Truss Decomposition on Shared-Memory Parallel Systems

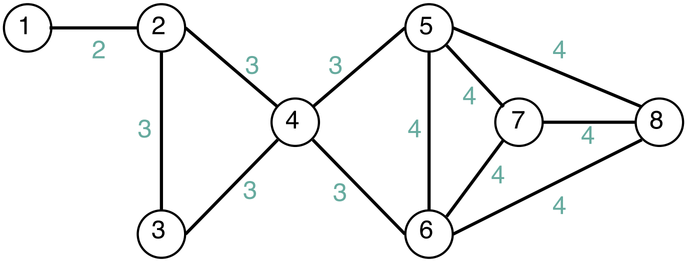
Shaden Smith[1,2], Xing Liu[2], Nesreen K. Ahmed[2], Ancy Sarah Tom[1], Fabrizio Petrini[2], and George Karypis[1]

[1]Department of Computer Science & Engineering, University of Minnesota
[2]Intel Parallel Computing Lab
shaden@cs.umn.edu

GraphChallenge Finalist, HPEC 2017

Truss decomposition

We are interested in computing the complete *truss decomposition* of a graph on shared-memory parallel systems.



Notation:

- A k-truss is a subgraph in which each edge is contained in at least $(k-2)$ triangles in the same subgraph.
- The truss number of an edge, $\Gamma(e)$, is the maximum k-truss that contains e.

Serial peeling algorithm

Peeling builds the truss decomposition bottom-up.

```
1: Compute initial supports and store in $sup(\cdot)$
2: $k \leftarrow 3$
3: while $|E| > 0$ do
4:     for each edge $e$ not in current $k$-truss do
5:         for each edge $e' \in \Delta_e$ do
6:             $sup(e') \leftarrow sup(e') - 1$
7:         end for
8:         $\Gamma(e) \leftarrow k - 1$
9:         Remove $e$ from $E$
10:    end for
11:    $k \leftarrow k + 1$
12: end while
```

Multi-Stage Peeling (MSP)

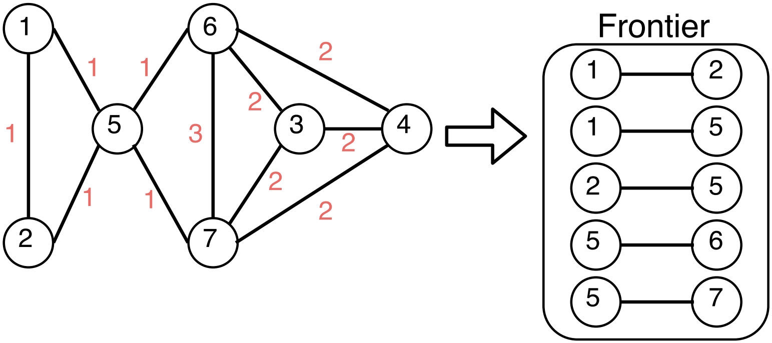
We break the peeling process into several bulk-synchronous substeps.

High-level idea:

- Store the graph as an adjacency list for each vertex (i.e., CSR).
- Do a 1D decomposition on the vertices.
- Operations which modify graph state (e.g., edge deletion and support updates) are grouped by source vertex.
    - Batching localizes updates to a specific adjacency list and eliminates race conditions.

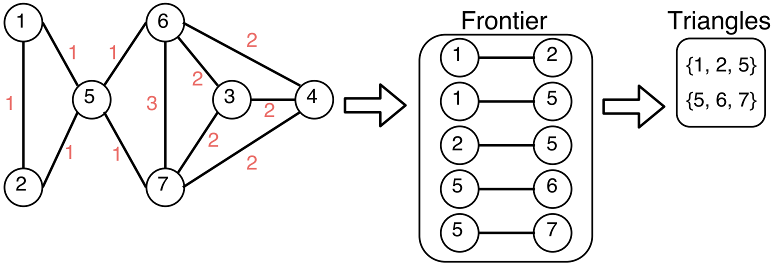
Multi-Stage Peeling (MSP)

Step 1: frontier generation



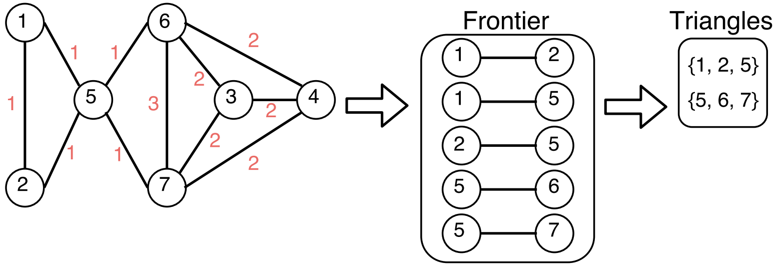
Multi-Stage Peeling (MSP)

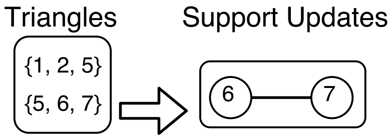
Step 2: triangle enumeration



Multi-Stage Peeling (MSP)

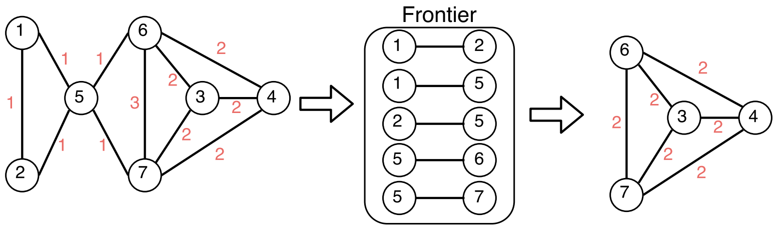
Step 3: support updates





Multi-Stage Peeling (MSP)

Step 4: edge deletion



Experimental Setup

Software:

- Parallel baseline: asynchronous nucleus decomposition (AND)[1], written in C and parallelized with OpenMP
- MSP is written in C and parallelized with OpenMP
- Compiled with `icc` v17.0

Hardware:

- 56-core shared-memory system (2$\times$ 28-core Skylake Xeon)
- 192GB DDR4 memory

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

Graphs

More datasets in paper.

Graph	$\|V\|$	$\|E\|$	$\|\Delta\|$	$k_{\max}$
`cit-Patents`	3.8M	16.5M	7.5M	36
`soc-Orkut`	3.0M	106.3M	524.6M	75
`twitter`	41.7M	1.2B	34.8B	1998
`rmat22`	2.4M	64.1M	2.1B	485
`rmat23`	4.5M	129.3M	4.5B	625
`rmat24`	8.9M	260.3M	9.9B	791
`rmat25`	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.

Parallel baseline comparison

MSP is up to 28× faster than AND and 20× faster than 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 full truss decomposition. **Peeling** is the optimized serial implementation. **AND** and **MSP** are executed on 56 cores.

Wrapping up

Multi-stage peeling (MSP):

- processes graph mutations in batches to avoid race conditions
    - resulting algorithm is free of atomics and mutexes
- can decompose a billion-scale graph on a single node in minutes

Relative to the state-of-the-art:

- Up to 28$\times$ speedup over the state-of-the-art parallel algorithm
- Serial optimizations achieve over 1400$\times$ speedup over the provided Matlab benchmark (*in paper*).

shaden@cs.umn.edu

Backup

Peeling algorithm

```
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:         for $e \in \mathcal{F}_k$ do
7:             for $e' \in \Delta_e$ do
8:                 $sup(e') \leftarrow sup(e') - 1$
9:             end for
10:            $E \leftarrow E \setminus \{e\}$
11:            $\Gamma(e) \leftarrow k-1$
12:            $\mathcal{F}_k \leftarrow \{e \in E : sup(e) < k-2\}$
13:        end for
14:    end while
15:    $k \leftarrow k+1$
16: end while
```

Parallelization challenges

A natural first approach to parallelization is to peel edges concurrently.

There are several challenges when parallelizing:

- graph data structure is dynamic
- supports must be decremented safely
- triangles may be counted multiple times

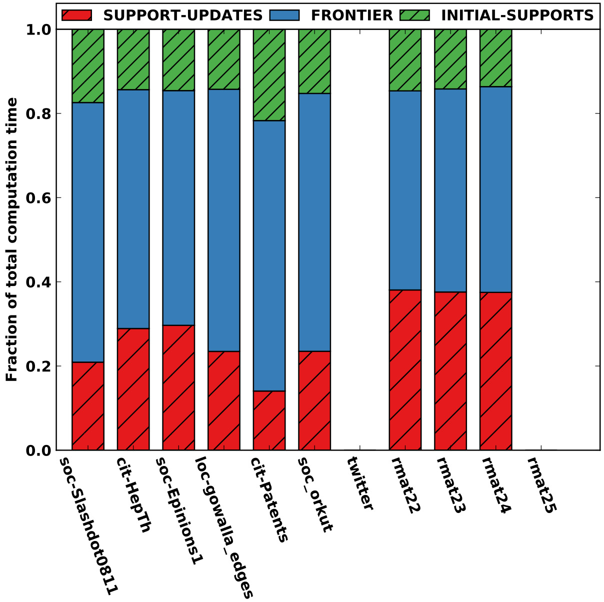
Serial benchmark comparison

The optimized peeling implementation achieves 1400× speedup over the GraphChallenge benchmark (both serial).

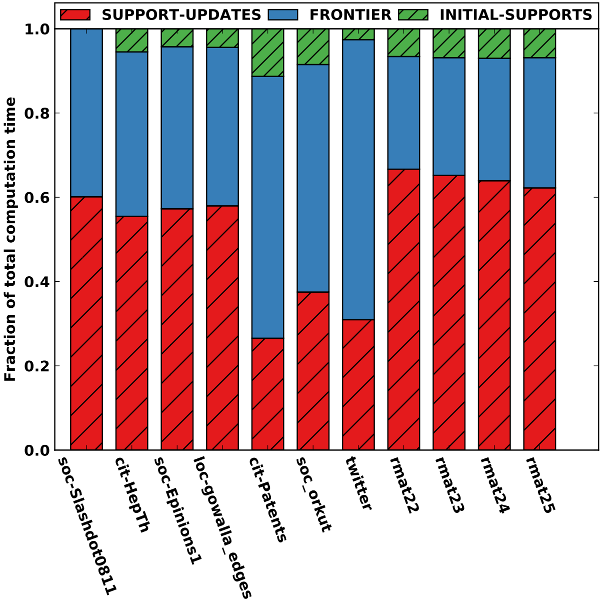
Graph	**Octave**	**Peeling**	**Speedup**
`soc-Slashdot0811`	169.23	0.22	769.1×
`cit-HepTh`	448.23	0.40	1120.6×
`soc-Epinions1`	675.03	0.46	1467.4×
`loc-gowalla`	787.95	0.79	997.4×
`cit-Patents`	972.66	4.03	241.4×

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

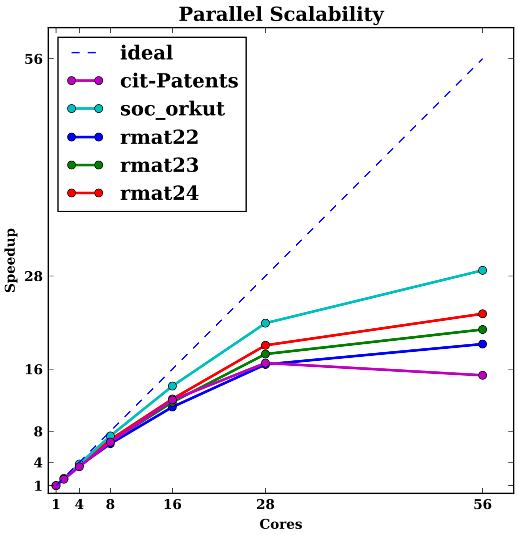
Serial breakdown



Parallel breakdown

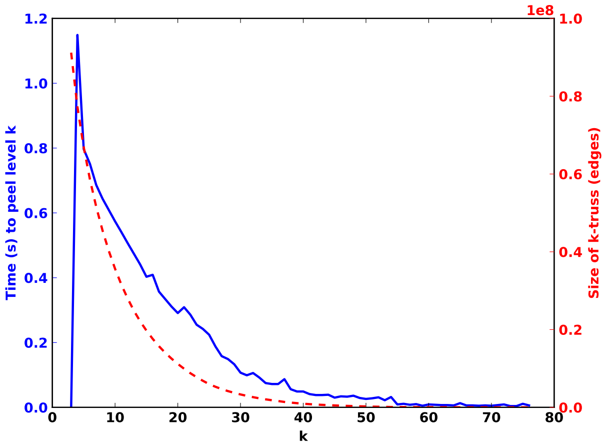


Strong scaling



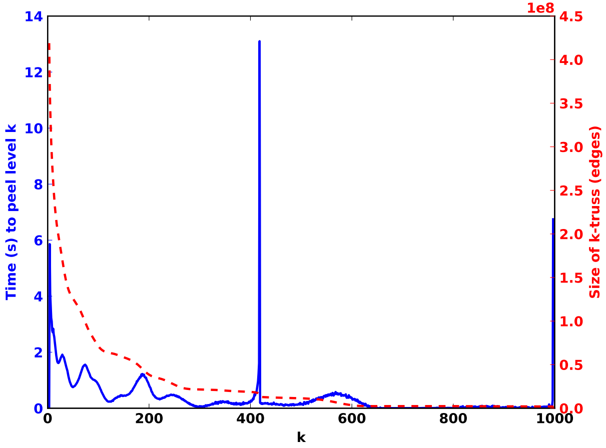
Cost per truss

The time per *k*-truss on `soc-orkut` is unsurprising.



Cost per truss

`rmat25` is more challenging.


Time (s) to peel level k
Size of k-truss (edges)
1e8
k