Truss Decomposition on Shared-Memory Parallel Systems

Shaden Smith[*†], Xing Liu[†], Nesreen K. Ahmed[†], Ancy Sarah Tom[*], Fabrizio Petrini[†], George Karypis[*]
[*] Department of Computer Science and Engineering, University of Minnesota
[†] Parallel Computing Lab, Intel Corporation
{shaden, karypis}@cs.umn.edu, tomxx030@umn.edu, {xing1.liu, nesreen.k.ahmed, fabrizio.petrini}@intel.com

***Abstract*—The scale of data used in graph analytics grows at an unprecedented rate. More than ever, domain experts require efficient and parallel algorithms for tasks in graph analytics. One such task is the *truss decomposition*, which is a hierarchical decomposition of the edges of a graph and is closely related to the task of triangle enumeration. As evidenced by the recent GraphChallenge, existing algorithms and implementations for truss decomposition are insufficient for the scale of modern datasets. In this work, we propose a parallel algorithm for computing the truss decomposition of massive graphs on a shared-memory system. Our algorithm breaks a computation-efficient serial algorithm into several bulk-synchronous parallel steps which do not rely on atomics or other fine-grained synchronization. We evaluate our algorithm across a variety of synthetic and real-world datasets on a 56-core Intel Xeon system. Our serial implementation achieves over $1400\times$ speedup over the provided GraphChallenge serial benchmark implementation and is up to $28\times$ faster than the state-of-the-art shared-memory parallel algorithm.**

I. Introduction

Truss decomposition is a powerful tool for discovering community structure and can provide insights during graph analytics. The ability to efficiently compute truss decomposition is critical as graphs become increasingly large and sparse. Appropriately, the recent GraphChallenge [1] calls for researchers to develop novel hardware and software to enable truss decomposition on modern datasets.

Most algorithms for computing truss decomposition are based on the concept of *peeling* edges [2], [3]. At each stage, edges are eliminated from the graph and their incident edges are updated. The resulting algorithm is computationally efficient, but is challenging to parallelize due the unstructured nature of the computation and the dynamic nature of the graph.

Several works have explored the optimization of truss decomposition on distributed systems such as MapReduce [2], [4] and Pregel [4], [5]. However, these methods are not designed for high-performance computing systems and do not consider implementation details which are critical for achieving high throughput on modern architectures.

Recently, Sariyuce et al. [6] presented the asynchronous nucleus decomposition algorithm (AND), which is the state-of-the-art parallel algorithm for truss and other related graph decomposition on shared-memory systems. Nucleus decomposition is a generic framework for graph decompositions that is capable of utilizing higher-order structures such as cliques [7] and generalizes the k-core and truss approaches to discover dense subgraphs. Instead of the traditional peeling process, AND iteratively traverses the graph structure and updates the truss decomposition based on the H-index values computed recursively from the local neighborhoods of edges [8]. The localized nature of AND allows for lock-free parallel execution at the cost of more computation than the traditional peeling algorithm. As a result, AND is not guaranteed to outperform a serial peeling implementation despite its high scalability.

In this work, we propose a shared-memory parallel algorithm that is based on peeling. We break the peeling process into multiple bulk-synchronous stages, and thus call the algorithm *multi-stage peeling* (MSP). In summary, our contributions include:

1) We describe an optimized serial implementation of the peeling algorithm for truss decomposition. Our serial implementation outperforms the GraphChallenge benchmark implementation by over $1400\times$ and is able to decompose graphs with over 100 million edges in only a few minutes time.
2) We propose MSP, a parallel algorithm for truss decomposition on shared-memory systems. MSP uses the efficient serial peeling algorithm as a basis, and like AND, is lock- and atomic-free. MSP is demonstrated to outperform AND by up to $28\times$.
3) We perform an extensive evaluation of our algorithms on a diverse set of graphs using a modern hardware platform. We compute the full truss decomposition of a graph with 1.2 billion edges on a single shared-memory system. To the best of our knowledge, this is the largest truss decomposition performed on any graph in the literature.

II. Background and Notation

Let $G = (V, E)$ be an undirected, unweighted, and simple graph with no self-loops, where V and E are the vertex and edge set respectively. Thus, $|V|$ is the number of vertices, and $|E|$ is the number of edges. We use $A(v)$ to denote the set of neighbors of vertex $v \in V$, and so $\deg(v) = |A(v)|$ is the degree of v. We assume vertices and edges are unique in order to identify them by their indices.

A triangle in G is a cycle/clique of three vertices. Let $\{u, v, w\} \subset V$, $\{u, v, w\}$ is a triangle if and only if all three edges exist in G (i.e., $(u, v), (u, w), (v, w) \in E$). We use the general notation Δ to denote the set of all triangles in the

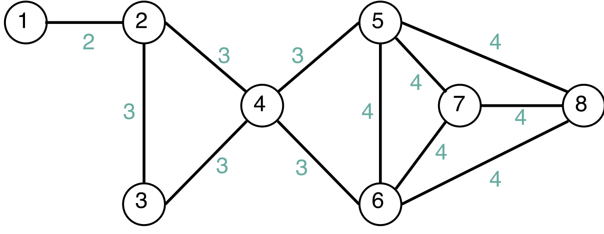


Fig. 1: The truss decomposition of a graph. Each edge e is labeled with its truss number, $\Gamma(e)$.

graph G, and so $|\Delta|$ denotes the number of triangles in G. Similarly, we use Δ_e to denote the set of all triangles incident to edge $e \in E$.

Using the definition of triangles, we define the support of an edge $e = (u, v) \in E$, denoted by $\text{sup}(e)$, as $\text{sup}(e) = |A(u) \cap A(v)| = |\Delta_e|$. Thus, the support of an edge $e = (u, v) \in E$ is the number of triangles incident to e.

We now define the notion of k-truss, which was first introduced by Cohen (see [9] for details). The k-truss of a graph G, denoted by G_k, is the largest subgraph of G, such that for every edge $e \in E_{G_k}$, $\text{sup}(e) \geq (k-2)$, where $k \geq 2$. Clearly, the 2-truss of G is equivalent to the graph itself. Further, we define the *truss number* of an edge e in G, denoted by $\Gamma(e)$, as the maximum value k such that $e \in G_k$. Thus, if $\Gamma(e) = k$, then $e \in G_k$ but $e \notin G_{k+1}$. Finally, we use k_{max} to denote the maximum truss number of any edge in the graph G.

In this work, we present serial and parallel algorithmic optimizations for truss decomposition on shared-memory systems. Given a graph G, we compute the full k-truss decomposition of G for all $2 \leq k \leq k_{\text{max}}$. An example truss decomposition is illustrated in Figure 1.

Table I provides a summary of the graphs and their main properties including the number of triangles and the maximum truss number.

III. Optimized Serial Peeling Algorithm

We now detail the serial peeling algorithm that provides a basis for our parallel algorithm. We work from the algorithm by Wang and Cheng [2], presented in Algorithm 1, and discuss the design decisions which were made to achieve high performance.

1) Tracking supports: The peeling process begins with the generation of a *frontier* (Line 4). The frontier $\mathcal{F}_k$ is the set of edges at iteration k whose support values are smaller than $k-2$. In order to efficiently generate the frontier, we follow a similar strategy as existing works that track edges in *support buckets* [2], [10]. For each unique support value τ, a bucket is allocated to store edges whose support values are equal to τ. We implement support buckets as doubly-linked lists so that deletions and insertions can be done in $\mathcal{O}(1)$ time. As support value $\text{sup}(e)$ is decremented, edge e will be moved into bucket $\max\{\text{sup}(e), k-3\}$. This selection of bucket allows the peeling process to proceed by only examining a single bucket during

Algorithm 1: Serial Peeling

1. Compute initial supports and store in sup;
2. $k \leftarrow 3$;
3. **while** $|E| > 0$ **do**
4. $\mathcal{F}_k \leftarrow \{e \in E : sup(e) < k-2\}$;
5.   **while** $|\mathcal{F}_k| > 0$ **do**
6.   **foreach** $e \in \mathcal{F}_k$ **do**
7.   **foreach** $e' \in \Delta_e$ **do**
8. $sup(e') \leftarrow sup(e') - 1$;
9.   **end**
10. $E \leftarrow E \setminus \{e\}$;
11. $\Gamma(e) \leftarrow k-1$;
12.   **end**
13. $\mathcal{F}_k \leftarrow \{e \in E : \text{sup}(e) < k-2\}$;
14.   **end**
15. $k \leftarrow k+1$;
16. **end**

the next iteration (i.e., edges in the frontier can be extracted in the next iteration by only accessing bucket $k-3$).

2) Triangle enumeration: Triangle enumeration (Line 7) is a major cost during the peeling process. Given an edge $e = (u, v) \in \mathcal{F}_k$, we need to enumerate the triangles incident to e, which is equivalent to computing the intersection of $A(u)$ and $A(v)$. We implement the set intersection as a linear merge of the two adjacency lists [11]. Since we know how many triangles will be found by simply examining the current support value of e, the merge exits as soon as $\text{sup}(e)$ triangles are found, instead of performing the complete merge operation. Moreover, we follow the practice of reordering vertices by degree [11], [12] and perform the merge starting from the end of the adjacency lists. Intuitively, a degree-based ordering results in intersections occurring more frequently at the end of the adjacency lists, thus allowing an early exit from the merge operation.

3) Edge deletion: We facilitate edge deletions (Line 10) by storing each vertex's adjacency list as an array-based doubly-linked list. The array-based doubly-linked list provides edge deletion in $\mathcal{O}(1)$ time while still providing random access to edge data, which is used in other algorithmic stages.

IV. Parallel Multi-Stage Peeling

A natural approach when parallelizing the edge-centric Algorithm 1 is to peel the edges in $\mathcal{F}_k$ concurrently. This approach brings a number of challenges to consider:

- Line 6: if two edges belonging to the same triangle are peeled concurrently, the remaining edge will have its support decremented twice for the same triangle.
- Line 8: the support of an edge may be decremented concurrently by multiple threads.
- Line 10: race conditions can occur if multiple threads concurrently remove edges from the same adjacency list.

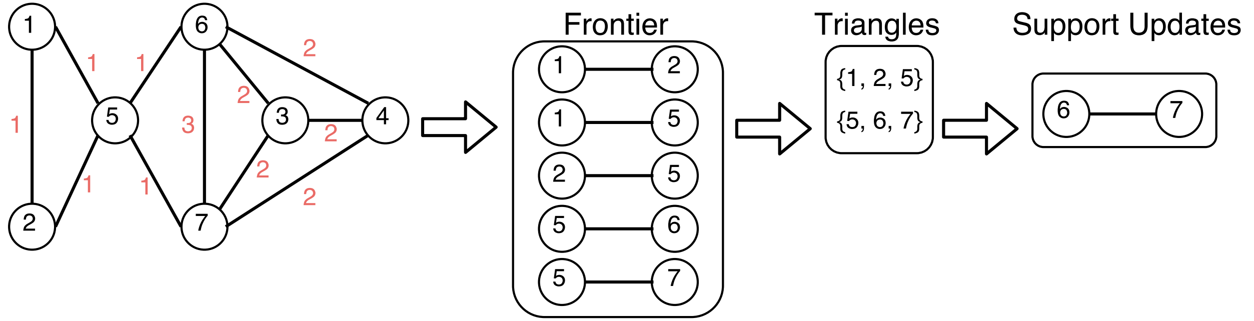


Fig. 2: One iteration of Multi-Stage Peeling (MSP). Edges are labeled with support values (not to be confused with truss numbers). The initial graph is the same as Figure 1 after reordering vertices by degree and removing edges with zero support.

- Line 10: there may be threads searching the graph for triangles while edges are removed, leaving the graph traversal in an inconsistent state.

We propose to split the computation into several bulk-synchronous substeps. By separating the various steps of the peeling process and using a static decomposition of vertices and edges, we are able to parallelize the peeling process without relying on fine-grained synchronizations such as mutexes and atomics. We call this algorithm *multi-stage peeling* (MSP).

MSP relies on communication between threads, implemented in the form of message queues. We denote a set of message queues for p threads $Q = \{q_1, \dots, q_{|Q|}\}$, where $|Q| \geq p$. We additionally require two functions which map vertices and edges to queues, denoted $\Psi : V \to Q$ and $\Phi : E \to Q$, respectively. We use a cyclic distribution in this work, i.e., $\Psi(v) = v \pmod{|Q|}$ and $\Phi(e) = e \pmod{|Q|}$. The MSP algorithm relies on two sets of queues, denoted Q^E and $Q^{\mathcal{F}}$, respectively.

Message queues are implemented such that each thread maintains a local portion of each $q_i \in Q$, and thus any thread can send messages to any queue without synchronization. The thread-local queues can then be aggregated into a single set of incoming messages after a thread barrier.

A. Frontier Generation

We distribute edges among a set of $|Q|$ support buckets in order to parallelize the frontier generation. It is advantageous to use $|Q|$ buckets and to assign edge e to bucket $\Phi(e)$ so that threads can update support buckets in a lock-free manner later in the computation. Since edges are not peeled uniformly throughout the graph, load imbalance could occur if we used the static partitioning provided by Φ for the more expensive substep of triangle enumeration. We therefore use the distributed support buckets to fill $\mathcal{F}_k$, which is a single thread-global work array that can be processed in a load balanced manner via a dynamically-scheduled for-loop.

For each edge $e \in \mathcal{F}$, we enumerate Δ_e using the optimized routines from Section III. Importantly, we must be careful to delete each triangle only once. Otherwise, incident edges will incorrectly have their supports decremented multiple times. To address this challenge, we introduce an ordering of the edges in a triangle. Suppose we have a triangle $\{u, v, w\}$, with $u < v < w$. We define the ordering of the edges as $(u, v) < (u, w) < (v, w)$. Note that this ordering naturally occurs when the graph is stored as an adjacency list. We select the triangle for deletion if and only if e is the lowest ordered edge in the triangle also found in $\mathcal{F}_k$.

For each triangle selected for deletion, we insert the two edges incident to e into queues Q^E. The destination queue is chosen based on the lower-numbered vertex of the edge. For example, suppose peeled edge (u, v) is part of triangle $\{u, v, w\}$. Then the thread inserts messages (u, w) and (v, w) into queues $q^E_{\Psi(u)}$ and $q^E_{\Psi(v)}$, respectively. Finally, the edge e is placed into a second queue $q^{\mathcal{F}}_{\Psi(u)}$. We note that a simple optimization is to only communicate edges if they are not already in $\mathcal{F}_k$, i.e., they are not already being removed from the graph.

After all edges in $\mathcal{F}_k$ have been processed and their triangles queued for support updates, the threads reach a barrier and we begin the next stage in frontier generation. Each $q^{\mathcal{F}} \in Q^{\mathcal{F}}$ is assigned to a thread. The thread extracts each edge $(u, v) \in q^{\mathcal{F}}$ and deletes it from $A(u)$. The modification of $A(u)$ can be done without locks because all edges present in $A(u)$ will be assigned to $q^{\mathcal{F}}_{\Psi(u)}$, which in turn is assigned to a single thread.

B. Support Updates

By this stage in MSP, all edges in $\mathcal{F}_k$ have been removed from the graph structure and the remaining edges of each deleted triangle have been queued for support updates. The support update stage occurs in two substeps: decrementing support values and updating support buckets.

Each $q^E \in Q^E$ is assigned to a thread for support decrements. After decreasing the support value of edge e, a support bucket is updated to reflect the new support value of e. Atomics are again unnecessary, as the partitioning of buckets among threads ensures that an edge will only ever be decremented by a single thread, and likewise a support bucket will only be updated by a single thread.

C. Load Balancing Message Queues

MSP relies on a static assignment of edges and vertices to message queues for lock-free computation. However, the irregular nature of the peeling process means that a static

TABLE I: Summary of datasets.

Graph	$\|V\|$	$\|E\|$	$\|\Delta\|$	$k_{\max}$
soc-Slashdot0811 [14]	77.3K	469.2K	551.7K	35
cit-HepTh [14]	27.7K	352.2K	1.5M	30
soc-Epinions1 [14]	75.8K	405.7K	1.6M	33
loc-gowalla [15]	196.6K	950.3K	2.3M	29
cit-Patents [14]	3.8M	16.5M	7.5M	36
soc-Orkut [15]	3.0M	106.3M	524.6M	75
twitter [16]	41.7M	1.2B	34.8B	1998
rmat22 [1]	2.4M	64.1M	2.1B	485
rmat23 [1]	4.5M	129.3M	4.5B	625
rmat24 [1]	8.9M	260.3M	9.9B	791
rmat25 [1]	17.0M	523.5M	21.6B	996

K, **M**, and **B** denote thousands, millions, and billions, respectively. The first group of graphs is taken from real-world datasets, and the second group is synthetic.

partitioning of the data is likely to result in load imbalance. To address this challenge, we overdecompose by assigning $|Q|$ to a multiple of p, and dynamically assign queues to threads. We empirically found $|Q| = 2p$ to give the best performance in our evaluation.

V. Experiments & Results

A. Experimental Setup

1) Hardware and Software Configuration: Our experimental test-bed is a dual-socket server with Intel Xeon Platinum 8180 processors running at 1.7 GHz. Each processor has 28 cores and is equipped with 192 GB of DDR4 memory. The system can deliver a total of 3046 GFLOP/s peak double precision performance and 210 GB/s STREAM Triad bandwidth [13]. Our experimental runs were limited to two hours due to system configuration.

Our k-truss implementations were written in C and compiled with Intel C++ Compiler 17.0. The provided serial baseline code was implemented in Matlab and run with GNU Octave 4.2.1. The Octave binary was built from source code with Intel C++ Compiler 17.0 and Math Kernel Library 2017.

2) Datasets: We evaluate our work with eleven graphs that span both synthetic and real-world datasets. Summarized in Table I, the graphs come from sources such as the Network Data Repository [15], SNAP [14], and pre-generated synthetic datasets provided by the GraphChallenge specification [1]. All graphs are stored initially using an adjacency list representation (i.e., CSR).

3) MSP Evaluation: We first examine the relative costs of the various substeps during the k-truss computation. Figure 3a shows the breakdown of times for serial computation. The initial support computation (i.e., triangle counting) is consistently the most inexpensive step, followed by support updates and lastly, frontier generation. As we move to parallel execution (Figure 3b), the support updates dominate the runtime in eight of eleven datasets. This is attributed to lower scalability of the support update stage. Scaling of the support update stage is reliant on thread-to-thread communication, which can be challenging in multi-socket NUMA systems such as the one used in our evaluation. Additionally, if updates are clustered around a few vertices, then the cyclic distribution used in MSP may not provide load balance.

TABLE II: Speedup over the provided serial baseline code.

Graph	Octave	Peeling	Speedup
soc-Slashdot0811	169.23	0.22	769.1×
cit-HepTh	448.23	0.40	1120.6×
soc-Epinions1	675.03	0.46	1467.4×
loc-gowalla	787.95	0.79	997.4×
cit-Patents	972.66	4.03	241.4×

Values are runtime in seconds. **Octave** is the serial Octave benchmark provided by the GraphChallenge specification [1]. **Peeling** is the proposed implementation of Algorithm 1. Speedup is measured relative to **Octave**.

TABLE III: Speedup over the serial peeling algorithm.

Graph	Peeling	AND		MSP	
cit-Patents	2.89	**0.23**	12.6×	0.58	5.0×
soc-Orkut	228.06	64.31	3.5×	**11.30**	20.2×
twitter	-	-	-	**1566.72**	
rmat22	403.59	398.46	1.0×	**42.22**	9.6×
rmat23	980.68	1083.66	0.9×	**85.14**	11.5×
rmat24	2370.54	4945.70	0.5×	**175.29**	13.5×
rmat25	5580.47	-	-	**352.37**	15.8×

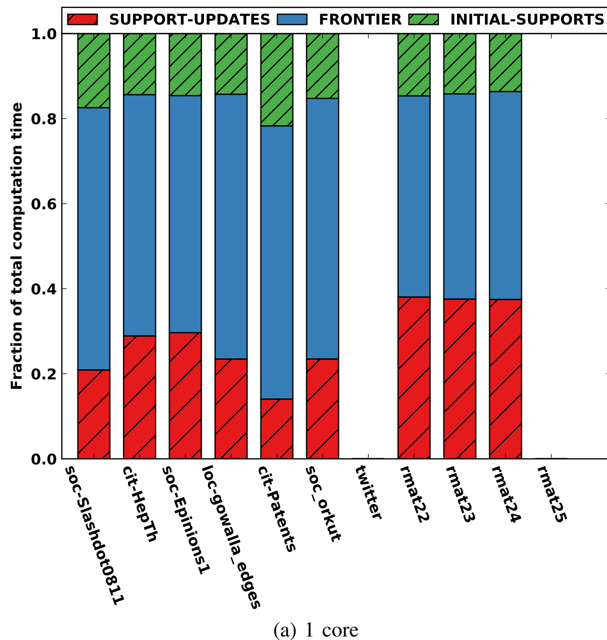
Values are runtimes, in seconds, of the truss decomposition. The time for initial support computation is omitted, as the same kernel is used for all datasets [19]. **Peeling** denotes the runtime achieved with Algorithm 1. **AND** [6] denotes the runtime of the asynchronous nucleus decomposition algorithm on 56 cores. **MSP** denotes the runtime of the parallel multi-stage peeling algorithm on 56 cores. All speedups are measured relative to **Peeling**. A dash indicates that the run was unable to be completed within the two hour time limit. The fastest result for each graph is bolded.

We present strong scaling results of the MSP algorithm in Figure 4. We include all graphs that require at least one second to complete serially. As seen in the figure, the curves start to flatten from 28 cores. This is attributed to a combination of communication across sockets and NUMA effects.

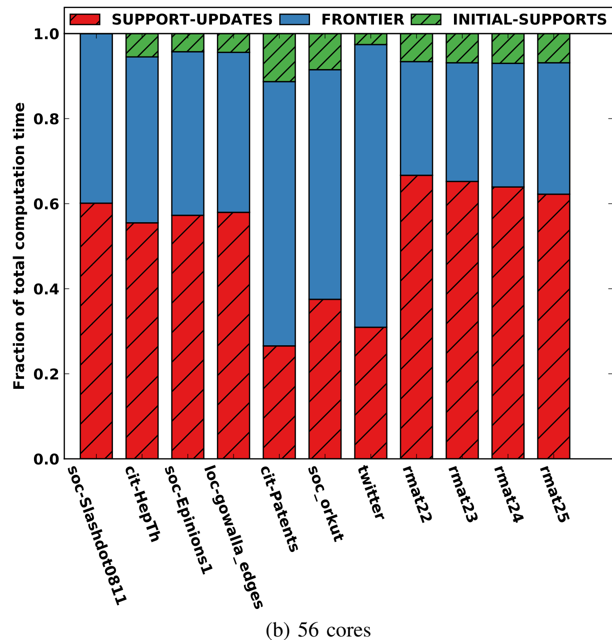
Figure 5 presents the time required for MSP to compute each k-truss. The figure shows that the computations on the low values of k dominate the runtime. This is expected because social network graphs obey power-law degree distributions and the edges connected to low degree vertices are likely to have lower support values. Notably, rmat25 has a pronounced outlier in the middle of the computation. The outlier corresponds to a large number of edges peeled at once. All of the evaluated synthetic graphs exhibited the same outlier in the middle of the peeling process.

4) Evaluation against the state-of-the-art: Table II compares the runtimes of the serial baseline code provided by the GraphChallenge specification [1] and the serial peeling implementation developed in this work (Section III). The serial baseline code is a linear algebraic formulation of the k-truss problem [17], [18] written in Matlab and was chosen because it has the highest performance of the supplied benchmarks. We only compare against our serial implementation for fairness. Our peeling implementation achieves up to $1467.4\times$ speedup over the provided serial baseline code.

Lastly, in Table III we compare MSP to our serial peeling implementation and asynchronous nucleus decomposition



(a) 1 core



(b) 56 cores

Fig. 3: Relative costs of the primary substeps of the k-truss computation. Pre-processing such as IO and graph construction are omitted. Missing bars indicate that the run was unable to complete within the two hour time limit.

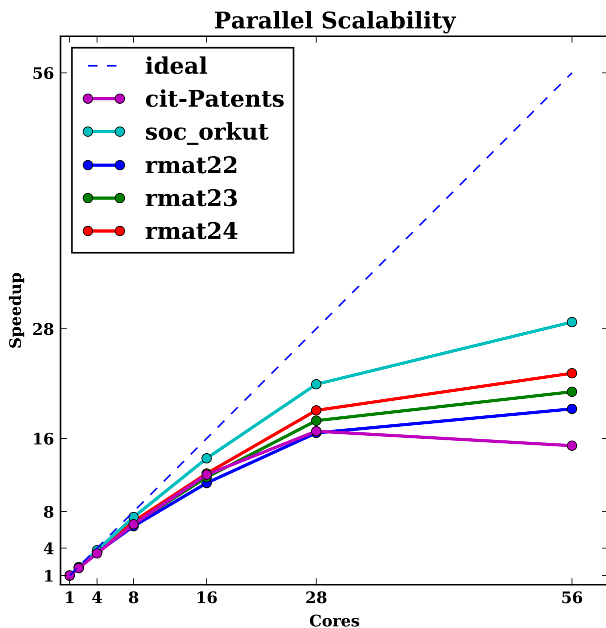


Fig. 4: Scalability of MSP for five graphs. `twitter` and `rmat25` are omitted due to serial runtimes exceeding the two hour time limit.

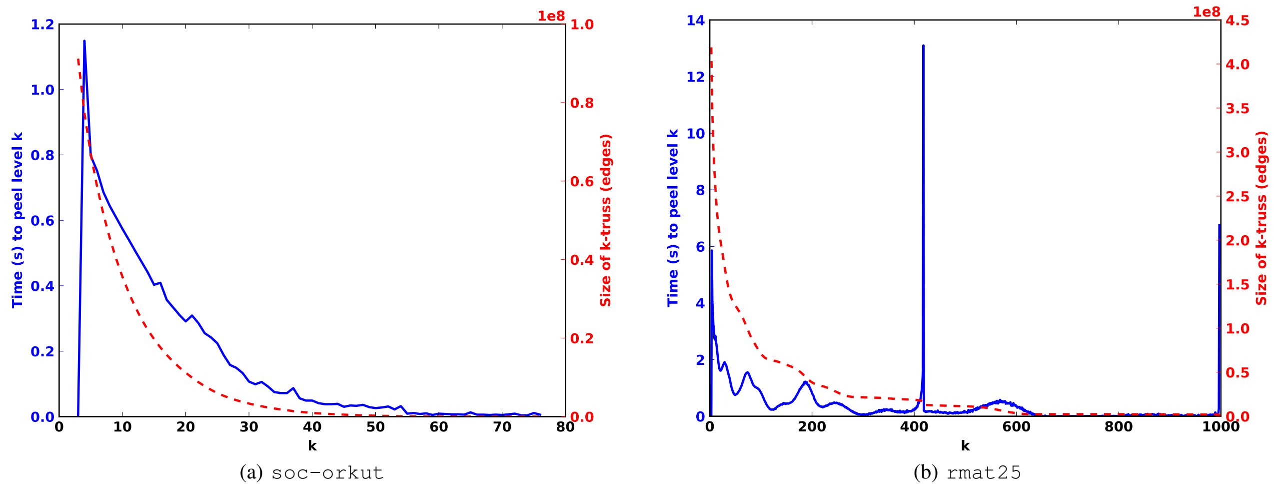
(AND), a state-of-the-art shared-memory algorithm for truss and other related graph decompositions [6]. We implemented AND with the same triangle enumeration primitives as MSP for a fair comparison. We again selected only the datasets which required at least one second of serial computation. AND outperforms the competing algorithms on `cit-Patents`, but as the graphs grow in size MSP eventually outperforms AND by 28× on `rmat24`. Larger graphs could not be completed by AND within the two hour time limit.

VI. Conclusion

Computing the truss decomposition of a graph is an essential step in many data analytics workloads. As evidenced by the GraphChallenge [1], the performance of existing algorithms for truss decompositions are insufficient for modern graph datasets on current and future hardware architectures. We presented MSP, a parallel algorithm for computing truss decomposition on shared-memory systems. MSP maintains the computational efficiency of the best serial algorithms while avoiding fine-grained synchronization. Resultingly, MSP is up to 28× faster than the state-of-the-art parallel algorithm. MSP is able to decompose a graph with 1.2 billion edges on a single compute node.

Acknowledgments

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota.



(a) `soc-orkut` (b) `rmat25`

Fig. 5: Time required to peel each k-truss and the size of each k-truss in edges.

References

[1] 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," *IEEE HPEC*, 2017.

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

[3] R. A. Rossi, "Fast triangle core decomposition for mining large graphs," in *Advances in Knowledge Discovery and Data Mining (PAKDD)*. Springer, 2014, pp. 310–322.

[4] P.-L. Chen, C.-K. Chou, and M.-S. Chen, "Distributed algorithms for k-truss decomposition," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 471–480.

[5] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 2014, pp. 613–624.

[6] A. E. Sariyuce, C. Seshadhri, and A. Pinar, "Parallel local algorithms for core, truss, and nucleus decompositions," *arXiv preprint arXiv:1704.00386*, 2017.

[7] N. K. Ahmed, J. Neville, R. A. Rossi, and N. Duffield, "Efficient graphlet counting for large networks," in *ICDM*, 2015, pp. 1–10.

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

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

[10] V. Batagelj and M. Zaversnik, "An O(m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.

[11] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 149–160.

[12] S. Parimalarangan, G. M. Slota, and K. Madduri, "Fast parallel graph triad census and triangle counting on shared-memory platforms," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 1500–1509.

[13] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

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

[15] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. [Online]. Available: http://networkrepository.com

[16] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.

[17] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[18] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–9.

[19] 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 *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.