Parallel Algorithms for Hierarchical Nucleus Decomposition

Anonymous Author(s)

ABSTRACT

Nucleus decompositions have been shown to be a useful tool for finding dense subgraphs. The coreness value of a clique represents its density based on the number of other cliques it is adjacent to. One useful output of nucleus decomposition is to generate a hierarchy among dense subgraphs at different resolutions. However, existing parallel algorithms for nucleus decomposition do not generate this hierarchy, and only compute the coreness values. This paper presents a scalable parallel algorithm for hierarchy construction, with practical optimizations, such as interleaving the coreness computation with hierarchy construction and using a concurrent union-find data structure in an innovative way to generate the hierarchy. We also introduce a parallel approximation algorithm for nucleus decomposition, which achieves much lower span in theory and better performance in practice. We prove strong theoretical bounds on the work and span (parallel time) of our algorithms.

On a 30-core machine with two-way hyper-threading on real-world graphs, our parallel hierarchy construction algorithm achieves up to a 58.84x speedup over the state-of-the-art sequential hierarchy construction algorithm by Sariyüce et al. and up to a 30.96x self-relative parallel speedup. On the same machine, our approximation algorithm achieves a 3.3x speedup over our exact algorithm, while generating coreness estimates with a multiplicative error of 1.33x on average.

1 INTRODUCTION

Dense subgraph and substructure detection is a fundamental tool in graph mining, with many applications in areas including social network analysis [13, 60], fraud detection [24], and computational biology [3, 21]. The recent work of Sariyüce *et al.* [52] introduced the *nucleus decomposition problem*, a generalization of the k-core [45, 53] and k-truss [12] problems which better captures higher-order structures in graphs. In this problem, a c-(r, s) nucleus is defined to be the maximal induced subgraph such that every r-clique in the subgraph is contained in at least c s-cliques. The goal of the (r, s) nucleus decomposition problem is to (1) identify for each r-clique in the graph, the largest c such that it is in a c-(r, s) nucleus (known as the coreness value) and (2) generate a *nucleus hierarchy* over the nuclei, where for $c' < c$ a c'-(r, s) nucleus A is a descendant of a c-(r, s) nucleus B if A is a subgraph of B. Figure 1 shows the hierarchy for $(1, 2)$ nucleus. The nucleus hierarchy is an unsupervised method for revealing dense substructures at different resolutions in the graph. Since the hierarchy is a tree, it is easy to visualize and explore as part of structural graph analysis tasks [49].

Sariyüce and Pinar present the first algorithm for solving the nucleus decomposition problem [49]. However, their algorithm is sequential, and in order to scale to the large graph sizes of today, it is important to design parallel algorithms that take advantage of modern parallel hardware. The existing parallel algorithms for nucleus decomposition include those by Sariyüce *et al.* [51] and by Shi *et al.* [55], but these algorithms only compute the coreness values. Importantly, they do not generate the hierarchy, which limits their applicability and which is non-trivial to generate in parallel. In this paper, we design the first work-efficient parallel algorithm for hierarchy construction in nucleus decomposition, where the work, or the total number of operations, matches the best sequential time complexity. Our algorithm runs in $O(m\alpha^{s-2})$ expected work and $O(k \log n + \rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p. (parallel time), where α is the arboricity of the input graph G, $\rho_{(r,s)}(G)$ is the (r, s) peeling complexity of G, and k is the maximum (r, s)-clique core number in G. The key to our theoretical efficiency is our careful construction of subgraphs representing the s-clique-connectivity of r-cliques, that allows us to exploit linear-work graph connectivity instead of more expensive union-finds as used in prior work. Our approach gives as a byproduct the most theoretically-efficient serial algorithm for computing the hierarchy, improving upon the previous best known serial bounds by Sariyüce and Pinar [49].

We also present a practical parallel algorithm that interleaves the hierarchy construction with the computation of the coreness values. Prior work by Sariyüce and Pinar [49] also includes a (serial) interleaved hierarchy algorithm. However, their algorithm requires storing all adjacent r-cliques with different coreness values, which could potentially be proportional to the number of s-cliques in G (i.e., use $O(m\alpha^{s-2})$ space), and which results in sequential dependencies in their post-processing step to construct the hierarchy. Our parallel algorithm fully interleaves the hierarchy construction with the coreness computation, and uses only two additional arrays of size proportional to the number of r-cliques in G. Our algorithm uses a concurrent union-find data structure in an innovative way. Also, our post-processing step to construct the hierarchy tree is fully parallel. Our main insight is a technique to fully extract the connectivity information from adjacent r-cliques with different core numbers while computing the coreness values.

Note that the span of the above algorithms can be large for graphs with large peeling complexity ($\rho_{(r,s)}(G)$). We introduce an approximate algorithm for nucleus decomposition and show that it can achieve work efficiency and polylogarithmic span. Our algorithm relaxes the peeling order by allowing all r-cliques within a $(\binom{s}{r} + \epsilon)$ factor of the current value of k to be peeled in parallel. We show that our algorithm generates coreness estimates that are an $(\binom{s}{r} + \epsilon)$-approximation of the true coreness values.

We experimentally study our parallel algorithms on real-world graphs using different (r, s) values, for up to $r < s \leq 7$. On a 30-core machine with two-way hyper-threading, our exact algorithm which generates both the coreness numbers and the hierarchy achieves up to a 30.96x self-relative parallel speedup, and a 3.76–58.84x speedup over the state-of-the-art sequential algorithm by Sariyüce and Pinar [49]. In addition, we show that on the same machine our approximate algorithm is up to 3.3x faster than our exact algorithm for computing coreness values, while generating coreness estimates with a multiplicative error of 1–2.92x (with a median error of 1.33x). Our algorithms are able to compute the (r, s) nucleus decomposition hierarchy for $r > 3$ and $s > 4$ on graphs with over a hundred million edges for the first time.

We summarize our contributions below:

- The first exact and approximate parallel algorithms for nucleus decomposition that generates the hierarchy with strong theoretical bounds on the work, span, and approximation guarantees.
- A number of practical optimizations that lead to fast implementations of our algorithms.
- Experiments showing that our new exact algorithm achieves 3.7–58.8x speedups over state of the art on a 30-core machine with two-way hyper-threading.
- Experiments showing that our new approximate algorithm achieves up to 3.3x speedups over our exact algorithm, while generating coreness estimates with a multiplicative error of 1–2.9x (with a median error of 1.3x).

Our anonymized code is available at: https://anonymous.4open.science/r/arb-nucleus-hierarchy-6522/.

2 RELATED WORK

The k-core [45, 53] and k-truss problems [12, 64, 67] are classic problems that relate to dense substructures in graphs. Many algorithms have been developed for k-cores and k-trusses in the static [5, 8, 9, 14, 16, 17, 20, 23, 33–35, 42, 46, 57, 61, 63, 68] and dynamic [1, 2, 26–28, 32, 38, 39, 41, 43, 44, 48, 58, 63, 65, 66] settings. Similar ideas have been studied in bipartite graphs [29, 36, 40, 50, 56, 62].

A related problem is the k-clique densest subgraph problem [59], which defines the density of subgraphs based on the number of k-cliques in it, rather than the number of edges. Fang *et al.* [19] further generalize this notion to arbitrary fixed-sized subgraphs (although they only present experiments for cliques). Similar to k-core and k-truss, algorithms for approximating the k-clique densest subgraph are based on iteratively peeling (removing) elements from the graph. Shi *et al.* [54] present efficient parallel algorithms for solving the k-clique densest subgraph problem.

Sariyüce *et al.* define the (r, s) nucleus decomposition problem and show that it can be used to find higher quality dense substructures in graphs that previous approaches. They provide efficient sequential [49, 52] and parallel algorithms [51] for this problem. Sariyüce *et al.* [51] present two parallel algorithms for the nucleus decomposition problem: (1) a global peeling-based algorithm and (2) a local update model that iterates until convergence. They also introduce a sequential algorithm for constructing the nucleus decomposition hierarchy [49], but as far as we know there are no parallel algorithms for constructing this hierarchy. Their algorithm is not work-efficient as it uses union-find. We also note that their space-usage can in theory be as large as $O(n\alpha^{s-2})$, i.e., proportional to the number of s-cliques in the graph. Chu *et al.* [11] present a parallel algorithm for generating the k-core decomposition hierarchy, which is a special case of nucleus decomposition for $r = 1$ and $s = 2$, but their algorithm does not trivially generalize to higher r and s. Their serial and parallel algorithms both use union-find and run in $O(m\alpha(n))$ work (where $\alpha(n)$ is the inverse Ackermann function), which is not work-efficient, and their parallel algorithm has depth that depends on the peeling-complexity of the input. Shi *et al.* [55] present an improved parallel algorithm for nucleus decomposition, with good theoretical guarantees and practical performance, but it does not generate the hierarchy. Their algorithm is work-efficient, in that it runs in work proportional to enumerating all s-cliques, i.e., $O(m\alpha^{s-2})$ work. They leverage a work-efficient parallel clique counting algorithm [54], along with a multi-level hash table structure to store data associated with cliques space efficiently, and techniques for traversing this structure in a cache-friendly manner. More recently, Sariyüce generalizes the r-cliques and s-cliques in nucleus decomposition to any pair of subgraphs [47]. There has also been work on nucleus decomposition in probabilistic graphs [18].

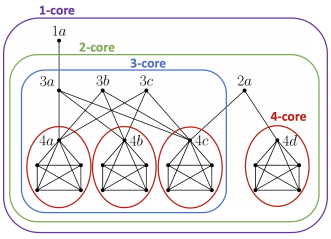

Figure 1: An example of the $(1, 2)$ nucleus (k-core) hierarchy.

3 PRELIMINARIES

Model of Computation. We use the work-span model of computation for our theoretical analysis, which is a standard model for analyzing shared-memory parallel algorithms [15, 30]. The ***work*** W of an algorithm is the total number of operations, and the ***span*** (parallel time) S of an algorithm is the longest dependency path. Using a randomized work-stealing scheduler [7], we can obtain a running time of $W/P + O(S)$ in expectation. Our goal is to develop ***work-efficient*** parallel algorithms under this model, or algorithms with a work complexity that asymptotically matches the best-known sequential time complexity for the given problem.

Graph Notation and Definitions. We consider simple and undirected graphs $G = (V, E)$. We let $n = |V|$ and $m = |E|$. For analysis, we assume $m = \Omega(n)$. The ***arboricity*** (α) of a graph is the minimum number of spanning forests needed to cover the graph. α is upper bounded by $O(\sqrt{m})$ and lower bounded by $\Omega(1)$ [10]. An ***k-clique*** is a set of vertices such that all $\binom{k}{2}$ edges exist among them. Enumerating k-cliques can be done in $O(m\alpha^{k-2})$ work and $O(\log^2 n)$ span with high probability (w.h.p.) [54].[1]

A c-(r, s) ***nucleus*** is a maximal subgraph H of an undirected graph formed by the union of s-cliques S, such that each r-clique R in H has induced s-clique degree at least c (i.e., each r-clique is contained within at least c induced s-cliques). The goal of the (r, s) ***nucleus decomposition problem*** is to compute the following: (1) the (r, s)-***clique core number*** of each r-clique R, or the maximum c such that R is contained within a c-(r, s) nucleus and (2) a hierarchy over the nuclei, where for $c' < c$, a c'-(r, s) nucleus A is a descendant of a c-(r, s) nucleus B if A is a subgraph of B. In this paper, similar to all prior work on nucleus computations, we take r and s to be constants and ignore constants depending on these values in our bounds. The k-core and k-truss problems correspond to the k-$(1, 2)$ and k-$(2, 3)$ nucleus, respectively. Figure 1 shows the hierarchy for $(1, 2)$ nucleus (k-core). We call two r-cliques ***s-clique-adjacent*** if there exists an s-clique S such that both r-cliques are subgraphs

[1] We say $O(f(n))$ ***with high probability (w.h.p.)*** to indicate $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$, where n is the input size.

of S. We also define the ***s*-clique-degree** of a r-clique R to be the number of s-cliques that R is contained within.

We also consider approximate nucleus decompositions in this paper. If the true (r, s)-clique core number of an r-clique R is k_R, then a ***γ*-approximate (r, s)-clique core number** of R is a value that is at least k_R and at most γk_R, where $\gamma > 1$.

An ***a*-orientation** of an undirected graph is a total ordering on the vertices such that when edges in the graph are directed from vertices lower in the ordering to vertices higher in the ordering, the out-degree of each vertex is bounded by a. Shi *et al.* and Besta et al. [4] provide parallel work-efficient algorithms for computing an $O(\alpha)$-orientation, which take $O(m)$ work and $O(\log^2 n)$ span.

A ***connected component*** in an undirected graph is a maximal subgraph such that all vertices in the subgraph are reachable from one another. Computing the connected components in a graph can be done in $O(m)$ work and $O(\log n)$ span w.h.p. [22].

A ***union-find*** data structure is used to represent collections of sets and supports the following two operations: ***unite(x,y)*** joins the sets that contain x and y and ***find(x)*** returns the identifier of the set containing x. Trees are used to implement union-find data structures, where elements have a parent pointer, and the root of a tree corresponds to the representative of the set. Path compression can be done during unite and find operations to shortcut the pointers of elements traversed to point to the root of the set. We use the concurrent union-find data structure by Jayanti et al. [31].

We use the notation $[n]$ to refer to the range of integers $[1, \ldots, n]$.

Parallel Primitives. We use the following parallel primitives in our algorithms. We use ***parallel hash tables***, which support n insertion, deletion, and membership queries, in $O(n)$ work and $O(\log n)$ span w.h.p. [25]. ***List ranking*** takes a linked list as input and returns the distance of each element to the end of the linked list. For a linked list of n elements, list ranking can be solved in $O(n)$ work and $O(\log n)$ span [30]. In our algorithms, we use list ranking to compute a unique identifier for each element in a linked list so that we can write them to an array in parallel.

Compare-and-swap(x, old, new) attempts to atomically write the value new into memory location x if x current stores the value old; it returns true if the write succeeds and false otherwise (meaning the value of x was changed by another thread after it was read into old). We assume compare-and-swaps take $O(1)$ work and span.

A ***parallel bucketing structure*** maintains a mapping from identifiers to buckets, which we use to group r-cliques according to their incident s-clique counts [16]. The buckets can change, and the structure can efficiently update these buckets. We take identifiers to be values associated with r-cliques, and use the structure to repeatedly extract all r-cliques in the minimum bucket, which can cause the buckets of other r-cliques to change (other r-cliques sharing vertices with extracted r-cliques in our algorithm).

4 OVERVIEW OF CONTRIBUTIONS

We present several novel algorithms for efficient parallel hierarchy construction for (r, s) nucleus decomposition. The prior state-of-the-art work on parallel (r, s) nucleus decomposition, ARB-NUCLEUS [55], only computes the (r, s)-clique core numbers and does not construct the hierarchy, which is non-trivial to accomplish. We first present in Section 5 a new theoretically efficient parallel hierarchy construction algorithm, ARB-NUCLEUS-HIERARCHY, that given the (r, s)-clique core numbers from [55], constructs the full hierarchy. In this two-phase algorithm, the hierarchy construction is entirely separate from the computation of the core numbers. The main innovation is a clever technique to store subgraphs corresponding to each core, such that the each subgraph only needs to be materialized when processing the corresponding level in the hierarchy tree. This limits the number of times we must iterate over each r-clique, and allows us to use a linear-work connectivity subroutine, giving us our theoretically efficient bounds.

We also present in Section 6 a new parallel approximation algorithm, APPROX-ARB-NUCLEUS, that computes approximate (r, s)-clique core numbers in polylogarithmic span without increasing the work. Notably, our parallel approximation algorithm is the first to achieve the dual guarantees of work-efficiency and polylogarithmic span. We then combine APPROX-ARB-NUCLEUS with our theoretically efficient hierarchy construction subroutine, to form an approximate hierarchy algorithm, ARB-APPROX-NUCLEUS-HIERARCHY.

We observe that it is often not practically efficient to conduct a two-phase hierarchy algorithm (separating the hierarchy construction from the computation of the core numbers), and it is instead faster to construct the hierarchy directly while computing the core numbers. However, this presents a new series of challenges, where core numbers obtained later in the algorithm given by [55] may affect all levels of the hierarchy tree, causing significant global changes in the tree structure as core numbers are computed. We present practical solutions in Section 7 limiting the cascading effects of these changes by compactly storing the tree using two simple data structures, a union-find structure and a hash table, and grouping r-cliques on-the-fly to reduce the propagating changes. We then introduce an efficient parallel post-processing step, to explicitly construct the final hierarchy tree from the two data structures.

Finally, we implement all of our algorithms, and present a comprehensive experimental evaluation in Section 8.

5 NUCLEUS DECOMPOSITION HIERARCHY

We now describe our theoretically efficient parallel nucleus decomposition hierarchy algorithm, ARB-NUCLEUS-HIERARCHY. ARB-NUCLEUS-HIERARCHY computes the hierarchy by first running an efficient parallel nucleus decomposition algorithm, ARB-NUCLEUS [55], in order to obtain the (r, s)-clique core numbers corresponding to each r-clique. It then constructs a data structure consisting of k levels, where k is the maximum (r, s)-clique core number. Each level is represented by a hash table mapping sets of r-cliques to a linked list of r-cliques, which is used to efficiently store s-clique-adjacent r-cliques. ARB-NUCLEUS-HIERARCHY proceeds in levels through this data structure to construct the hierarchy tree from the bottom up, by performing linear-work connectivity on each level and updating the connectivity information in prior levels as a result.

Our Algorithm. We now provide a more detailed description of our algorithm. The pseudocode is in Algorithm 1. Note that we have highlighted in blue the parts that derive directly from prior work; the remaining pseudocode is novel to this paper. We refer to the example of $(r, s) = (1, 2)$ nucleus (k-core) decomposition in Figure 1. For simplicity, we have omitted labeling the other vertices

Algorithm 1 Parallel (r, s) nucleus decomposition hierarchy algorithm

1: Initialize r, s ▷ r and s for (r, s) nucleus decomposition
2: **procedure** ARB-NUCLEUS-HIERARCHY($G = (V, E)$)
3: $ND \leftarrow$ ARB-NUCLEUS(G) ▷ Compute the nucleus core numbers, where ND maps r-cliques to their core numbers
4: $k \leftarrow$ maximum core number in ND
5: For each $i \in [k]$, let L_i denote a hash table, where the keys are r-cliques and the values are linked lists
6: **parfor** all s-cliques S in G **do**
7: **parfor** all pairs of r-cliques R, R' in S where $ND[R'] \leq ND[R]$ **do**
8: Add R' to the linked list keyed by R in $L_{ND[R']}$
9: Initialize the hierarchy tree T with leaves corresponding to each r-clique
10: For each $i \in [k]$, let ID_i denote a hash table, where the keys are r-cliques and the values are r-cliques
11: Initialize each ID_i to contain each key in L_i, mapped to itself
12: **for** $i \in \{k, k-1, \ldots, 1\}$ **do**
13: Relabel each r-clique in each linked list in L_i with its corresponding value in ID_i
14: Apply parallel list ranking to transform the linked lists in L_i to arrays, which forms a graph H (where the edges are given by each key paired with each element in its corresponding linked list)
15: Run parallel linear-work connectivity on H
16: **parfor** each connected component $C = \{R_1, \ldots, R_c\}$ in H where $|C| \geq 2$ **do**
17: Construct a new parent in T for all of the nodes corresponding to each R_ℓ (for $\ell \in [c]$), and represent the parent as the r-clique R_1
18: **parfor** each $j \leq i$ **do**
19: Concatenate in parallel the linked lists corresponding to all R_ℓ (for $\ell \in [c]$) in L_j, as the updated value for the key R_1 in L_j
20: For each R_ℓ (for $\ell \in [c]$), update the value of R_ℓ in ID_j to be R_1
21: **return** T

L_1 =

key	3a
value	{1a}

, L_2 =

key	4c	4d
value	{2a}	{2a}

L_3 =

key	4a	4b	4c
value	{3a, 3b, 3c}	{3a, 3b, 3c}	{3a, 3b, 3c}

, $L_4 = \emptyset$

***i* = 4**:
Connected component = $\emptyset$

***i* = 3**:

ID_3 =

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
value	1a	2a	3a	3b	3c	4a	4b	4c	4d

Connected component = $\{3a, 3b, 3c, 4a, 4b, 4c\}$

***i* = 2**:

ID_2 =

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
value	1a	2a	3a	3a	3a	3a	3a	3a	4d

Connected component = $\{2a, 3a, 4d\}$

***i* = 1**:

ID_1 =

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
value	1a	2a	2a	3a	3a	3a	3a	3a	2a

Connected component = $\{1a, 2a\}$

Figure 2: An example of the L_i data structures maintained by ARB-NUCLEUS-HIERARCHY while computing the k-core hierarchy on the graph in Figure 1. For each round i of the hierarchy construction, the connected components of H and the ID_i table used to construct H is shown (except ID_4, which maps every r-clique to itself).

participating in the 5-cliques represented by $4a$, $4b$, $4c$, and $4d$. The number in front of each vertex label is its core number.

ARB-NUCLEUS-HIERARCHY first calls ARB-NUCLEUS on Line 3, to compute the (r, s)-clique core numbers of each r-clique. Note that this is the only instance where ARB-NUCLEUS-HIERARCHY uses a subroutine from the prior work [55], and the rest of the pseudocode is novel. It stores these values in a hash table, ND, keyed by the r-cliques. Then, it constructs a data structure consisting of k hash

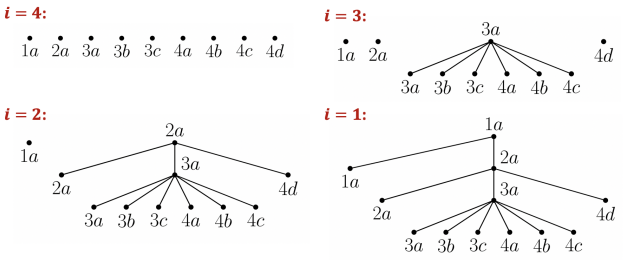

Figure 3: An example of the intermediate hierarchy trees T after each round i in ARB-NUCLEUS-HIERARCHY, while computing the k-core hierarchy on the graph in Figure 1.

tables on Line 5, where k is the maximum (r, s)-clique core number. Each hash table, L_i, maps r-cliques to a linked list of r-cliques, where i corresponds to a core number. The first stage of ARB-NUCLEUS-HIERARCHY inserts all s-clique-adjacent r-cliques into the hash tables on Lines 6–8. Specifically, for adjacent r-cliques R and R' where $ND[R'] \leq ND[R]$, we insert R' into the hash table corresponding to R''s core number, $L_{ND[R']}$, with $\{R\}$ as the key. If an entry for $\{R\}$ already exists in $L_{ND[R']}$, we append R' to the existing linked list. The hash tables are shown in Figure 2. For instance, $R' = 1a$ is adjacent to $R = 3a$, so we add $1a$ to the linked list keyed by $3a$ in L_1.

In order to iterate in parallel over all r-cliques and over all s-cliques containing each r-clique, our algorithm uses a s-clique enumeration algorithm based on previous work by Shi *et al.* [54], which recursively finds and lists c-cliques in parallel and which can efficiently extend given r-cliques to find the s-cliques they are contained within. ARB-NUCLEUS-HIERARCHY then takes all combinations of r vertices in each discovered s-clique to find adjacent r-cliques in Lines 7–8. Note that this s-clique enumeration subroutine is already used in ARB-NUCLEUS in order to compute the (r, s)-clique core numbers of each r-clique, and in practice, we can construct the hash tables L_i while computing the core numbers in ARB-NUCLEUS (rather than running the s-clique enumeration subroutine twice).

After constructing the initial set of hash tables, ARB-NUCLEUS-HIERARCHY proceeds to construct the nucleus decomposition hierarchy on Lines 9–20. The broad idea is to construct the hierarchy starting at the leaf nodes, each of which correspond to an r-clique, and to merge tree nodes from the bottom up (i.e., in decreasing order of core number). The algorithm begins by considering only connected components formed by r-cliques with the greatest core number, k, which dictate the leaf nodes to be merged into supernodes at the second-to-last level of the hierarchy tree based on their s-clique connectivity. In subsequent rounds, when processing core number i, the algorithm considers connected components formed by r-cliques with core numbers $\geq i$, which similarly dictate the merges to be performed in the next level of the hierarchy tree. ARB-NUCLEUS-HIERARCHY efficiently maintains connected components from higher core numbers when computing connected components at lower core numbers, by concatenating the relevant linked lists in the hash tables corresponding to the lower core numbers when processing higher core numbers. Figure 3 shows the construction of this hierarchy throughout the different rounds (for $i = 4, \ldots, 1$),

and Figure 2 lists the connected components computed in each round. In our example, since we omit the other vertices in each component represented by $4a$, $4b$, $4c$, and $4d$ (for simplicity), we begin with singleton leaf nodes after processing round $i = 4$.

In more detail, on Line 9, we begin with a hierarchy tree T consisting only of leaf nodes corresponding to each r-clique. We also initialize a data structure consisting of k hash tables on Lines 10–11, where each hash table ID_i maps r-cliques to r-cliques. The idea is to maintain a mapping of each r-clique to the component that it is contained within in L_i, for the corresponding level. Initially, each ID_i maps each key in L_i to itself.

Then, ARB-NUCLEUS-HIERARCHY considers each core number i, starting from k and going down to 1 (Line 12). For each i, we relabel each r-clique in each linked list in L_i with its corresponding value in ID_i (Line 13). Then, it uses parallel list ranking to convert all linked lists in L_i to arrays, forming a graph H where edges are given by each key paired with each element in its corresponding linked list (Line 14). Note that the edges actually denote s-clique-adjacent r-cliques (or components of r-cliques). The vertices correspond to r-cliques, which represent components of r-cliques in G. In our example, we process the hash table L_3 in round $i = 3$. ID_3 maps every vertex to itself, so we do not relabel any of the labels in L_3. The graph H that we construct consists of the vertices $3a$, $3b$, $3c$, $4a$, $4b$, and $4c$, with edges given by $\{3a, 3b, 3c\} \times \{4a, 4b, 4c\}$.

ARB-NUCLEUS-HIERARCHY proceeds by running an efficient parallel linear-work connectivity algorithm on H (Line 15), and processes the given connected components. Note that each connected component C consists of a set of vertices in H, which represents a set of r-cliques, say $\{R_1, \ldots, R_c\}$. In our example, in round $i = 3$, we have a single connected component in H consisting of all of the vertices, $3a$, $3b$, $3c$, $4a$, $4b$, and $4c$. Then, for each such connected component representing more than one r-clique, in the hierarchy tree T, we construct a new parent for the nodes corresponding to each R_ℓ for $\ell \in [c]$ (Line 17). Note that each R_ℓ could correspond to a subtree containing multiple tree nodes, owing to parent nodes constructed in previous steps. We represent the new parent by the r-clique R_1, which we designate arbitrarily as the representative for the connected component C. In Figure 3, under $i = 3$, we construct a new parent node labeled by $3a$, which we designate as the representative of the component $\{3a, 3b, 3c, 4a, 4b, 4c\}$.

Finally, for each core number $j < i$, we update the connectivity information on each hash table L_j, by concatenating all linked lists in L_j corresponding to the r-cliques in the component C (Line 19). More precisely, we update the value of the key R_1 to be the concatenation, or we insert the concatenation into L_j with R_1 as the key if R_1 does not already exist in the hash table (this is possible if R_1 had no neighbors with core number j). We use tombstones to delete the other keys R_ℓ ($\ell \neq 1$) in each L_j. Additionally, on Line 20, we update the component ID in ID_j of each R_ℓ to be R_1. For $i = 3$, we see that in our example there are no lists to concatenate, but in ID_2, we map each of $3b$, $3c$, $4a$, $4b$, and $4c$ to the representative $3a$.

We repeat this process until we have processed all k rounds. In our example, for round $i = 2$, we relabel $4c$ in L_2 with $3a$, as given by ID_2, and we construct a subgraph H with vertices $2a$, $3a$, and $4d$, and edges $(2a, 3a)$ and $(2a, 4d)$. Again, there is only one connected component, given by $\{2a, 3a, 4d\}$. In the hierarchy tree for round $i = 2$, we construct a new parent labeled with the representative $2a$, whose children are the leaves $2a$ and $4d$, and the previously constructed parent node $3a$. Again, there are no lists to concatenate in L_1, but in ID_1, we map both $3a$ and $4d$ to the representative $2a$. Then, for round $i = 1$, we relabel $3a$ in L_1 with $2a$, as given by ID_1. We construct a subgraph H with vertices $1a$ and $2a$, and a single edge between them. There is only one connected component, $\{1a, 2a\}$, and in the hierarchy tree for round $i = 1$, we construct a new parent labeled with the representative $1a$, whose children are the leaf $1a$ and the previously constructed parent node $2a$. This concludes the construction of the hierarchy T in our example.

Theoretical Efficiency. We now analyze the theoretical efficiency of our hierarchy algorithm, ARB-NUCLEUS-HIERARCHY. Note that as in Shi *et al.*'s [55] work, $\rho_{(r,s)}(G)$ is defined to be the ***(r, s) peeling complexity*** of G, or the number of rounds needed to peel the graph where in each round, all r-cliques with the minimum s-clique count are peeled (removed). Importantly, $k \leq \rho_{(r,s)}(G) \leq O(m\alpha^{r-2})$, since at least one r-clique is peeled in each round, and the number of rounds is at least the maximum (r, s)-clique core number in G.

THEOREM 5.1. ARB-NUCLEUS-HIERARCHY *computes the* (r, s) *nucleus decomposition hierarchy in* $O(m\alpha^{s-2})$ *expected work and* $O(k \log n + \rho_{(r,s)}(G) \log n + \log^2 n)$ *span w.h.p., where* $\rho_{(r,s)}(G)$ *is the* (r, s) *peeling complexity and* k *is the maximum* (r, s)*-clique core number.*

PROOF. First, the theoretical complexity of computing the (r, s)-clique core numbers of each r-clique (Line 3) is given by Shi *et al.* [55], which they show takes $O(m\alpha^{s-2})$ work and $O(\rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p. for constant r and s. Note that the work bound is given by the version of their algorithm that takes space proportional to the number of s-cliques in G, which we incur regardless to store the hash tables L_i. Additionally, iterating through all s-cliques in G (Line 6) is superseded by the work required to compute the (r, s)-clique core numbers. For every pair of r-cliques in each s-cliques, we hash each r-clique and append to a linked list (Lines 7–8), which in total takes $O(m\alpha^{s-2})$ work and $O(\log^2 n)$ span w.h.p. for constant r and s.

We now discuss the work and span of constructing the hierarchy tree T level-by-level, in k rounds (Lines 12–20). The key idea here is that the linked lists in each hash table L_i are iterated over at most once across all rounds, and the concatenation of linked lists in intermediate rounds incurs minimal costs (since concatenating linked lists does not require iterating through the linked lists). The sum of the lengths of the linked lists in each L_i remains invariant throughout this portion of the construction, so in total, the cost of iterating over the linked lists is bounded by $O(m\alpha^{s-2})$ work, matching the work needed to construct each linked list in each L_i originally. Also, the number of keys in each L_i only monotonically decreases, so processing the connected components of the constructed subgraphs H takes at most $O(m\alpha^{s-2})$ work as well.

In more detail, for fixed i, let the sum of the lengths of the linked lists in L_i be ℓ_i, and let the number of keys in L_i be y_i. For each round $i \in [k]$, applying ID_i and parallel list ranking on the linked lists in L_i (Lines 13–14) takes work proportional to ℓ_i and $O(\log n)$ span w.h.p.. The number of edges in the subgraph H constructed from the linked lists in L_i is at most ℓ_i, so performing connectivity on H (Line 15) takes work linear in ℓ_i and $O(\log n)$ span w.h.p. [22]. Also, there are at most $O(y_i)$ vertices in H, so processing each connected component and updating the hierarchy tree T (Line 17)

takes $O(y_i)$ work and $O(1)$ span. In total, for Lines 13–17, we incur $O(\sum_i(\ell_i + y_i)) = O(m\alpha^{s-2})$ expected work and $O(k \log n)$ span w.h.p.

It remains to bound the cost of of the loop in Lines 18–20. First, note that across all rounds, each value corresponding to every key in each L_i is concatenated at most once (Line 19). This is because we concatenate linked lists and store the concatenation under exactly one existing key. We empty the corresponding values for the previously associated keys, which allows us to maintain the invariant that the sum of the lengths of the linked lists in each L_i remains fixed. Also, the previously associated keys are never used again, since we update the value associated with each r-clique in ID_j. Thus, the concatenations are bounded by $O(m\alpha^{s-2})$ work across all rounds, matching the work of constructing all L_i. Additionally, iterating through all levels $j \leq i$ for each component C does not increase the work. This is because we only reach this loop if $|C| \geq 2$, which means that we are effectively merging multiple r-cliques together in each hash table L_j for $j \leq i$, and assigning the r-clique R_1 as the new representative for the other r-cliques in the component (updating ID_j). Thus, we can assign the work of iterating through all $j \leq i$ to the r-cliques that are being merged (R_ℓ where $\ell \neq 1$). Once the r-clique R_ℓ is merged, due to the concatenation on Line 19 and the update in ID_j on Line 20, it never participates as a vertex in H again in future rounds, so is never re-processed in a future connected component of H; this is due to the mapping on Line 13. The amount of work that we assign per merged r-clique R_ℓ is at most the core number of R_ℓ. This is because the amount of work we assign to R_ℓ is given by the number of rounds in which we merge, or i, and R_ℓ only appears in L_i if $ND[R_\ell] \geq i$, by construction on Line 8. Thus, in total, for each r-clique, we incur work upper bounded by the core number. We have in total $O(\sum_{r\text{-clique } R \in G} ND[R]) = O(m\alpha^{s-2})$ work, since the sum of the (r,s)-clique core numbers in G is necessarily bounded by the number of s-cliques for constant r and s. This is because each s-clique contributes to at most $\binom{s}{r}$ r-clique's core numbers, so the summation across all core numbers is upper bounded by $\binom{s}{r}\cdot$ (the number of s-cliques). Thus, in total, the loop in Lines 18–20 incurs $O(m\alpha^{s-2})$ work and $O(k \log n)$ span, where the span is due to list ranking.

In total, we have $O(m\alpha^{s-2})$ expected work and $O(k \log n + \rho_{(r,s)}(G) \log n + \log^2 n)$ span w.h.p., as desired. □

Comparison to Prior Work. The prior state-of-the-art algorithm is the sequential (r,s) nucleus decomposition hierarchy algorithm by Sariyüce and Pinar [49]. They provide an algorithm similar to ARB-NUCLEUS-HIERARCHY in that it first computes the (r,s)-clique core numbers of each r-clique, and then builds the hierarchy tree from the bottom up. They show that the time complexity is upper bounded by the time complexity of computing the (r,s)-clique core numbers. Note that they omit a factor of $O(\alpha(n_s, n_r))$ where α is the inverse Ackermann function and n_s and n_r are the number of s-cliques and r-cliques, respectively, in the graph; this factor is necessary for their algorithm, since they use union-find to construct the hierarchy tree. Thus, the time complexity of Sariyüce and Pinar's algorithm is $O(m\alpha^{s-2}\alpha(n_s, n_r))$ (this is achieved using the state-of-the-art algorithm for computing the (r,s)-clique core numbers [55]). Our ARB-NUCLEUS-HIERARCHY algorithm avoids the additional inverse Ackermann factor, by efficiently constructing graphs to represent different levels of the hierarchy tree and using linear-work graph connectivity to construct the hierarchy tree. Thus, we improve the sequential running time of constructing the (r,s) nucleus decomposition hierarchy to $O(m\alpha^{s-2})$, and ARB-NUCLEUS-HIERARCHY is work-efficient.

6 APPROXIMATE NUCLEUS DECOMPOSITION

Given the potentially large span (i.e., longest critical path) of computing the exact nucleus decomposition hierarchy, we develop a new parallel approximate nucleus decomposition hierarchy algorithm, ARB-APPROX-NUCLEUS-HIERARCHY. Instead of computing exact (r,s)-clique-core numbers of each r-clique, the main idea of the new algorithm is to compute an approximation of the (r,s)-clique core number. Specifically, we compute a $(\binom{s}{r}+\varepsilon)$-approximate (r,s)-clique core number of each r-clique; that is to say, if we let the true (r,s)-clique core number of each r-clique be k_R, our approximation is at least k_R and at most $(\binom{s}{r}+\varepsilon)\cdot k_R$.

Our approximate computation uses the same peeling paradigm from Shi *et al.* [55] for exact nucleus decomposition, but with an important modification that allows it to take only $O(\log^2 n)$ peeling rounds, thus significantly improving upon the span. As a result, we only have polylogarithmically many core numbers, leading to a hierarchy tree with polylogarithmic height. Our hierarchy construction for ARB-APPROX-NUCLEUS-HIERARCHY is exactly the same as that of ARB-NUCLEUS-HIERARCHY, and the only salient difference is that for ARB-APPROX-NUCLEUS-HIERARCHY, we replace the ARB-NUCLEUS subroutine in Line 3 of Algorithm 1 with our approximate nucleus decomposition subroutine, APPROX-ARB-NUCLEUS.

Our Algorithm. We present our pseudocode for APPROX-ARB-NUCLEUS in Algorithm 2. Again, we have highlighted in blue the parts that derive directly from prior work, and the remainder is novel to this work. Note that it takes as input a parameter $\delta > 0$, which controls the ε in the $(\binom{s}{r}+\varepsilon)$-approximation. Specifically, APPROX-ARB-NUCLEUS gives a $(\binom{s}{r}+\delta)\cdot(1+\delta) = (\binom{s}{r}+\varepsilon)$-approximation, which we prove in Theorem 6.3.

First, on Line 3, APPROX-ARB-NUCLEUS computes a low out-degree orientation of the graph G, which directs the edges such that every vertex has out-degree at most $O(\alpha)$, using an efficient algorithm by Shi *et al.* [54]. Then, on Lines 4–5, it counts the number of s-cliques per r-clique in G, and stores the result in a parallel hash table U. It uses an s-clique counting subroutine, REC-LIST-CLIQUES, from Shi *et al.*'s previous work [54]. The key difference between APPROX-ARB-NUCLEUS and ARB-NUCLEUS is on Line 6, where the buckets in the bucketing structure hold r-cliques with a range of s-clique-degrees instead of a single s-clique-degree. Specifically, for an input parameter δ, we define the range of each bucket B_i to be $[(\binom{s}{r}+\delta)\cdot(1+\delta)^i, (\binom{s}{r}+\delta)\cdot(1+\delta)^{i+1}]$, where $i \in [s\log_{1+\delta} n]$ since $\binom{n}{s} = O(n^s)$ is a trivial upper bound on the maximum s-clique-degree possible in any given graph.

The peeling algorithm then proceeds as it does in [55], except using our modified bucketing structure. While not all r-cliques have been peeled, APPROX-ARB-NUCLEUS processes the set of r-cliques A (that have not yet been peeled) within the lowest bucket B_i (starting with $i = 0$), and peels them from the graph (Lines 8–20). For each

Algorithm 2 Approximate parallel (r, s) nucleus decomposition algorithm

```
1: Initialize r, s                                   ▷ r and s for (r, s) nucleus decomposition
2: procedure APPROX-ARB-NUCLEUS(G = (V, E), δ)
3:     DG ← ARB-ORIENT(G)                            ▷ Apply an arboricity-orientation algorithm
4:     Initialize U to be a parallel hash table with r-cliques as keys, and s-clique
       counts as values
5:     REC-LIST-CLIQUES(DG, s, U)                    ▷ Count s-cliques, and store the counts per
       r-clique in U
6:     Let ND be a bucketing structure mapping each r-clique to a bucket based on
       # of s-cliques, where each bucket B_i contain all r-cliques with s-clique-degree in
       the range [((s choose r) + δ) · (1 + δ)^i, ((s choose r) + δ) · (1 + δ)^{i+1}], for all i ∈ [s log_{(s choose r)+δ} n]
7:     finished ← 0, num_rounds ← 0, i ← 0
8:     while finished < |U| do
9:         A ← r-cliques in the bucket B_i in ND (to be peeled)
10:        finished ← finished + |A|
11:        num_rounds ← num_rounds + 1
12:        parfor all r-cliques R in A do
13:            parfor all s-cliques S containing R do
14:                parfor all r-cliques R' in S where R' ≠ R do
15:                    Update s-clique count of R' in U
16:        Peel A and update the buckets of r-cliques with updated s-clique counts
17:        if num_rounds ≥ O(log_{1+δ/(s choose r)}(n)) or B_i is empty then
18:            Add the remaining r-cliques in B_i (if it is non-empty) to B_{i+1}
19:            i ← i + 1
20:            num_rounds ← 0
21:    return ND
```

r-clique R in A, we iterate over all s-clique-adjacent r-cliques R', and update the recorded s-clique-degree of R' given R's removal (Lines 12–15). Then, we peel (remove) the r-cliques in A from the graph and update the buckets of all unpeeled r-cliques based on the updated s-clique-degrees on Line 16. Notably, if a r-clique R's s-clique-degree falls below the range of the current bucket of r-cliques that is being peeled, we do not rebucket R into a lower bucket, and instead aggregate these r-cliques within the current bucket. As such, in any given round of peeling, we are actually peeling all r-cliques with s-clique-degree $\leq (\binom{s}{r} + \delta) \cdot (1 + \delta)^i$, which is important for our theoretical bounds. Note that we process a given bucket B_i at most $O(\log_{1+\delta/\binom{s}{r}}(n))$ times; if we have exceeded this threshold, or if B_i is empty, then we move on to processing the next bucket, B_{i+1} (Lines 17–20). If there are unpeeled r-cliques remaining in B_i once we reach this threshold, we include them in the next bucket B_{i+1} (Line 18). Note that the approximate (r, s)-clique core number that we compute for each r-clique is given by the upper bound of the bucket in which it was peeled. In practice, we can improve this by taking the minimum of the upper bound of the bucket, and the s-clique-degree of each r-clique (in the original graph).

Once we have peeled all r-cliques, this concludes our subroutine. ARB-APPROX-NUCLEUS-HIERARCHY is then given by replacing ARB-NUCLEUS in Line 3 of Algorithm 1 with APPROX-ARB-NUCLEUS.

Theoretical Guarantees and Efficiency. We now discuss the theoretical guarantees and theoretical efficiency of APPROX-ARB-NUCLEUS, and by extension, ARB-APPROX-NUCLEUS-HIERARCHY.

We introduce the following lemmas to help prove that APPROX-ARB-NUCLEUS guarantees a $(\binom{s}{r} + \varepsilon)$-approximation of the true (r, s)-clique core numbers of each r-clique.

In particular, Lemma 6.1 bounds the number of r-cliques with (r, s)-clique core numbers $\leq \ell$ for a fixed ℓ. We use this to prove Lemma 6.2, which bounds the proportion of r-cliques with core numbers $\leq \ell$, but with s-clique-degree $> \ell(\binom{s}{r} + \delta)$. We can then set ℓ such that at any given step of our peeling process, we obtain a bound on the maximum proportion of r-cliques with core number at most ℓ that is not within the current bucket to be peeled. In essence, this gives us a bound on the number of times that a bucket must be reprocessed, such that moving on to the next bucket does not degrade the approximation factor, which gives us our approximation guarantees.

Note that Lemmas 6.1 and 6.2 apply to any stage of the peeling process in APPROX-ARB-NUCLEUS; that is to say, even if we have peeled a set of r-cliques from the graph G, the lemmas hold true for the remaining unpeeled r-cliques in G, with the updated s-clique-degrees (which are the original s-clique-degrees minus the number of incident s-cliques that have been removed from the graph due to the set of peeled r-cliques).

LEMMA 6.1. *Let S_ℓ be the set of remaining, or unpeeled, r-cliques with (r, s)-clique core numbers $\leq \ell$, considering an arbitrary fixed stage in the peeling process. Then, the number of s-cliques incident to S_ℓ is $\leq \ell \cdot |S_\ell|$.*

PROOF. We prove this by considering the exact (r, s) nucleus decomposition algorithm given by Sariyüce *et al.* [52]. In their algorithm, at each step, an r-clique with the minimum s-clique-degree in the graph is peeled. In peeling an r-clique R, for every s-clique S that R participated in, the algorithm decrements the s-clique-degree of the remaining r-cliques $R' \in S$ that have not yet been peeled. The (r, s)-clique core number of each r-clique R is given by the maximum k_R such that at some point before R was peeled in this algorithm, all r-cliques that were not yet peeled had s-clique-degree at least k_R. In order for a given r-clique R to have (r, s)-clique core number k_R, it must have had s-clique-degree at most k_R when it was peeled from the graph. If R has s-clique-degree greater than k_R when it was peeled, since R must have had the minimum s-clique-degree when it was peeled, this would mean that before R was peeled, every r-clique had s-clique-degree greater than k_R, so by definition, R's core number must be greater than k_R, which is a contradiction.

Note that Sariyüce *et al.* [52] prove that considering the r-cliques in the order in which they are peeled, the (r, s)-clique core numbers of the r-cliques are monotonically increasing.

Using these observations, the set S_ℓ of r-cliques is given by a contiguous sequence of r-cliques peeled in this algorithm, where upon being peeled, each r-clique has induced s-clique-degree at most ℓ. As such, we can assign each s-clique S incident to S_ℓ to the r-clique R in which S contributed to R's induced s-clique-degree when R was peeled. As a result, the total number of s-cliques incident to S_ℓ must be $\leq \ell \cdot |S_\ell|$, since the induced s-clique-degree of each such r-clique when it was peeled is upper bounded by ℓ. □

LEMMA 6.2. *Let F_ℓ be the set of unpeeled, or remaining, r-cliques with s-clique-degree $> \ell(\binom{s}{r} + \delta)$ and with (r, s)-clique core number $\leq \ell$, considering an arbitrary fixed stage in the peeling process. Then, $|F_\ell| \leq \binom{s}{r} \cdot |S_\ell| / (\binom{s}{r} + \delta)$.*

PROOF. Let S_ℓ be the set of unpeeled r-cliques with (r, s)-clique core numbers $\leq \ell$. First, note that $F_\ell \subseteq S_\ell$ by definition. By Lemma 6.1, we know that at most $\ell \cdot |S_\ell|$ s-cliques are incident to S_ℓ. Thus, the sum of the s-clique-degrees of the r-cliques in S_ℓ is at most $\binom{s}{r} \cdot \ell \cdot |S_\ell|$, since each s-clique can contribute to the s-clique-degree of at most

$\binom{s}{r}$ r-cliques. Then, since each r-clique in F_ℓ has s-clique-degree $> \ell(\binom{s}{r}+\delta)$ by definition, the maximum number of r-cliques in F_ℓ is $\binom{s}{r}\cdot\ell\cdot|S_\ell|/(\ell(\binom{s}{r}+\delta)) = \binom{s}{r}\cdot|S_\ell|/(\binom{s}{r}+\delta)$, as desired. □

Theorem 6.3. arb-approx-nucleus-hierarchy *computes a* $(\binom{s}{r}+\varepsilon)$*-approximate* (r,s) *nucleus decomposition hierarchy in* $O(m\alpha^{s-2})$ *expected work and* $O(\log^3 n)$ *span w.h.p.*

Proof. Note that the work bound for arb-approx-nucleus-hierarchy follows directly from that for arb-nucleus [55] and arb-nucleus-hierarchy. This is because the only difference between approx-arb-nucleus and arb-nucleus is the boundaries used in the bucketing structure, which affects the number of r-cliques that are peeled in any given round, but does not affect the amount of work needed to rediscover s-cliques containing peeled r-cliques. The latter dominates the work of both approx-arb-nucleus and arb-nucleus, so as a result, the work of approx-arb-nucleus is precisely the work of arb-nucleus. To be more specific, as given by [55], approx-arb-nucleus takes $O(m\alpha^{s-2})$ work w.h.p. assuming space proportional to the number of s-cliques.[2] Additionally, arb-approx-nucleus-hierarchy is precisely arb-nucleus-hierarchy, except using approx-arb-nucleus to compute the (r,s)-clique core numbers. Thus, following the proof of Theorem 5.1, we see that arb-approx-nucleus-hierarchy overall takes $O(m\alpha^{s-2})$ expected work w.h.p.

For our span bound, first note that there are only a logarithmic number of possible i values (since $\binom{n}{s} = O(n^s)$ upper bounds the maximum possible s-clique-degree). For each bucket B_i, we by construction process B_i at most $O(\log_{1+\delta/\binom{s}{r}}(n))$ times (Line 17). Thus, in total, we have $O(\log^2 n)$ rounds of peeling in approx-arb-nucleus. The span of each peeling round is $O(\log n)$ w.h.p. to retrieve the next bucket of s-cliques and to perform hash table operations to update the s-clique counts, as discussed in more detail by Shi *et al.* [55]. Therefore, the total span of peeling is $O(\log^3 n)$ span w.h.p.

The work and span of orienting the graph and counting the number of s-cliques follows from Shi *et al.*'s s-clique enumeration algorithm [54], which takes $O(m\alpha^{s-2})$ expected work and $O(\log^2 n)$ span w.h.p. Thus, in total, arb-approx-nucleus-hierarchy takes $O(m\alpha^{s-2})$ expected work and $O(\log^3 n)$ span w.h.p., as desired.

It remains to argue that approx-arb-nucleus computes an $(\binom{s}{r}+\varepsilon)$-approximate (r,s)-clique core number for every r-clique. Our proof here follows arguments by Ghaffari *et al.* [23], in their approximate k-core decomposition algorithm. We generalize their arguments to apply to (r,s) nucleus decomposition.

We prove this using induction on i, considering each bucket B_i. Our inductive hypothesis is that before beginning to peel B_i, we have already peeled all r-cliques with (r,s)-clique core numbers $\leq (1+\delta)^i$. We show here that after a logarithmic number of rounds peeling all r-cliques in B_i, we have necessarily peeled all r-cliques with (r,s)-clique core numbers $\leq (1+\delta)^{i+1}$. The key to this argument is Lemma 6.2, where we set $\ell = (1+\delta)^{i+1}$. Notably, all r-cliques in B_i have s-clique-degree at most $(\binom{s}{r}+\delta)\cdot(1+\delta)^{i+1} = (\binom{s}{r}+\delta)\cdot\ell$, by construction of B_i. Thus, Lemma 6.2 bounds the number of unpeeled r-cliques outside of the bucket B_i (that is to say, in a bucket B_j where $j > i$), but with (r,s)-clique core number $\leq \ell$ (these r-cliques are precisely those in F_ℓ). Specifically, after each round of peeling B_i, at most $\binom{s}{r}\cdot|S_\ell|/(\binom{s}{r}+\delta)$ r-cliques remaining outside of B_i have core number $\leq \ell$, so it takes $O(\log_{1+\delta/\binom{s}{r}}(n))$ rounds until no r-cliques with core number $\leq \ell$ are outside of B_i (that is to say, F_ℓ is empty). This means that after a logarithmic number of rounds of peeling all r-cliques in B_i, we have necessarily peeled all r-cliques with core number $\leq \ell$.

Now, in our algorithm, we assign the approximate core number of these peeled r-cliques to be the upper boundary of B_i, or $(\binom{s}{r}+\delta)\cdot(1+\delta)^{i+1}$. Note that the core numbers of the r-cliques that we peel while processing B_i are $> (1+\delta)^i$ (by our inductive hypothesis) and $\leq \ell = (1+\delta)^{i+1}$. Thus, our approximation is within a $(\binom{s}{r}+\delta)\cdot(1+\delta) = (\binom{s}{r}+\varepsilon)$ factor of the true core number. Thus, approx-arb-nucleus gives a $(\binom{s}{r}+\varepsilon)$-approximation of the true core numbers of each r-clique.

□

7 PRACTICAL IMPLEMENTATIONS

While the algorithm presented in Section 5 (Algorithm 1) is efficient in theory, we present a number of optimizations that improve its practical performance. Algorithm 1 requires two passes over the r-cliques and their s-clique-adjacent neighbors, first to compute the (r,s)-clique core numbers and then to construct the hierarchy. We present algorithms that interleave these two computations, so that only a single pass is required. Specifically, we present two algorithms that are not as theoretically efficient as Algorithm 1, but are faster in practice, particularly when the difference between r and s is large, as we demonstrate in Section 8.2.

7.1 Interleaved Hierarchy Framework

Our algorithms use the same framework, given in Algorithm 3, and the main difference between the two algorithms is the implementation of the key subroutines link and construct-tree. The framework is based on the peeling process used to compute the (r,s)-clique core numbers in Shi *et al.*'s work [55]. The main idea is that when we peel an r-clique R, while computing the updated s-clique counts due to peeling R, we are already iterating over all s-clique-adjacent r-cliques R'. Note that additionally, the nucleus decomposition algorithm in Shi *et al.* [55] uses a bucketing structure that maintains the intermediate (r,s)-clique core numbers of each r-clique throughout the peeling process (which begin as simply the s-clique count of each r-clique, and throughout the peeling process are updated to the actual (r,s)-clique core numbers). Then, if the intermediate (r,s)-clique core number of R' is less than or equal to that of R, as maintained in the bucketing structure, the intermediate (r,s)-clique core numbers of R and R' are actually the final (r,s)-clique core numbers of R and R' respectively. Thus, each such R and R' pair are connected in the nucleus decomposition hierarchy up to the level given by $\min(ND[R], ND[R'])$, and after the peeling process completes, we will have all of the relevant connectivity information to completely compute the nucleus decomposition hierarchy. In this sense, it suffices to define a link subroutine to process the s-clique-adjacent r-cliques given the intermediate (r,s)-clique core numbers throughout the peeling algorithm. Based on link, construct-tree constructs the final hierarchy tree.

[2] As an aside, following the arguments in [55], if we instead restrict our space usage to be proportional to the number of r-cliques, we can modify the bucketing structure to use a batch-parallel Fibonacci heap [56], which would increase the work bound to $O(m\alpha^{s-2}+\log^3 n)$ amortized expected work w.h.p.

Algorithm 3 Parallel (r, s) nucleus decomposition hierarchy algorithm framework

```
1: Initialize r, s                                   ▷ r and s for (r, s) nucleus decomposition
2: procedure ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK(G = (V, E))
3:     DG ← ARB-ORIENT(G)                            ▷ Apply an arboricity-orientation algorithm
4:     Initialize U to be a parallel hash table with r-cliques as keys, and s-clique counts as values
5:     REC-LIST-CLIQUES(DG, s, U)                    ▷ Count s-cliques, and store the counts per r-clique in U
6:     Let ND be a bucketing structure mapping each r-clique to a bucket based on # of s-cliques
7:     finished ← 0
8:     while finished < |U| do
9:         A ← r-cliques in the next bucket in ND (to be peeled)
10:        finished ← finished + |A|
11:        parfor all r-cliques R in A do
12:            parfor all s-cliques S containing R do
13:                parfor all r-cliques R′ in S where R′ ≠ R do
14:                    if ND[R′] ≤ ND[R] then
15:                        LINK(R′, R, ND)
16:                    else Update s-clique count of R′ in U
17:        Update the buckets of r-cliques with updated s-clique counts, peeling A
18:    return CONSTRUCT-TREE(ND)                     ▷ Return the hierarchy tree T, constructed based on LINK
```

In more detail, ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK first uses an efficient low out-degree orientation algorithm by Shi *et al.* [54] to direct the graph G such that every vertex has out-degree at most $O(\alpha)$ (Line 3). Then, it counts the number of s-cliques per r-clique in G and stores the counts in a parallel hash table U, where the keys are r-cliques and the values are the counts (Lines 4–5). Note that ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK uses a subroutine REC-LIST-CLIQUES based on previous work by Shi *et al.* [54], to count the number of s-cliques per r-clique. Also, our algorithm initializes a parallel bucketing structure ND that maps r-cliques to buckets, initially based on their s-clique counts (Line 6). We use the bucketing structure by Dhulipala et al. [16]. This structure ND stores the aforementioned intermediate (r, s)-clique core numbers of each r-clique, and supports efficient operations to update buckets and return the lowest unpeeled bucket. Our algorithm then proceeds with a classic peeling paradigm, where while not all r-cliques have been peeled, it processes the r-cliques (that have not yet been peeled) incident to the lowest number of s-cliques and peels them from the graph (Lines 8–17). For a set A of r-cliques with the lowest number of incident s-cliques (Line 9), we iterate over all s-clique-adjacent r-cliques R' to each r-clique R in A (Lines 11–13).

Note that if $ND[R'] \leq ND[R]$, this means that R' was either previously peeled or is currently being peeled (and is also in A); this is because at any given peeling step, we process the bucket of unpeeled r-cliques with the minimum incident s-clique count. This also means that $ND[R']$ and $ND[R]$ are the actual (r, s)-clique core numbers of R' and R respectively, which follows directly from the correctness of the peeling paradigm [52]. Thus, each such R and R' pair are connected in the nucleus decomposition hierarchy up to the level given by $\min(ND[R], ND[R'])$, which we process using the LINK subroutine (Lines 14–15). The LINK subroutine will construct the hierarchy, and we describe it in Sections 7.2 and 7.3.

On the other hand, if $ND[R] > ND[R']$, then this means that R' has not yet been peeled, and the s-clique removed by R must be properly accounted for. In this case, ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK updates the s-clique count of R' in the hash table U (Line 16). After processing all R and R' pairs, we then update the buckets of the r-cliques with updated s-clique counts in U (Line 17). We omit the details of these steps for conciseness, since they are described in Shi *et al.*'s parallel nucleus decomposition algorithm [55].

Algorithm 4 Basic link and tree construction.

```
1: Initialize k union-find data structures, uf_i for i ∈ [k], where k is the maximum (r, s)-clique core number
2: procedure LINK-BASIC(R, Q, ND)
3:     parfor i ∈ [min(ND[R], ND[Q])] do
4:         uf_i.unite(R, Q)
5: procedure CONSTRUCT-TREE-BASIC(ND)
6:     Initialize the hierarchy tree T with leaves corresponding to each r-clique
7:     for i ∈ {k, k − 1, . . . , 1} do
8:         parfor each connected component C = {R_1, . . . , R_c} in uf_i do
9:             Construct a new parent in T, to be the parent of the roots of the leaf nodes corresponding to each R_ℓ (for ℓ ∈ [c])
10:    return T
```

7.2 Basic Version of LINK

The salient detail that remains is how we perform the LINK subroutine (Line 15) and how we construct the hierarchy tree T with CONSTRUCT-TREE (Line 18). Note that the algorithms that we provide for these subroutines, here and in Section 7.3, are novel.

In Algorithm 4, we present a basic LINK subroutine, LINK-BASIC, and the corresponding CONSTRUCT-TREE subroutine, CONSTRUCT-TREE-BASIC. LINK-BASIC maintains a parallel union-find data structure uf_i per core number $i \in [k]$, which corresponds to a level of the hierarchy tree T. Each uf_i connects r-cliques that are s-clique-adjacent considering only r-cliques with core numbers $\geq i$. To construct these uf_i's, given two r-cliques R and Q, LINK-BASIC simply unites R and Q in each uf_i where $i \leq \max(ND[R], ND[Q])$ (Lines 3–4). Then, given the uf_i for all $i \in [k]$, we construct the hierarchy tree T from the bottom-up, starting with leaf nodes corresponding to r-cliques. CONSTRUCT-TREE-BASIC begins with $i = k$, where for each connected component in uf_k (Line 8), we construct a parent in T where its children are the leaf nodes corresponding to the r-cliques in the connected component (Line 9). Then, for $i = k - 1, \ldots, 1$ (Line 7), we construct a new parent for each connected component in uf_i, where its children are the parents of the leaf nodes corresponding to the r-cliques that compose the component (Line 9). This produces the desired T.

However, LINK-BASIC is not efficient, since it requires a union-find data structure per level, and for every pair of r-cliques, we could perform up to k UNITE operations. Indeed, in Section 8.2, we empirically show that LINK-BASIC performs many unnecessary UNITE operations in practice. If we let n_r and n_s denote the number of r-cliques and the number of s-cliques in the graph respectively, LINK-BASIC incurs additional space proportional to $O(kn_r)$ and total work upper bounded by $O(kn_s)$ (since there are at most $O(n_s)$ pairs of s-clique-adjacent r-cliques). In the next subsection, we introduce more efficient LINK and CONSTRUCT-TREE subroutines.

7.3 Efficient Version of LINK

Our improved subroutines LINK-EFFICIENT and CONSTRUCT-TREE-EFFICIENT are shown in Algorithm 5. We refer to an example of $(r, s) = (1, 2)$ nucleus (k-core) decomposition, given by the graph

Algorithm 5 Efficient link and tree construction.

```
1: Initialize a union-find data structure, uf, of length equal to the number of r-cliques
2: Initialize a hash table, L, where the keys and values are r-cliques
3: procedure LINK-EFFICIENT(R, Q, ND)
4:     if R or Q is empty then return
5:     if ND[Q] < ND[R] then Swap R and Q
6:     R ← uf.parent(R), Q ← uf.parent(Q)
7:     if ND[R] = ND[Q] then
8:         uf.unite(R, Q)
9:         if uf.parent(R) ≠ R then LINK-EFFICIENT(L[R], uf.parent(R), ND)
10:        if uf.parent(Q) ≠ Q then LINK-EFFICIENT(L[Q], uf.parent(Q), ND)
11:    else                                              ▷ ND[R] < ND[Q]
12:        while true do
13:            LQ ← L[Q]
14:            Q ← uf.parent(Q)
15:            if COMPARE-AND-SWAP(L[Q], empty, R) then
16:                if uf.parent(Q) ≠ Q then
17:                    LINK-EFFICIENT(R, uf.parent(Q), ND)
18:                break
19:            else if ND[LQ] < ND[R] then
20:                if COMPARE-AND-SWAP(L[Q], LQ, R) then
21:                    if uf.parent(Q) ≠ Q then
22:                        LINK-EFFICIENT(R, uf.parent(Q), ND)
23:                    LINK-EFFICIENT(R, LQ, ND)
24:                    break
25:            else
26:                LINK-EFFICIENT(R, L[Q], ND)
27:                break
28: procedure CONSTRUCT-TREE-EFFICIENT(ND)
29:    Initialize the hierarchy tree T with leaves corresponding to each r-clique
30:    parfor each connected component C = {R_1, ..., R_c} in uf do
31:        Construct a parent node uf_C in T, where its children are the leaves corresponding to the r-cliques in C
32:    parfor each parent node uf_C in T do
33:        if L[C] is non-empty then
34:            R ← uf.parent(L[C]) ▷ Note that C is the r-clique representing the component in uf
35:            Make uf_C a child of uf_R in T ▷ Note that uf_R necessarily exists since R represents a component in uf
36:    return T
```

in Figure 1. Recall that we have omitted labeling other vertices in the 5-cliques represented by $4a$, $4b$, $4c$, and $4d$.

The main idea of LINK-EFFICIENT is instead of maintaining k union-find data structures, we maintain a single parallel union-find data structure uf and an additional hash table L that maps r-cliques to r-cliques. First, uf stores connected r-cliques considering only other r-cliques with equal core numbers. For instance, in Figure 1, the vertices $3a$, $3b$, and $3c$ are connected and all have core number 3, so we would store these as a component in uf. Note that for all core numbers i, we can store this information using a single union-find data structure because the sets of r-cliques with distinct core numbers are disjoint. We can arbitrarily represent each connected component in uf by a single r-clique in that component.

The main idea of L is to connect the components in uf to the "nearest" core with a different core number that it is contained within (if it exists). For instance, in Figure 1, we note that the component in uf corresponding to $4a$ (consisting of vertices with core number 4) is contained within the 3-core consisting of the component $\{3a, 3b, 3c\}$. In L, we would store one of $3a$, $3b$, or $3c$ in an entry corresponding to key $4a$, indicating that this is the "nearest" core that $4a$ must join in the hierarchy. We note that $4a$ is also contained within a larger 2-core and a larger 1-core, but its "nearest" core, or the smallest core such that the component $4a$ is a proper subset of that core, is given by the 3-core. It is sufficient to

After Round 3:
$A = \{3a, 3b, 3c\}$

$uf =$

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
parent	1a	2a	3a	3b	3c	4a	4b	4c	4d

$L =$

key	3a
value	1a

After $(3a, 4c)$:

$uf =$

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
parent	1a	2a	3a	3b	3c	4a	4b	4c	4d

$L =$

key	3a	4c
value	1a	3a

After $(3b, 4c)$:

$uf =$

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
parent	1a	2a	3b	3b	3c	4a	4b	4c	4d

$L =$

key	3a	3b	4c
value	1a	1a	3a

After $(2a, 4c)$:

$uf =$

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
parent	1a	2a	3b	3b	3c	4a	4b	4c	4d

$L =$

key	2a	3a	3b	4c
value	1a	1a	2a	3a

After Round 4:
$A = \{4a, 4b, 4c, 4d\}$

$uf =$

r-clique	1a	2a	3a	3b	3c	4a	4b	4c	4d
parent	1a	2a	3b	3b	3b	4a	4b	4c	4d

$L =$

key	2a	3a	3b	4a	4b	4c	4d
value	1a	1a	2a	3a	3b	3b	2a

Figure 4: An example of the uf and L data structures maintained by LINK-EFFICIENT when computing the k-core hierarchy on the graph in Figure 1. The data structures are shown after the third and fourth rounds of peeling in ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK, and after intermediate calls to LINK-EFFICIENT within the fourth round.

store only the "nearest" core to $4a$ in L, because the component in uf corresponding to $\{3a, 3b, 3c\}$ is responsible for storing its "nearest" core in L as well, to the 2-core it is contained within. On the other hand, the component corresponding to $4d$ is not contained within the 3-core, and its "nearest" core would be the 2-core containing the component $2a$, so $4d$ would store in L the value $2a$.

More formally, for each r-clique R representing a connected component in uf, let R' be a r-clique with the maximum $ND[R']$ such that $ND[R'] < ND[R]$, and R' is connected to R through s-cliques considering only r-cliques with core number $\geq ND[R']$. Then, each such r-clique R is a key in L, and L stores the corresponding R' that satisfies these conditions as the value. Note that if there are multiple such R' where $ND[R']$ is maximized under these conditions, it is irrelevant which R' is stored in L, because it is simple to look up the component that R' corresponds to using uf. That is to say, it is irrelevant which of $3a$, $3b$, and $3c$ we store for $4a$ in L, because we can look up the parent of $3a$, $3b$, and $3c$ in uf, which would resolve to the same parent. LINK-EFFICIENT updates uf and L, depending on the core numbers of the given r-cliques R and Q.

Tree Construction. We describe first how CONSTRUCT-TREE-EFFICIENT constructs the hierarchy tree T given the specifications for uf and L. We refer to the construction shown in Figure 5, where uf and L are given under "After Round 4" in Figure 4.

The main idea for the construction is that if we begin with a hierarchy tree T consisting of only leaf nodes corresponding to each r-clique, the highest (bottom-most) level in which that leaf node may join a non-trivial connected component is the level corresponding to the leaf node's r-clique's (r, s)-clique core number.

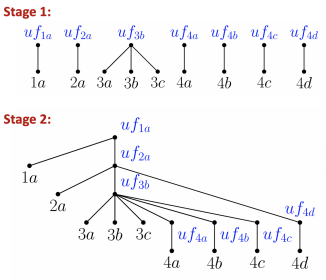

Figure 5: An example of the k-core hierarchy tree on the graph in Figure 1, constructed by CONSTRUCT-TREE-EFFICIENT. Stage 1 shows an intermediate tree constructed in the CONSTRUCT-TREE-EFFICIENT subroutine, and Stage 2 depicts the final hierarchy tree.

That is to say, starting from $i = k$ and iterating to $i = 1$, an r-clique R is necessarily a singleton component from $i = k$ to $i = ND[R] + 1$. The union-find structure uf denotes the parent of each leaf node on the level corresponding to its core number.

Then, once we have the connected components per level corresponding to each core number, it remains to describe how components with different core numbers are contained within each other. This containment is necessarily hierarchical, where components corresponding to larger core numbers are contained within successive components corresponding to smaller core numbers, as can be seen in Figure 1. This containment is precisely what L describes; it points each component within a given core i to a component on the greatest core number j such that $j < i$, where the two components are connected on level j (that is to say, connected through r-cliques with core numbers $\geq j$). Thus, using L, we can point each parent constructed using our uf to another parent, which represents the first instance (from the bottom-up in T) in which the original parent is merged into another component.

In more detail, in CONSTRUCT-TREE-EFFICIENT, we begin with a hierarchy tree T consisting of isolated r-cliques (Line 29). For each component $C = \{R_1, \ldots, R_c\}$ in uf, representing a connected component of r-cliques with the same core number, we construct a parent in T, where its children are leaves corresponding to the r-cliques in C (Lines 30–31). C is one of these cliques, and is chosen arbitrarily. The new parent is uf_C. This process is shown in Stage 1 in Figure 5; each vertex has itself as its parent in uf, except the vertices $3a$, $3b$, and $3c$, which have $3b$ as their parent. Thus, we create a new parent node uf_{3b} for the leaves $3a$, $3b$, and $3c$.

Then, for each parent node uf_C, we look up the nearest connected component that it should hierarchically connect to using L. What this means is that the component uf_C should join the connected component of $L[C]$ (if it exists). If $R = uf.\text{parent}(L[C])$, then the connected component of $L[C]$ is represented by uf_R. Thus, we make uf_C a child of uf_R in T (Lines 32–35). This construction is shown in Stage 2 in our example in Figure 5. We note that $L[2a] = 1a$ and $uf[1a] = 1a$, so we make uf_{2a} the child of uf_{1a}. Similarly, $L[3b] = 2a$ and $uf[2a] = 2a$, so we make uf_{3b} the child of uf_{2a}, and $L[4d] = 2a$ and $uf[2a] = 2a$, so we also make uf_{4d} the child of uf_{2a}. Finally, we have $L[4a] = 3a$, $L[4b] = 3b$, and $L[4c] = 3b$, where $uf[3a] = uf[3b] = 3b$, so we make uf_{4a}, uf_{4b}, and uf_{4c} the children of uf_{3b}. In this manner, we construct T.

We make a subtle note that it is not strictly necessary to maintain parent nodes in the hierarchy tree with exactly one child; we can remove all such parents P and make the child C a direct child of its grandparent G. This is because it is inherently implied that the component that the child C represents in unchanged until it joins the component represented by G. For instance, uf_{4d}'s sole child is $4d$. If we set $4d$ to be a direct child of its grandparent, uf_{2a}, and remove uf_{4d}, then it is implied that the vertex $4d$ remains its own component until it joins the 2-core containing $2a$ (and notably, $4d$ represents an unchanged component in the 3-core and in the 4-core of the graph). In this sense, the final hierarchy tree produced in Figure 5 is actually equivalent to that in Figure 3.

LINK **Subroutine.** We now describe LINK-EFFICIENT. The key to LINK-EFFICIENT is to properly maintain uf and L according to their definitions given new information obtained by connected r-cliques. The main subtlety is that after updating the connectivity information of R or Q, in uf or in L, there may be cascading effects resulting in new calls to LINK-EFFICIENT.

In more detail, LINK-EFFICIENT first ensures that $ND[R] \leq ND[Q]$ (Line 5). We need only perform operations on the parents of the components of the r-cliques in uf, so we set R and Q to their respective parents in uf (Line 6).

The first case is if $ND[R] = ND[Q]$ (Line 7). Then, we need only unite R and Q in uf, to maintain that uf tracks the connectivity between r-cliques with the same core number (Line 8). However, the new parent P may now need to update its value in L based on $L[R]$ and $L[Q]$. This is because it is possible that $L[P]$ must be updated; for instance, if $L[R]$ has a strictly greater core number than the current $L[P]$, $L[P]$ must be updated to be $L[R]$. LINK-EFFICIENT performs these updates by calling itself on $L[R]$ and P, and on $L[Q]$ and P, if $P \neq R$ and $P \neq Q$, respectively (Lines 9–10).

The second case is if $ND[R] < ND[Q]$ (Line 11). There are two main considerations. First, R may replace the current value of $L[Q]$, which we call LQ, if $ND[R] > ND[LQ]$. Second, R and LQ must be linked, since they are connected through Q and could affect each other's parent or value in uf or L respectively. We handle these cases with a series of if statements and COMPARE-AND-SWAPs.

We first perform a COMPARE-AND-SWAP, checking if $L[Q]$ is empty and replacing it with R if so (Line 15); if this COMPARE-AND-SWAP succeeds, then there is still the possibility that Q's parent in uf changed before the COMPARE-AND-SWAP completed, in which case Q's new parent is unaware of its connection to R. In this scenario, we must call LINK-EFFICIENT again on R and the new parent of Q, since R could potentially modify the r-clique stored in L corresponding to Q's new parent (Lines 16–17).

If the previous COMPARE-AND-SWAP failed, then we check if $ND[LQ] < ND[R]$ (Line 19), which if true, means that R is a candidate to replace LQ in L. We perform another COMPARE-AND-SWAP to replace LQ with R (Line 20), and if it succeeds, we must again check if Q's parent in uf has potentially changed before the COMPARE-AND-SWAP completed. Again, if this occurs, we must call LINK-EFFICIENT on R and the new parent Q (Lines 21–22). We must also call LINK-EFFICIENT on R and LQ (Line 23), to store R's connectivity to LQ. This is because R's "nearest" core as stored in L may be superseded by LQ. If the COMPARE-AND-SWAP fails, we simply try again, hence the while loop (Line 12).

The last case is if $ND[LQ] \geq ND[R]$ (Line 25), in which case R is not a candidate to replace LQ in L, and we store R's connectivity to Q by calling LINK-EFFICIENT on R and $L[Q]$. Note that this is necessary because R and $L[Q]$ could be united in uf if they have the same core number, or R could be a "nearest" core to $L[Q]$.

This concludes our efficient LINK subroutine, LINK-EFFICIENT. We show in Section 8.2 that LINK-EFFICIENT performs many fewer UNITE and LINK operations, and achieves significant speedups, over LINK-BASIC. We provide an example of running LINK-EFFICIENT.

Example of LINK-EFFICIENT. As an example of these cascading effects, we refer to an intermediate state of uf and L on the example graph in Figure 1, given after the third round of peeling in ARB-NUCLEUS-DECOMP-HIERARCHY-FRAMEWORK (immediately before peeling the final set of vertices in the graph, given by the components $4a$, $4b$, $4c$, and $4d$). This state is shown in the data structures under "After Round 3" in Figure 4. In uf, every vertex is its own parent, and in L, we have identified that the component corresponding to $3a$ is connected to $1a$. Note that many link operations occur from peeling $4a$, $4b$, $4c$, and $4d$, including for the (R, Q) pairs $(3a, 4c)$, $(3b, 4c)$, and $(2a, 4c)$, which we consider in this example. We show in Figure 4 the state of uf and L after each of these three calls to LINK-EFFICIENT (including the cascading calls that each of these calls generate). We list the R and Q for each LINK-EFFICIENT operation, as well as the cascading calls that they invoke. We assume the operations happen sequentially for clarity, although in practice they can happen in parallel.

- $R = 3a$, $Q = 4c$: We now know that $4c$ is connected to $3a$'s component, and $4c$ does not have a previously set "nearest" core, so we set $L[4c] = 3a$ (Line 15).
- $R = 3b, Q = 4c$: We note that $L[4c]$ is now already set to a "nearest" core with core number 3, so there is nothing new to update for $L[Q]$. However, we gain from this new link the knowledge that $3a$ and $3b$ are connected (since $L[4c] = 3a$), so we must cascade a new LINK-EFFICIENT call to $(R, L[Q]) = (3b, 3a)$ (Line 26), so that $3a$ and $3b$ can be set to the same component in uf.
  - $R = 3a$, $Q = 3b$: We now call UNITE on $3a$ and $3b$ in uf (Line 8). Say that arbitrarily, $3b$ is set as the new parent of $3a$ and $3b$ in uf. Since the parents in uf are responsible for maintaining the connection to the "nearest" core, we must transfer $L[R] = L[3a]$ to $L[Q] = L[3b]$. We do so by calling LINK-EFFICIENT on $(L[R], uf.\text{parent}(R)) = (1a, 3b)$ (Line 9).
    - $R = 1a$, $Q = 3b$: Now, we can set $L[Q] = L[3b]$ to $R = 1a$, since $3b$'s "nearest" core is now $1a$ (Line 15).
- $R = 2a$, $Q = 4c$: Since $4c$ already has an entry in L that's "nearer" to it than $2a$, there is nothing new to update for $L[Q]$. However, we now know that $3a$ and $2a$ are connected, so we must cascade a new LINK-EFFICIENT call to $(R, L[Q]) = (2a, 3a)$ (Line 26).
  - $R = 2a$, $Q = 3a$: Since the parent of Q in uf is $3b$, we can treat this as $R = 2a$ and $Q = 3b$ (since we only need to maintain connections in L for the parents in uf). Now, we find that $3b$ has recorded its "nearest" core as $L[3b] = 1a$, but $2a$ is "nearer". Thus, we update $L[3b]$ to be $2a$ (Line 20), but now we know that $2a$ is connected to $1a$. So, we call LINK-EFFICIENT on $(R, L[Q]) = (2a, 1a)$ (Line 23).
    - $R = 1a$, $Q = 2a$: We discover that $2a$'s "nearest" core is given by $1a$. We set $L[Q] = L[2a]$ to $R = 1a$ (Line 15).

Note that it is necessary for us to perform these cascading calls to LINK-EFFICIENT, because the only way for $3a$ and $3b$ to discover that they should be connected is through one of the 4-core components, and the only way for $3b$ to realize that the component with $2a$ is its "nearest" core is also through one of the 4-core components. Similarly, the only way for $2a$ to realize that the component with $1a$ is its "nearest" core is through first one of the 4-core components, then through the 3-core component, in which $3a$, which we now know is connected to $3b$, had the original adjacency to $1a$. Thus, information must be constantly propagated through uf and L.

Comparison to Prior Work. Sariyüce and Pinar [49] also provide a hierarchy construction algorithm, NH, that is performed interleaved with the peeling process. They maintain a union-find data structure that stores the connectivity of all r-cliques considering only r-cliques with the same core number. However, for adjacent r-cliques with different core numbers, they simply store all pairs of such r-cliques in a list. They process this list after the peeling process to construct the hierarchy tree, and their method for processing this list requires a global view, since NH first sorts the pairs of r-cliques in the list based on their core numbers. Storing this list incurs additional space potentially proportional to the number of s-cliques in the graph, which represents a significant overhead.

Our main innovation in LINK-EFFICIENT is that we need only incur additional space overhead proportional to the number of r-cliques in the graph, because we process adjacent r-cliques with different core numbers while performing the peeling process. We are able to process this information into a hash table L proportional to the number of r-cliques, so our memory overhead overall is $2n_r$, where n_r is the number of r-cliques. In contrast, NH uses $\binom{s}{r} \cdot n_s + n_r$ additional space, where n_s is the number of s-cliques.

In addition, NH is sequential, whereas LINK-EFFICIENT is thread-safe and carefully resolves conflicts in updating uf and L. Also, the post-processing step to construct the hierarchy tree in NH involves many sequential dependencies, where even merges on the same level of the tree may conflict with each other, whereas our post-processing step, CONSTRUCT-TREE-EFFICIENT, is fully parallel.

7.4 Practical Version of ARB-NUCLEUS-HIERARCHY

Finally, we make certain modifications to our theoretically efficient (r, s) nucleus decomposition hierarchy algorithm, ARB-NUCLEUS-HIERARCHY (Algorithm 1), to improve its performance in practice. We maintain the two-pass paradigm of first computing the (r, s)-clique core numbers of each r-clique and then constructing the hierarchy tree T. However, we do not explicitly store linked lists containing all pairs of s-clique-adjacent r-cliques, since this represents too much of a memory overhead to be practical, particularly for larger r and s. We also do not explicitly generate the graph H given by these linked lisits (Line 14).

Instead, we use a single union-find data structure to maintain the connected components (throughout the loop on Lines 12–20), and for each $i \in \{k, k-1, \ldots, 1\}$ (Line 12), we iterate through all r-cliques R with core number i and their s-clique-adjacent r-cliques R'. We perform a parallel sort on the r-cliques based on their core numbers, which allows us to efficiently extract r-cliques with the same core numbers; this adds a small additional memory overhead, which we observe in Section 8.2. For each such pair of

	n	m
amazon	334,863	925,872
dblp	317,080	1,049,866
youtube	1,134,890	2,987,624
skitter	1,696,415	11,095,298
livejournal	3,997,962	34,681,189
orkut	3,072,441	117,185,083
friendster	65,608,366	1.806×10^9

Table 1: Sizes of our input graphs, which are from SNAP [37].

r-cliques R and R' where $ND[R'] \geq ND[R]$, we directly unite them in our union-find data structure to obtain the desired connected components; this replicates the same information stored in the linked lists (on Lines 6–8). We construct the requisite new parents in the hierarchy tree T given the computed connected components, and we reuse the same union-find data structure for subsequent i.

8 EVALUATION

8.1 Environment and Graph Inputs

We run our experiments on a Google Cloud Platform instance of a 30-core machine with two-way hyper-threading, with 3.9 GHz Intel Cascade Lake processors and 240 GB of main memory. We use all cores when testing parallel implementations, unless specified otherwise. Our implementations are written in C++ and we compile our code using g++ (version 7.4.0) with the `-O3` flag. We use parallel primitives and the work-stealing scheduler from ParlayLib by Blelloch *et al.* [6]. We terminate any experiment that takes over 4 hours. We test our algorithms on real-world graphs from the Stanford Network Analysis Project (SNAP) [37], shown in Table 1.

We implement all three versions of our exact (r, s) nucleus decomposition hierarchy algorithms, including our theoretically-efficient arb-nucleus-hierarchy (Algorithm 1, with the practical modifications described in Section 7.4), which we call anh-te for succinctness, and our nucleus decomposition hierarchy framework arb-nucleus-decomp-hierarchy-framework (Algorithm 3) using both link-basic (Algorithm 4) and link-efficient (Algorithm 5), which we call anh-bl and anh-el, respectively.

We also implement our approximate (r, s) nucleus decomposition hierarchy algorithm, arb-approx-nucleus-hierarchy (using Algorithm 2). We integrate approx-arb-nucleus with each of anh-te, anh-el, and anh-bl for the hierarchy construction, giving us three implementations, approx-anh-te, approx-anh-el, and approx-anh-bl, respectively.

We compare our hierarchy algorithms to nh, the state-of-the-art sequential (r, s) nucleus decomposition hierarchy implementation by Sariyüce and Pinar [49] for $(1, 2)$, $(2, 3)$, and $(3, 4)$ nucleus decomposition. Note that nh does not generalize to other r and s values. nh, like anh-bl and anh-el, constructs the hierarchy while computing the (r, s)-clique-core numbers of the graph. For the special case of k-core, we also compare to phcd, the state-of-the-art parallel k-core hierarchy implementation by Chu *et al.* [11].

8.2 Comparison of anh-te, anh-el, and anh-bl

Figure 6 compares our exact (r, s) nucleus decomposition hierarchy algorithms, anh-te, anh-el, and anh-bl, against each other, for various r and s. We show $r < s \leq 5$ here, but we ran all of our algorithms for $r < s \leq 7$. Also, the running times listed in these figures do not include the time needed to compute the low out-degree orientation or to compute the initial s-clique-degrees of each r-clique, which are the same across all of our algorithms (note that we do include these times when comparing to other work in Section 8.3). However, our running times do include the time required to compute the (r, s)-clique-core numbers of each r-clique, which notably for anh-te is given by arb-nucleus from [55].

Overall, we find that anh-el is faster if the difference between s and r is small (generally, if $s - r \leq 2$), and anh-te is faster in all other cases. The exception is for k-core (or (1, 2)-nucleus decomposition), where anh-te is 2.38–21.95x faster than anh-el. This is because the k-core decomposition requires far lower overhead to compute the core numbers per vertex compared to higher r and s, since we need only maintain the degree of each vertex. The benefit of anh-el is due to the improved locality in iterating over and processing s-cliques once, rather than recomputing the s-cliques twice. This is not a benefit for k-core, because iterating over edges is a much simpler and cache-friendly pattern. Also, anh-bl is significantly slower than both anh-te and anh-el, and runs out of memory for many values of r and s, since it has a much larger memory footprint from storing a union-find structure per core number. Overall, anh-el is up to 2.37x faster than anh-te, and anh-te is up to 41.55x faster than anh-el, where we see the largest speedups in anh-te over anh-el when s is much larger than r. anh-bl is up to 14.55x slower than anh-el, and up to 11.96x slower than anh-te.

The cases in which anh-el outperforms anh-te and vice versa, and the reason for the slowness of anh-bl, is due to the number of link and unite operations. Indeed, for the dblp and youtube graphs, particularly for larger r and when the difference between r and s is small, the number of times in which anh-te calls link and unite is 1.08–13.67x the number of times in which anh-el calls link and unite. For smaller r and when the difference between r and s is large, anh-el calls link and unite between 1.02–18.94x more than anh-te. Looking at fixed r and increasing s, we observe that anh-el performs many more cascading calls to itself as s increases, since $\binom{s}{r}$ increases and anh-el more likely needs to connect two r-cliques across multiple levels of the hierarchy tree. anh-bl is much slower than both anh-te and anh-el, performing up to 39.75x the number of link and unite calls. anh-bl repeatedly performs unite operations equal to the core number of each r-clique, which can be redundant, since if two r-cliques are connected in a higher core, they are necessarily connected in the lower core.

In terms of memory usage, considering the memory overhead of building and constructing the hierarchy (not including the space required to store the graph or the (r, s)-clique-core numbers of each r-clique), on dblp and youtube, anh-bl uses 1.53–10.03x the amount of overhead that anh-el uses, and anh-te uses 1.08–1.11x the amount of overhead that anh-el uses. We observe that anh-el is the most memory efficient overall, since it only maintains two arrays proportional to the number of r-cliques (uf and L). anh-te incurs almost the same memory overhead, with a minor additional cost attributed to maintaining r-cliques sorted by their core numbers (which we discuss in Section 7.4), and anh-bl incurs much more overhead to maintain union-find data structures proportional to the number of r-cliques per core number.

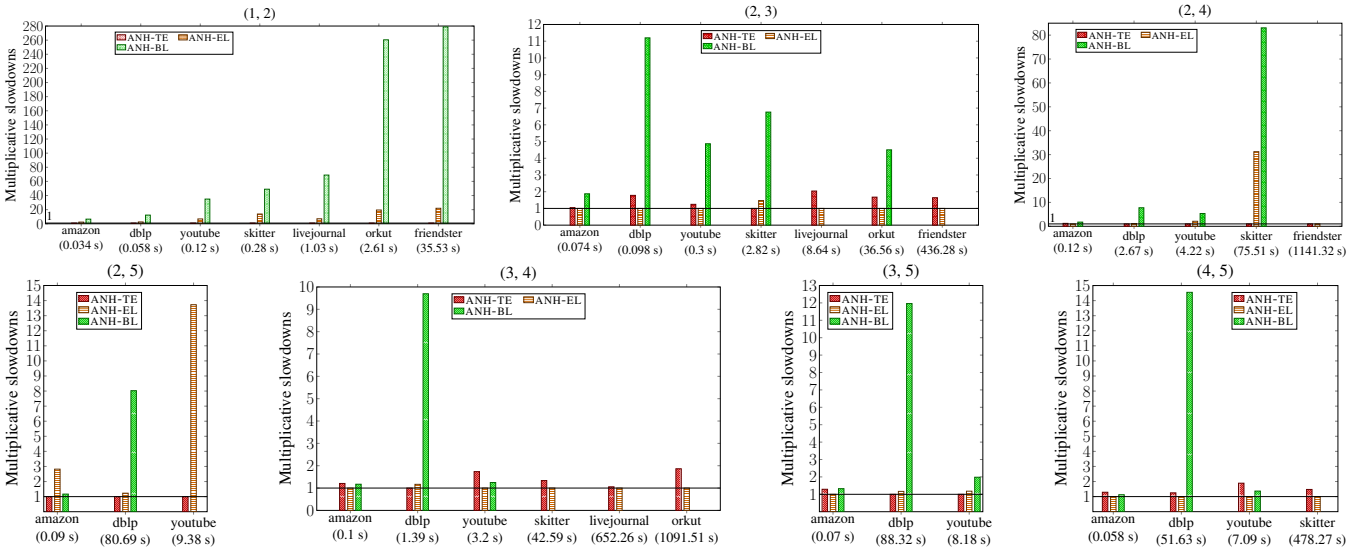

Figure 6: Multiplicative slowdowns of our parallel nucleus decomposition hierarchy implementations ANH-TE, ANH-EL, and ANH-BL, over the fastest of the three for each graph, for $r < s \leq 5$. We have omitted bars where our implementations run out of memory or time out after 4 hours, and we have omitted graphs where only one of ANH-TE, ANH-EL, and ANH-BL completes. Below each graph in parentheses is the fastest running time among the three implementations. We have also included a line marking a multiplicative slowdown of 1.

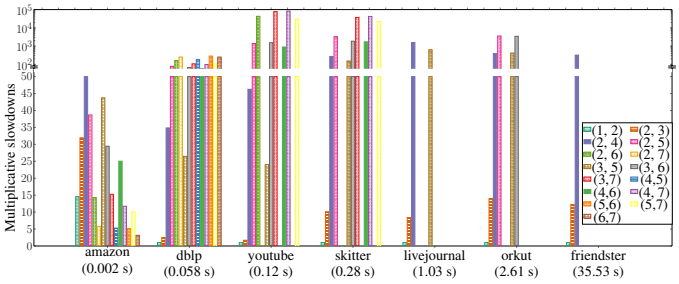

Figure 7: Multiplicative slowdowns of parallel ARB-NUCLEUS-HIERARCHY for each (r, s) combination over the fastest running time for parallel ARB-NUCLEUS-HIERARCHY across all $r < s \leq 7$ for each graph, considering the fastest of ANH-TE, ANH-EL, and ANH-BL. The fastest running time is labeled in parentheses below each graph. We have omitted bars where ARB-NUCLEUS-HIERARCHY runs out of memory or times out after 4 hours.

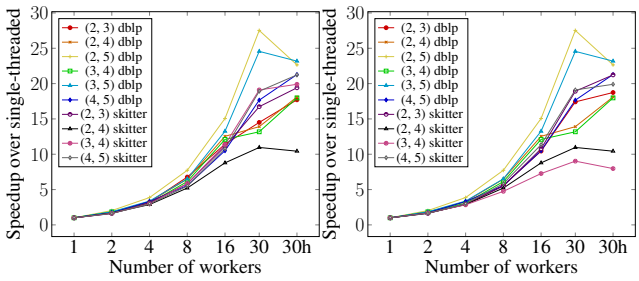

Figure 8: Speedup of ANH-TE on the left and ANH-EL on the right over their respective single-threaded running times, on dblp and skitter for various r and s. "30h" denotes 30-cores with two-way hyper-threading.

8.3 Performance of Exact Hierarchy

Figure 7 shows the best running times for all graphs, over $r < s \leq 7$, considering all of our exact (r, s) nucleus decomposition hierarchy algorithms, ANH-TE, ANH-EL, and ANH-BL, excluding the time needed to compute our low out-degree orientation and the initial s-clique-degrees of each r-clique. In general, larger (r, s) values correspond to longer running times. However, some of the times for larger values of (r, s) are faster than for smaller values of (r, s) (especially on amazon) because the maximum coreness values for the larger values of (r, s) are small and the algorithms finish quickly.

Figure 8 shows the scalability of ANH-TE and ANH-EL over different numbers of threads on dblp and skitter, and we see good scalability overall. Across all of our graphs and for $r < s \leq 7$, we observe up to 24.75x self-relative speedups (and a median of 15.57x) for ANH-TE and up to 30.96x self-relative speedups (and a median of 14.13x) for ANH-EL. Generally, we observe greater self-relative speedups for larger r and s and for larger graphs.

Comparison to other implementations. Figure 9 shows the comparison of our parallel $(1, 2)$, $(2, 3)$ and $(3, 4)$ nucleus decomposition hierarchy implementations to other implementations. Note that here, we include in our implementations the time needed to compute the low out-degree orientation and to compute the initial s-clique-degrees of each r-clique. We do not include the time required to load the graph in both our and other implementations.

For $(1, 2)$ nucleus (k-core) decomposition, we compare to the parallel PHCD [11] and the sequential NH [49]. Our fastest implementation for k-core is ANH-TE, and we see that ANH-TE is up to 2.57x slower than PHCD overall, but 1.87x faster than PHCD on dblp.

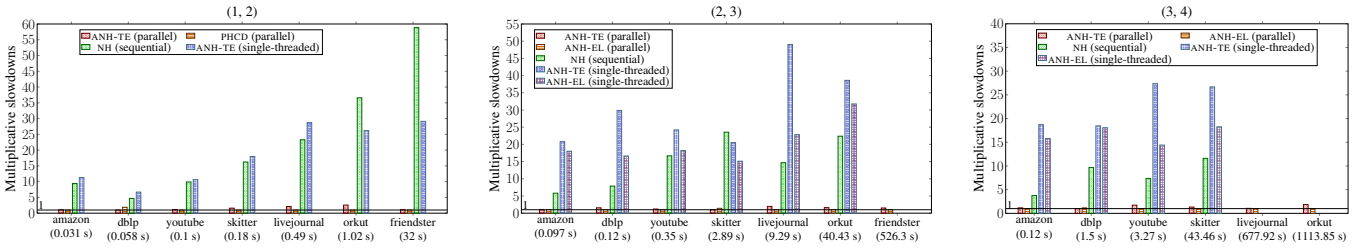

Figure 9: Multiplicative slowdowns, comparing ANH-TE, ANH-EL, the parallel PHCD [11], and the sequential ND [49], for (1, 2), (2, 3), and (3, 4) nucleus decomposition. We give the multiplicative slowdown over the fastest implementation for each graph and each (r, s), where the fastest running time is labeled in parentheses below each graph. We include end-to-end running times in this comparison, excluding only the time required to load the graph. We have omitted bars where the implementation runs out of memory or times out after 4 hours. We have also included a line marking a multiplicative slowdown of 1.

We note that PHCD is optimized for the k-core decomposition, rather than general (r, s) nucleus decomposition, whereas ANH-TE generalizes for larger r and s. Like ANH-TE, PHCD constructs the hierarchy tree from the bottom-up after computing the core numbers of each vertex, but unlike ANH-TE, PHCD leverages the information from computing the core numbers to optimize for the k-core hierarchy, by reordering vertices based on their core numbers. This optimization allows them to more efficiently divide the work of constructing the hierarchy across different threads, and allows them to reduce the work in practice when iterating over the neighbors of a vertex v with larger core numbers than that of v. Compared to the sequential NH, ANH-TE is 4.67–58.84x faster, particularly on larger graphs.

For (2, 3) and (3, 4) nucleus decomposition, we compare our parallel ANH-TE and ANH-EL to the sequential NH [49]. Considering the fastest of ANH-TE and ANH-EL for each graph, our implementations are 3.76–23.54x faster, demonstrating that we achieve good speedups from our use of parallelization. Sequentially, our fastest algorithm is between 2.02x faster and 4.2x slower than NH.

8.4 Performance of Approximate Hierarchy

We considered $\delta = 0.1$, 0.5, and 1 for our experiments for approximate (r, s) nucleus decomposition (where δ is the approximation parameter in Algorithm 2). We first compare APPROX-ARB-NUCLEUS to ARB-NUCLEUS [55] and see a speedup of up to 16.16x for $\delta = 0.1$, up to 8.35x for $\delta = 0.5$, and up to 10.88x for $\delta = 1$.

Besides the computation of the (r, s)-clique-core numbers, APPROX-ANH-TE, APPROX-ANH-EL, and APPROX-ANH-BL are identical to ANH-TE, ANH-EL, and ANH-BL, respectively. In other words, their hierarchy construction procedure is the same. We observe up to a 3.3x speedup considering the fastest of our approximate algorithms for each graph and $r < s \leq 7$, over the fastest of our exact algorithms. Notably, for $\delta = 0.1$, we are able to compute the $(2, 5)$ nucleus decomposition hierarchy on friendster in 8783.2 seconds, where our exact implementations timeout at 4 hours. The improvements in running time using our approximate algorithms are lower than when comparing APPROX-ARB-NUCLEUS to ARB-NUCLEUS [55] because even in our approximate algorithms, s-cliques must be exactly counted per r-clique, and much of the time is spent doing this.

In terms of accuracy, the average error in the (r, s)-clique-core number per r-clique is relatively low for all of our δ values. For $\delta = 0.1$, across all of our graphs and for $r < s \leq 7$, our coreness estimates per r-clique are have a multiplicative error of 1–2.92x on average (with a median of 1.33x) compared to the exact coreness numbers. For $\delta = 0.5$, the coreness estimates range from having a multiplicative error of 1–2.92x on average as well, with a median of 1.34x, and for $\delta = 1$, the coreness estimates range from having a multiplicative error of 1–3.05x on average, with a median of 1.35x. The multiplicative errors of the maximum (r, s)-clique-core number, across all graphs and for $r < s \leq 7$, are also reasonably low, with a median of 1.6x for $\delta = 0.1$, a median of 2x for $\delta = 0.5$, and a median of 2x for $\delta = 1$. The maximum multiplicative error for a given r-clique is 6.73x for $\delta = 0.1$, 6.98x for $\delta = 0.5$, and 7.32x for $\delta = 1$, but these arise for large s, notably when $r = 5$ and $s = 7$ for all δ, and these errors are still much lower than the theoretical guarantee of $(\binom{s}{r} + \delta) \cdot (1 + \delta)$ given by Theorem 6.3.

9 CONCLUSION

We have presented new parallel exact and approximate algorithms for nucleus hierarchy construction with strong theoretical guarantees. We have developed optimized implementations of our algorithms, which interleave the coreness number computation with the hierarchy construction. Our experiments showed that our implementations outperform state-of-the-art implementations while achieving good parallel scalability.

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.

[2] Sabeur Aridhi, Martin Brugnara, Alberto Montresor, and Yannis Velegrakis. 2016. Distributed k-Core Decomposition and Maintenance in Large Dynamic Graphs. In *ACM International Conference on Distributed and Event-Based Systems*. 161–168.

[3] Gary D Bader and Christopher WV Hogue. 2003. An Automated Method for Finding Molecular Complexes in Large Protein Interaction Networks. *BMC Bioinformatics* 4, 1 (2003), 1–27.

[4] Maciej Besta, Armon Carigiet, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, and Torsten Hoefler. 2020. High-Performance Parallel Graph Coloring with Strong Guarantees on Work, Depth, and Quality. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[5] Mark Blanco, Tze Meng Low, and Kyungjoo Kim. 2019. Exploration of Fine-Grained Parallelism for Load Balancing Eager k-Truss on GPU and CPU. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[6] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. Brief Announcement: ParlayLib – A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[7] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.

[8] Yulin Che, Zhuohang Lai, Shixuan Sun, Yue Wang, and Qiong Luo. 2020. Accelerating Truss Decomposition on Heterogeneous Processors. *Proc. VLDB Endow.* 13, 10 (June 2020), 1751–1764.

[9] Pei-Ling Chen, Chung-Kuang Chou, and Ming-Syan Chen. 2014. Distributed Algorithms for k-Truss Decomposition. In *IEEE International Conference on Big Data (BigData)*. 471–480.

[10] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (Feb. 1985), 210–223.

[11] Deming Chu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2022. Hierarchical Core Decomposition in Parallel: From Construction to Subgraph Search. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1138–1151. https://doi.org/10.1109/ICDE53745.2022.00090

[12] Jonathan Cohen. 2008. Trusses: Cohesive Subgraphs for Social Network Analysis. *National Security Agency Technical Report* 16, 3.1 (2008).

[13] Ye Conghuan. 2011. Dense Subgroup Identifying in Social Network. In *International Conference on Advances in Social Networks Analysis and Mining*. 555–556.

[14] Alessio Conte, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. Discovering k-Trusses in Large-Scale Networks. In *IEEE High Performance extreme Computing Conference (HPEC)*. 1–6.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.)*. MIT Press.

[16] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 293–304.

[17] Laxman Dhulipala, Quanquan C. Liu, Sofya Raskhodnikova, Jessica Shi, Julian Shun, and Shangdi Yu. 2022. Differential Privacy from Locally Adjustable Graph Algorithms: k-Core Decomposition, Low Out-Degree Ordering, and Densest Subgraphs. In *63rd IEEE Annual Symposium on Foundations of Computer Science*. 754–765.

[18] Fatemeh Esfahani, Venkatesh Srinivasan, Alex Thomo, and Kui Wu. 2022. Nucleus Decomposition in Probabilistic Graphs: Hardness and Algorithms. In *IEEE International Conference on Data Engineering (ICDE)*. 218–231.

[19] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. 2019. Efficient Algorithms for Densest Subgraph Discovery. *Proc. VLDB Endow.* 12, 11 (July 2019), 1719–1732.

[20] Martin Farach-Colton and Meng-Tsung Tsai. 2014. Computing the Degeneracy of Large Graphs. In *Latin American Symposium on Theoretical Informatics*. 250–260.

[21] Eugene Fratkin, Brian T Naughton, Douglas L Brutlag, and Serafim Batzoglou. 2006. MotifCut: Regulatory Motifs Finding with Maximum Density Subgraphs. *Bioinformatics* 22, 14 (2006), e150–e157.

[22] Hillel Gazit. 1991. An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph. *SIAM J. Comput.* 20, 6 (1991), 1046–1067.

[23] Mohsen Ghaffari, Silvio Lattanzi, and Slobodan Mitrović. 2019. Improved Parallel Algorithms for Density-Based Network Clustering. In *Proceedings of the 36th International Conference on Machine Learning*. 2201–2210.

[24] David Gibson, Ravi Kumar, and Andrew Tomkins. 2005. Discovering Large Dense Subgraphs in Massive Graphs. In *Proc. VLDB Endow.* VLDB Endowment, 721–732.

[25] J. Gil, Y. Matias, and U. Vishkin. 1991. Towards a Theory of Nearly Constant Time Parallel Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 698–710.

[26] Q. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen. 2020. Faster Parallel Core Maintenance Algorithms in Dynamic Graphs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 31, 6 (2020), 1287–1300.

[27] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-Truss Community in Large and Dynamic Graphs. In *ACM SIGMOD International Conference on Management of Data*. 1311–1322.

[28] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 276–287.

[29] Yihao Huang, Claire Wang, Jessica Shi, and Julian Shun. 2023. Efficient Algorithms for Parallel Bi-core Decomposition. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 17–32.

[30] J. Jaja. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[31] Siddhartha V. Jayanti and Robert E. Tarjan. 2016. A Randomized Concurrent Algorithm for Disjoint Set Union. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 75–82.

[32] H. Jin, N. Wang, D. Yu, Q. Hua, X. Shi, and X. Xie. 2018. Core Maintenance in Dynamic Graphs: A Parallel Approach Based on Matching. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 29, 11 (2018), 2416–2428.

[33] H. Kabir and K. Madduri. 2017. Parallel k-Core Decomposition on Multicore Platforms. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1482–1491.

[34] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-Truss Decomposition on Multicore Systems. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[35] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. k-Core Decomposition of Large Networks on a Single PC. *Proc. VLDB Endow.* 9, 1 (2015), 13–23.

[36] Kartik Lakhotia, Rajgopal Kannan, Viktor K. Prasanna, and César A. F. De Rose. 2020. RECEIPT: REfine CoarsE-grained IndePendent Tasks for Parallel Tip decomposition of Bipartite Graphs. *Proc. VLDB Endow.* 14, 3 (2020), 404–417.

[37] Jure Leskovec and Andrej Krevl. 2019. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[38] R. Li, J. Yu, and R. Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *IEEE Transactions on Knowledge & Data Engineering (TKDE)* 26, 10 (oct 2014), 2453–2465.

[39] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical Core Maintenance on Large Dynamic Graphs. *Proc. VLDB Endow.* 14, 5 (2021), 757–770.

[40] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2020. Efficient (α, β)-Core Computation in Bipartite Graphs. *Proc. VLDB Endow.* 29, 5 (2020), 1075–1099.

[41] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems. In *ACM Symposium on Parallelism in Algorithms and Architectures*. 191–204.

[42] Tze Meng Low, Daniele G. Spampinato, Anurag Kutuluru, Upasana Sridhar, Doru Thom Popovici, Franz Franchetti, and Scott McMillan. 2018. Linear Algebraic Formulation of Edge-Centric k-Truss Algorithms with Adjacency Matrices. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[43] Qi Luo, Dongxiao Yu, Xiuzhen Cheng, Zhipeng Cai, Jiguo Yu, and Weifeng Lv. 2020. Batch Processing for Truss Maintenance in Large Dynamic Graphs. *IEEE Transactions on Computational Social Systems* 7, 6 (2020), 1435–1446.

[44] Qi Luo, Dongxiao Yu, Hao Sheng, Jiguo Yu, and Xiuzhen Cheng. 2021. Distributed Algorithm for Truss Maintenance in Dynamic Graphs. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. 104–115.

[45] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (July 1983).

[46] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2012. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 24, 2 (2012), 288–300.

[47] Ahmet Erdem Sariyüce. 2021. Motif-Driven Dense Subgraph Discovery in Directed and Labeled Networks. In *The Web Conference (WWW)*. 379–390.

[48] Ahmet Erdem Sarıyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2016. Incremental k-Core Decomposition: Algorithms and Evaluation. *Proc. VLDB Endow.* 25, 3 (2016), 425–447.

[49] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 97–108.

[50] Ahmet Erdem Sariyüce and Ali Pinar. 2018. Peeling Bipartite Networks for Dense Subgraph Discovery. In *ACM International Conference on Web Search and Data Mining (WSDM)*. 504–512.

[51] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. *Proc. VLDB Endow.* 12, 1 (2018), 43–56.

[52] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. 2017. Nucleus Decompositions for Identifying Hierarchy of Dense Subgraphs. *ACM Trans. Web* 11, 3, Article 16 (July 2017), 16:1–16:27 pages.

[53] Stephen B. Seidman. 1983. Network Structure and Minimum Degree. *Soc. Networks* 5, 3 (1983), 269 – 287.

[54] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Parallel Clique Counting and Peeling Algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*. 135–146.

[55] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2022. Theoretically and Practically Efficient Parallel Nucleus Decomposition. *Proc. VLDB Endow.* 15, 3 (feb 2022), 583–596.

[56] Jessica Shi and Julian Shun. 2020. Parallel Algorithms for Butterfly Computations. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APoCS)*. 16–30.

[57] Shaden Smith, Xing Liu, Nesreen K Ahmed, Ancy Sarah Tom, Fabrizio Petrini, and George Karypis. 2017. Truss Decomposition on Shared-Memory Parallel Systems. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

[58] Bintao Sun, T.-H. Hubert Chan, and Mauro Sozio. 2020. Fully Dynamic Approximate k-Core Decomposition in Hypergraphs. *ACM Trans. Knowl. Discov. Data (TKDD)* 14, 4, Article 39 (May 2020).

[59] Charalampos Tsourakakis. 2015. The k-Clique Densest Subgraph Problem. In *The Web Conference (WWW)*. 1122–1132.

[60] Liptia Venica and Gusti Ayu Putri Saptawati. 2021. Finding Dense Subgraph for Community Detection on Social Network Based on Information Diffusion. In *International Conference on Data and Software Engineering (ICoDSE)*. 1–6.

[61] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (May 2012), 812–823.

[62] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient Bitruss Decomposition for Large-Scale Bipartite Graphs. In *IEEE International Conference on Data Engineering (ICDE)*. 661–672.

[63] D. Wen, L. Qin, Y. Zhang, X. Lin, and J. Yu. 2019. I/O Efficient Core Graph Decomposition: Application to Degeneracy Ordering. *IEEE Transactions on Knowledge & Data Engineering (TKDE)* 31, 01 (jan 2019), 75–90.

[64] Yang Zhang and Srinivasan Parthasarathy. 2012. Extracting Analyzing and Visualizing Triangle k-Core Motifs within Networks. In *IEEE International Conference on Data Engineering (ICDE)*. 1049–1060.

[65] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *ACM SIGMOD International Conference on Management of Data*. 1024–1041.

[66] Y. Zhang, J. X. Yu, Y. Zhang, and L. Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *IEEE International Conference on Data Engineering (ICDE)*. 337–348.

[67] Feng Zhao and Anthony KH Tung. 2012. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. *Proc. VLDB Endow.* 6, 2 (2012), 85–96.

[68] Zhaonian Zou. 2016. Bitruss Decomposition of Bipartite Graphs. In *Database Systems for Advanced Applications (DASFAA)*. 218–233.