Fast Parallel Index Construction for Efficient K-truss-based Local Community Detection in Large Graphs

Md Abdul Motaleb Faysal
University of Nevada, Las Vegas
Las Vegas, Nevada, USA
faysal@unlv.nevada.edu

Maximilian Bremer
Lawrence Berkeley National Lab
Berkeley, California, USA
mb2010@lbl.gov

Cy Chan
Lawrence Berkeley National Lab
Berkeley, California, USA
cychan@lbl.gov

John Shalf
Lawrence Berkeley National Lab
Berkeley, California, USA
jshalf@lbl.gov

Shaikh Arifuzzaman
University of Nevada, Las Vegas
Las Vegas, Nevada, USA
shaikh.arifuzzaman@unlv.edu

ABSTRACT

Finding cohesive subgraphs is a crucial graph analysis kernel widely used for social and biological networks (graphs). There exist various approaches for discovering insightful substructures in a network, such as finding cliques, community discovery, and truss decomposition. Finding cliques is a computationally intractable problem, making it difficult to identify cohesive subgraphs in large graphs. One possible solution is k-truss decomposition, which is a relaxed form of finding cliques that can be solved in polynomial time. Further, unlike global community detection–which focuses on breaking down the entire graph into disjoint communities–a local or goal-oriented community search aims at finding the community of an entity of interest. In this work, we identify a k-truss-induced community discovery technique that can detect local communities in polynomial time. However, most previous studies have explored k-truss-induced local community formation in a serial setting, making them unsuitable for large graphs. In this paper, we design a parallel k-truss-induced local community construction method using multi-core parallelism. To the best of our knowledge, this is the first attempt to parallelize this algorithmic approach with extensive performance analysis. Our experiments demonstrate a significant performance improvement, with speedups from 19x to 55x for graphs with hundreds of millions to billions of edges, using NERSC Perlmutter compute nodes.

CCS CONCEPTS

• **Computing methodologies** → *Parallel algorithms*; • **Mathematics of computing** → **Graph algorithms**.

KEYWORDS

Graph algorithms, parallel algorithms, k-truss, local community discovery, large graphs, connected components, sparse graphs

ICPP 2023, August 7–10, 2023, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0843-5/23/08…$15.00
https://doi.org/10.1145/3605573.3605637

ACM Reference Format:
Md Abdul Motaleb Faysal, Maximilian Bremer, Cy Chan, John Shalf, and Shaikh Arifuzzaman. 2023. Fast Parallel Index Construction for Efficient K-truss-based Local Community Detection in Large Graphs. In *52nd International Conference on Parallel Processing (ICPP 2023), August 7–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605573.3605637

1 INTRODUCTION

Community discovery is a widely used application for grouping or clustering entities of similar categories [8, 29, 32, 35, 36]. Some examples include finding groups of people having similar interests in social networks, marketing products to groups of consumers based on their categories, clustering similar kinds of proteins and recognizing the functionality of unknown proteins, web spam detection in the cyber-security domain, and so on. In numerous real-world applications, the focus lies on determining the communities to which an entity (vertex in a graph) belongs, rather than identifying the independent disjoint communities of the entire graph [1, 22]. For instance, a user in a social network may be interested in the social groups or communities they participate in rather than all the communities in the network. This entity-centered personalized search is more meaningful as the communities a user participates in represent the social or behavioral context of the user. While the disjoint community problem usually applies a global criterion [18, 19] or optimization function to discover all qualified communities, the overlapping community problem generally constructs and maintains an index-based structure with an objective to retrieve community subgraphs containing the query vertex [1]. We refer to the latter problem as a local or goal-oriented community search. A key difference between these problems is that in global community discovery, a vertex belongs to only one community at a time (disjoint), whereas in local community discovery, a vertex may belong to multiple communities simultaneously (overlapping). In Figure 1, we illustrate the community membership for these two kinds of community discovery problems.

There have been goal-oriented local community discovery models proposed based on graph motifs such as *k-core* [5, 34, 42], *clique*/*quasi clique* [12, 44], and *k-truss* [1, 23, 45]. A k-truss-based index construction for local community search has merits over other techniques. For instance, the most obvious cohesive subgraph, *clique* [27], has the drawbacks of being very restrictive (every vertex

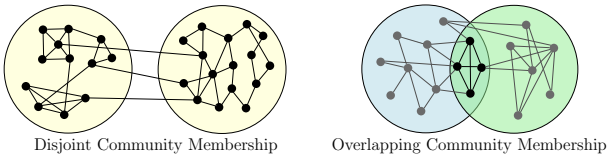


Figure 1: The left subfigure illustrates the disjoint communities in oval shapes–a vertex (dark circles) may belong to only one community. The right subfigure illustrates overlapping communities where some vertices, marked as dark, belong to multiple communities simultaneously.

within 1−distance) and too common (small clique) or too rare (large clique) in a real-world scenario. Moreover, it is not a polynomially tractable problem [9]. The *k-core* problem is a generalization of the clique. Despite being polynomially solvable, the *k-core* subgraphs have the disadvantage of lacking cohesion, an important property of the community subgraph [11]. *K-truss*, a relaxed form of the *clique*, can be computed in polynomial time. *K-truss* uses a higher-order graph motif of triangle connectivity as the building block for the formulation of a community instead of primitive features such as vertex set or edge set thus enabling a comprehensive model of multiple overlapping communities.

There are recent studies on *k-truss*-based goal-oriented community search [1, 20, 22]. The main limitation of the contemporary studies of *k-truss*-oriented index construction or community search is the sequential nature of those algorithms. One constituent subproblem of this local community search formulation is *k-truss* decomposition. The *k-truss* decomposition is a well-studied problem for parallel algorithm design. There exist works on parallel *k-truss* decomposition in both shared-memory [24, 41, 47] and distributed-memory [10, 16, 31] settings. There are also GPU-based studies [2, 14, 46] for *k-truss* decomposition. The work by Akbas et al. [1] proposed *EquiTruss*, a *k-truss*-based index structure that demonstrates better performance over *TCP-Index* proposed by Huang et al. [22] as a formulation to build the index for the local community search. Both of those studies are sequential and limited in scalability. We devise a shared-memory-based parallel algorithm for multicore setting using *EquiTruss* formulation that works on large graphs. We identify that *EquiTruss* can be computationally expensive for larger graphs as evident in Figure 2. While there exist parallel algorithms for *k-truss* decomposition, no known work exists for parallel *EquiTruss*. Consequently, in this study, we focus on the scalable algorithm design for the *EquiTruss* problem.

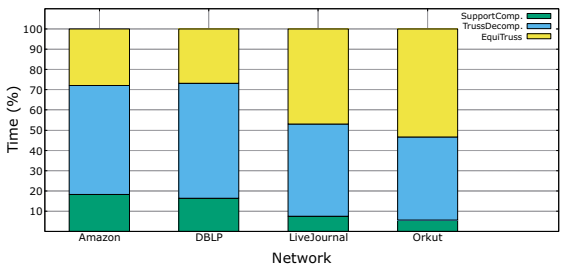


Figure 2: Compute kernel timing breakdown (in percentage) for our original *EquiTruss* based implementation. Computing *EquiTruss* (shown in yellow) is computationally as expensive as *k-truss* decomposition (blue) for large graphs.

We construct *EquiTruss* in parallel using a connected component (CC) algorithm [39], which we refer to as *Baseline EquiTruss*. Later, we incorporate cache-optimized storage and extraction of neighborhood information for better execution time. We refer to this approach as *C-Optimal EquiTruss*. Finally, we use a state-of-the-art sampling-based parallel connected component algorithm [43] to construct supernode in *EquiTruss*, which we call *Afforest EquiTruss* that outperforms the earlier two versions. We summarize the contributions of our work as follows:

- We novelly identify the k-triangle-induced index construction (*EquiTruss*) as a connected component problem on a graph in which edges are treated as entities instead of vertices. The connectivity among the edges is established through k-triangle connectivity.
- We design an OpenMP-based parallel *EquiTruss* approach for constructing the supergraph (index) without any loss of accuracy. To our knowledge, our novel algorithm is the first parallel algorithm for building such index structures to facilitate local community search.
- For our parallel *EquiTruss* implementations, we use state-of-the-art connected components approach, Afforest [43] and the prior state-of-the-art connected component approach, Shiloach-Vishkin (SV) [39]. We present a comparative analysis of the performance using these approaches.
- We construct the supergraph in a combination of parallel supernode and parallel superedge formulation resulting in up to 30× speedup in *NERSC Perlmutter* compute node compared to the sequential counterpart and up to 55× speedup compared to the *Baseline EquiTruss*.

2 BACKGROUND

Our parallel algorithm design is based on the serial *EquiTruss* [1] approach. We describe the notations used throughout this paper in Table 1. Then, in Table 2, we list all of our different implementations of *EquiTruss*. We present a few relevant definitions next, followed by a discussion of *EquiTruss* index construction strategy.

2.1 Preliminaries

The problem of *EquiTruss* index construction for an online community search considers the graph $G(V, E)$ to be simple, undirected, and unweighted with the number of vertices $|V|$ and the number of edges $|E|$. Below are some definitions relevant to the context of *EquiTruss*.

Definition 1 (Triangle [3]). *Given vertices u, v, and w s.t. (u, v), (v, w), and (u, w) are edges in E, a triangle Δ is a set of these three edges forming a cycle, i.e., $\Delta = \{(u, v), (v, w)\ (u, w)\} \subseteq E$.*

Definition 2 (Support). *The support of an edge, e, is the number of triangle(s) having e as their constituent edge. We denote the support of e as $|\Delta|_e$ or support(e).*

Definition 3 (k-truss). *A k-truss is a subgraph such that each edge has a support of at least $k-2$ within the subgraph. Formally, given a subgraph $G'(V', E') \subseteq G$, G' is a k-truss if $|\Delta|_e \geq k-2$ for all $e \in E'$. A maximal k-truss is a k-truss that is not a proper subgraph of another k-truss: formally, there exists no subgraph G'' such that G' and G'' are k-trusses and $G' \subsetneq G''$.*

Table 1: Notations and abbreviations used in this paper.

Notation	Description
G (V, E)	A simple undirected graph, G
$\mathbb{G}(\mathbb{V}, \mathbb{E})$	A summary/supergraph $\mathbb{G}$
$\mathbb{V}$	Set of supernode(s)
$\mathbb{E}$	An edge list of superedge(s)
τ	A dictionary storing trussness for edge $e \in E$
Π	A dict. for parent component ID of $e \in E$
$support(e), \lvert\Delta\rvert_e$	No. triangles having e as their constituent edge
k	Trussness of an edge
Φ_k	An edge set of same trussness k
ν	A supernode satisfying k-triangle connectivity
$\Delta_s \leftrightarrow \Delta_t$	Δ_s and Δ_t are triangle connected
$e \leftrightarrow e'$	Edges e and e' are triangle connected
$e \overset{k}{\leftrightarrow} e'$	Edges e and e' are k-triangle connected
CC	Connected Component
SV	Shiloach-Vishkin algorithm
LP	Label Propagation algorithm

Table 2: Naming of different algorithmic implementations.

Name	Description
Original *EquiTruss*	Our C++ implementation based on work [1]
Baseline *EquiTruss*	Our shared-memory-parallel *EquiTruss* based on Shiloach-Vishkin CC algorithm
C-Optimal *EquiTruss*	Our memory and computation optimized *EquiTruss* from its predecessor Baseline
Afforest *EquiTruss*	Our shared-memory-parallel *EquiTruss* based on Afforest [43] CC algorithm

Definition 4 (Trussness). *Given an edge $e \in E$, the trussness of an edge, $\tau(e)$ is defined to be the largest k such that there exists a k-truss in G that contains e. The trussness of a graph $\tau(G)$ is defined as* $\min_{e \in E} \tau(e)$.

Definition 5 (Triangle Adjacency). *Two triangles Δ_1 and Δ_2 are adjacent if they share a common edge, i.e., $\Delta_1 \cap \Delta_2 \neq \emptyset$.*

Definition 6 (Triangle Connectivity). *Given 2 triangles Δ_s and Δ_t within G, they are triangle connected, i.e., $\Delta_s \leftrightarrow \Delta_t$ if there exists a sequence of triangles, $\Delta_1, \ldots, \Delta_n$ in G with $n \geq 2$ such that $\Delta_1 = \Delta_s$, $\Delta_n = \Delta_t$, and for $1 \leq i < n$, $\Delta_i \cap \Delta_{i+1} \neq \emptyset$. If $e \in \Delta_s$ and $e' \in \Delta_t$, then e, e' are triangle connected or $e \leftrightarrow e'$. If all edges in the path between $e \leftrightarrow e'$ have trussness of k, then $e \overset{k}{\leftrightarrow} e'$.*

Definition 7 (k-truss Community). *For an integer $k \geq 3$, a subgraph $G' \subseteq G$ is a k-truss community if G' is a k-truss and for all $e, e' \in E'$, $e \overset{k}{\leftrightarrow} e'$.*

The goal of the *EquiTruss* algorithm is to create a *summary graph* $\mathbb{G}(\mathbb{V}, \mathbb{E})$ that will enable the fast construction of the k-truss communities associated with a given vertex.

Definition 8 (Supernode). *A supernode $\nu \in \mathbb{V}$ is a set of edges in E such that*

(1) *For all $e_1, e_2 \in \nu$, $\tau(e_1) = \tau(e_2)$,*
(2) *For all $e_1, e_2 \in \nu$, $e_1 \leftrightarrow e_2$ in the maximal k-truss of G,*
(3) *The supernode ν is maximal, i.e., there does not exist an edge $e \in G \setminus \nu$ such that $\tau(e) = \tau(\nu)$ and $e \leftrightarrow \nu$.*

Note that due to the maximality requirement of the supernodes, the set of supernodes $\mathbb{V}$ partitions E.

Definition 9 (Superedge). *Given supernodes $\nu_1, \nu_2 \in \mathbb{V}$, we say there exists a superedge between them if $\nu_1 \leftrightarrow \nu_2$ in the κ-truss where $\kappa = \min(\tau(\nu_1), \tau(\nu_2))$ and $\tau(\nu_1) \neq \tau(\nu_2)$.*

2.2 Index Construction Method

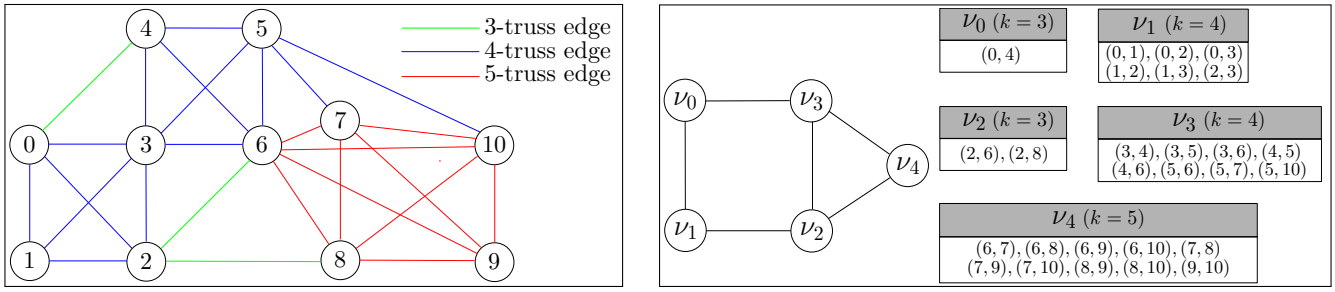
Here we discuss the index construction phase of the *EquiTruss* approach. The pseudocode is presented in Algorithm 1. The constructed index is the main data structure to retrieve all the communities of a query entity (vertex). Algorithm 1 receives a graph $G(V, E)$ and returns a supergraph with supernodes connected by superedges. The supernodes are groups of edges formed by following the condition of *k-triangle* connectivity. Aside from the input graph $G(V, E)$, a dictionary of edges, τ, with their corresponding k-trussness pre-computed by a k-truss decomposition technique is also taken as input. The initialization for all edges takes place between ln. $1 - 5$ in Algorithm 1 where a list (initially empty) of supernode IDs are maintained for superedge computation in a later phase of the algorithm. The entire edge set E is grouped into subsets based on their corresponding trussness, k (ln. $4 - 5$). All of those subsets of edges of different trussnesses are traversed in an iterative fashion (ln. 7) starting from $k_{min} \geq 3$ to k_{max}. For an edge set, Φ_k of certain trussness, k, edges are fetched (ln. 8), constructed to a supernode with supernode ID chronologically assigned, and added to the set of supernodes, $\mathbb{V}$ (ln. $9 - 11$). For an edge in a supernode, ν, a Breadth First Traversal (BFS) is performed to connect other edges belonging to that supernode following *k-triangle* connectivity (ln. $13 - 24$), i.e., all edges that are triangle connected with the current edge e having the same trussness, k of e are added to the supernode ν containing e. Ln. $20 - 23$ and ln. $26 - 29$ of Algorithm 1 describe this process. For an edge, $e\prime$ forming *k-triangle* connectivity with e and $\tau(e\prime) > k$, an entry is added to the list of the supernode(s) that e is connected to (ln. $31 - 32$). When $e\prime$'s list of the supernode is processed, a superedge entry is created connecting the supernode containing $e\prime$ to the supernode containing e (ln. $17 - 19$). An illustrative example of supernode, superedge, and summary graph structure is presented in Figure 3.

3 METHODOLOGY

3.1 Overview of the parallel algorithm

We break down our parallel index construction method into three different algorithmic snippets. Algorithm 2 discusses the design of creating the set of supernodes in parallel. Then in Algorithm 3, we discuss our parallel algorithm design to create the set of superedges $\mathbb{E}$. Finally, in Algorithm 4, we discuss our parallel approach to creating the supergraph $\mathbb{G}(\mathbb{V}, \mathbb{E})$.

Creating Supernode: In Algorithm 2, we demonstrate supernode creation using Shiloach-Vishkin (SV) [39] approach to parallel CC. While there are other approaches for parallel CC such as Label Propagation [33, 50] or *BFS*, we select SV [39] for running our edge-induced CC to form supernodes. SV has linear work efficiency as



(a) **(b)**

Figure 3: Illustration of how the summary graph is constructed by a sample graph example presented in [1]. Figure 3a depicts the group of edges with different *k-trussness* values with different colors. For instance, the blue marked edges are the members of the 4-truss subgraph but not the 5-truss subgraph. Following *Definition 8*, supernodes ν_0, ν_1, ν_2, ν_3, and ν_4 are constructed as shown in Figure 3b. Superedges are formed between those supernodes following *Definition 9*.

LP but is independent of the graph diameter (D). Variants [6, 40] of parallel CC using *BFS* have linear work efficiency, but parallelism is limited by the increasing number of connected components. It is important to note that, our edge-based CC with a composition of *k-triangle* connectivity fits nicely with the formulation of SV to establish supernodes. Furthermore, we experiment with a cache-optimized variant of SV and the state-of-the-art CC approach, Afforest [43] to demonstrate better run time both in sequential and parallel executions.

Algorithm 2 receives the original graph $G(V, E)$ and a dictionary of edges with their k-trussness, τ. The algorithm starts with initializing each edge to its own parent component (ln. $1-2$) and grouping edges based on their trussness (ln. $3-5$). Similar to Algorithm 1, all different subsets of edges based on their trussness are processed iteratively from $k_{min} \geq 3$ to k_{max} (ln. 6). All edges under certain truss groups are processed in parallel (ln. 10) to find the other edges forming triangles with edge $e(u, v)$ (ln. 11). We adopted the SV approach because it is highly amenable to parallelism and theoretically works well independent of graph topology [43]. The SV has two phases that alternate, *hooking* and *shortcut*. The hooking phase (ln. $12-20$) connects the edge e_1 (ln. 16) to the same parent component of e if the condition for $k-triangle$ connectivity (ln. 15) is satisfied. Similar action is done for edge e_2 (ln. $18-19$). In either case, the boolean variable *hooking* is set (ln. 17, 20) for a successful attempt to connect the edge to the parent component of e to run another round of *hooking* and *shortcut* phases. The shortcut phase (ln. $21-23$) is run on parallel across all edges in a Φ_k set with continuous linking up (ln. $22-23$) to the parent until all edges under a specific component are directly connected to the root. It is important to note that, both the *hooking* and the *shortcut* phases have a benign race condition that does not affect the correctness.

Creating Superedge: Algorithm 3 shows the design aspects of creating superedges in parallel. A list/vector of subsets of superedges is allocated (ln. 1) with a size equal to the number of available parallel threads. Each thread can add to its own subset of superedge(s) and thereby avoid race conditions. Both Algorithm 2 and Algorithm 3 are invoked consecutively upon the same Φ_k set. All edges of the Φ_k set are processed in parallel to find their triangle composing edges (e_1 and e_2), retrieve their trussness from τ and compute the minimum of trussness (ln. $3-8$). A superedge is established between the supernode containing current edge e with trussness k to the supernode containing the edge e_1, or e_2 having a minimum trussness $k_1 < k$, or $k_2 < k$. A thread creating a superedge adds it to its subset of superedge(s) (ln. 10, 12).

Creating Supergraph (Index): In Algorithm 4, we discuss the parallel merging of the thread local subset of superedge(s) constructed in Algorithm 3 to create the supergraph $\mathbb{G}(\mathbb{V}, \mathbb{E})$. A list *sm_graph* of the size of the total number of threads is allocated (ln. 1). Each thread has a thread-local vector of vectors of superedge {ID1, ID2}, *sm_graph_t* (ln. 6) where the outer vector has vector entries equal to the total number of threads, and *ID1* and *ID2* represent the supernode IDs. All of the superedges of each thread-local subset constructed in Algorithm 3 are hashed to the vector corresponding to a destination thread (ln. $7-11$). Each thread then combines (ln. $13-14$) all of its corresponding superedges annotated by all threads into *combined_sm_graph_t* allocated in ln. 2 and removes duplicates (ln. $15-16$). Finally, all threads merge their superedge(s) to the final supergraph (ln. 19).

3.2 Algorithm Complexity Analysis

Computing support/triangle has the best time complexity of $O(|E|^{1.5})$ [37]. Algorithm 1 compute supernode(s) using *BFS*. *BFS* has a time complexity of $O(|V| + |E|)$ for a graph $G(V, E)$ with number of vertices $|V|$ and number of edges $|E|$. However, for the edge-induced graph of *EquiTruss*, the constituent component of supernodes are edges of the original graph $G(V, E)$. Therefore, the time complexity is $O(|E| + |E|^{1.5})$ where $|E|^{1.5}$ is the maximum number of triangles possible for a graph with $|E|$ edges [17]. The time complexity of the CRCW (concurrent read, concurrent write) based Shiloach-Vishkin CC is $O(\frac{|E| \log |V|}{p} + \log |V|)$ for p parallel processing units [21]. In case of Algorithm 2 for the edge-induced graph of *EquiTruss*, the time complexity using p-thread is $O(\frac{|E|^{1.5} \log |E|}{p} + \log |E|)$. Most of the component identification work happens for Afforest proportional to $O(|V|)$ [43]. The edge-induced graph of *EquiTruss* would take $O(|E|)$ time and an additional $O(|E|^{1.5})$ time complexity to compute triangles. Therefore, the time complexity is $O(\frac{|E|^{1.5}+|E|}{p})$ for p parallel units. The space requirement for both groups of Algorithm 1 and Algorithm 2, 3, 4 is proportional to the number of edges in the original graph $G(V, E)$, i.e, $O(|E|)$ for storing relevant dictionary and data structure and an additional memory requirement for storing the summary graph $\mathbb{G}(\mathbb{V}, \mathbb{E})$ is $O(|\mathbb{E}|)$, therefore, $O(|E| + |\mathbb{E}|)$ in total.

Algorithm 1: Construct Index for *EquiTruss*

Data: A graph, $G(V, E)$ and a dictionary of edges, τ, with their k-truss values
Result: A supergraph, *EquiTruss*: $\mathbb{G}(\mathbb{V}, \mathbb{E})$

1 **for** $e(u, v) \in E$ **do**
2 $e.processed \leftarrow FALSE$
3 $e.list \leftarrow \emptyset$
4 **if** $(\tau(e) = k)$ **then**
5 $\Phi_k \leftarrow \Phi_k \cup e$
6 $spNdID \leftarrow 0$
7 **for** $k = k_{min}$ *to* k_{max} **do**
8 **while** $(\exists_e \in \Phi_k)$ **do**
9 $e.processed \leftarrow TRUE$
10 Create a supernode v, where $v.spNdID \leftarrow spNdID++$
11 $\mathbb{V} \leftarrow \mathbb{V} \cup \{v\}$
12 Initialize an empty queue, Q
13 $Q.enqueue(e)$
14 **while** $(Q \neq \emptyset)$ **do**
15 $e(u, v) \leftarrow Q.dequeue()$
16 $v \leftarrow v \cup \{e\}$
17 **for** $ID \in e.list$ **do**
18 Create a superedge (v, μ), where μ is an existing supernode with $\mu.spNdID = ID$
19 $\mathbb{E} \leftarrow \mathbb{E} \cup \{(v, \mu)\}$
20 **for** $w \in N(u) \cap N(v)$ **do**
21 **if** $\tau(u, w) \geq k$ *and* $\tau(v, w) \geq k$ **then**
22 ProcessEdge((u, w), spNdID, Q)
23 ProcessEdge((v, w), spNdID, Q)
24 $\Phi_k \leftarrow \Phi_k - \{e\}$
25 **Procedure** `ProcessEdge`$((u, v), spNdID, \&Q)$
26 **if** $(\tau(u, v) = k)$ **then**
27 **if** $(u, v).processed = FALSE$ **then**
28 $(u, v).processed \leftarrow TRUE$
29 $Q.enqueue((u, v))$
30 **else**
31 **if** $(spNdID \notin (u, v).list)$ **then**
32 $(u, v).list \leftarrow (u, v).list \cup \{spNdID\}$

3.3 Optimization of Compute Kernel

We break down our *Baseline EquiTruss* implementation into several compute kernels (illustration in section 4). The compute kernels are *Support*, *Initialization*, *SpNode*, *SpEdge*, *SmGraph*, and *SpNodeRemap*. We identify *SpNode* (described in Algorithm 2) as the most expensive kernel and aim to improve it by incorporating a few optimizations. We use the *CSRGraph* class from *GAP* Benchmark Suite [7] for efficient storage and operations. Instead of searching for the trussness (k) on the entire edge set in a dictionary/hashmap for an edge (ln. 4, 15, 18 in Algorithm 2), the search space is reduced to only the neighborhood list by *CSR* storage from *GAP*. The dictionary to store

Algorithm 2: Construct SuperNode(s) in parallel

Data: A graph $G(V, E)$ and a dictionary of edges with their k-truss values
Result: A dictionary of edges, Π, with each edge having their supernode ID/parent component ID assigned

/⋆ **Each edge initially forms its own component** ⋆/
1 **for** $e(u, v) \in E$ **do**
2 $\Pi(e) \leftarrow e$
/⋆ **Group edge set, E, into different subsets based on their trussness, e.g.,** $k = 3, 4, \ldots, k_{max}$ ⋆/
3 **for** $e(u, v) \in E$ **do**
4 **if** $(\tau(e) = k)$ **then**
5 $\Phi_k \leftarrow \Phi_k \cup e$
/⋆ **Run ShiloachVishkin (SV) connected component for each** Φ_k **set** ⋆/
6 **for** $k = k_{min}$ *to* k_{max} **do**
7 $hooking \leftarrow true$
8 **while** $(hooking)$ **do**
9 $hooking \leftarrow false$
 /⋆ **Hooking phase for SV** ⋆/
10 **for** $e(u, v) \in \Phi_k$ *in parallel* **do**
11 Compute a list of all common neighbors, **W**, that make triangle(s) with e
12 **for** $(w \in W)$ *in parallel* **do**
13 $e_1 \leftarrow (u, w) \in E$
14 $e_2 \leftarrow (v, w) \in E$
15 **if** $(\Pi(e) < \Pi(e_1)$ *and* $\Pi(e_1) = \Pi(\Pi(e_1))$ *and* $\tau(e) = \tau(e_1))$ **then**
16 $\Pi(\Pi(e_1)) \leftarrow \Pi(e)$
17 $hooking \leftarrow true$
18 **if** $(\Pi(e) < \Pi(e_2)$ *and* $\Pi(e_2) = \Pi(\Pi(e_2))$ *and* $\tau(e) = \tau(e_2))$ **then**
19 $\Pi(\Pi(e_2)) \leftarrow \Pi(e)$
20 $hooking \leftarrow true$
 /⋆ **Shortcut phase for SV** ⋆/
21 **for** $e \in \Phi(k)$ *in parallel* **do**
22 **while** $(\Pi(\Pi(e)) \neq \Pi(e))$ **do**
23 $\Pi(e) \leftarrow \Pi(\Pi(e))$

and retrieve parent component/supernode ID for the entire edge set has been replaced from a hashmap to a contiguous memory buffer.

The *Shiloach-Vishkin* connected component (CC) design from *GAP* has been adapted to deal with our special situation where we treat an edge to be an entity in the connected component instead of the usual vertex in SV connected component. The SV adaptation skip further processing if $\Pi(e) = \Pi(e_1)$ (ln. 15 or 18 in Algorithm 2). This resulted in an optimal design of *SpNode* construction by the Shiloach-Vishkin CC algorithm. We name it *SpNode C-Optimal*. SV algorithm for CC was improved by Afforest [43] by modifying the convergence logic to be applied separately to different subgraphs. It utilizes component approximation by subgraph sampling to reduce the number of edge processing while obtaining the exact solution.

Algorithm 3: Create SuperEdge(s) in parallel

Data: $\Phi(k)$ set of current k from Algo. 2
Result: A vector/list of thread local superedge subsets
Allocate vector<set<compID1, compID2≫**sp_edges**, a vector of size = number of threads
for $e(u,v) \in \Phi(k)$ *in parallel* **do**
  /⋆ **W is the list of neighbor(s) forming triangle(s) with** e ⋆/
  for $w \in W$ *in parallel* **do**
    $tid \leftarrow get\_thread\_id$
    $e_1 \leftarrow (u,w) \in E$
    $e_2 \leftarrow (v,w) \in E$
    $k \leftarrow \tau(e), k_1 \leftarrow \tau(e_1), k_2 \leftarrow \tau(e_2)$
    $lowest\_k \leftarrow min(k, k_1, k_2)$
    /⋆ **Create superedge downward,** $k > k_1$ ⋆/
    if $k > lowest\_k$ *and* $lowest\_k = k_1$ **then**
      $sp\_edges[tid].insert(\{\Pi(e_1), \Pi(e)\})$
    /⋆ **Create superedge downward,** $k > k_2$ ⋆/
    if $k > lowest\_k$ *and* $lowest\_k = k_2$ **then**
      $sp\_edges[tid].insert(\{\Pi(e_2), \Pi(e)\})$

The two phases of the original SV algorithm (*hooking* and *shortcut*) are modified by the corresponding *link* and *compress* phases to avoid overriding work by concurrent parallel units. Similar to the *SpNode C-Optimal*, we adapted the *Afforest* implementation from GAP to run our special case of the connected component algorithm.

4 PERFORMANCE EVALUATION

4.1 Experimental Settings

We implemented our algorithm using C++ programming language, OpenMP frameworks for multi-threading, and GNU g++ compiler for building the code. We used *Perlmutter* CPU compute node from National Energy Research Scientific Computing Center (NERSC). The CPU node consists of 2 AMD EPYC 7763 CPUs, 64 cores per CPU with base frequency 2.45GHz, 512 GB of DDR4 memory, and 204.8 GB/s memory bandwidth per CPU. The undirected network data sets listed in Table 3 are collected from SNAP [26].

4.2 Effect of Compute Kernel Optimization

Figure 4 illustrates the time percentage breakdown of operational kernels for different networks for the parallel *EquiTruss*. It is evident that constructing supernodes (*SpNode* in Figure 4) is the most expensive part of the overall algorithm. This kernel takes as much as 79% for the *YouTube* network and 87% for the *Orkut* network, respectively, of the overall index construction time. The second most expensive kernel is the superedge creation as described in Algorithm 3 ranging from as little as 6% for the *DBLP* network to 10% for the *YouTube* network of the overall time.

Figure 5 illustrates the performance improvement in terms of speedup for Algorithm 2 from the *SpNode Baseline* to *SpNode C-Optimal* to finally *SpNode Afforest* because of our optimizations. We observe the supernode construction time significantly reduces from 8655 seconds in *Baseline* to 2093 seconds in *SpNode Aff.* and 4371 seconds in *SpNode C-Opt.* resulting in 4.13× and 1.98× speedup,

Algorithm 4: Construct SuperGraph in parallel

Data: A vector/list of thread local superedge subsets **sp_edges** from Algo. 3
Result: A complete list of superedges from merging thread local superedge subsets
Allocate, a list **sm_graph** of size = num_threads
Allocate vector<vector<{ID1, ID2} ≫**combined_sm_graph_t**(num_threads)
Allocate a contiguous buffer, **final_sp_graph**, of type <ID1, ID2>and size = total_num_sp_edges
Inside each thread t in parallel
{
  Allocate thread-local vector<vector<{ID1, ID2} ≫ **sm_graph_t**(num_threads)
  for *each superedge* $\in sp\_edges[t]$ **do**
    $ID1 \leftarrow superedge.ID1$
    $ID2 \leftarrow superedge.ID2$
    $dest\_t \leftarrow (hash(ID1, ID2))\%num\_threads$
    $sm\_graph\_t[dest\_t] \leftarrow superedge$
  $sm\_graph[t] \leftarrow sm\_graph\_t$
  for $sm\_t \in sm\_graph$ **do**
    Copy all $sm\_t[t]$ into $combined\_sm\_graph\_t[t]$
  sort $combined\_sm\_graph\_t[t]$
  remove duplicates from $combined\_sm\_graph\_t[t]$
  /⋆ **Parallel reduction** ⋆/
  $total\_num\_sp\_edges$ += $combined\_sm\_graph\_t[t].size()$
}
Merge combined_sm_graph_t[t] into final_sp_graph in parallel

Table 3: Network dataset for our experiments. We used several social and information networks.

Network	# Vertices	# Edges
Amazon	334863	925872
DBLP	317080	1049866
YouTube	1134890	2987624
LiveJournal	3997962	34681189
Orkut	3072441	117185083
Friendster	65608366	1806067135

respectively, for the *Orkut* network in a single-thread execution. The optimization of Afforest for CC over SV delivered significantly better performance as observed from the blue bar in Figure 5. Similarly, for the *LiveJournal* network, the supernode construction time reduces to 453 seconds in *SpNode Aff.* and to 696 seconds in *SpNode C-Opt.* from the *Baseline SpNode* computation time of 1393 seconds resulting in 3.07× and 2× speedup, respectively.

4.3 Performance Analysis

Comparison with State-of-the-art: We obtained the original Java implementation of the sequential EquiTruss by Akbas et al. [1] to perform a horizontal comparison with our implementations.

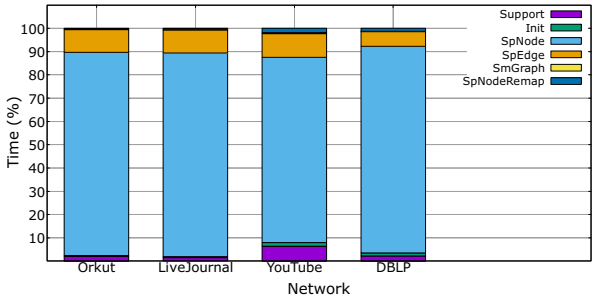


Figure 4: The operational kernels for the Baseline implementation of the parallel *EquiTruss* algorithm. The percentage (%) breakdown of single-thread run time for 4 different networks is illustrated. It is evident that the *SpNode* kernel is the major portion $(79-89)\%$ of the execution time.

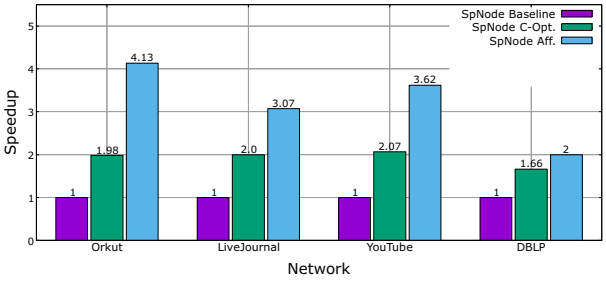


Figure 5: Runtime improvement in single-threaded execution in terms of speedup of the major operational kernel of EquiTruss using cache-optimized data structure and Afforest connected component algorithm.

We present the run time comparison in Table 4. For the *LiveJournal* network, the index construction phase (SpNd, SpEdge, and SmGraph) of the serial java code is 3.3× faster than our Baseline, 1.8× faster than C-Opt. EquiTruss, and 1.3× faster than Afforest EquiTruss in sequential settings. Our parallel (128-thread) versions are 11.55× faster (Baseline), 20.59× faster (C-Opt. EquiTruss), and 29.56× faster (Afforest EquiTruss), respectively, than the sequential Java implementation. For larger networks (e.g., *Orkut* with $117M$ edges), the sequential Java code runs out of memory while all of our implementations in Table 4 can process billion-size graphs (e.g. *com-Frienster*). For measuring the accuracy of the constructed supernodes or supergraphs, we compared the total number and constituent components (constituent edges) of supernodes and superedges of the sequential Java code by Akbas et al. [1] against our implementations in both sequential and parallel settings. The results are identical in all cases. *EquiTruss* or parallel *EquiTruss* depends on deterministic sub-kernels: k-truss connected components. Since there is no approximation involved at any stage, the formulation of the k-triangle connectivity ensures the exactness of the connected components (supernodes). Therefore, we only report the number of supernodes and superedges in Table 5 and do not dedicate additional space to report accuracy which is 100% for all cases.

Speedup: We list the number of supernodes and superedges in the summary graph along with speedup gain for our parallel *Baseline EquiTruss*, an optimized version over Baseline *C-Opt. EquiTruss*, and *Afforest EquiTruss* in Table 5. The speedup gain for the *Baseline* is 13.92×, 27.31×, and 29.63× for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. The *C-Opt. EquiTruss* exhibits 8.82×, 22.25×, and 22.61× speedup over the sequential (single-threaded) counterpart for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. Finally, using the *Aff. EquiTruss*, we observe 7.06×, 19.55×, and 18.27× speedup over the sequential (single-threaded) counterpart for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. In all of those cases, the maximum speedup is observed for using the maximum number of threads (i.e., physical cores) in a compute node which is 128. The *Baseline* version delivers better speedup as this is the less efficient one performing more computation than the other 2 versions. It is important to note that the *Baseline* version still has a significantly lower run-time than our C++ implementation of *EquiTruss* based on Akbas et al. [1] (*Original EquiTruss* in Table 2). If we just consider the speedup gain from the sequential *Baseline* to our final optimized version (*Aff. EquiTruss*) with 128 threads, it would be 16.10×, 47.8×, and 55.24× for *YouTube*, *LiveJournal*, and *Orkut* networks, respectively. These are significant speedup gains using our parallel implementation over the sequential versions.

Strong Scalability: Figure 6 illustrates the strong scalability plots for the increasing number of threads from 1 to 128. There are 3 different curves under each sub-Figure representing the scalability for 3 different design phases (*Baseline EquiTruss*, *C-Opt. EquiTruss*, and *Aff. EquiTruss*) of the parallel *EquiTruss* problem. The execution time reduces from 3283 seconds to 179 seconds with *Aff. EquiTruss* by using 128 threads shown in Figure 6a for the *Orkut* network. Similarly, the execution time scalabilities are shown for *LiveJournal* network in Figure 6b for the 3 different design phases of the *EquiTruss* problem. The run time reduces from 895 seconds using a single thread to 40 seconds using 128 threads for the *C-Opt. EquiTruss* (blue curve) in Figure 6b. And finally, the execution time reduces from 36.56 seconds to 2.62 seconds for the *YouTube* network as shown in Figure 6c using the *Baseline* version of *EquiTruss*. Figure 7 demonstrates strong scalability for the *SpNode* construction run-time for the billion-size *Friendster* network. Here we only show the *SpNode* construction cost due to the maximum 12 hours of node occupancy limit in a regular compute node in NERSC *Perlmutter* supercomputer. In Figure 7, the *SpNode* computation time using *C-Opt. EquiTruss* cannot be shown for single-thread and 2-thread due to the occupancy hour limit. In Figure 8, we show the run-time reduction for the 3 major kernels as described in Algorithm 2, 3, and 4 for our 3 different versions of the parallel *EquiTruss* using 1, 8, 32, and 128 threads respectively. The *SpNode* kernel (light purple) dominates over the other 2 kernels *SpEdge* (light green) and *SmGraph* (light blue) in a single thread. However, it starts to decrease significantly along with the other 2 kernels as we increase the number of parallel threads and becomes really small in 128 threads for both the example networks (Figure 8a and 8b).

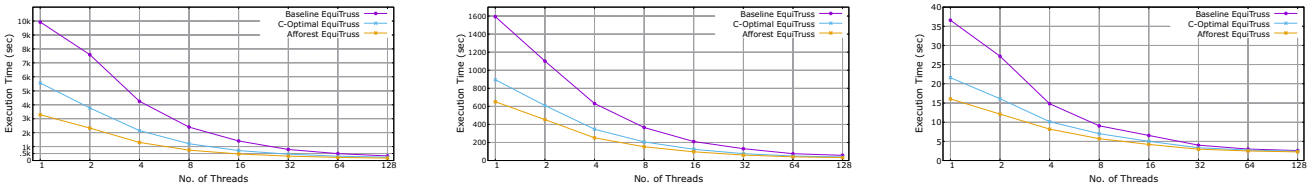
Parallel Efficiency: Figure 9 illustrates the parallel efficiency for 3 different networks using histogram plots. The parallel efficiency (ε) of an algorithm compares the parallel run time to the sequential run time assuming perfect scalability [4]. To formulate, parallel efficiency $\varepsilon = \frac{T_{seq}}{pT(p)}$, where p is the number of parallel units, $T(p)$ is the time with p parallel units, and T_{seq} is the sequential run time. In each plot, there are 3 histogram bars grouped together

Table 4: Comparing the combined run time of the major computational phases (SpNd, SpEdge, and SmGraph) for *Index* construction. The comparison is performed in single-threaded settings between our implementations and the original Java implementation by Akbas et al. [1].

Network	Baseline (sec)	C-Opt. EquiTruss (sec)	Aff. EquiTruss (sec)	Akbas et al. [1] (sec)
Amazon	6.77	3.96	3.24	1.46
DBLP	10.92	7.37	6.57	2.33
LiveJournal	1549	851	608	467
Orkut	9631	5268	2990	MLE

Table 5: The number of supernodes and superedges in summary graphs for different networks. Comparison of the slowest execution time (1-thread) to faster execution time (128-thread) in seconds and the corresponding speedup for different versions of our parallel *EquiTruss* implementation.

Network	No. of Sp nodes	No. of Sp edges	Base. Eq.			C-Opt. Eq.			Aff. Eq.		
			1-t(s)	128-t(s)	Speedup	1-t(s)	128-t(s)	Speedup	1-t(s)	128-t(s)	Speedup
Amazn.	115060	103513	7.26	0.52	13.86	4.45	0.46	9.7	3.74	0.40	9.16
DBLP	126904	105409	11.52	0.62	18.53	7.96	0.51	15.52	7.16	0.49	14.46
YouTb.	400408	940550	36.56	2.62	13.92	21.60	2.44	8.82	16.07	2.27	7.06
LvJrnl.	4765102	13405280	1593.43	58.34	27.31	895.03	40.21	22.25	651.69	33.33	19.55
Orkut	17227001	76631446	9924.57	334.89	29.63	5561.59	245.97	22.61	3283.14	179.64	18.27



(a) Orkut **(b) LiveJournal** **(c) YouTube**

Figure 6: Illustrating runtime reduction and scalability using 3 different design phases of the parallel *EquiTruss* for 3 different networks. For instance, the execution times reduce from 9924, 5561, and 3283 seconds to 334, 245, and 179 seconds, respectively, for baseline *EquiTruss*, C-Opt. *EquiTruss*, and Afforest *EquiTruss* using 128 threads for the *Orkut* network (Figure 6a).

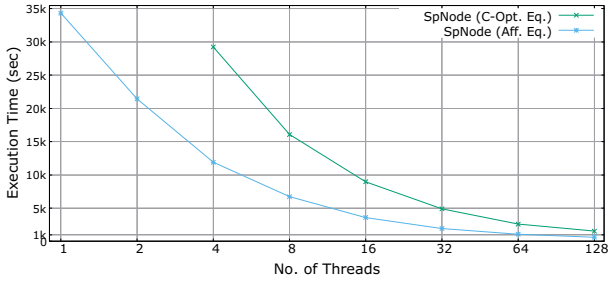
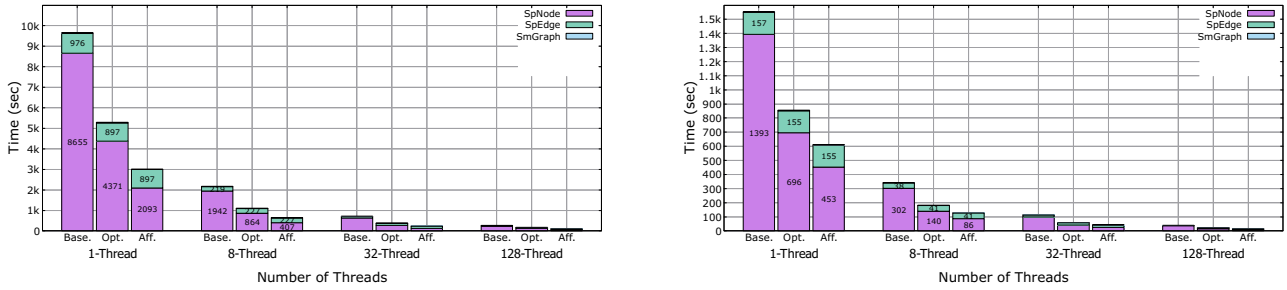


Figure 7: Execution time for the *SpNode* kernel for billion-size *Friendster* network using *C-Opt. EquiTruss* and *Aff. EquiTruss*. For the *Aff. EuiTruss*, single thread run-time of 34332 seconds reduces to only 612 seconds using 128 threads.

representing our three versions of *EquiTruss* implementation. In Figure 9a, we observe 70% parallel efficiency for *Aff. EquiTruss* and 73% parallel efficiency for *C-Opt. EquiTruss* using 2 threads for the *Orkut* network. For the same network, those corresponding parallel efficiencies become 22% and 27%, respectively, using 64 threads, 14% and 17%, respectively, using 128 threads. The utilization of 128 threads in our diverse EquiTruss versions showcases the potential for even greater scalability when employing a shared-memory system with a higher number of available threads.

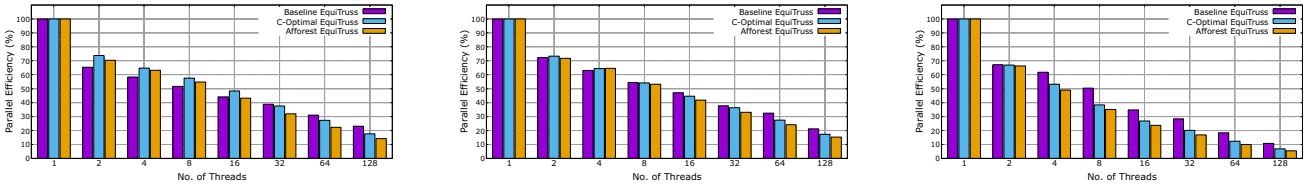
5 OTHER RELATED WORK

A few early studies [15, 30] on clique-based overlapping community search are based on the clique percolation method where after finding k-cliques, all adjacent k-cliques (sharing $k-1$ nodes) are merged. Zhang et al. [51] propose clique percolation clustering to detect overlapping communities in PPI networks. Kumpula et al. [25] propose a clique-based approach for both weighted and unweighted graphs. All these strategies are too restrictive on the clique size. Maity et al. [28] extend the work [30] for the complete graph and inherit the limitation as well. Community search strategies [38, 48] depending on maximal clique suffer from computational intractability. *K-core* based local community search techniques [5, 49] optimize the metrics such as density, modularity, or conductance but fail to avoid non-relevant vertices, and cannot detect overlapping membership communities. Online community search based on a community model named α-adjacency-γ-quasi-k-clique is proposed by Cui et al. [13] where the formulation has been



(a) Orkut (b) LiveJournal

Figure 8: Timing breakdown of the major compute kernels (*SpNode, SpEdge, SmGraph*). The reduction in execution time for those kernels is presented for different numbers of threads (1, 8, 32, and 128). The *SpNode* time reduces from 2093 seconds in 1 thread to 407 seconds in 8 threads, then to 127 seconds in 32 threads, and finally to 60 seconds in 128 threads for the *Afforest EquiTruss* for *Orkut* network (Fig. 8a). Similarly, the *SpNode* time reduces from 696 seconds in 1 thread to 140 seconds in 8 threads, then to 42 seconds in 32 threads, and finally to 16 seconds in 128 threads for the *C-Opt. EquiTruss* for *LiveJournal* network (Fig. 8b).



(a) Orkut (b) LiveJournal (c) YouTube

Figure 9: Illustrating parallel efficiency using 3 different designs of the parallel *EquiTruss* for 3 different networks. For instance, the parallel efficiencies are 38.89%, 37.66%, and 32%, respectively, for baseline *EquiTruss*, C-Opt. *EquiTruss*, and Afforest *EquiTruss* for *Orkut* network while using 32 threads (Figure 9a).

found to be NP-hard [23] and the approximation proposed [13] has non-promising solution quality. The truss-based community search called *TCP-Index* [23] maintains trussness information into groups of tree-structured indexes called maximum spanning tree (MST). The limitations of *TCP-Index* are, the constituent edges of a graph G have to be maintained redundantly in multiple MSTs and during the community search phase, a costly truss reconstruction phase needs to be performed. The work in [1] avoids such limitations by maintaining an edge in a supernode structure, but is limited in scalability for the algorithm's sequential nature.

6 CONCLUSION

Designing parallel algorithms for local community discovery is not as well-explored as global community discovery. There are existing studies that discuss the problem of constructing community subgraphs using higher-order graph primitives: *cliques*, *quasi clique*, or *k-core*. An alternative approach, *k-truss* decomposition, addresses the issues of computational intractability or lack of cohesiveness which are inherent in those other approaches. Fueled by the promising aspect of cohesiveness in k-triangle-connected subgraph structures, we combine it with the state-of-the-art parallel connected component approaches for our formulation of parallel *EquiTruss* in shared-memory settings. Our parallel *EquiTruss* algorithm scales well to large systems and on large datasets. The algorithm demonstrates up to 55× speedup while processing billion-size graphs on 128 physical cores of NERSC Perlmutter compute node with 512GB of memory.

ACKNOWLEDGMENTS

This work has been partially supported by National Science Foundation (NSF) under Award Number 2323533 and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Award Number DE-AC02-05CH11231. We express our gratitude to Professor Esra Akbas for providing us with *EquiTruss* Java implementation.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-Based Community Search: A Truss-Equivalence Based Indexing Approach. *Proc. VLDB Endow.* 10, 11 (aug 2017), 1298–1309. https://doi.org/10.14778/3137628.3137640

[2] Mohammad Almasri, Omer Anjum, Carl Pearson, Zaid Qureshi, Vikram S. Mailthody, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2019. Update on k-truss Decomposition on GPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2019.8916285

[3] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. 2019. Fast parallel algorithms for counting and listing triangles in big graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14, 1 (2019), 1–34.

[4] Seung-Hee Bae, Daniel Halperin, Jevin West, Martin Rosvall, and Bill Howe. 2013. Scalable Flow-Based Community Detection for Large-Scale Network Analysis. In *2013 IEEE 13th International Conference on Data Mining Workshops*. 303–310. https://doi.org/10.1109/ICDMW.2013.138

[5] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. 2015. Efficient and effective community search. *Data Mining and Knowledge Discovery* 29, 5 (01 Sep 2015), 1406–1433. https://doi.org/10.1007/s10618-015-0422-1

[6] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10. https://doi.org/10.1109/SC.2012.50

[7] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[8] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008

[9] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM* 16, 9 (sep 1973), 575–577. https://doi.org/10.1145/362342.362367

[10] Pei-Ling Chen, Chung-Kuang Chou, and Ming-Syan Chen. 2014. Distributed algorithms for k-truss decomposition. In *2014 IEEE International Conference on Big Data (Big Data)*. 471–480. https://doi.org/10.1109/BigData.2014.7004264

[11] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16, 3.1 (2008).

[12] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online Search of Overlapping Communities. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/2463676.2463722

[13] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local Search of Communities in Large Graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 991–1002. https://doi.org/10.1145/2588555.2612179

[14] Timothy A. Davis. 2018. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and K-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2018.8547538

[15] Imre Derényi, Gergely Palla, and Tamás Vicsek. 2005. Clique Percolation in Random Networks. *Phys. Rev. Lett.* 94 (Apr 2005), 160202. Issue 16. https://doi.org/10.1103/PhysRevLett.94.160202

[16] Zhihui Du, Joseph Patchett, Oliver Alvarado Rodriguez, and David A. Bader. [n. d.]. In *The 9th Annual Chapel Implementers and Users Workshop (CHIUW)*.

[17] Mathematics Stack Exchange. [n. d.]. Number of triangles in a graph based on number of edges. https://math.stackexchange.com/questions/823481/number-of-triangles-in-a-graph-based-on-number-of-edges

[18] Md Abdul Motaleb Faysal and Shaikh Arifuzzaman. 2019. Distributed community detection in large networks using an information-theoretic approach. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 4773–4782.

[19] Md Abdul M Faysal, Shaikh Arifuzzaman, Cy Chan, Maximilian Bremer, Doru Popovici, and John Shalf. 2021. HyPC-Map: A Hybrid Parallel Community Detection Algorithm Using Information-Theoretic Approach. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[20] Wafaa M. A. Habib, Hoda M. O. Mokhtar, and Mohamed E. El-Sharkawi. 2022. Discovering top-weighted k-truss communities in large graphs. *Journal of Big Data* 9, 1 (03 Apr 2022), 36. https://doi.org/10.1186/s40537-022-00588-1

[21] Yujie Han and Robert A. Wagner. 1990. An Efficient and Fast Parallel-Connected Component Algorithm. *J. ACM* 37, 3 (jul 1990), 626–642. https://doi.org/10.1145/79147.214077

[22] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying K-Truss Community in Large and Dynamic Graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1311–1322. https://doi.org/10.1145/2588555.2610495

[23] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying K-Truss Community in Large and Dynamic Graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1311–1322. https://doi.org/10.1145/2588555.2610495

[24] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-truss decomposition on multicore systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091052

[25] Jussi M. Kumpula, Mikko Kivelä, Kimmo Kaski, and Jari Saramäki. 2008. Sequential algorithm for fast clique percolation. *Physical Review E* 78, 2 (aug 2008). https://doi.org/10.1103/physreve.78.026109

[26] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[27] R. Duncan Luce and Albert D. Perry. 1949. A method of matrix analysis of group structure. *Psychometrika* 14, 2 (01 Jun 1949), 95–116. https://doi.org/10.1007/BF02289146

[28] Suman Maity and Santanu Rath. 2014. Extended Clique percolation method to detect overlapping community structure. *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)* (2014), 31–37.

[29] M. E. J. Newman. 2013. Spectral methods for community detection and graph partitioning. *Physical Review E* 88, 4 (Oct 2013). https://doi.org/10.1103/physreve.88.042822

[30] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *Nature* 435, 7043 (01 Jun 2005), 814–818. https://doi.org/10.1038/nature03607

[31] Roger Pearce and Geoffrey Sanders. 2018. K-truss decomposition for Scale-Free Graphs at Scale in Distributed Memory. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2018.8547572

[32] Martin Rosvall and Carl T Bergstrom. 2008. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences* 105, 4 (2008), 1118–1123. https://doi.org/10.1073/pnas.0706851105 arXiv:https://www.pnas.org/content/105/4/1118.full.pdf

[33] Piyush Sao, Oded Green, Chirag Jain, and Richard Vuduc. 2016. A Self-Correcting Connected Components Algorithm. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale* (Kyoto, Japan) *(FTXS '16)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/2909428.2909435

[34] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental K-Core Decomposition: Algorithms and Evaluation. *The VLDB Journal* 25, 3 (jun 2016), 425–447. https://doi.org/10.1007/s00778-016-0423-8

[35] Naw Safrin Sattar and Shaikh Arifuzzaman. 2019. Overcoming mpi communication overhead for distributed community detection. In *Software Challenges to Exascale Computing: Second Workshop, SCEC 2018, Delhi, India, December 13-14, 2018, Proceedings 2*. Springer Singapore, 77–90.

[36] Naw Safrin Sattar and Shaikh Arifuzzaman. 2022. Scalable distributed Louvain algorithm for community detection in large graphs. *The Journal of Supercomputing* 78, 7 (2022), 10275–10309.

[37] Thomas Schank and Dorothea Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms*, Sotiris E. Nikoletseas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 606–609.

[38] Hua-Wei Shen, Xue-Qi Cheng, and Jia-Feng Guo. 2009. Quantifying and identifying the overlapping community structure in networks. *Journal of Statistical Mechanics: Theory and Experiment* 2009, 07 (jul 2009), P07042. https://doi.org/10.1088/1742-5468/2009/07/p07042

[39] Yossi Shiloach and Uzi Vishkin. 1982. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms* 3 (1982), 57–67.

[40] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and Coloring-Based Parallel Algorithms for Strongly Connected Components and Related Problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 550–559. https://doi.org/10.1109/IPDPS.2014.64

[41] Shaden Smith, Xing Liu, Nesreen K. Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. 2017. Truss decomposition on shared-memory parallel systems. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2017.8091049

[42] Mauro Sozio and Aristides Gionis. 2010. The Community-Search Problem and How to Plan a Successful Cocktail Party. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, DC, USA) *(KDD '10)*. Association for Computing Machinery, New York, NY, USA, 939–948. https://doi.org/10.1145/1835804.1835923

[43] Michael Sutton, Tal Ben-Nun, and Amnon Barak. 2018. Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 12–21. https://doi.org/10.1109/IPDPS.2018.00012

[44] Charalampos Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria Tsiarli. 2013. Denser than the Densest Subgraph: Extracting Optimal Quasi-Cliques with Quality Guarantees. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Chicago, Illinois, USA) *(KDD '13)*. Association for Computing Machinery, New York, NY, USA, 104–112. https://doi.org/10.1145/2487575.2487645

[45] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (may 2012), 812–823. https://doi.org/10.14778/2311906.2311909

[46] Runze Wang, Linchen Yu, Qinggang Wang, Jie Xin, and Long Zheng. 2021. Productive High-Performance k-Truss Decomposition on GPU Using Linear Algebra. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC49654.2021.9622792

[47] Jian Wu, Alison Goshulak, Venkatesh Srinivasan, and Alex Thomo. 2018. K-Truss Decomposition of Large Networks on a Single Consumer-Grade Machine. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. 873–880. https://doi.org/10.1109/ASONAM.2018.8508642

[48] Peng Wu and Li Pan. 2014. Detecting highly overlapping community structure based on Maximal Clique Networks. In *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*. 196–199. https://doi.org/10.1109/ASONAM.2014.6921582

[49] Yubao Wu, Ruoming Jin, Jing Li, and Xiang Zhang. 2015. Robust Local Community Detection: On Free Rider Effect and Its Elimination. *Proc. VLDB Endow.* 8, 7 (feb 2015), 798–809. https://doi.org/10.14778/2752939.2752948

[50] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. *Proc. VLDB Endow.* 7, 14 (oct 2014), 1821–1832. https://doi.org/10.14778/2733085.2733089

[51] Shihua Zhang, Xuemei Ning, and Xiang-Sun Zhang. 2006. Identification of functional modules in a PPI network by clique percolation clustering. *Comput Biol Chem* 30, 6 (Nov. 2006), 445–451.