efficient algorithms and practical high-performance implementations. Xiao et al. [19] unify a large body of previously-developed triangle counting algorithms and observe that the ordering of vertices and orientation of edges has a significant impact on performance. We use a parallel triangle counting implementation proposed for the problem of triad census in directed graphs [20].

## 3. Truss Decomposition Approach

We begin by describing the support computation approach, which is in turn based on a recent parallel triangle counting algorithm [20].

**Parallel triangle counting and support computation.** Fast algorithms for triangle counting use degree- or $k$-core-based vertex ordering and combine this with edge orientation. With increasing $k$-core-ordering of vertices, a canonical triangle representation of $v < u < w$ gives a low operation count [16], [19], [20]. We define $N^+(u) = \{v : \langle u, v\rangle \in E, v > u\}$, $d^+(u) = |N^+(u)|$, $N^-(u) = \{v : \langle u, v\rangle \in E, v < u\}$, and $d^-(u) = |N^-(u)|$.

We use two variants of triangle counting called Adjacency Marking (AM) and Adjacency Intersection (AI). The pseudocodes for these variants that exploit increasing coreness ordering are given in Algorithm 1.

For every vertex $u$, we mark the neighbors $N^+(u)$ using a thread-local array $X$. We then visit each $v \in N^-(u)$ and consider adjacencies $w \in N^+(v)$. If $w$ is marked, $vuw$ forms a triangle $\triangle_{wuv}$ and $v < u < w$. The time complexity of this algorithm is $\Theta(\sum_v (d^+(v) + d^+(v)^2)) = \Theta(m + \sum_v (d^+(v)^2))$.

In triangle counting, the array $X$ could be a bit vector. However, for support computation, we use $X$ to store the edge id of the edge $\langle u, w\rangle$. Also, no atomic operations are necessary in triangle counting. However, three atomic operations are used for each triangle discovered, to count the support of each edge in the triangle. This adds overhead to support computation that is not present in triangle counting. Please refer to [5] for support computation pseudocode.

**Truss decomposition.** In the PKT algorithm, we compute trussness values in a bottom-up manner. We process edges belonging to the $l$-class before processing edges belonging to $(l + 1)$-class. The steps are given in Algorithm 2. The overall strategy is similar to the ParK [22] algorithm for parallel $k$-core decomposition. The output array $S$ has the trussness values of all the edges. Note that the trussness of an edge $e$ is upper bounded by $S[e] + 2$.

The algorithm starts by computing the support of the edges in parallel, and stores the support in the array $S$. Next, it uses the procedures SCAN and PROCESSSUBLEVEL

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

**Algorithm 1** Triangle counting: Adjacency Marking (AM) and Adjacency Intersection (AI) variants.

```
procedure TriCountAM(G)
    tcl ← 0                                          ▷ thread-local count
    for all i ∈ V do                                 ▷ thread-local mark array
        X[i] ← 0
    for all u ∈ V in parallel do                     ▷ Parallelize
        for all w ∈ N+(u) do
            X[w] ← 1
        for all v ∈ N−(u) do
            for all w ∈ N+(v) do
                if X[w] = 1 then
                    tcl ← tcl + 1
        for all w ∈ N+(u) do
            X[w] ← 0
    tc ← ∑ tcl
    return tc

procedure TriCountAI(G)
    tcl ← 0                                          ▷ thread-local count
    for all v ∈ V in parallel do                     ▷ Parallelize
        for all u ∈ N+(v) do                         ▷ Exploit ordering
            tcl ← tcl + |N+(v) ∩ N+(u)|
    tc ← ∑ tcl                                       ▷ sum all thread-local counts
    return tc
```

**Algorithm 2** PKT: Parallel $k$-truss decomposition.

```
procedure PKT(G)
    S ← φ                                            ▷ Global trussness array
    curr ← φ, next ← φ                               ▷ Temp arrays of size m
    SupportComp(G, S)
    todo ← m; k ← 0
    while todo > 0 do
        SCAN(S, k, curr)
        while |curr| > 0 do
            todo ← todo − |curr|
            ProcessSubLevel(S, k, curr, next)
            Swap curr and next
        k ← k + 1
```

to find edges in a $k$-class. To find edges in a $k$-class, the algorithm uses a SCAN phase to scan the $S$ array and find edges with support $k-2$. These edges are processed in the procedure PROCESSSUBLEVEL. The processing of these edges in PROCESSSUBLEVEL may add new edges to the $k$-class. This continues until no more edges can be added to $k$-class. The arrays $curr$ and $next$ are used to keep track of edges in the current level. We do not give the pseudocode for PROCESSSUBLEVEL here, but the pseudocode and a detailed explanation of its working are available in [5]. Note that we make one key change to the PROCESSSUBLEVEL routine in [5]: instead of an adjacency marking-based approach for this step, we use an intersection-based approach.

The time complexity for support computation is the same as triangle counting: $\Theta(\sum_v (d^+(v) + d^+(v)^2))$. The cumulative time taken by the SCAN procedure is $mt_{\max}$, since the array $S$ is of size $m$. The procedure PROCESSSUBLEVEL computes the intersection of the end points of each edge exactly once, and so the time complexity is: $\sum_{e=\langle u,v\rangle}(d(u)+d(v)) = \sum_v d(v)^2$, since we consider an edge $e = \langle u, v\rangle$ and $u < v$. Since $mt_{\max} \ll \sum_v d(v)^2$ is small for most real-world graphs, the time complexity is dominated by $\sum_v d(v)^2$. Further, since the wedge count $|\wedge| = (\sum_v d(v)^2 - 2m)/2$, and so $|\wedge|$ is also an estimate of the work performed.

Each thread processes the triangles that contains an edge. The number of triangles formed by an edge may vary quite a bit among the edges. This introduces load imbalance in our algorithm, and we use OpenMP's dynamic loop scheduling to alleviate load imbalance.

## 4. Results and Performance Analysis

**Experimental Setup.** We present results on a dual-socket Intel shared-memory server with 128 GB main memory. The server has 2.2 GHz Intel Xeon E5-2650 v4 (Broadwell) processors. Each processor has twelve cores, 30 MB L3 cache, and hyperthreading is turned off. The main memory bandwidth using the STREAM Triad benchmark is 116 GB/s.

All the programs are compiled using the Intel C/C++ compiler (version 16.0.3) with -O3 optimization. We use OpenMP for parallelization, and pin threads to cores using the *compact* pinning strategy. For the scan phase, we use *static* thread scheduling. For support computation and edge processing, we use *dynamic* scheduling with chunk sizes set to 10 and 4, respectively.

We ran our triangle counting and truss decomposition programs on all the graphs in the current Graph Challenge collection. Here, we choose to report the running times only for the graph instances for which PKT takes longer than 1 minute on 24 cores. We identify four such graphs: Friendster and three Graph500 graphs (of SCALE 23, 24, and 25). We report some graph statistics in Table 2. In addition to these four graphs, we also include the soc-friendster graph from Network Repository [23] in our test suite. Although the source for the Friendster and soc-friendster graph appears to be the same, they differ quite a bit in terms of aggregate statistics. We believe the SNAP [24] com-Friendster graph was preprocessed differently to create the Friendster graph in the Graph Challenge collection.

In Table 2, we list the reciprocal of the global clustering coefficient for each graph. Note that the Friendster graph has a very low global clustering coefficient. The number of triangles is also significantly lower than the count in the soc-friendster instance, and $t_{\max}$ is also very low. The Friendster instance has the largest vertex count. The edge counts of Friendster and soc-friendster are comparable. We also list $c_{\max}$, the maximum coreness, for each graph. $t_{\max}$ is significantly lower than $c_{\max}$ for all graphs.

We report normalized performance in terms of Giga Wedges Examined Per Second (GWEPS), a rate analogous

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE 2. THE GRAPHS USED FOR BENCHMARKING PERFORMANCE. SOC-FRIENDSTER IS TAKEN FROM [23], AND THE REST OF THE GRAPHS ARE FROM THE GRAPH CHALLENGE WEBSITE. WE ALSO GIVE THE WEDGE COUNT ($|\wedge|$), TRIANGLE COUNT ($|\triangle|$), VERTEX COUNT ($n$), EDGE COUNT ($m$), MAXIMUM DEGREE ($d_{\text{MAX}}$), MAXIMUM CORENESS ($c_{\text{MAX}}$), MAXIMUM TRUSSNESS ($t_{\text{MAX}}$), AND THE INVERSE OF THE GLOBAL CLUSTERING COEFFICIENT FOR EACH GRAPH.

| Graph | $\|\wedge\|$ ($\times 10^9$) | $\|\triangle\|$ ($\times 10^9$) | $m$ ($\times 10^6$) | $n$ ($\times 10^6$) | $d_{\max}$ | $c_{\max}$ | $t_{\max}$ | $\frac{\|\wedge\|}{3\|\triangle\|}$ |
|---|---|---|---|---|---|---|---|---|
| Friendster | 446.1 | 0.0002 | 1800.0 | 119.4 | 3618 | 168 | 5 | 775 605.4 |
| soc-friendster | 720.7 | 4.2 | 1806.1 | 65.6 | 5214 | 304 | 129 | 57.6 |
| graph500-scale-23-ef16 | 685.7 | 4.5 | 129.3 | 4.6 | 272 176 | 1095 | 625 | 50.2 |
| graph500-scale-24-ef16 | 1824.0 | 9.9 | 260.3 | 8.9 | 431 690 | 1432 | 791 | 61.2 |
| graph500-scale-25-ef16 | 4830.7 | 21.6 | 523.5 | 17.0 | 684 732 | 1855 | 996 | 74.6 |

TABLE 3. PKT $k$-TRUSS DECOMPOSITION PERFORMANCE.

| Graph | Perf (GWEPS) | Time (s) | 24-core Rel. Speedup | Speedup wrt [5] |
|---|---|---|---|---|
| Friendster | 1.69 | 265 | 15.52× | 3.34× |
| soc-friendster | 1.41 | 511 | 17.89× | 1.99× |
| graph500-scale23-ef16 | 2.64 | 260 | 14.82× | 1.62× |
| graph500-scale24-ef16 | 2.69 | 679 | 15.14× | 2.09× |
| graph500-scale25-ef16 | 2.77 | 1742 | 14.96× | 2.44× |

TABLE 4. PKT: BREAKDOWN OF TIME IN EACH PHASE.

| Graph | percentage time in phase | | |
|---|---|---|---|
| | Support | Scan | Processing |
| Friendster | 17.1% | 1.3% | 81.6% |
| soc-friendster | 11.9% | 2.0% | 86.1% |
| graph500-scale23-ef16 | 3.8% | 1.3% | 95.0% |
| graph500-scale24-ef16 | 3.4% | 1.2% | 95.4% |
| graph500-scale25-ef16 | 2.9% | 1.2% | 95.9% |

TABLE 5. TRIANGLE COUNTING PERFORMANCE.

| Graph | Perf (GWEPS) | Time (s) | 24-core Rel. Speedup | Speedup wrt AI-based |
|---|---|---|---|---|
| Friendster | 17.0 | 26.2 | 16.37× | 1.72× |
| soc-friendster | 23.3 | 31.0 | 17.04× | 2.26× |
| graph500-scale23-ef16 | 239.5 | 2.9 | 16.88× | 4.13× |
| graph500-scale24-ef16 | 268.6 | 6.8 | 17.06× | 4.56× |
| graph500-scale25-ef16 | 306.0 | 15.8 | 16.83× | 4.83× |

to Graph500 GTEPS. For our $k$-truss decomposition implementation, this rate is easy to interpret, because the dominant term in the number of operations performed is proportional to the number of wedges. However, for our triangle counting implementation, the number of wedges is only an upper bound on the operation count, since we avoid inspecting a large fraction of wedges with the vertex ordering and canonical triangle labeling optimizations. Nevertheless, we believe GWEPS is a better metric to use than GTEPS for both triangle counting and truss decomposition.

We report performance of only our algorithms in this paper, and the graphs analyzed is also very limited. For additional performance comparisons to the $k$-truss decomposition algorithms in [25] and [6], and a larger collection of test graphs, please see the empirical evaluation section of [5]. soc-Friendster is the common graph in both these papers.

**$k$-truss decomposition performance.** In Table 3, we give the GWEPS rate, execution time, and the relative speedup on 24 cores with our PKT implementation. The rate is between 1.41 to 2.77 GWEPS, suggesting that normalizing by wedge count is indeed reasonable for this implementation. The speedup over single-threaded execution varies from $14.82\times$ to $17.89\times$. Recall that $k$-truss decomposition in this paper refers to computing the trussness of every edge, but we do not explicitly list the trusses. Listing trusses would require executing connected components on a subset of the graph edges for a range of $k$ values, from 3 to $t_{\max}$.

An interesting aspect to note is that the performance of Friendster and soc-friendster are fairly close, despite the orders-of-magnitude difference in number of triangles and $t_{\max}$. There may be further room for improvement with Friendster, given the low $t_{\max}$.

Our current PKT implementation uses an adjacency intersection-based edge processing strategy instead of the adjacency marking-based strategy in [5]. This change turns out to be beneficial for all 5 graphs considered in this study. We give the speedup using the intersection-based approach over the marking-based approach in the fifth column of Table 3. This value ranges from $1.62\times$ to $3.34\times$. We need to further investigate why we have a range of speedups.

In Table 4, we list the breakdown of the percentage of time spent in each phase: support computation, cumulative scan, and cumulative edge processing. Edge processing is the dominant phase for all the graphs. The scan phase cost is insignificant because of the relatively low $t_{\max}$ for all the graphs, and the regular memory accesses. Support computation time is relatively higher for the Friendster instances compared to the Graph500 instances, and this partially explains the higher overall performance rates (in Table 3) for the Graph500 instances. Since support computation can exploit vertex ordering and canonical triangle labeling, the ratio of support computation time to the processing phase time is an indicator of the impact of these work-reducing optimizations.

The geometric mean performance rate for these five graph instances is **2.16 GWEPS**.

**Triangle counting performance.** We next report triangle counting performance in Table 5. The performance rate and time correspond to 24-core execution. The performance

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

is significantly better on the Graph500 instances than on the Friendster graphs. The range of relative speedups is narrower than the truss decomposition case. The performance reported corresponds to the adjacency marking-based implementation. The marking-based approach turns out to be consistently faster than the intersection-based approach, as expected [20], and the speedup ranges from $1.72\times$ to $4.83\times$. However, it is unclear from this table why the performance for the Friendster instances is lower than the Graph500 instances. Additionally, comparing the rates for triangle counting and truss decomposition for the SCALE 25 Graph500 instance, we find that triangle counting is nearly two orders of magnitude faster.

**Analyzing triangle counting performance.** In order to better explain triangle counting performance results, we report several work estimates and speedups achieved with optimizations in Table 6. Before running triangle counting or truss decomposition, we perform a preprocessing step where we relabel the vertices to be in increasing order of coreness. We can get the coreness values of all vertices by performing $k$-core decomposition, and we use our recently developed PKC [26] approach. The running times for $k$-core decomposition are listed in column 8. After getting coreness values, the time to reorder the graph is given in column 9. The time for both these steps is proportional to the number of graph edges. Further, our ordering implementation is currently untuned, and is thus significantly slower than the actual $k$-core decomposition. The second column of the table gives the triangle counting speedup achieved when using the coreness-based vertex ordering. It is as high as $11.26\times$. For this reason, the previously-reported triangle counting times use this $k$-core ordering and not the natural ordering in the graph.

In columns 3-5 of Table 6, we give estimates of work performed by the adjacency marking-based approach, taking ordering into consideration. The work ratio column estimates the work savings with ordering, and is an indicator of possible speedup with ordering. Column 6 gives another coarser work estimate. This is the work performed by an algorithm that does not consider a canonical labeling for a triangle. Column 7 gives the work savings when using $k$-core ordering, over the simple counting approach. For the Graph500 instances, this value is higher than the Friendster instances. This partially explains the high GWEPS rate reported in Table 5.

Drilling down even further, notice that $\sum d^+(v)^2$ is not a tight bound on the number of edges visited, or the number of lookups to the mark array. In Table 7, we compute a finer-grained work estimate taking a break statement into consideration. This value is 5.1 to 7.4 times lower than the $\sum d^+(v)^2$ bound. However, even column 2 of Table 7 does not fully explain why triangle counting time for Friendster is higher than the SCALE 25 Graph500 instance. We suspect the lowered performance is due to the larger number of vertices in the Friendster and soc-friendster instances, leading to marking array bitvectors that do not fit in L3 cache.

## 5. Future Research Directions

We believe our overall performance results are competitive for both triangle counting and truss decomposition, but we also think there is much room for performance improvement.

**Parallel $k$-truss decomposition.** There are several opportunities to further improve performance of our PKT approach:

- How much room for performance improvement is possible by tuning the dynamic loop scheduling chunk sizes?
- A marking-based edge processing implementation could benefit from explicitly deleting edges from the graph. Would edge deletion improve performance?
- We use three separate bit vectors of size $m$ to mark the state of edges [5]. Can these bit vectors be combined or altogether avoided, and will this reduce the overall running time?
- Can we reduce the number of conditional statements in the inner loop of the edge processing routine?

Our PKT implementation does not use a hash table. How would the performance of a highly tuned hash table-based approach, such as Wang and Cheng [6], compare to PKT?

For graph instances that have low overall triangle counts in comparison to the wedge count, the explicit triangle listing approach may be faster than approaches such as PKT. A shared-memory multicore implementation of the explicit triangle listing approach will also be of interest.

In PKT, we currently perform $\Theta(|\triangle|)$ atomic updates to the edge support values. Can we use a hybrid approach combining the work-inefficient local truss decomposition approach of Sarıyüce et al. [13] with the level-synchronous PKT to reduce fine-grained synchronization?

In their Graph Challenge submission, Smith et al. [27] present a new truss decomposition algorithm called MSP. MSP avoids fine-grained synchronization using a two-step frontier creation strategy: triangles are first added to thread-local queues, and following barrier synchronization, edge supports are updated. MSP also uses a static load-balancing strategy. We plan to directly compare PKT and MSP in future work.

Green et al. [28] present a level-synchronous algorithm that relies on a dynamic triangle counting subroutine and a graph representation that is amenable to fast edge deletions. Their GPU implementation is significantly faster than the reference Graph Challenge code.

Our implementation is designed for execution on a large memory multicore server. It will be interesting to note the best-performing approach for the following classes of parallel platforms:

- an Intel Xeon Phi system: due to the larger number of cores and lower per-core cache (in comparison to multicore servers), adjacency marking-based approaches may be infeasible to implement.
- a distributed-memory cluster with the graph replicated on each node: i.e., how would performance scale for the graph instances considered in this work?
- a distributed-memory cluster with partitioned graph and data structures; External memory settings.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE 6. IMPACT OF "INCREASING $k$-CORE" VERTEX IDENTIFIER ORDERING ON PARALLEL TRIANGLE COUNTING PERFORMANCE. WE ALSO LIST WORK ESTIMATES FOR DIFFERENT TRIANGLE COUNTING APPROACHES, PARALLEL $k$-CORE COMPUTATION TIME, AND GRAPH REORDERING TIME.

| Graph | Speedup w/ Ordering | $\sum_{v \in V}(d^+(v)^2)(\times 10^9)$ KCO | NAT | Work Ratio | $\sum d(v)^2$ $(\times 10^9)$ | $\sum d(v)^2 / \sum d^+(v)^2$ | $k$-core time (s) | Ordering time (s) |
|---|---|---|---|---|---|---|---|---|
| Friendster | 5.19× | 223.4 | 572.6 | 2.6 | 895.8 | 4.0 | 15.7 | 111.7 |
| soc-friendster | 3.83× | 398.6 | 815.0 | 2.0 | 1444.9 | 3.6 | 21.2 | 112.6 |
| graph500-scale23-ef16 | 8.79× | 107.6 | 789.9 | 7.3 | 1371.7 | 12.8 | 0.9 | 15.7 |
| graph500-scale24-ef16 | 9.27× | 319.8 | 1999.3 | 6.3 | 3648.6 | 11.4 | 1.7 | 32.6 |
| graph500-scale25-ef16 | 11.26× | 754.0 | 5036.6 | 6.7 | 9662.5 | 12.8 | 3.4 | 69.5 |

TABLE 7. A FINER-GRAINED WORK ESTIMATE FOR MARKING-BASED TRIANGLE COUNTING.

| Graph | Edges Vis. $(\times 10^9)$ | $\sum(d^+(v)^2)$ $(\times 10^9)$ | Ratio |
|---|---|---|---|
| Friendster | 42.8 | 223.4 | 5.2 |
| soc-friendster | 78.7 | 398.6 | 5.1 |
| graph500-scale23-ef16 | 17.2 | 107.6 | 6.2 |
| graph500-scale24-ef16 | 43.1 | 319.8 | 7.4 |
| graph500-scale25-ef16 | 102.0 | 754.0 | 7.4 |

- A single GPU; A single compute node with multiple GPUs; A multi-node GPU cluster.

**Support computation.** Computing the number of triangles every edge participates in, or edge support computation, is a required step in nearly all truss decomposition algorithms. We currently modify the adjacency marking-based approach for triangle counting to perform support computation. A drawback of our approach is the heavy use of atomics. We can avoid atomics if we sacrifice work efficiency. Another alternative is to use an adjacency intersection-based approach. It will be interesting to further explore the design space of support computation algorithms.

**Triangle listing.** While there has been tremendous progress related to exact and approximate triangle counting algorithms, as well as triangle listing algorithms, detailed performance comparisons of competing parallel implementations are still unavailable. Further, it is unclear how one could report normalized performance. Additionally, optimality notions with regards to vertex ordering are lacking. Similar to truss decomposition, it will be interesting to track top hardware-implementation combinations to process a fixed graph instance.

## 6. Conclusion

We reported performance results for parallel triangle counting and truss decomposition on the largest Graph Challenge instances. We introduced a performance metric, Giga Wedges Examined Per Second (GWEPS), to assess performance of both triangle counting and truss decomposition. For truss decomposition, we achieved a geometric mean performance rate of 2.16 GWEPS on a 24-core shared-memory server. For triangle counting, the best rate is nearly two orders of magnitude higher than the relatively constant rate for truss decomposition.

## Acknowledgments

This research is supported by the US National Science Foundation grants ACI-1253881 and CCF-1439057. This research was conducted with Advanced CyberInfrastructure computational resources provided by The Institute for CyberScience at The Pennsylvania State University (http://ics.psu.edu).

## References

[1] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[2] D. Matula and L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," *J. ACM*, vol. 30, no. 3, pp. 417–427, 1983.

[3] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," National Security Agency, Tech. Rep., 2008.

[4] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static Graph Challenge: Subgraph Isomorphism," in *Proc. IEEE High Performance Extreme Computing (HPEC)*, 2017.

[5] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," arXiv.org e-Print archive, https://arxiv.org/abs/1707.02000, July 2017.

[6] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 812–823, May 2012.

[7] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, no. 1, pp. 210–223, 1985.

[8] J. Cohen, "Graph twiddling in a MapReduce world," *Computing in Science & Engineering*, vol. 11, no. 4, pp. 29–41, 2009.

[9] K. Saito, T. Yamada, and K. Kazama, "Extracting communities from complex networks by the $k$-dense method," in *Proc. Int'l. Workshop on Mining Complex Data (MCD)*, 2006.

[10] Y. Zhang and S. Parthasarathy, "Extracting analyzing and visualizing triangle k-core motifs within networks," in *Proc. Int'l. Conf. on Data Engineering (ICDE)*, 2012.

[11] A. Verma and S. Butenko, "Network clustering via clique relaxations: A community based approach," in *Graph Partitioning and Graph Clustering*, D. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds. AMS, 2013, ch. 9, pp. 129–139.

[12] A. E. Sarıyüce, C. Seshadhri, A. Pinar, and Ü. V. Çatalyürek, "Finding the hierarchy of dense subgraphs using nucleus decompositions," in *Proc. Int'l. Conf. on World Wide Web (WWW)*, 2015.

[13] A. E. Sarıyüce, C. Seshadhri, and A. Pinar, "Parallel local algorithms for core, truss, and nucleus decompositions," arXiv.org e-Print archive, https://arxiv.org/abs/1704.00386, 2017.

[14] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," in *Proc. Symp. on Principles of Distributed Computing (PODC)*, 2011.

[15] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theoretical Computer Science*, vol. 407, no. 1–3, pp. 458–473, 2008.

[16] M. Ortmann and U. Brandes, "Triangle listing algorithms: Back from the diversion," in *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2014.

[17] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Proc. Int'l. Conf. on Data Engineering (ICDE)*, 2015, pp. 149–160.

[18] Y. Cui, D. Xiao, and D. Loguinov, "On efficient external-memory triangle listing," in *Proc. Int'l. Conf. on Data Mining (ICDM)*, 2016.

[19] D. Xiao, Y. Cui, D. B. Cline, and D. Loguinov, "On asymptotic cost of triangle listing in random graphs," in *Proc. Symp. on Principles of Database Systems (PODS)*, 2017.

[20] S. Parimalarangan, G. M. Slota, and K. Madduri, "Fast parallel graph triad census and triangle counting on shared-memory platforms," in *Proc. Workshop on Parallel and Distributed Processing for Computational Social Systems (ParSocial)*, 2017.

[21] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam, "Fast linear algebra-based triangle counting with KokkosKernels," in *Proc. IEEE High Performance Extreme Computing (HPEC) Graph Challenge*, 2017.

[22] N. S. Dasari, D. Ranjan, and M. Zubair, "ParK: An efficient algorithm for k-core decomposition on multicore processors," in *Proc. Int'l. Workshop on High Performance Big Graph Data Management, Analysis, and Mining (BigGraphs)*, 2014.

[23] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. AAAI Conf. on Artificial Intelligence (AAAI)*, 2015. [Online]. Available: http://networkrepository.com

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

[25] R. A. Rossi, "Fast triangle core decomposition for mining large graphs," in *Proc. Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining (PAKDD)*, 2014.

[26] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *Proc. Workshop on Parallel and Distributed Processing for Computational Social Systems (ParSocial)*, 2017.

[27] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *Proc. IEEE High Performance Extreme Computing (HPEC) Graph Challenge*, 2017.

[28] O. Green, J. Fox, E. Kim, F. Busato, N. Bombieri, K. Lakhotia, S. Zhou, S. Singapura, H. Zeng, R. Kannan, V. Prasanna, and D. Bader, "Quickly finding a truss in a haystack," in *Proc. IEEE High Performance Extreme Computing (HPEC) Graph Challenge*, 2017.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE