Local Algorithms for Hierarchical Dense Subgraph Discovery

Ahmet Erdem Sarıyüce
University at Buffalo
Buffalo, NY
erdem@buffalo.edu

C. Seshadhri
University of California
Santa Cruz, CA
sesh@ucsc.edu

Ali Pinar
Sandia National Laboratories
Livermore, CA
apinar@sandia.gov

ABSTRACT

Finding the dense regions of a graph and relations among them is a fundamental problem in network analysis. Core and truss decompositions reveal dense subgraphs with hierarchical relations. The incremental nature of algorithms for computing these decompositions and the need for global information at each step of the algorithm hinders scalable parallelization and approximations since the densest regions are not revealed until the end. In a previous work, Lu et al. proposed to iteratively compute the h-indices of neighbor vertex degrees to obtain the core numbers and prove that the convergence is obtained after a finite number of iterations. This work generalizes the iterative h-index computation for truss decomposition as well as nucleus decomposition which leverages higher-order structures to generalize core and truss decompositions. In addition, we prove convergence bounds on the number of iterations. We present a framework of local algorithms to obtain the core, truss, and nucleus decompositions. Our algorithms are local, parallel, offer high scalability, and enable approximations to explore time and quality trade-offs. Our shared-memory implementation verifies the efficiency, scalability, and effectiveness of our local algorithms on real-world networks.

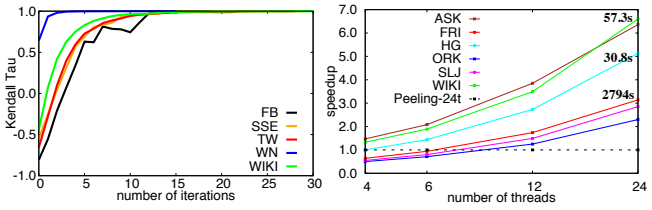
PVLDB Reference Format:
Ahmet Erdem Sarıyüce, C. Seshadhri, and Ali Pinar. Local Algorithms for Hierarchical Dense Subgraph Discovery. *PVLDB*, 12(1): 43-56, 2018.
DOI: https://doi.org/10.14778/3275536.3275540

1. INTRODUCTION

A characteristic feature of the real-world graphs is sparsity at the global level yet density in the local neighborhoods [15]. Dense subgraphs are indicators for functional units or unusual behaviors. They have been adopted in various applications, such as detecting DNA motifs in biological networks [12], identifying the news stories from microblogging streams in real-time [2], finding price value motifs in financial networks [10], and locating spam link farms in web [24, 13, 9]. Dense regions are also used to improve efficiency of compute-heavy tasks like distance query computation [21] and materialized per-user view creation [14].

Detecting dense structures in various granularities and finding the hierarchical relations among them is a fundamental problem in graph mining. For instance, in a citation network, the hierarchical relations of dense parts in various granularities can reveal how new research areas are initiated or which research subjects became popular in time [38]. k-core [39, 27] and k-truss decompositions [34, 6, 44, 48] are effective ways to find many dense regions in a graph and construct a hierarchy among them. k-core is based on the vertices and their degrees, whereas k-truss relies on the edges and their triangle counts.

Higher-order structures, also known as motifs or graphlets, have been used to find dense regions that cannot be detected with edge-centric methods [4, 42]. Computing the frequency and distribution of triangles and other small motifs is a simple yet effective approach used in data analysis [19, 30, 1, 33]. Nucleus decomposition is a framework of decompositions that is able to use higher-order structures to find dense subgraphs with hierarchical relations [37, 38]. It generalizes the k-core and k-truss approaches and finds higher-quality dense subgraphs with more detailed hierarchies. However, existing algorithms in the nucleus decomposition framework require global graph information, which becomes a performance bottleneck for massive networks. They are also not amenable for parallelization or approximation due to their interdependent incremental nature. We introduce a framework of algorithms for nucleus decomposition that uses **only local information**. Our algorithms provide faster and ap-



(a) Convergence rates (b) Scalability performance

Figure 1: *On the left, we present the convergence rates for k-truss decomposition on five graphs. Kendall-Tau similarity score compares the obtained and the exact decompositions; becomes 1.0 when they are the same. Our local algorithms compute the almost-exact decompositions in around 10 iterations. On the right, we show the runtime performances w.r.t. partially parallel peeling algorithms. On average, k-truss computations are 4.8x faster when we switch from 4 threads to 24 threads.*

Proceedings of the VLDB Endowment, Vol. 12, No. 1
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3275536.3275540

proximate solutions and their local nature enables query-driven processing of vertices/edges.

1.1 Problem and challenges

The standard method to compute a k-core decomposition is a sequential algorithm, known as the peeling process. To find a k-core, all vertices with degree less than k are removed repeatedly until no such vertex remains. This process is repeated after incrementing k until no vertices remain. Batagelj and Zaversnik introduced a bucket-based $\mathcal{O}(|E|)$ algorithm for this process [3]. It keeps track of the vertex with the minimum degree at each step, thus requires global information about the graph at any time. k-truss decomposition has a similar peeling process with $\mathcal{O}(|\triangle|)$ complexity [6]. To find a k-truss, all edges with less than k triangles are removed repeatedly and at each step, algorithm keeps track of the edge with the minimum triangle count, which requires information from all around the graph. Nucleus decomposition [37] also facilitates the peeling process on the given higher-order structures. **The computational bottleneck in the peeling process is the need for the global graph information.** This results in inherently sequential processing. Parallelizing the peeling process in a scalable way is challenging since each step depends on the results of the previous step. Parallelizing each step in itself is also infeasible since synchronizations are needed to decrease the degrees of the vertices that are adjacent to multiple vertices being processed in that step.

Iterative h-index computation: Lu et al. introduced an alternative formulation for k-core decomposition [26]. They proposed iteratively computing h-indices on the vertex degrees to find the core numbers of vertices (even though they do not call out the correspondence of their method to h-indices). Degrees of the vertices are used as the initial core number estimates and each vertex updates its estimate as the h-index value for its neighbors' core number estimates. This process is repeated until convergence. At the end, each vertex has its core number. They prove that convergence to the core numbers is guaranteed, and analyze the convergence characteristics of the real-world networks and show runtime/quality trade-offs.

We generalize Lu et al.'s work for *any* nucleus decomposition, including k-truss. We show that convergence is guaranteed for all nucleus decompositions and prove the first upper bounds for the number of iterations. Our framework of algorithms locally compute *any* nucleus decomposition. We propose that iteratively computing h-indices of vertices/edges/r-cliques based on their degrees/triangle/s-clique counts converges in the core/truss/nucleus numbers ($r < s$). Local formulation also enables the parallelization. Intermediate values provide an approximation to the exact nucleus decomposition to trade-off between runtime and quality. Note that this is not possible in the peeling process, because no intermediate solution can provide an overall approximation to the exact solution, e.g., the densest regions are not revealed until the end.

1.2 Contributions

Our contributions can be summarized as follows:

- **Generalizated nucleus decomposition:** We generalize the iterative h-index computation idea [26] for *any* nucleus decomposition by using only local information. Our approach is based on iteratively computing the h-indices on the degrees of vertices, triangle counts of edges, and s-clique counts of r-cliques ($r < s$) until convergence. We prove that the iterative computation by h-indices guarantees exact core, truss, and nucleus decompositions.
- **Upper bounds for convergence:** We prove an upper bound for the number of iterations needed for convergence. We define the concept of *degree levels* that models the worst case for convergence. Our bounds are applicable to *any* nucleus decomposition and much tighter than the trivial bounds that rely on the total number of vertices.
- **Framework of parallel local algorithms:** We introduce a framework of efficient algorithms that only use local information to compute *any* nucleus decomposition. Our algorithms are highly parallel due to the local computation and are implemented in OpenMP for shared-memory architectures.
- **Extensive evaluation on real-world networks:** We evaluate our algorithms and implementation on various real-world networks. We investigate the convergence characteristics of our new algorithms and show that close approximations can be obtained only in a few iterations. This enables exploring trade-offs between time and accuracy. Figure 1a presents the convergence rates for the k-truss decomposition. In addition, we present a metric that approximates solution quality for informed decisions on accuracy/runtime trade-offs. We also evaluate runtime performances of our algorithms, present scalability results, and examine trade-offs between runtime and accuracy. Figure 1b has the results at a glance for the k-truss case. Last, but not least, we highlight a query-driven scenario where our local algorithms are used on a subset of vertices/edges to estimate the core and truss numbers.

2. BACKGROUND

We work on a simple undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. We define r-clique as a complete graph among r vertices for $r > 0$, i.e., each vertex is connected to all the other vertices. **We use R (and S) to denote r-clique (and s-clique).**

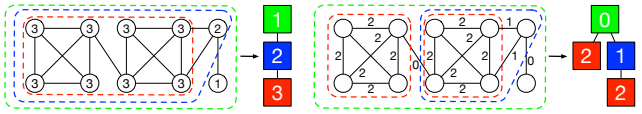
2.1 Core, Truss, and Nucleus Decompositions

DEFINITION 1. *k-core of G is a maximal connected subgraph of G, where each vertex has at least degree k.*

A vertex can reside in multiple k-cores, for different k values, which results in a hierarchy. Core number of a vertex is defined to be the the largest k value for which there is a k-core that contains the vertex. Maximum core of a vertex is the maximal subgraph around it that contains vertices with equal or larger core numbers. It can be found by a traversal that only includes the vertices with larger or equal core numbers. Figure 2a illustrates k-core examples.

DEFINITION 2. *k-truss of G is a maximal connected subgraph of G where each edge is in at least k triangles.*

Cohen [6] defined the standard maximal k-truss as one-component subgraph such that each edge participates in at least $k - 2$ triangles, but here we just assume it is k triangles for the sake of simplicity. An edge can take part in multiple k-trusses, for different k values, and there are also hierarchical relations between k-trusses in a graph. Similar to the core number, truss number of an edge is defined to



(a) k-cores (b) k-trusses

Figure 2: *Illustrative examples for k-core and k-truss. On the left, red, blue, and green regions show the 3-, 2-, and 1-cores. Core numbers are also shown for each vertex. For the same graph, trusses are presented on the right. Entire graph is a 0-truss. Five vertices on the right form a 1-truss, in blue. There are also two 2-trusses and one of them is a subset of the 1-truss. Truss numbers of the edges are also shown.*

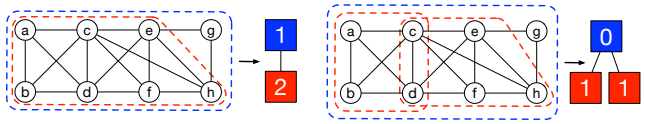
be the largest k value for which there exists a k-truss that includes the edge and maximum truss of an edge is the maximal subgraph around it that contains edges with larger or equal truss numbers. We show some truss examples in Figure 2b. Computing the core and truss numbers are known as the core and truss decomposition.

Unifying k-core and k-truss: Nucleus decomposition is a framework that generalizes core and truss decompositions [37]. k-(r, s) nucleus is defined as the maximal subgraph of the r-cliques where each r-clique takes part in at least k s-cliques. We first give some basic definitions and then formally define the k-(r, s) nucleus subgraph.

DEFINITION 3. *Let $r < s$ be positive integers.*

- *$\mathcal{R}(G)$ and $\mathcal{S}(G)$ are the set of r-cliques and s-cliques in G, respectively (or $\mathcal{R}$ and $\mathcal{S}$ when G is unambigous).*
- ***$\mathcal{S}$-degree** of $R \in \mathcal{R}(G)$ is the number of $S \in \mathcal{S}(G)$ such that S contains R ($R \subset S$). It is denoted as $d_s|_G(R)$ (or $d_s|(R)$ when G is obvious).*
- *Two r-cliques R, R' are **$\mathcal{S}$-connected** if there exists a sequence $R = R_1, R_2, \ldots, R_k = R'$ in $\mathcal{R}$ such that for each i, some $S \in \mathcal{S}$ contains $R_i \cup R_{i+1}$.*
- *Let k, r, and s be positive integers such that $r < s$. A **k-(r, s) nucleus** is a subgraph G' which contains the edges in the maximal union $\mathcal{S}$ of s-cliques such that*
  - *$\mathcal{S}$-degree of any r-clique $R \in \mathcal{R}(G')$ is at least k.*
  - *Any r-clique pair $R, R' \in \mathcal{R}(G')$ are $\mathcal{S}$-connected.*

For $r = 1, s = 2$, k-$(1, 2)$ nucleus is a maximal (induced) connected subgraph with minimum vertex degree k. This is exactly the k-core. Setting $r = 2, s = 3$ gives maximal subgraphs where every edge participates in at least k triangles, which corresponds to k-truss, and all edges are triangle connected, which is also introduced as a k-truss community [18]. Note that the original k-truss definition is different from $(2, 3)$ nucleus since it does not require triangle connectivity. In this work, we focus on core, truss, and κ_s indices (Definition 4) and our algorithms work for *any connectivity constraint*. We skip details for brevity.



(a) k-trusses (b) k-(3,4) nuclei

Figure 3: *Illustrative examples for k-truss and k-$(3, 4)$ nucleus. On the left, entire graph is a single 1-truss and all except vertex g forms a 2-truss. For the same graph, k-$(3, 4)$ nuclei are shown on the right. Entire graph is a 0-(3,4) nucleus and there are two 1-(3,4) nuclei (in red): subgraph among vertices a, b, c, d and subgraph among c, d, e, f, h. Note that those two subgraphs are reported as separate, not merged, since there is no four-clique that contains a triangle from each nuclei (breaking $\mathcal{S}$-connectedness).*

Table 1: *Notations*

Symbol	Description
G	graph
R	r-clique
S	s-clique
$\mathcal{R}(G)$ (or $\mathcal{R}$)	set of r-cliques in graph G
$\mathcal{S}(G)$ (or $\mathcal{S}$)	set of s-cliques in graph G
$\mathcal{C}(G)$ (or $\mathcal{C}$)	$\mathcal{R}(G) \cup \mathcal{S}(G)$
$d_s\|_G(R)$ (or $d_s(R)$)	$\mathcal{S}$-degree of R: number of s-cliques that contains R in graph G
$\delta_{r,s}(G)$	minimum $\mathcal{S}$-degree of an r-clique in graph G
$\mathcal{N}_s(R)$	neighbor R's s.t. $\exists$ an s-clique $S \supseteq (R \cup R')$
$\kappa_s(R)$	largest k s.t. R is contained in a k-(r, s) nucleus
$\kappa_2(u)$, $\kappa_3(e)$	Core number of vertex u, truss number of edge e
$\mathcal{H}(K)$	largest h s.t. at least h numbers in set K are $\geq h$
$\mathcal{U}$	update operator, $(\tau : \mathcal{R} \to \mathbb{N}) \to (\mathcal{U}\tau : \mathcal{R} \to \mathbb{N})$

Nucleus decomposition for $r = 3$ and $s = 4$ has been shown to give denser (and non-trivial) subgraphs than the k-cores and k-trusses, where density of $G = (V, E)$ is defined as $2|E|/\binom{|V|}{2}$ [37]. Figure 3 presents the difference between k-truss and k-$(3, 4)$ nucleus on a toy graph. It is used to analyze citation networks of APS journal papers and discovered hierarchy of topics i.e., a large subgraph on complex networks has children subgraphs on synchronization networks, epidemic spreading and random walks, which cannot be observed with core and truss decompositions [38]. Nucleus decomposition for larger r and s values are costly and only affordable for small networks with a few thousand edges. Enumerating r-cliques and checking their involvements in s-cliques for $r, s > 4$, can become intractable for larger graphs, making k-$(3, 4)$ is a sweet spot.

In graph G, minimum $\mathcal{S}$-degree of an r-clique $R \in \mathcal{R}(G)$ is denoted as $\boldsymbol{\delta_{r,s}(G)}$, e.g., the minimum degree of a vertex in G is $\delta_{1,2}(G)$. We also use $\boldsymbol{\mathcal{N}_s(R)}$ to denote the set of neighbor r-cliques of R such that $R' \in \mathcal{N}_s(R)$ if $\exists$ an s-clique S s.t. $S \supset R$ and $S \supset R'$. As in k-core and k-truss definitions, an r-clique can take part in multiple k-(r, s) nuclei for different k values. We define the κ_s index of r-clique analogous to the core numbers of vertices and truss numbers of edges [37].

DEFINITION 4. *For any r-clique $R \in \mathcal{R}(G)$, the **κ_s-index** of R, denoted as $\kappa_s(R)$, is the largest k value such that R is contained in a k-(r, s) nucleus.*

Core number of a vertex u is denoted by $\kappa_2(u)$ and the truss number of an edge e is denoted by $\kappa_3(e)$. We use the notion of k-(r, s) nucleus and κ_s-index to introduce our generic theorems and algorithms for *any* r, s values. The set of k-(r, s) nuclei is found by the peeling algorithm [37] (given in Algorithm 1). It is a generalization of the k-core and k-truss decomposition algorithms, and finds the κ_s indices of r-cliques in non-decreasing order.

The following lemma is standard in the k-core literature and we prove the analogue for k-(r, s) nucleus. It is a convenient characterization of the κ_s indices.

Algorithm 1: PEELING(G, r, s)

Input: G: graph, $r < s$: positive integers
Output: $\kappa_s(\cdot)$: array of κ_s indices for r-cliques
Enumerate all r-cliques in G
For every r-clique R, set $d_s(R)$ ($\mathcal{S}$-degrees)
Mark every r-clique as unprocessed
for each *unprocessed r-clique R with minimum $d_s(R)$* **do**
  $\kappa_s(R) = d_s(R)$
  Find set $\mathcal{S}$ of s-cliques containing R
  for each $C \in \mathcal{S}$ **do**
    if *any $R \subset C$ is processed* **then** continue
    for each *r-clique $R' \subset C$, $R' \neq R$* **do**
      if $d_s(R') > d_s(R)$ **then** $d_s(R') = d_s(R') - 1$
  Mark R as processed
return array $\kappa_s(\cdot)$

LEMMA 1. *$\forall\ R \in \mathcal{R}(G)$, $\kappa_s(R) = max_{\mathcal{R}(G') \ni R} \delta_{r,s}(G')$, where $G' \subseteq G$.*

PROOF. Let T be the $\kappa_s(R)$-(r,s) nucleus containing R. By definition, $\delta_{r,s}(T) = \kappa_s(R)$, so $\max_{G'} \delta_{r,s}(G') \geq \kappa_s(R)$. Assume the contrary that there exists some subgraph $T' \ni R$ such that $\delta_{r,s}(T') > \kappa_s(R)$ (WLOG, we can assume T' is connected; otherwise, we denote T' to be the component containing R). There must exist some maximal connected $T'' \supseteq T'$ that is a $\delta_{r,s}(T')$-nucleus. This would imply that $\kappa_s(R) \geq \delta_{r,s}(T') > \kappa_s(R)$, a contradiction. □

2.2 h-index computation

The main idea in our work is the iterative h-index computation on the $\mathcal{S}$-degrees of r-cliques. h-index metric is introduced to measure the impact and productivity of researchers by the citation counts [17]. A researcher has an h-index of k if she has at least k papers and each paper is cited at least k times such that there is no $k' > k$ that satisfies these conditions. We define the function $\mathcal{H}$ to compute the h-index as follows:

DEFINITION 5. *Given a set K of natural numbers, $\mathcal{H}(K)$ is the largest $k \in \mathbb{N}$ such that $\geq k$ elements of K are $\geq k$.*

Core number of a vertex can be defined as the largest k such that it has at least k neighbors whose core numbers are also at least k. In the following section, we formalize this observation, and build on it to design algorithms to compute not only core decompositions but also truss or nucleus decomposition for any r and s values.

3. FROM THE h-INDEX TO THE κ_s-INDEX

Our main theoretical contribution is two-fold. First, we introduce a generic formulation to compute the k-(r,s) nucleus by an iterated h-index computation on r-cliques. Secondly, we prove convergence bounds on the number of iterations.

We define the *update operator* $\mathcal{U}$. This takes a function $\tau : \mathcal{R} \to \mathbb{N}$ and returns another function $\mathcal{U}\tau : \mathcal{R} \to \mathbb{N}$, where $\mathcal{R}$ is the set of r-cliques in the graph.

DEFINITION 6. *The update $\mathcal{U}$ is applied on the r-cliques in a graph G such that for each r-clique $R \in \mathcal{R}(G)$:*
1. For each s-clique $S \supset R$, set $\rho(S,R) = \min_{R' \subset S, R' \neq R} \tau(R')$.
2. Set $\mathcal{U}\tau(R) = \mathcal{H}(\{\rho(S,R)\}_{S \supset R})$.

Observe that $\mathcal{U}\tau$ can be computed in parallel over all r-cliques in $\mathcal{R}(G)$. It is convenient to think of the $\mathcal{S}$-degrees (d_s) and κ_s indices as functions $\mathcal{R} \to \mathbb{N}$. We initialize $\tau_0 = d_s$, and set $\tau_{t+1} = \mathcal{U}\tau_t$.

The results of Lu et al. [26] prove that, for the k-core case ($r = 1, s = 2$), there is a sufficiently large t such that $\tau_t = \kappa_2$ (core number). We generalize this result for *any* nucleus decomposition. Moreover, we prove the first convergence bounds for $\mathcal{U}$.

The core idea of [26] is to prove that the $\tau_t(\cdot)$ values never increase (monotonicity) and are always lower bounded by core numbers. We generalize their proof for any (r,s) nucleus decomposition.

THEOREM 1. *For all t and all r-cliques R:*
- *(Monotonicity) $\tau_{t+1}(R) \leq \tau_t(R)$.*
- *(Lower bound) $\tau_t(R) \geq \kappa_s(R)$.*

PROOF. (Monotonicity) We prove by induction on t. Consider the base case $t = 0$. Note that for all R, $\tau_1(R) = \mathcal{U}d_s(R) \leq d_s(R)$. This is because in the second step, the $\mathcal{H}$ operator acts on a set of $d_s(R)$, and this is largest possible value it can return. Now for induction (assume the property is true up to t). Fix an r-clique R, and s-clique $S \supset R$. For $\tau_t(R)$, one computes the value $\rho(S,R) = \min_{R' \subset S, R' \neq R} \tau_{t-1}(R)$. By the induction hypothesis, the values $\rho(S,R)$ computed for τ_{t+1} is at most the value computed for τ_t. Note that the $\mathcal{H}$ operator is monotone; if one decreases values in a set K, then $\mathcal{H}(K)$ cannot increase. Since the ρ values cannot increase, $\tau_{t+1}(R) \leq \tau_t(R)$.

(Lower bound) We will prove that for any $G' \subseteq G$, $\mathcal{R}(G') \ni R$, $\tau_t(R) \geq \delta_{r,s}(G')$. Lemma 1 completes the proof.

We prove the above by induction on t. For the base case, $\tau_0(R) = d_s|_G(R) \geq d_s|_{G'}(R) \geq \delta_{r,s}(G')$. Now for induction. By the induction hypothesis, $\forall\ R \in \mathcal{R}(G'), \tau_t(R) \geq \delta_{r,s}(G')$. Consider the computation of $\tau_{t+1}(R)$, and the values $\rho(S,R)$ computed in the step one. For every s-clique S, note that $\rho(S,R) = \min_{R' \subset S, R' \neq R} \tau_t(R')$. By the induction hypothesis, this is at least $\delta_{r,s}(G')$. By definition of $\delta_{r,s}(G')$, $d_s|_{G'}(R) \geq \delta_{r,s}(G')$. Thus, in step two, $\mathcal{H}$ returns at least $\delta_{r,s}(G')$. □

Note that this is an intermediate result and we will present our final result in Lemma 2 at the end.

3.1 Convergence bounds by the degree levels

A trivial upper bound for convergence is the number of r-cliques in the graph, $|\mathcal{R}(G)|$, because after n iterations n r-cliques with the lowest κ_s indices will converge. We present a tighter bound for convergence. Our main insight is to define the *degree levels* of r-cliques, and relate these to the convergence of τ_t to κ_s. We prove that the κ_s indices in the i-th level converge within i iterations of the update operation. This gives quantitative bounds on the convergence.

DEFINITION 7. *For a graph G,*
- *$\mathcal{C}(G) = \mathcal{R}(G) \cup \mathcal{S}(G)$, i.e., set of all r-cliques and s-cliques.*
- *$S \in \mathcal{C}(G)$ if and only if $R \in \mathcal{C}(G), \forall\ R \subset S$.*
- *If R is removed from $\mathcal{C}(G)$, all $S \supset R$ are also removed from $\mathcal{C}(G)$.*
- **Degree levels** *are defined recursively as follows. The i-th level is the set L_i.*
  - *L_0 is the set of r-cliques that has the minimum $\mathcal{S}$-degree in $\mathcal{C}$.*
  - *L_i is the set of r-cliques that has the minimum $\mathcal{S}$-degree in $\mathcal{C} \setminus \bigcup_{j<i} L_j$.*

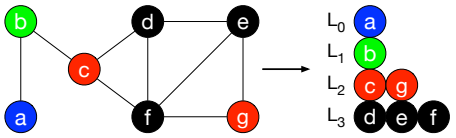


Figure 4: *Illustration of degree levels for the k-core decomposition. $L_0 = \{a\}$ since it has the minimum degree initially and the only such vertex. Its removal makes the b with the minimum degree, so $L_1 = \{b\}$. After removing vertex b, there are two vertices with the least degree; $L_2 = \{c, g\}$. Lastly, removing those leaves three vertices with the same degree and $L_3 = \{d, e, f\}$.*

Figure 4 shows the degree levels for k-core decomposition on a toy graph. We first prove the κ_s indices cannot decrease as the level increases. The following proof is closely related to the fact the minimum degree removal algorithm (peeling) finds all cores/trusses/nuclei.

THEOREM 2. *Let $i \leq j$. For any $R_i \in L_i$ and $R_j \in L_j$, $\kappa_s(R_i) \leq \kappa_s(R_j)$.*

PROOF. Let $L' = \bigcup_{r \geq i} L_r$, the union of all levels i and above, and G' is the graph such that $L' = \mathcal{R}(G')$. By definition of the levels, $d_s|_{G'}(R_i) = \delta_{r,s}(G')$ and $d_s|_{G'}(R_j) \geq d_s|_{G'}(R_i)$. There exists some $\kappa_s(R_i)$-nucleus T containing R_i. We split into two cases.

Case 1: $\mathcal{R}(T) \subseteq L'$. Thus, $\kappa_s(R_i) = \delta_{r,s}(T) \leq \delta_{r,s}(G') = d_s|_{G'}(R_i)$. Note that $\kappa_s(R_j) = \min_{P \ni R_j} \delta_{r,s}(P)$, so $\kappa_s(R_j) \geq \delta_{r,s}(G')$. Hence, $\kappa_s(R_i) \leq \kappa_s(R_j)$.

Case 2: $\mathcal{R}(T) \setminus L' \neq \emptyset$. Thus, there exists some r-clique $R' \in \mathcal{R}(T) \cap L_b$, where $b < i$. Choose the R' that minimizes this value of b. Since T is a $\kappa_s(R_i)$-nucleus, $d_s|_T(R') \geq \kappa_s(R_i)$. Consider $M = \bigcup_{r \geq b} L_r$. Note that $\mathcal{R}(T) \subseteq M$, since we chose R' to minimize b. Let Q is the graph such that $M = \mathcal{R}(Q)$. We have $d_s|_Q(R') \geq d_s|_T(R') \geq \kappa_s(R_i)$. Since $R' \in L_b$, $d_s|_Q(R') = \delta_{r,s}(Q)$. Since $j > b$ and $R_j \in M$, $\kappa_s(R_j) \geq \delta_{r,s}(Q)$. Combining the above, we deduce $\kappa_s(R_i) \leq \kappa_s(R_j)$. □

The main convergence theorem is the following. As explained earlier, it shows that the i-th level converges within i iterations.

THEOREM 3. *Fix level L_i. For all $t \geq i$ and $R \in L_i$, $\tau_t(R) = \kappa_s(R)$.*

PROOF. We prove by induction on i. For the base case $i = 0$; note that for any R of minimum $\mathcal{S}$-degree in G, $\kappa_s(R) = d_s|_G(R) = \tau_0(R)$. For induction, assume the theorem is true up to level i. Thus, for $t \geq i$ and $\forall\ R \in \bigcup_{j \leq i} L_j$, $\tau_t(R) = \kappa_s(R)$. Select arbitrary $R_a \in L_{i+1}$, and set $L' = \bigcup_{j \geq i+1} L_j$.

We partition the s-cliques containing R_a into the "low" set $\mathcal{S}_\ell$ and "high" set $\mathcal{S}_h$. s-cliques in $\mathcal{S}_\ell$ contain some r-clique outside L', and those in $\mathcal{S}_h$ are contained in L'. For every s-clique $S \in \mathcal{S}_\ell$, there is a $R_b \subset S$ such that $R_b \in L_k$ for $k \leq i$. By the inductive hypothesis, $\tau_t(R_b) = \kappa_s(R_b)$. By Theorem 2 applied to $R_b \in L_k$ and $R_a \in L_{i+1}$, $\kappa_s(R_b) \leq \kappa_s(R_a)$.

Now we focus on the computation of $\tau_{t+1}(R_a)$, which starts with computing $\rho(S, R_a)$ in step one of Definition 6. For every $S \in \mathcal{S}_\ell$, by the previous argument, there is some r-clique $R_b \subset S$, $R_b \neq R_a$, such that $\tau_t(R_b) \leq \kappa_s(R_a)$. Thus, $\forall\, S \in \mathcal{S}_\ell$, $\rho(S, R_a) \leq \kappa_s(R_a)$. This crucially uses the *min* in the setting of $\rho(S, R_a)$, and is a key insight into the generalization of iterated $\mathcal{H}$-indices for any nucleus decomposition.

The number of edges in $\mathcal{S}_h$ is exactly $d_s|_{G'}(R_a) = \delta_{r,s}(G')$. Applying Lemma 1 to $R_a \in L'$, we deduce $\kappa_s(R_a) \geq d_s|_{G'}(R_a)$. All in all, for all $S \in \mathcal{S}_\ell$, $\rho(S, R_a)$ is at most $\kappa_s(R_a)$. On the other hand, there are at most $\kappa_s(R_a)$ s-cliques in $\mathcal{S}_h$. The application of the $\mathcal{H}$ function in the second step yields $\tau_{t+1}(R_a) \leq \kappa_s(R_a)$. But the lower bound of Theorem 1 asserts $\tau_{t+1}(R_a) \geq \kappa_s(R_a)$, and hence, these are equal. This completes the induction. □

We have the following lemma to show that convergence is guaranteed in a finite number of iterations.

LEMMA 2. *Given a graph G let l be the maximum i, such that $L_l \neq \emptyset$ and $\tau_l(R) \geq \kappa_s(R)$ for all r-cliques (e.g., $\tau_0 = d_s$) and set $\tau_{t+1} = \mathcal{U}\tau_t$. For some $t \leq l$, $\tau_t(R) = \kappa_s(R)$, for all r-cliques.*

4. LOCAL ALGORITHMS

We introduce generalized local algorithms to find the κ_s indices of r-cliques for any (r, s) nucleus decomposition. For each r-clique, we iteratively apply h-index computation. Our local algorithms are parallel thanks to the independent nature of the h-index computations. We also explore time and quality trade-offs by using the iterative nature. We first present the deterministic synchronous algorithm which does not depend on the order of processing the r-cliques. It implements the $\mathcal{U}$ operator in Definition 6. Then we adapt our algorithm to work in an asynchronous manner that converges faster and uses less space. For those familiar with linear algebra, the synchronous and asynchronous algorithms are analogous to Jacobi and Gauss-Seidel iterations for iterative solvers. At the end, we discuss some heuristics and key implementation details for shared-memory parallelism in OpenMP.

4.1 Synchronous Nucleus Decomposition (SND)

We use the update operator $\mathcal{U}$ to compute the k-(r, s) nuclei of a graph G in a synchronous way. Algorithm 2 (SND) implements the Definition 6 for functions $\tau_0 = d_s$ and $\tau_{t+1} = \mathcal{U}\tau_t$ to find the κ_s indices of r-cliques in graph G.

SND algorithm iterates until no further updates occur for any τ index, which means all the τ indices converged to κ_s. Computation is performed synchronously on all the r-cliques and at each iteration i, τ_i indices are found for all r-cliques. We declare two arrays, $\tau(\cdot)$ and $\tau^p(\cdot)$, to store the indices being computed and the indices that were computed in the previous iteration, respectively (Lines 1 and 4). $\tau(\cdot)$ are initialized to the $\mathcal{S}$-degrees of the r-cliques since $\tau_0 = d_s$ (Line 2). At each iteration, newly computed $\tau(\cdot)$ indices are backed up in $\tau^p(\cdot)$ (Line 7), and the new $\tau(\cdot)$ indices are computed. During the iterative process, convergence is checked by the flag $\mathcal{F}$ (Line 5), which is initially set to TRUE (Line 3) and stays TRUE as long as there is an update on a τ index (Lines 6, 13, and 14).

Computation of the new $\tau(\cdot)$ indices for each r-clique can be performed in parallel (Lines 8 to 15). For each r-clique R, we apply the two step process in the Definition 6. First, for each s-clique S that contains R, we compute the ρ values that is the minimum τ^p index of an r-clique $R' \subset S$ ($R' \neq R$) and collect them in a set L (Lines 10 to 12). Then, we assign the h-index of the set L as the new τ index of the r-clique (Line 15). The algorithm continues until there are no updates on the τ index ' (Lines 13 and 14). Once the

Algorithm 2: SND: Synchronous Nucleus Decomp

Input: G: graph, r, s: positive integers ($r < s$)
Output: $\kappa_s(\cdot)$: array of κ_s indices for r-cliques

1 $\tau(\cdot) \leftarrow$ indices $\forall\ R \in \mathcal{R}(G)$ `// current iteration`
2 $\tau(R) \leftarrow d_s(R)\ \forall\ R \in \mathcal{R}(G)$ `// set to the` $\mathcal{S}$`-degrees`
3 $\mathcal{F} \leftarrow$ TRUE `// stays` TRUE `if any` $\tau(R)$ `is updated`
4 $\tau^p(\cdot) \leftarrow$ backup indices $\forall\ R \in \mathcal{R}(G)$ `// prev. iter.`
5 **while** $\mathcal{F}$ **do**
6 $\quad\mathcal{F} \leftarrow$ FALSE
7 $\quad\tau^p(R) \leftarrow \tau(R)\ \forall\ R \in \mathcal{R}(G)$
8 $\quad$**for each** $R \in \mathcal{R}(G)$ *in parallel* **do**
9 $\quad\quad L \leftarrow$ empty set
10 $\quad\quad$**for each** s-clique $S \supset R$ **do**
11 $\quad\quad\quad\rho \leftarrow \min_{R' \in \mathcal{N}_s(R)} \tau^p(R')$
12 $\quad\quad\quad L$. add (ρ)
13 $\quad\quad$**if** $\tau^p(R) \neq \mathcal{H}(L)$ **then**
14 $\quad\quad\quad\mathcal{F} \leftarrow$ TRUE
15 $\quad\quad\tau(R) \leftarrow \mathcal{H}(L)$
16 $\kappa_s(\cdot) \leftarrow \tau(\cdot)$
17 **return** array $\kappa_s(\cdot)$

τ indices converge, we assign them to κ_s indices and finish (Lines 16 and 17).

Time complexity: SND algorithm starts with enumerating the r-cliques (not shown in the pseudocode) and its runtime is denoted by $RT_r(G)$ (this part can be parallelized as well, but we ignore that for now). Then, each iteration (Lines 5 to 15) is performed t times until convergence where t is the total number of iterations for which we provided bounds in Section 3.1. In each iteration, each r-clique $R \in \mathcal{R}$ is processed once, *which is parallelizable*. Suppose R has vertices $v_1, v_2, \cdots, v_r$. We can find all s-cliques containing R by looking at all $(s-r)$-tuples in each of the neighborhoods of v_i (Indeed, it suffices to look at just one such neighborhood). This takes $(\sum_R \sum_{v \in R} d(v)^{s-r})/p = (\sum_{v \in V} \sum_{R \ni v} d(v)^{s-r})/p = (\sum_{v \in V} d_R(v) d(v)^{s-r})/p$ time if p threads are used for parallelism. Note that the h-index computation can be done incrementally without storing all ρ values in set L (see Section 4.4). Overall, the time complexity of SND using p threads is:

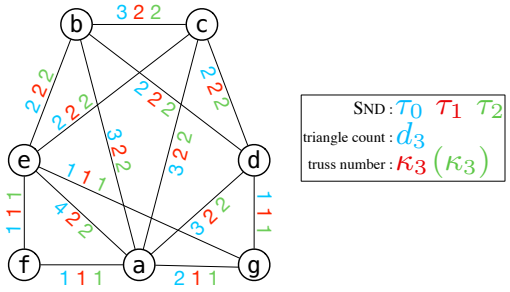


Figure 5: SND *(Algorithm 2) for the k-truss decomposition ($r = 2, s = 3$). We find the κ_3 indices. Triangle counts of all the edges are computed (d_3) and set as their τ_0 values (blue). For each edge, we first compute τ_1 indices (red) based on the τ_0 indices. The* bc *edge, for instance, has three triangles and for each of those we find the neighbor with the minimum τ_0 index and compute the h-index. So, τ_1(bc) $= \mathcal{H}\{min(\tau_0(\text{ba}), \tau_0(\text{ca})), min(\tau_0(\text{bd}), \tau_0(\text{cd})), min(\tau_0(\text{be}), \tau_0(\text{ce}))\} = \mathcal{H}\{3, 2, 2\} = 2$. No updates happen in the second iteration (green), so convergence is obtained in a single iteration.*

Algorithm 3: AND: Asynchronous Nucleus Decomp.

Input: G: graph, r, s: positive integers ($r < s$)
Output: $\kappa_s(\cdot)$: array of κ_s indices for r-cliques

1 $\tau(\cdot) \leftarrow$ indices $\forall\ R \in \mathcal{R}(G)$ `// current iteration`
2 $\tau(R) \leftarrow d_s(R)\ \forall\ R \in \mathcal{R}(G)$ `// set to the` $\mathcal{S}$`-degrees`
3 $\mathcal{F} \leftarrow$ TRUE `// stays` TRUE `if any` $\tau(R)$ `is updated`
4 c$(R) \leftarrow$ TRUE $\forall\ R \in \mathcal{R}(G)$
5 **while** $\mathcal{F}$ **do**
6 $\quad\mathcal{F} \leftarrow$ FALSE
7 $\quad$**for each** $R \in \mathcal{R}(G)$ *in parallel* **do**
8 $\quad\quad$**if** c(R) *is* FALSE **then** cont. **else** c$(R) \leftarrow$ FALSE
9 $\quad\quad L \leftarrow$ empty set
10 $\quad\quad$**for each** s-clique $S \supset R$ **do**
11 $\quad\quad\quad\rho \leftarrow \min_{R' \in \mathcal{N}_s(R)} \tau(R')$
12 $\quad\quad\quad L$. add (ρ)
13 $\quad\quad$**if** $\tau(R) \neq \mathcal{H}(L)$ **then**
14 $\quad\quad\quad\mathcal{F} \leftarrow$ TRUE, c$(R) \leftarrow$ TRUE
15 $\quad\quad\quad$**for each** $R' \in \mathcal{N}_s(R)$ **do**
16 $\quad\quad\quad\quad$**if** $\mathcal{H}(L) \leq \tau(R')$ **then**
17 $\quad\quad\quad\quad$c$(R') \leftarrow$ TRUE
18 $\quad\quad\tau(R) \leftarrow \mathcal{H}(L)$
19 $\kappa_s(\cdot) \leftarrow \tau(\cdot)$
20 **return** array $\kappa_s(\cdot)$

$$\mathcal{O}\Big(RT_r(G) + t\big(\sum_{v \in V} d_R(v) d(v)^{s-r}\big)/p\Big) \tag{1}$$

When $t = p$, complexity is same as the sequential peeling algorithm's (Algorithm 1) and SND is work-efficient.

Space complexity: In addition to the space that is needed to store r-cliques (taking $\mathcal{O}(r\,|\mathcal{R}(G)|)$), we need to store τ indices for the current and the previous iterations, which takes $\mathcal{O}(|\mathcal{R}(G)|)$ space, i.e., number of r-cliques. ρ values are not need to be stored in set L since the h-index computation can be done incrementally. So, the total space complexity is $\mathcal{O}(|\mathcal{R}(G)|)$ (since $r = \mathcal{O}(1)$).

Figure 5 illustrates the SND algorithm for k-truss decomposition ($r = 2, s = 3$) on a toy graph, where the participations of edges (2-cliques) in triangles (3-cliques) are examined. Triangle counts of all the edges (d_3) are computed and set as their τ_0 values (in blue). For each edge, first we compute τ_1 indices (in red) based on the τ_0 indices (Lines 5 to 15). For instance, the **ae** edge has four triangles and for each of those we find the neighbor with minimum τ_0 index (Lines 10 to 12); thus $L = \{(min(\tau_0(\textbf{eb}), \tau_0(\textbf{ab})), min(\tau_0(\textbf{ec}), \tau_0(\textbf{ac})), min(\tau_0(\textbf{eg}), \tau_0(\textbf{ag})), min(\tau_0(\textbf{ef}), \tau_0(\textbf{af}))\} = \{2, 2, 1, 1\}$ and $\tau_1(\textbf{ae}) = \mathcal{H}(L) = 2$ (Line 15). Since the τ index is updated, we set flag $\mathcal{F}$ TRUE to continue iterations. In the second iteration (τ_2 indices), no update occurs, i.e., $\tau_2(\textbf{e}) = \tau_1(\textbf{e})$ for all edges, thus the algorithm termnates. So, one iteration is enough for the convergence and we have $\kappa_3 = \tau_1$ for all the edges.

4.2 Asynchronous Nucleus Decomposition (AND)

In the SND algorithm, updates on the τ indices are synchronous and all the r-cliques are processed by using the same snapshot of τ indices. However, when an r-clique R is being processed in iteration i, a neighbor r-clique $R' \in \mathcal{N}_s(R)$ might have already completed its computation in that iteration and updated its τ index. By Theorem 1, we know that the τ index can only decrease as the algorithm proceeds. Lower $\tau(R')$ indices in set L can decrease $\mathcal{H}(L)$, and

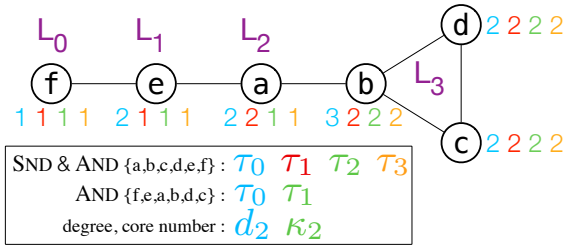


Figure 6: SND *(Algorithm 2) and* AND *(Algorithm 3, w/o orange lines) for the k-core decomposition ($r = 1, s = 2$). We find the κ_2 indices (core numbers) of vertices (edge is 2-clique). τ_0 indices are initialized to the degrees (d_2 in blue).* SND *algorithm uses the τ_{i-1} indices to compute the τ_i indices and converges in two iterations (τ_1 in red, τ_2 in green, τ_3 in orange). Same happens when we use* AND *and follow the* {a,b,c,d,e,f} *order to process the vertices. On the other hand, if we choose the order by degree levels,* {f,e,a,b,c,d}*, convergence is obtained in a single iteration.*

it can help $\tau(R)$ to converge faster. So, it is better to use the *up-to-date* τ indices for faster convergence. In addition, there would be no need to store the τ indices computed in the previous iteration, saving $|\mathcal{R}(G)|$ space.

We introduce the AND algorithm (Algorithm 3) to leverage the up-to-date τ indices for faster convergence (ignore the orange lines for now). At each iteration, we propose to use the latest available information in the neighborhood of an r-clique. Removing the green lines in the SND algorithm and inserting the blue lines in the AND algorithm are sufficient to switch from synchronous to asynchronous computation. We do not need to use the $\tau_p(\cdot)$ to back up the indices in the previous iteration anymore, so Lines 4 and 7 in Algorithm 2 are removed. Computation is done on the latest τ indices, so we adjust the Lines 11 and 13 in Algorithm 2 accordingly, to use the up-to-date τ indices.

In the same iteration, each r-clique can have a different view of the $\tau(\cdot)$ and updates are done *asynchronously* in an arbitrary order. Number of iterations for convergence depends on the processing order (Line 7 in Algorithm 3) and never more than the SND algorithm.

Time complexity: *The worst case for* AND *happens when all the r-cliques see the τ indices of their neighbors that are computed in the previous iteration, which exactly corresponds to the* SND *algorithm.* Thus the time complexity of AND is same as SND's (Equation (1)). However, in practice we expect fewer iterations.

Space complexity: The only difference with SND is that we do not need to store the τ values in the previous iteration anymore. So, it is still $\mathcal{O}(|\mathcal{R}(G)|)$.

Figure 6 illustrates AND algorithm with two different orderings and the SND algorithm on the k-core case ($r = 1, s = 2$). Focus is on vertices (1-cliques) and their edge (2-clique) counts (degrees). We start with SND. Vertex degrees are set as τ_0 indices (blue). For each vertex u we compute the $\tau_1(u) = \mathcal{H}(\{\tau_0(v) : v \in \mathcal{N}_2(u)\}$ (red), i.e., h-index of its neighbors' degrees. For instance, vertex **a** has two neighbors, **e** and **b**, with degrees 2 and 3. Since $\mathcal{H}(\{2,3\}) = 2$, we get $\tau_1(\mathbf{a}) = 2$. For vertex **b**, we get $\tau_1(\mathbf{b}) = \mathcal{H}(\{2,2,2\}) = 2$. After computing all the τ_1 indices, τ values of vertices **e** and **b** are updated, thus we compute the τ_2 indices, shown in green. We observe an update for the vertex **a**; $\tau_2(\mathbf{a}) = \mathcal{H}(\{\tau_1(\mathbf{e}), \tau_1(\mathbf{b})\}) = \mathcal{H}(\{1,2\}) = 1$ and

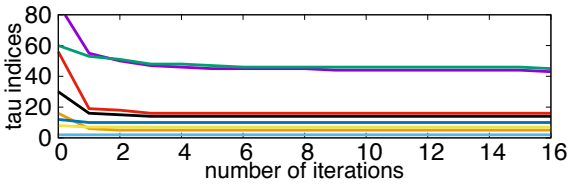


Figure 7: *Changes in τ indices of randomly selected edges in* `facebook` *graph during the k-truss decomposition. Wide plateaus appear during the convergence, especially at the end.*

continue computation. For τ_3 indices (orange), no update is observed which means that $\kappa_s = \tau_2$, and SND converges in two iterations. Regarding the AND algorithm, say we choose to follow the increasing order of degree levels (noted in purple) where $L_0 = \{\mathbf{f}\}$, $L_1 = \{\mathbf{e}\}$, $L_2 = \{\mathbf{a}\}$, $L_3 = \{\mathbf{b,c,d}\}$. Computing the τ_1 indices on this order enables us to reach the convergence in a single iteration. For instance, $\tau_1(\mathbf{a}) = \mathcal{H}(\{\tau_1(\mathbf{e}), \tau_0(\mathbf{b})\}) = \mathcal{H}(\{1,3\}) = 1$. However, if we choose to process the vertices in a different order than the degree levels, say $\{\mathbf{a,b,c,d,e,f}\}$, we have $\tau_1(\mathbf{a}) = \mathcal{H}(\{\tau_0(\mathbf{e}), \tau_0(\mathbf{b})\}) = \mathcal{H}(\{2,3\}) = 2$, and need more iteration(s) to converge. Indeed, **a** is the only updated vertex. In the second iteration, we get $\tau_2(\mathbf{a}) = \mathcal{H}(\{\tau_1(\mathbf{e}), \tau_1(\mathbf{b})\}) = \mathcal{H}(\{1,2\}) = 1$, an update, thus continue iterating. Third iteration does not change the τ indices, so AND with $\{\mathbf{a,b,c,d,e,f}\}$ order converges in two iterations, just as the SND.

4.3 Avoiding redundant work by notifications

SND and AND algorithms converge when no r-clique updates its τ index anymore. Consequently, update on all r-cliques continue even when only one update occurs and we need an extra iteration to detect convergence. Figure 7 shows the τ indices of randomly selected edges in the `facebook` graph during k-truss decomposition ($r = 2, s = 3$). There are plenty of wide plateaus where τ indices stay constant, which implies redundant computations. How can we avoid this redundancy? Observe that repeating τ indices or plateaus are not sufficient, because an update can still occur after maintaining the same τ index for a number of iterations, creating a plateau. In order to efficiently detect the convergence and skip any plateaus during the computation, we introduce a notification mechanism where an r-clique is notified to recompute its τ index, if any of its neighbors has an update.

Orange lines in Algorithm 3 present the notification mechanism added to AND. $\texttt{c}(\cdot)$ array is declared in (Line 4) to track whether an $R \in \mathcal{R}(G)$ has updated its τ index. $\texttt{c}(R) =$ FALSE means R did not update its τ index, it is an idle r-clique, and there is no need to recompute its τ value for that iteration (Line 8). A non-idle r-clique is called active. Thus, all $\texttt{c}(\cdot)$ is set to TRUE at the beginning to initiate the computations for all r-cliques. Each r-clique marks itself idle at the beginning of an iteration (Line 8) and waits for an update in a neighbor. When the $\tau(R)$ is updated, τ indices of *some* neighbor r-cliques in $\mathcal{N}_s(R)$ might be affected and they should be notified. If $R' \in \mathcal{N}_s(R)$, if $\tau(R') < \mathcal{H}(L)$ (new $\tau(R)$) then $\tau(R') \leq \tau(R)$ already in the previous iteration (Theorem 1), and thus no change can happen in the h-index computation. Therefore, we only need to notify the neighbors that have τ indices greater than or equal to $\mathcal{H}(L)$ (Lines 15 to 17). This version of our algorithm requires an additional $O(|\mathcal{R}(G)|)$ space for $\texttt{c}(\cdot)$ array and does not offer a theoretical improvement in time-complexity. However,

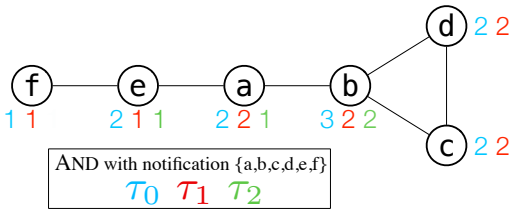


Figure 8: *k-core decomposition ($r = 1, s = 2$) by* AND *(Algorithm 3) that uses the notification mechanism. After the first iteration, the only active vertex is a. In the second iteration, computation updates $\tau(a)$ and thus notifies vertices b and e. In the third iteration, their τ indices are recomputed and no update happens. All the vertices become idle, thus convergence is obtained. 9 τ computations performed in 3 iterations by* AND *with notification mechanism, while 24 τ computations are done in 4 iterations if notification mechanism is not used (Figure 6).*

avoiding redundant computations yields faster runtimes in practice.

Figure 8 illustrates the notification mechanism on the graph in Figure 6, processing the vertices in the **a,b,c,d,e,f** order. Again, vertex degrees are set as τ_0 indices (blue) and we compute $\tau_1(u) = \mathcal{H}(\{\tau_0(v) : v \in \mathcal{N}_2(u)\}$, i.e., h-index of its neighbors' degrees, (red) for each vertex u. No update happens for vertex **a** and no vertices are notified. $\tau(\mathbf{b})$ is updated as 2 and we check if any neighbors of **b** has a τ index ≥ 2 (Line 16). All its neighbors have such τ indices, thus all are notified: **a, c, d**. Vertices **c** and **d** do not update their τ indices. Then, $\tau(\mathbf{e})$ is updated as 1 and since $\tau_0(\mathbf{e}) \geq \tau_1(\mathbf{a}) > \tau_1(\mathbf{e})$, vertices **a** and **f** are notified for recomputing its τ index. At that instant, vertices **a** and **f** are active. Next, vertex **f** is processed and does not change its τ index, so all the vertices except **a** are idle now. In the second iteration, we only process **a** and compute $\tau_2(\mathbf{a}) = \mathcal{H}\{\tau_1(\mathbf{e}), \tau_1(\mathbf{b})\} = \mathcal{H}\{1, 2\} = 1$. Update in $\tau(\mathbf{a})$ notifies vertices **b** and **e** since both have $\geq \tau$ indices. In the third iteration, we recompute τ indices for **b** and **e**, but there is no update. So all the vertices become idle, implying convergence. Overall, it takes 9 τ computations and 3 iterations for the AND with notification mechanism, while 24 τ computations and 4 iterations are needed without the notification mechanism (Figure 6). So the notification mechanism is helpful to avoid redundant computations.

PARTIALAND on a set of r-cliques: Local nature of the AND algorithm enables selection of a set of r-cliques and its application only to this set until convergence. This is useful in query-driven scenarios where the focus is on a single (or a few) vertex/edge. We define PARTIALAND as the application of AND algorithm on a set of given r-cliques, say P. We only modify the orange lines in Algorithm 3 where `c` of an r-clique is re-computed only if it is in set P. This way we just limit the AND computation on a small set. We give an application of PARTIALAND in Section 5.3 where the computation is limited on a given r-clique and its neighbors.

4.4 Heuristics and implementation

We introduce key implementation details for the shared-memory parallelism and heuristics for efficient h-index computation. We used OpenMP [7] to utilize the shared-memory architectures. The loops, annotated as parallel in Algorithms 2 and 3, are shared among the threads, and each thread is responsible for its partition of r-cliques. No synchronization or atomic operation is needed. Default scheduling policy in OpenMP is *static* and it distributes the iterations of the loop to the threads in chunks, i.e., for two threads, one takes the first half and the other takes the second. This approach does not work well for our algorithms, since the notification mechanism may result in significant load imbalance among threads. If most of the idle r-cliques are assigned to a certain thread, this thread quickly finishes, and remains idle until the iteration ends. To prevent this, we adopted the dynamic scheduling where each thread is given a new workload once it idle. We set chunk size to 100 and observed no significant difference for other values. No thread stays idle this way, improving parallel efficiency.

h-index computation for a list of numbers is traditionally done by sorting the numbers in the non-increasing order and checking the values starting from the head of the list to find the largest h value for which at least h items exist with at least h value. Main bottleneck in this routine is the sorting operation which takes $\mathcal{O}(nlogn)$ time for n numbers. We use a linear time algorithm that uses a hashmap and does not include sorting to compute the h-index. h is initialized as zero and we iterate over the items in the list. At each step, we attempt to increase the current h value based on the inspected item. For the present h value in a step, we keep track of the number of items examined so far that have value equal to h. We use a hashmap to keep track of the number of items that has at least h value, and ignore values smaller than h. This enables the computation of the h-index in linear time and provides a trade-off between time and space. In addition, after the initialization, we check to see if the current τ index can be preserved. Once we see at least τ items with index at least τ, no more checks needed.

Table 2: *Statistics about our dataset; number of vertices, edges, triangles and four-cliques (K_4).*

	$\|V\|$	$\|E\|$	$\|\triangle\|$	$\|K_4\|$
`as-skitter` (ASK)	1.7M	11.1M	28.8M	148.8M
`facebook` (FB)	4K	88.2K	1.6M	30.0M
`friendster` (FRI)	65.6M	1.8B	4.1B	8.9B
`soc-LiveJournal` (SLJ)	4.8M	68.5M	285.7M	9.9B
`soc-orkut` (ORK)	2.9M	106.3M	524.6M	2.4B
`soc-sign-epinions` (SSE)	131.8K	711.2K	4.9M	58.6M
`soc-twitter-higgs` (HG)	456.6K	12.5M	83.0M	429.7M
`twitter` (TW)	81.3K	1.3M	13.1M	104.9M
`web-Google` (WGO)	916.4K	4.3M	13.4M	39.9M
`web-NotreDame` (WND)	325.7K	1.1M	8.9M	231.9M
`wiki-200611` (WIKI)	3.1M	37.0M	88.8M	162.9M

5. EXPERIMENTS

We evaluate our algorithms on three instances of the (r, s) nucleus decomposition: k-core $((1, 2))$, k-truss $((2, 3))$, and $(3, 4)$ nucleus, which are shown to be the practical and effective [37, 38]. We do not store the s-cliques during the computation for better scalability in terms of the memory space. Instead, we find the participations of the r-cliques in the s-cliques on-the-fly [37]. Our dataset contains a diverse set of real-world networks from SNAP [25] and Network Repository [32] (see Table 2), such as internet topology network (`as-skitter`), social networks (`facebook`, `friendster`, `soc-LiveJournal`, `soc-orkut`), trust network (`soc- sign-epinions`), Twitter follower-followee networks (`soc-twitter -higgs`, `twitter`), web networks (`web-Google`, `web- NotreDame`), and a network of wiki pages (`wiki-200611`).

Table 3: *Number of iterations for the theoretical upper bound, Degree Levels* (DL) *(Section 3.1),* SND*, and* AND *algorithms.*

		ASK	FB	SLJ	ORK	HG	WGO	WIKI
k-core	DL	1195	352	3479	5165	1713	384	2026
	SND	63	21	99	147	73	23	55
	AND	33	11	51	73	37	14	30
k-truss	DL	1605	859	5401	4031	2215	254	2824
	SND	118	33	86	207	101	20	562
	AND	58	19	44	103	53	11	410
$(3,4)$	DL	1734	1171	7426	3757	2360	157	1559
	SND	72	38	123	196	109	11	122
	AND	41	23	73	116	51	6	107

All experiments are performed on a Linux operating system running on a machine with Intel Ivy Bridge processor at 2.4 GHz with 64 GB DDR3 1866 MHz memory. There are two sockets on the machine and each has 12 cores, making 24 cores in total. Algorithms are implemented in C++ and compiled using gcc 6.1.0 at the -O2 level. We used OpenMP v4.5 for the shared-memory parallelization. Code is available at `http://sariyuce.com/pnd.tar`.

We first investigate the convergence characteristics of our new algorithms in Section 5.1. We compare the number of iterations that our algorithms need for the convergence and also examine the convergence rates for the κ values. In addition, we investigate how the densest subgraphs evolve and present a metric that can be monitored to determine the "good-enough" decompositions so that trade-offs between quality and runtime can be enjoyed. Then, we evaluate the runtime performance in Section 5.2. In particular, we examine the impact of notification mechanism (Section 4.3) on the AND algorithm, show the scalability for our best performing method, and compare it with respect to the partially parallel peeling algorithms. We also examine the runtime and accuracy trade-off for our algorithms. Last, but not least, we highlight a query-driven scenario in Section 5.3 where our algorithms are used on a subset of vertices/edges to estimate the core and truss numbers.

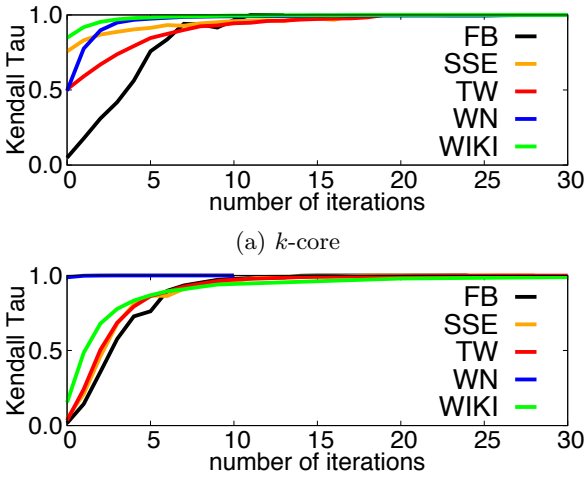
5.1 Convergence analysis

Here we study the following questions:

- How does the number of iterations change between asynchronous computation (AND) and synchronous (SND)? How do they relate to our theoretical bounds of Section 3.1?
- What is the rate of convergence regarding the τ values? How quickly do they approach to the κ values?
- How is the evolution of the important subgraphs (with high density) during the convergence?
- Is there a generic way to infer the "good-enough" decompositions so that the computation can be halted for trade-off between runtime and quality?

5.1.1 Number of iterations

As described in Section 4.2, the number of iterations for convergence can (only) be decreased by the asynchronous algorithm AND. We compare SND (Algorithm 2) and AND (Algorithm 3) for three nucleus decompositions. All runs are performed in sequential, and for AND we use the natural ordering of the r-cliques in datasets that is the order of vertices/edges/triangles given or computed based on the ids in the data files. Note that, we also checked AND with random r-clique orderings and did not observe significant differences. We also compute the number of degree levels (Definition 7) that we prove as an upper bound in Section 3.1.



(a) k-core

(b) $(3,4)$ nucleus

Figure 9: *Convergence rates for five graphs in our dataset. Kendall-Tau similarity score compares the τ values in a given iteration with the exact decomposition (κ values); becomes 1.0 when they are equal. Our algorithms compute almost-exact decompositions in around 10 iterations for k-core, k-truss (in Figure 1), and $(3,4)$ nucleus decompositions.*

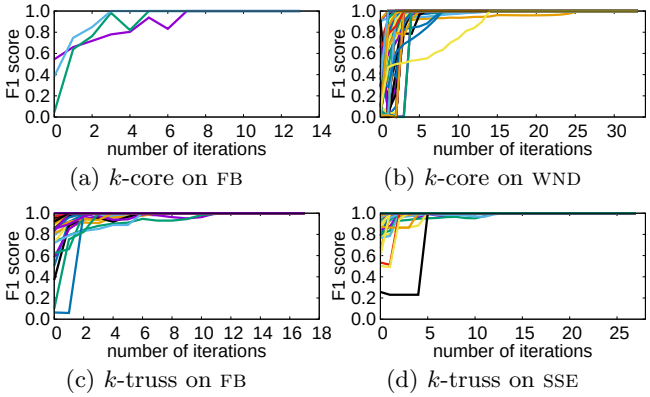
Table 3 presents the results for k-core, k-truss, and $(3,4)$ nucleus decompositions. Number of degree levels gives much tighter bounds than the obvious limits – number of r-cliques. We observe that both algorithms converge in far fewer iterations than our upper bounds – SND converges in 5% of the bounds given for all decompositions, on average. Regarding the comparison, AND algorithm converges in 50% fewer iterations than the SND for k-core and k-truss decompositions and 35% fewer iterations for $(3,4)$ nucleus decomposition. Overall, we see the clear benefit of asynchronous computation on all the decompositions, thus use AND algorithm in the following experiments.

5.1.2 Convergence rates for the τ values

In the previous section, we studied the number of iterations required for exact solutions. Now we will investigate how fast our estimates, τ values converge to the exact, κ values. We use Kendall-Tau similarity score to compare the τ and κ values for each iteration, which becomes 1, when they are equal. Figure 9 and Figure 1 present the results for five representative graphs in our dataset. We observe that our local algorithms compute almost exact decompositions in less than 10 iterations for all decompositions, and we need 5, 9, and 6 iterations to achieve 0.90 similarity for k-core, k-truss, and $(3,4)$ nucleus decompositions, respectively.

5.1.3 Evolution of the densest regions

In the hierarchy of dense subgraphs computed by our algorithms, the leaves are the most important part (see Figure 2 and Figure 3), since those subgraphs have the highest edge density ($|E|/\binom{|V|}{2}$), pointing to the significant regions. Note that the r-cliques in a leaf subgraph have same κ values and they are the local maximals, i.e., have greater-or-equal κ value than all their neighbors. For this reason, we monitor how the nodes/edges in the leaf subgraphs form their



(a) k-core on FB (b) k-core on WND

(c) k-truss on FB (d) k-truss on SSE

Figure 10: *Evolution of densest subgraphs (leaves). Each line shows the evolution of a leaf. We limit to subgraphs with at least 10 vertices to filter out the trivial ones. Almost all leaves are captured in the first few iterations.*

max-cores/max-trusses during the convergence process. In the k-core decomposition; for a given leaf subgraph L, we find the max-cores, M_v^i, of all $v \in L$ at each iteration i with respect to the τ_i values. Then we measure the F1 score between each M_v^i and L, and report the averages for each leaf L in iteration i, i.e., $\sum_{v \in L} M_v^i/|L|$. We follow a similar way for the k-truss case; find the max-trusses for all edges in each leaf, track their F1 scores during convergence with respect to the leaf, and report the averages.

Figure 10 presents the results for a representative set of graphs (similar trends observed for other graphs). Each line shows the evolution of a leaf subgraph during the convergence process and we only considered the subgraphs with at least 10 vertices to filter out the trivial ones. We observe that almost all leaves are captured in the first few iterations. Regarding the `facebook` network, it has 3 leaves in the k-core case and all can be found in 7 iterations, and 5 iterations are enough to identify 28 of 33 leaves in k-truss decomposition. This trend is also observed for the other graphs; 5 iterations find 78 of the 85 leaves in k-core decomposition of `web-NotreDame`, and 39 of the 42 leaves in k-truss decomposition of `soc-sign-epinions`.

5.1.4 Predicting convergence

Number of iterations for convergence depends on the graph structure, as shown in Table 3. We cannot know whether a particular r-clique has converged by tracking the stability of its τ index since there can be temporary plateaus (see Figure 7). However, we know which r-cliques are active or idle in each iteration thanks to the notification mechanism in AND algorithm. We propose using the ratio of active r-cliques as an indicator.

Table 4: *Impact of the notification mechanism.* AND-*nn does not use notifications. Using 24 threads, notification mechanism yields speedups up to* 3.98 *and* 3.16 *for* k*-truss and* $(3,4)$ *cases.*

(seconds)	k-truss			$(3,4)$		
Graphs	AND-nn	AND	*Speedup*	AND-nn	AND	*Speedup*
FB	0.45	0.35	1.29	34.4	22.2	1.55
TW	3.89	2.23	1.74	178.7	59.6	3.00
SSE	2.50	1.46	1.72	105.5	49.6	2.13
WGO	3.15	1.25	2.52	25.7	16.9	1.53
WND	2.38	0.60	3.98	220.5	69.8	3.16

We examine the relation between the ratio of active r-cliques and the accuracy ratios of the r-cliques. Figure 11 presents the results for a set of graphs on all decompositions. We observe that when the ratio of active r-cliques goes below 40% during the computation, 91%, 89%, and 92% accurate results are obtained for k-core, k-truss, and $(3,4)$ nucleus decompositions, on average. When the ratio goes below 10%, over 98% accuracy is achieved in all decompositions. The results show the ratio of active r-cliques is a helpful guide to find almost-exact results can be obtained faster. Watching for 10 or 40% of active r-cliques yields nice trade-offs between runtime and quality. Watching the 40% threshold provides 3.67, 4.71, and 4.98 speedups with respect to full computation in k-core, k-truss, and $(3,4)$ nucleus decompositions, respectively, and the speedups for 10% threshold are 2.26, 2.81, and 3.25 (more details in Section 5.2.3).

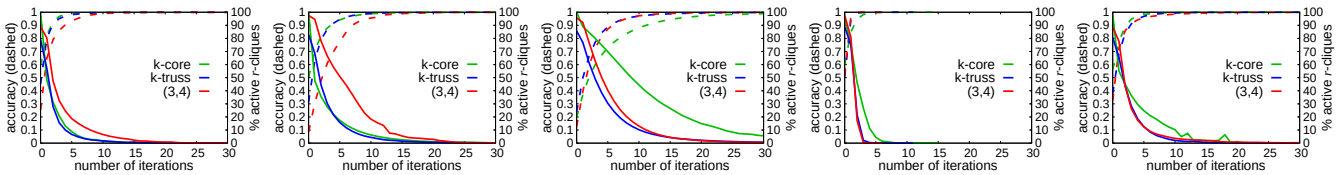
5.2 Runtime performance

We evaluate the performance of our algorithms and seek to answer the following questions:

- What is the impact of the notification mechanism (in Section 4.3) on AND algorithm?
- How does the AND algorithm scale with more threads? How does it compare to sequential peeling?
- What speedups are achieved when a certain amount of accuracy is sacrificed?

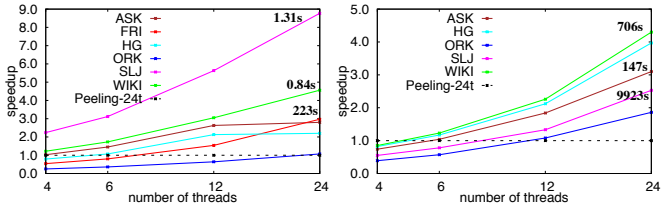
5.2.1 Impact of the notification mechanism

We check the impact of the notification mechanism for k-truss and $(3,4)$ cases. We use 24 threads and Table 4 presents the results where AND is the entire Algorithm 3 with notification mechanism and AND-nn does not have the notifications – missing the orange lines in Algorithm 3. We observe that Algorithm 3 brings great improvements, reaching up to 3.98 and 3.16 over speedups over AND-nn for k-truss and $(3,4)$ cases. We use AND algorithm (with notification mechanism) in the rest of the experiments.



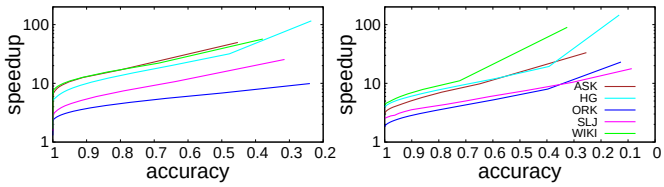
(a) `as-skitter` (b) `soc-LiveJournal` (c) `soc-orkut` (d) `web-Google` (e) `wiki-200611`

Figure 11: *Changes in the ratio of active* r*-cliques and the accuracy of* τ *indices during the computation, When the ratio of active* r*-cliques goes below* 40%*,* τ *indices provide* 91%, 89% *and* 92% *accurate results on average for* k*-core,* k*-truss and* $(3,4)$ *nucleus decompositions, respectively. If the ratio is below* 10%*, more than* 98% *accuracy is achieved in all decompositions.*



(a) k-core (b) $(3,4)$

Figure 12: *Speedups of the parallel computations with respect to the peeling computations (with 24 threads) for the k-core and $(3,4)$ nucleus decompositions. We used $4, 6, 12$, and 24 threads where each thread is assigned to a single core. On average, k-core computations are performed $3.83x$ faster when the number of threads goes from 4 to 24. This increase is $4.7x$ for $(3,4)$ case. Runtimes with 24 threads are annotated for some graphs. Speedup numbers increase with more threads and faster solutions are possible with more cores.*



(a) k-truss (b) $(3,4)$-nucleus

Figure 13: *Runtime/accuracy tradeoff. We show the potential for speedups with respect to the peeling computations for the k-truss and $(3,4)$ nucleus decompositions. Speedups at full accuracy correspond to the speedups with 24 threads in Figure 12. Number of iterations (and accuracy) decrease on the x-axis. We reach up to $15x$ and $9x$ speedups on k-truss and $(3,4)$ cases when 0.8 accuracy is allowed.*

5.2.2 Scalability and comparison with peeling

Given the benefit of notification mechanism, we now compare the runtime performances of AND (Algorithm 3) and the peeling process (Algorithm 1) on three decompositions. Our machine has 24 cores in total (12 in each socket) and we perform full computations until convergence with 4, 6, 12, and 24 threads. Note that our implementations for the baseline peeling algorithms are efficient; for instance [46] computes the truss decomposition of `as-skitter` graph in 281 secs where we can do it in 74 secs, without any parallelization. In addition, for `soc-orkut` and `soc-LiveJournal` graphs, we compute the truss decompositions in 352 and 81 secs whereas [18] needs 2291 and 1176 secs (testbeds in [46] and [18] are similar to ours). For the k-truss and $(3,4)$ nucleus decompositions, triangle counts per edge and four-clique counts per triangle need to be computed and we parallelize these parts for both the peeling algorithms and AND, for a fair comparison. Rest of the peeling computation is sequential. Figure 12 and Figure 1 present the speedups by AND algorithm over the (partially parallel) peeling computation **with 24 threads** on k-core, k-truss, and $(3,4)$ nucleus decompositions. For all, AND with 24 threads obtains significant speedups over the peeling computation. In particular, with 24 threads AND is 8.77x faster for the k-core case on the `soc-LiveJournal`, 6.3x faster for the k-truss decomposition on `as-skitter`, and 4.3x faster for the $(3,4)$ nucleus case on `wiki-200611` graph. In addition, our speedup numbers increase by more threads. On average, k-core computations are performed 3.83x faster when the number of threads are increased from 4 to 24. This increase is 4.8x and 4.7x for k-truss and $(3,4)$ cases. Our speedup numbers increase with more threads and *faster solutions are possible with more cores.*

Recent results: There is a couple recent studies, concurrent to our work, that introduced new efficient parallel algorithms for k-core [8] and k-truss [41, 45, 22] decompositions. Dhulipala et al. [8] have a new parallel bucket data structure for k-core decomposition that enables work-efficient parallelism, which is not possible with our algorithms. They present speedups to 23.6x on `friendster` graph with 72 threads. Regarding the k-truss decomposition, the HPEC challenge [35] attracted interesting studies that parallelize the computation [41, 45, 22]. In particular, Shaden et al. [41] reports competitive results with respect to the earlier version of our work [36]. Note that **our main focus in this work is a generic framework that enables local computation for k-core, k-truss, and $(3,4)$ nucleus decompositions**, which has not been discussed in the previous works. Although our algorithms are not work-efficient and more specialized solutions can give better speedups, our local algorithms are more generally applicable, enable trade-offs between runtime and accuracy, and also enable query-driven scenarios that can be used to analyze smaller subgraphs.

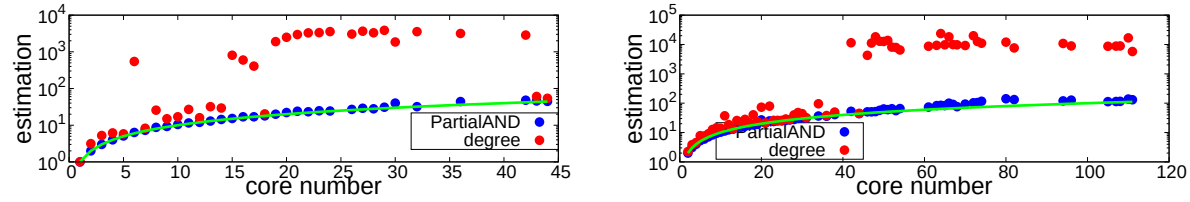
5.2.3 Runtime and accuracy trade-off

We check the speedups for the approximate decompositions in the intermediate steps during the convergence. We show how the speedups (with respect to peeling algorithm with 24 threads) change when a certain amount of accuracy in κ_s indices is sacrificed. Figure 13 presents the behavior for k-truss and $(3,4)$ nucleus decompositions on some representative graphs. We observe that speedups for the k-truss decomposition can reach up to 15x when 0.8 accuracy is allowed. For $(3,4)$ nucleus decomposition, up to 9x speedups are observed for the same accuracy score. Overall, our local algorithms are enable to enjoy different trade-offs between the runtime and accuracy.

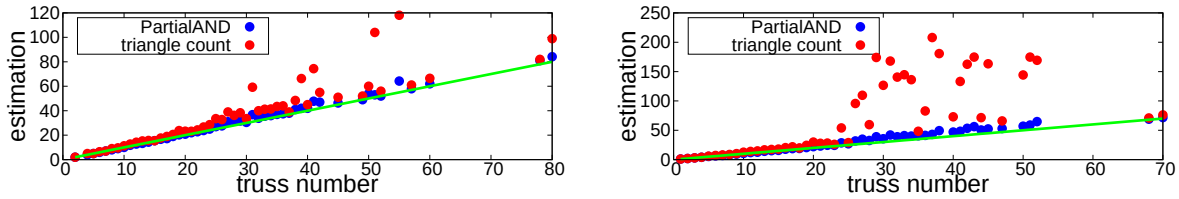
5.3 PARTIALAND to estimate κ_2 and κ_3 values

So far, we have studied the performance of our algorithms on the full graph. Now, we will look at how we can apply similar ideas to a portion of the graph using the PARTIALAND algorithm described at end of Section 4.3. We will apply PARTIALAND to the ego networks and show that it can be used to estimate κ_2 values (core number) of vertices and κ_3 values (truss number) of edges. Ego network of a vertex u is defined as the induced subgraph among u and its neighbors. It has been shown that ego networks in real-world networks exhibit low conductance [15] and also can be used for friend suggestion in online social networks [11]. Accurate and fast estimation of core numbers [29] is important in the context of network experiments (A/B testing) [43] where a random subset of vertices are exposed to a treatment and responses are analyzed to measure the impact of a new feature in online social networks.

For the core number estimation of a vertex u, we apply PARTIALAND on u and its neighbor vertices, i.e., $u \cup \mathcal{N}_2(u)$,



(a) Core number estimations for WGO and ASK



(b) Truss number estimations for TW and HG

Figure 14: *Accuracy of core and truss number estimations by* PARTIALAND. *Top two charts present the core number estimations by* PARTIALAND *and degree with respect to the ground-truth (green line) on* `web-Google` *and* `as-skitter`. *Bottom two present the truss number estimations for* `twitter` *and* `soc-twitter-higgs` *(green line is the ground-truth).* PARTIALAND *estimates the core and truss numbers accurately for a wide range. Results for other graphs are similar and omitted for brevity.*

and report $\kappa_2(u)$. Indeed, the application of PARTIALAND on ego-network for core number estimation is the same as the *propagating estimator* in [29] for distance-1 neighborhood. Here we generalize the same concept to estimate truss numbers of edges. Regarding the truss number estimations, we define the ego network of an edge e as the set of neighbor edges that participates in a common triangle ($\mathcal{N}_3(e)$). Thus, we apply PARTIALAND on $e \cup \mathcal{N}_3(e)$ and report $\kappa_3(e)$ as a truss number estimate. Figures 14a and 14b present the results for core and truss number estimations. We selected vertices/edges with varying core/truss numbers (on the x-axis) and check the accuracy of PARTIALAND estimations. Ground-truth values are shown with green lines (note that y-axes in Figure 14a are log-scale). We also show the degrees/triangle counts of the vertices/edges in red as a baseline. Overall, PARTIALAND yields almost exact estimations for a wide range of core/truss numbers. On the other hand, degree of a vertex gives a close approximation to the core number for smaller degrees, but it fails for large values. This trend is similar for the truss numbers and triangle counts.

Regarding the runtime, PARTIALAND on ego networks only takes a fraction of a second – way more efficient than computing the entire core/truss decomposition. For instance, it takes only 0.23 secs on average to estimate the core number of any vertex in the `soc-orkut` network whereas the full k-core decomposition needs 11.4 secs. It is even better in the k-truss case; PARTIALAND takes 0.017 secs on average to estimate a truss number of an edge in `soc-twitter-higgs` network where the full k-truss computation takes 73 secs.

6. RELATED WORK

Previous attempts to find the approximate core numbers (or k-cores) focus on the neighborhood of a vertex within a certain radius [29]. It is reported that if the radius is at least half of the diameter, close approximations can be obtained. However, given the small-world nature of the real-world networks, the local graph within a distance of half the diameter is too large to compute. In our work, we approximate the k-core, k-truss, and (r, s) nucleus decompositions in a rigorous and efficient way that does not depend on the diameter.

Most related study is done by Lu et al. [26], where they show that iterative h-index computation on vertices result in the core numbers. Their experiments on smaller graphs also show that h-index computation provides nice trade-offs for time and quality of the solutions. In our work, we generalized the iterative h-index computation approach for *any* nucleus decomposition that subsumes the k-core and k-truss algorithms. Furthermore, we give provable upper bounds on the number of iterations for convergence. Apart from that work, Govindan et al. [16] use the iterative h-index computation to design space-efficient algorithms for estimating core numbers. Distributed algorithms in [28] and out-of-core approaches in [23, 47, 5] also make use of similar ideas, but only for core decomposition. Montresor et al. [28] present a bound for the number of iterations, which is basically $|V|-1$, much looser than ours.

Regarding the parallel computations, Jiang et al. [20] introduced parallel algorithms to find the number of iterations needed to find the empty k-core in random hypergraphs. Their work relies on the assumption that the edge density is below a certain threshold and the focus is on the number of iterations only. Our local algorithms present an alternative formulation for the peeling process, and work for any k value. For the k-truss decomposition, Quick et al. [31] introduced algorithms for vertex-centric distributed graph processing systems. For the same setup, Shao et al. [40] proposed faster algorithms that can compute k-trusses in a distributed graph processing system. Both papers make use of the peeling-based algorithms for computation. Our focus is on the local computation where the each edge has access to only its neighbors and no global graph information is necessary, thus promise better scalability.

7. CONCLUSION

We introduced a generalization of the iterative h-index computations to identify *any* nucleus decomposition and prove convergence bounds. Our local algorithms are highly parallel and can provide fast approximations to explore time and quality trade-offs. Experimental evaluation on real-world networks exhibits the efficiency, scalability, and effectiveness of our algorithms for three decompositions. We believe that our local algorithms will be beneficial for many real-world applications that work in challenging setups. For example, shared-nothing systems can leverage the local computation.

8. REFERENCES

[1] N. K. Ahmed, J. Neville, R. A. Rossi, and N. G. Duffield. Efficient graphlet counting for large networks. In *IEEE International Conference on Data Mining, ICDM*, pages 1–10, 2015.

[2] A. Angel, N. Sarkas, N. Koudas, and D. Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *PVLDB*, 5(6):574–585, 2012.

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

[4] A. R. Benson, D. F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[5] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *IEEE International Conference on Data Engineering , ICDE*, pages 51–62, 2011.

[6] J. Cohen. Trusses: Cohesive subgraphs for social network analysis. Technical report, National Security Agency Technical Report, Fort Meade, MD, 2008.

[7] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[8] L. Dhulipala, G. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures ,SPAA*, pages 293–304, 2017.

[9] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *International Conference on World Wide Web, WWW*, pages 461–470, 2007.

[10] X. Du, R. Jin, L. Ding, V. E. Lee, and J. H. T. Jr. Migration motif: a spatial-temporal pattern mining approach for financial markets. In *ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pages 1135–1144, 2009.

[11] A. Epasto, S. Lattanzi, V. Mirrokni, I. O. Sebe, A. Taei, and S. Verma. Ego-net community mining applied to friend suggestion. *PVLDB*, 9(4):324–335, 2015.

[12] E. Fratkin, B. T. Naughton, D. L. Brutlag, and S. Batzoglou. Motifcut: regulatory motifs finding with maximum density subgraphs. *Bioinformatics*, 22(14):e150–e157, 2006.

[13] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *International Conference on Very Large Data Bases, VLDB*, pages 721–732, 2005.

[14] A. Gionis, F. Junqueira, V. Leroy, M. Serafini, and I. Weber. Piggybacking on social networks. *PVLDB*, 6(6):409–420, 2013.

[15] D. F. Gleich and C. Seshadhri. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 597–605, 2012.

[16] P. Govindan, S. Soundarajan, T. Eliassi-Rad, and C. Faloutsos. Nimblecore: A space-efficient external memory algorithm for estimating core numbers. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM*, pages 207–214, 2016.

[17] J. E. Hirsch. An index to quantify an individual's scientific research output. *Proceedings of the National Academy of Sciences of the United States of America*, 102(46):16569–16572, 2005.

[18] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *ACM SIGMOD International Conference on Management of Data*, pages 1311–1322, 2014.

[19] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *International Conference on World Wide Web, WWW*, pages 495–505, 2015.

[20] J. Jiang, M. Mitzenmacher, and J. Thaler. Parallel peeling algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 319–330, 2014.

[21] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *ACM SIGMOD International Conference on Management of Data*, pages 813–826, 2009.

[22] H. Kabir and K. Madduri. Shared-memory graph truss decomposition. In *IEEE International Conference on High Performance Computing, HiPC*, pages 13–22, 2017.

[23] W. Khaouid, M. Barsky, S. Venkatesh, and A. Thomo. K-core decomposition of large networks on a single PC. *PVLDB*, 9(1):13–23, 2015.

[24] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *International Conference on World Wide Web, WWW*, pages 1481–1493, 1999.

[25] J. Leskovec and A. Krevl. SNAP Datasets, June 2014.

[26] 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*, 7:10168, 2016.

[27] D. W. Matula and L. L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, July 1983.

[28] A. Montresor, F. D. Pellegrini, and D. Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.

[29] M. P. O'Brien and B. D. Sullivan. Locally estimating core numbers. In *IEEE International Conference on Data Mining, ICDM*, pages 460–469, 2014.

[30] A. Pinar, C. Seshadhri, and V. Vishal. Escape: Efficiently counting all 5-vertex subgraphs. In *International Conference on World Wide Web, WWW*, pages 1431–1440, 2017.

[31] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM*, pages 457–463, 2012.

[32] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI Conference on Artificial*

Intelligence, pages 4292–4293, 2015.

[33] R. A. Rossi, R. Zhou, and N. K. Ahmed. Estimation of graphlet statistics. *CoRR*, abs/1701.01772, 2017.

[34] K. Saito and T. Yamada. Extracting communities from complex networks by the k-dense method. In *IEEE International Conference on Data Mining Workshops, ICDMW*, pages 300–304, 2006.

[35] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner. Static graph challenge: Subgraph isomorphism. In *IEEE High Performance Extreme Computing Conference, HPEC*, 2017.

[36] A. E. Sariyüce, C. Seshadhri, and A. Pinar. Parallel local algorithms for core, truss, and nucleus decompositions. *CoRR*, abs/1704.00386, 2017.

[37] A. E. Sarıyüce, C. Seshadhri, A. Pınar, and Ü. V. Çatalyürek. Finding the hierarchy of dense subgraphs using nucleus decompositions. In *International Conference on World Wide Web, WWW*, pages 927–937, 2015.

[38] A. E. Sarıyüce, C. Seshadhri, A. Pınar, and Ü. V. Çatalyürek. Nucleus decompositions for identifying hierarchy of dense subgraphs. *ACM Transactions on Web*, 11(3):16:1–16:27, 2017.

[39] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[40] Y. Shao, L. Chen, and B. Cui. Efficient cohesive subgraphs detection in parallel. In *ACM SIGMOD International Conference on Management of Data*, pages 613–624, 2014.

[41] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis. Truss decomposition on shared-memory parallel systems. In *IEEE High Performance Extreme Computing Conference, HPEC*, pages 1–6, 2017.

[42] C. Tsourakakis. The k-clique densest subgraph problem. In *International Conference on World Wide Web, WWW*, pages 1122–1132, 2015.

[43] J. Ugander, B. Karrer, L. Backstrom, and J. Kleinberg. Graph cluster randomization: Network exposure to multiple universes. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 329–337, 2013.

[44] A. Verma and S. Butenko. Network clustering via clique relaxations: A community based. *Graph Partitioning and Graph Clustering*, 588:129, 2013.

[45] C. Voegele, Y. Lu, S. Pai, and K. Pingali. Parallel triangle counting and k-truss identification using graph-centric methods. In *IEEE High Performance Extreme Computing Conference, HPEC*, pages 1–7, 2017.

[46] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[47] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. Yu. I/o efficient core graph decomposition at web scale. In *IEEE International Conference on Data Engineering, ICDE*, pages 133–144, 2016.

[48] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *IEEE International Conference on Data Engineering, ICDE*, pages 1049–1060, 2012.