Parallel Algorithms for Hierarchical Nucleus Decomposition

JESSICA SHI, MIT CSAIL, USA
LAXMAN DHULIPALA, University of Maryland, College Park, USA
JULIAN SHUN, MIT CSAIL, USA

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, 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.

ACM Reference Format:

1 Introduction

Dense subgraph and substructure detection is a fundamental tool in graph mining, with many applications in areas including social network analysis [13, 61], fraud detection [24], and computational biology [3, 21]. 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 $(1,3)$ 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].

Authors’ addresses: Jessica Shi, MIT CSAIL, USA, jeshi@mit.edu; Laxman Dhulipala, University of Maryland, College Park, USA, laxman@umd.edu; Julian Shun, MIT CSAIL, USA, jshun@mit.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2836-6573/2024/2-ART32 $15.00
https://doi.org/10.1145/3639287
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 \(O(m\alpha^{-2})\) expected work and \(O(k \log n + \rho_{(r,s)}(G) \log n + \log^2 n)\) span w.h.p. (parallel time), where \(\alpha\) is the arboricity of the input graph \(G\), \(\rho_{(r,s)}(G)\) is the \((r,s)\)-clique core number 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 \(O(m\alpha^{-2})\) 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 \((\epsilon + \epsilon)\) factor of the current value of \(k\) to be peeled in parallel. We show that our algorithm generates coreness estimates that are an \((\epsilon + \epsilon)\)-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 30.9x self-relative parallel speedup, and a 3.76–58.8x 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 code is publicly available at https://github.com/jeshi96/arb-nucleus-hierarchy.

2 Related Work
The $k$-core [45, 53] and $k$-truss problems [12, 65, 68] are classic problems relating 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, 58, 62, 64, 69] and dynamic [1, 2, 26–28, 32, 38, 39, 41, 43, 44, 48, 59, 64, 66, 67] settings. Similar ideas have been studied in bipartite graphs [29, 36, 50, 57, 63].

A related problem is the $k$-clique densest subgraph problem [60], 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 $O(n\alpha^{-2})$, i.e., proportional to the number of $s$-cliques in the graph. Chu et al. [11] present a parallel algorithm 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 $O(n\alpha^{-2})$, i.e., proportional to the number of $s$-cliques in the graph. Chu et al. [11] present a parallel algorithm 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 $O(n\alpha^{-2})$, i.e., proportional to the number of $s$-cliques in the graph. Chu et al. [11] present a parallel algorithm 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 $O(n\alpha^{-2})$, i.e., proportional to the number of $s$-cliques in the graph. Chu et al. [11] present a parallel algorithm 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 $O(n\alpha^{-2})$, i.e., proportional to the number of $s$-cliques in the graph. Chu et al. [11] present a parallel algorithm 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 $O(n\alpha^{-2})$, i.e., proportional to the number of $s$-cliques in the graph. Chu et al. [11] present a parallel algorithm for solving the $k$-clique densest subgraph problem.
3 Preliminaries

Model of Computation. We use the work-span model of computation for our theoretical analysis, which is a standard model for 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] on $P$ processors, 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 ($\alpha$) of a graph is the minimum number of spanning forests needed to cover the graph. $\alpha$ 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 $O(ma^{k-2})$ work and $O(\log^2 n)$ span with high probability (w.h.p.) [54].

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 $(r, s)$ nucleus 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 $(1, 3)$ nucleus. We call two $r$-cliques $s$-clique-adjacent if there exists an $s$-clique $S$ such that both $r$-cliques are subgraphs 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 $\gamma$-approximate $(r, s)$-clique core number of $R$ is a value that is at least $k_R$ and at most $\gamma k_R$, 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 $O(a)$-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 $[1, \ldots, n]$.

\footnote{We say $O(f(n)) \text{ with high probability (w.h.p.)}$ to indicate $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$, where $n$ is the input size.}
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(\( x, \text{old}, \text{new} \)) attempts to atomically write the value \( \text{new} \) into memory location \( x \) if \( x \) current stores the value \( \text{old} \); 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 \( \text{old} \)). 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 \((r, s)\)-clique 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.
Algorithm 1 Parallel \((r, s)\) nucleus hierarchy algorithm

1: Initialize \(r, s\) \(\triangleright r\) and \(s\) for \((r, s)\) nucleus decomposition
2: procedure \textsc{arb-nucleus-hierarchy}(\(G = (V, E)\)) \(\triangleright\) Compute the nucleus core numbers, where \(ND\) maps \(r\)-cliques to their core numbers
3: \(k \leftarrow \) maximum core number in \(ND\)
4: For each \(i \in [k]\), let \(L_i\) denote a hash table, where the keys are \(r\)-cliques and the values are linked lists
5: parfor all \(s\)-cliques \(S\) in \(G\) do
6: parfor all pairs of \(r\)-cliques \(R, R'\) in \(S\) where \(ND[R'] \leq ND[R]\) do
7: Add \(R'\) to the linked list keyed by \(R\) in \(L_{ND[R']}\)
8: Initialize the hierarchy tree \(T\) with leaves corresponding to each \(r\)-clique
9: for \(i \in \{k, k-1, \ldots, 1\}\) do
10: Relabel each \(r\)-clique in each linked list in \(L_i\) with its corresponding value in \(ID_i\)
11: for each connected component \(C = \{R_1, \ldots, R_c\}\) in \(H\) where \(|C| \geq 2\) do
12: Construct a new parent in \(T\) for all of the nodes corresponding to each \(R_\ell\) (for \(\ell \in [c]\)), and represent the parent as the \(r\)-clique \(R_1\)
13: Concatenate in parallel the linked lists corresponding to all \(R_\ell\) (for \(\ell \in [c]\)) in \(L_j\), as the updated value for the key \(R_1\) in \(L_j\)
14: For each \(R_\ell\) (for \(\ell \in [c]\)), update the value of \(R_\ell\) in \(ID_j\) to be \(R_1\)
15: return \(T\)

Fig. 2. An example of the \(L_i\) data structures maintained by \textsc{arb-nucleus-hierarchy} while computing the \((1, 3)\)-nucleus 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).

5 Nucleus Decomposition Hierarchy

We now describe our theoretically efficient parallel nucleus decomposition hierarchy algorithm, \textsc{arb-nucleus-hierarchy}. \textsc{arb-nucleus-hierarchy} computes the hierarchy by first running an efficient parallel nucleus decomposition algorithm, \textsc{arb-nucleus} \cite{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. \textsc{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 \((1, 3)\)-nucleus decomposition in Figure 1. For simplicity, we have omitted labeling some of the vertices in the graph, and do not include them in our discussion. 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 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 \(s\)-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 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 super-nodes 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 the 4-(1, 3)-nucleus besides 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 note that there is nothing to be done for round $i = 4$, since $L_4$ is empty. Therefore, we begin by processing 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 a linear-work parallel 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 $\{R_1, \ldots, R_j\}$. 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 $R_\ell$ for $\ell \in [c]$ (Line 17). Note that each $R_\ell$ 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 $R_i$, 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 $R_i$ to be the concatenation, or we insert the concatenation into $L_j$ with $R_i$ as the key if $R_i$ does not already exist in the hash table (this is possible if $R_i$ had no neighbors with core number $j$). We use tombstones to delete the other keys $R_\ell$ ($\ell \neq 1$) in each $L_j$. Additionally, on Line 20, we update the component ID in $ID_j$ of each $R_\ell$ 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 \(s\)-slique count are peeled (removed). Importantly, \(k \leq \rho_{(r,s)}(G) \leq O(ma^s-2)\), 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 \(O(ma^s-2)\) 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 \(O(ma^s-2)\) 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 \(s\)-slicies in \(G\), which we incur regardless to store the hash tables \(L_i\). Additionally, iterating through all \(s\)-slicies in \(G\) (Line 6) is superseded by the work required to compute the \((r,s)\)-clique core numbers. For every pair of \(r\)-slicies in each \(s\)-slique, we hash each \(r\)-slique and append to a linked list (Lines 7–8), which in total takes \(O(ma^s-2)\) 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 \(O(ma^s-2)\) 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 \(O(ma^s-2)\) work as well.

In more detail, for fixed \(i\), let the sum of the lengths of the linked lists in \(L_i\) be \(\ell_i\), 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 \(\ell_i\) 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 \(\ell_i\), so performing connectivity on \(H\) (Line 15) takes work linear in \(\ell_i\) 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 \(O(\sum_i (\ell_i + y_i)) = O(ma^s-2)\) expected work and \(O(k \log n)\) span w.h.p.

It remains to bound the cost 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 \(ID_i\). Thus, the concatenations are bounded by \(O(ma^s-2)\) 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 \(R_j\) as the new representative for the other \(r\)-cliques in the component (upating \(ID_j\)). Thus, we can assign the work of iterating through all \(j \leq i\) to the \(r\)-cliques that are being merged (\(R_\ell\) where \(\ell \neq 1\)). Once the \(r\)-clique \(R_\ell\) is merged, due to the concatenation on Line 19 and the update in
where the span is due to list ranking. Thus, we improve the sequential running time of constructing the nucleus decomposition hierarchy, leading to a hierarchy tree with polylogarithmic height. Our hierarchy peeling rounds, thus significantly improving upon the span. As a result, we only have polylogarithmic expected work. In contrast, our algorithm has a much lower total expected work of $\rho(r,s)(G)\log n + \log^2 n$ 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 \textsc{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 $\alpha$ 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 $O(\alpha n_s, n_r)$ (this is achieved using the state-of-the-art algorithm for computing the $(r,s)$-clique core numbers [55]). Our \textsc{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 $O(\alpha^s)$, and \textsc{arb-nucleus-hierarchy} is work-efficient.

Shi et al. [55] present a parallel algorithm to compute the $(r,s)$-clique core numbers in $O(\alpha^s)$ expected work. A vanilla extension of their algorithm to compute the hierarchy would be to run connected components for each non-empty core number $i$, with vertices being the $(r,s)$-cliques that have core number $i$ or above, in a top-down manner. This incurs $O(\alpha^s)$ expected work per level, and there can be $\rho(r,s)(G)$ levels in the worst case, leading to $O(\rho(r,s)(G)\alpha^s)$ total expected work. In contrast, our algorithm has a much lower total expected work of $O(\alpha^s)$.

### 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, \textsc{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 $((\frac{s}{r}) + \varepsilon)$-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 $(\frac{s}{r} + \varepsilon) \cdot k_R$.

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 \textsc{arb-approx-nucleus-hierarchy} is exactly the same as that of \textsc{arb-nucleus-hierarchy}, and the only salient difference is that for \textsc{arb-approx-nucleus-hierarchy}, we replace the \textsc{arb-nucleus} subroutine in Line 3 of Algorithm 1 with our approximate nucleus decomposition subroutine, \textsc{approx-arb-nucleus}.

\textbf{Our Algorithm.} We present our pseudocode for \textsc{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 \( \epsilon \) in the \((\ell') + \epsilon\)-approximation. Specifically, \textsc{approx-arb-nucleus} gives a \((\ell' + \delta) \cdot (1 + \delta) = (\ell' + \epsilon)\)-approximation, which we prove in Theorem 6.1.

First, on Line 3, \textsc{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 \textit{et al.} \cite{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, \textsc{rec-list-cliques}, from Shi \textit{et al.}'s previous work \cite{54}. The key difference between \textsc{approx-arb-nucleus} and \textsc{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 \( \delta \), we define the range of each bucket \( B_i \) to be \( \left( (\ell' + \delta) \cdot (1 + \delta)^i, (\ell' + \delta) \cdot (1 + \delta)^{i+1} \right] \), where \( i \in \left[ \lfloor s \log_{1+\delta} n \rfloor \right] \) since \( (\ell') = O(n^2) \) 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 \cite{55}, except using our modified bucketing structure. While not all \( r \)-cliques have been peeled, \textsc{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 \( r \)-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 \( \leq (\ell' + \delta) \cdot (1 + \delta)^i \), which is important for our theoretical bounds. Note that we process a given bucket \( B_i \) at most \( O(\log_{1+\delta}(\ell'))(n) \) 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. \textsc{arb-approx-nucleus-hierarchy} is then given by replacing \textsc{arb-nucleus} in Line 3 of Algorithm 1 with \textsc{approx-arb-nucleus}.

\textbf{Theoretical Guarantees and Efficiency.} We now discuss the theoretical guarantees and theoretical efficiency of \textsc{approx-arb-nucleus}, and by extension, \textsc{arb-approx-nucleus-hierarchy}.

The main idea is to bound the proportion of \( r \)-cliques with core numbers \( \leq \ell \) for some fixed \( \ell \), but with \( s \)-clique-degree \( > \ell(\ell') + \delta \). We can then set \( \ell \) 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 \( \ell \) 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. Due to space constraints, we defer the proof of the following theorem to the full paper \cite{56}.
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 8.1.

**Algorithm 2 Approximate parallel $(r, s)$ nucleus algorithm**

1. Initialize $r, s$ for $(r, s)$ nucleus decomposition
2. procedure APPROX-ARB-NUCLEUS($G = (V, E), \delta$)
3. \[DG \leftarrow \text{ARB-ORIENT}(G)\] \hspace{1cm} \text{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($DG, s, U$) \hspace{1cm} \text{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 $\left(\binom{r}{2}^i + \delta \cdot (1+\delta)^i, \binom{r}{2}^i + \delta \cdot (1+\delta)^{i+1}\right)$, for all $i \in \lfloor \log (r, s)_n \rfloor$
7. finished $\leftarrow 0$, num_rounds $\leftarrow 0$, $i \leftarrow 0$
8. while finished $< |U|$ do
9. \[A \leftarrow r\text{-cliques in bucket } B_i \text{ in } ND \text{ to be peeled}\]
10. finished $\leftarrow$ finished + |$A$|
11. num_rounds $\leftarrow$ num_rounds + 1
12. parfor all $r$-cliques $R$ in $A$ do
13. \[\text{parfor all } s\text{-cliques } S \text{ containing } R \text{ do}\]
14. \[\text{parfor all } r\text{-cliques } R' \text{ in } S \text{ where } R' \neq R \text{ do}\]
15. Update $s$-clique count of $R'$ in $U$
16. Peel $A$, assign the upper bound of the bucket as the coreness value for peeled $r$-cliques, and update the buckets of $r$-cliques with updated $s$-clique counts
17. if num_rounds $\geq O(\log r, s) / \binom{r}{2} (n))$ 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. $i \leftarrow i + 1$
20. num_rounds $\leftarrow 0$
21. return $ND$

**Theorem 6.1.** ARB-APPROX-NUCLEUS-HIERARCHY computes a $(\binom{r}{2} + \epsilon)$-approximate $(r, s)$ nucleus decomposition hierarchy in $O(m^2 + m\log n)$ expected work and $O(\log n)$ span w.h.p.

Note that in the theorem above, $\epsilon$ does not affect the work bound, and affects the base of one of the logarithms in the span bound (which we omitted as we treat $\epsilon$ as a constant).

**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 8.1.

**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 \texttt{Link} subroutine to process the \( s \)-clique-adjacent \( r \)-cliques given the intermediate \((r, s)\)-clique core numbers throughout the peeling algorithm. Based on \texttt{Link}, \texttt{Construct-Tree} constructs the final hierarchy tree.

In more detail, \texttt{Arb-Nucleus-Decomp-Hierarchy-Framework} first uses an efficient low out-degree orientation algorithm by Shi \textit{et al.} \cite{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 \texttt{Arb-Nucleus-Decomp-Hierarchy-Framework} uses a subroutine \texttt{Rec-List-Cliques} based on previous work by Shi \textit{et al.} \cite{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 \textit{et al.} \cite{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 until 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 \cite{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 \texttt{Link} subroutine (Lines 14–15). The \texttt{Link} subroutine will construct the hierarchy, and we describe it in Sections 7.2 and 7.3.
Algorithm 4 Basic link and tree construction

1: Initialize \( k \) union-find data structures, \( u_f^i \) for \( i \in [k] \), where \( k \) is the maximum \((r,s)\)-clique core number

2: **procedure** \textsc{link-basic}(\( R, Q, ND \))

3: \hspace{1em} \parfor \( i \in [\min(ND[R], ND[Q])] \) \do \hspace{1em} \( u_f^i, \text{unite}(R, Q) \)

4: **procedure** \textsc{construct-tree-basic}(\( ND \))

5: \hspace{1em} \texttt{Initialize the hierarchy tree} \( T \) \texttt{with leaves corresponding to each} \( r \)-\texttt{clique}

6: \hspace{1em} \parfor \( i \in [k, k-1, \ldots, 1] \) \do

7: \hspace{2em} \parfor \( \text{each connected component} \ C = \{R_1, \ldots, R_c\} \) \in \( u_f^i \) \do

8: \hspace{3em} \text{Construct a new parent in} \( T \), \text{to be the parent of the roots of the leaf nodes corresponding to each} \( R_\ell \) \text{for} \( \ell \in [c] \)

9: \hspace{1em} \text{return} \( T \)

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, \textsc{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].

7.2 Basic Version of \textsc{link}

The remaining details are in how we perform the \textsc{link} subroutine (Line 15) and how we construct the hierarchy tree \( T \) with \textsc{construct-tree} (Line 18).

In Algorithm 4, we present a basic \textsc{link} subroutine, \textsc{link-basic}, and the corresponding \textsc{construct-tree} subroutine, \textsc{construct-tree-basic}. \textsc{link-basic} maintains a parallel union-find data structure \( u_f^i \) per core number \( i \in [k] \), which corresponds to a level of the hierarchy tree \( T \). Each \( u_f^i \) connects \( r \)-cliques that are \( s \)-clique-adjacent considering only \( r \)-cliques with core numbers \( \geq i \). To construct these \( u_f^i \)'s, given two \( r \)-cliques \( R \) and \( Q \), \textsc{link-basic} simply unites \( R \) and \( Q \) in each \( u_f^i \) where \( i \leq \max(ND[R], ND[Q]) \) (Lines 3–4). Then, given the \( u_f^i \) for all \( i \in [k] \), we construct the hierarchy tree \( T \) from the bottom up, starting with leaf nodes corresponding to \( r \)-cliques. \textsc{construct-tree-basic} begins with \( i = k \), where for each connected component in \( u_f^k \) (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 \( u_f^i \), 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, \textsc{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 \( k \) \texttt{unite} operations. Indeed, in Section 8.1, we empirically show that \textsc{link-basic} performs many unnecessary \texttt{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, \textsc{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 \textsc{link} and \textsc{construct-tree} subroutines.

7.3 Efficient Version of \textsc{link}

Our improved subroutines \textsc{link-efficient} and \textsc{construct-tree-efficient} are shown in Algorithm 5. We refer to an example of \((1,3)\)-nucleus decomposition in the graph in Figure 1. Recall that we have omitted labeling some vertices in the graph for simplicity.

The main idea of \textsc{link-efficient} is instead of maintaining \( k \) union-find data structures, we maintain a single parallel union-find data structure \( u_f \) and an additional hash table \( L \) that maps \( r \)-cliques to \( r \)-cliques. First, \( u_f \) 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
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($R, LQ, ND$)
23:             break
24:     else
25:         LINK-EFFICIENT($R, L[Q], ND$)
26:     break
27: end procedure
28: procedure CONSTRUCT-TREE-EFFICIENT(ND)
29:     Initialize the hierarchy tree $T$ with leaves corresponding to each $r$-clique
30:     parfor each connected component $C = \{R_1, \ldots, R_e\}$ 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:             $R ← uf.parent(L[C])$  $\triangleright$ Note that $C$ is the $r$-clique representing the component in $uf$
35:             Make $uf_C$ a child of $uf_R$ in $T$  $\triangleright$ Note that $uf_R$ necessarily exists since $R$ represents a component in $uf$
36:     return $T$

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 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
Fig. 4. An example of the $u_f$ and $L$ data structures maintained by \textsc{link-efficient} when computing the $(1,3)$-nucleus hierarchy on the graph in Figure 1. The data structures are shown after the third and fourth rounds of peeling in \textsc{arb-nucleus-decomp-hierarchy-framework}, and after intermediate calls to \textsc{link-efficient} within the fourth round.

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 $u_f$, let $R'$ be an $r$-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 $u_f$. 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 $u_f$, which would resolve to the same parent. \textsc{link-efficient} updates $u_f$ and $L$, depending on the core numbers of the given $r$-cliques $R$ and $Q$.

**Tree Construction.** We describe first how \textsc{construct-tree-efficient} constructs the hierarchy tree $T$ given the specifications for $u_f$ and $L$. We refer to the construction shown in Figure 5, where $u_f$ 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. 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 $u_f$ 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

![Fig. 5. An example of the $(1,3)$-nucleus hierarchy tree on the graph in Figure 1, constructed by \textsc{construct-tree-efficient}. Stage 1 shows an intermediate tree constructed in the \textsc{construct-tree-efficient} subroutine, and Stage 2 depicts the final hierarchy tree.](image-url)
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 \( R = uf.parent(L[C]) \), 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 \( uf_{4a}, uf_{3b} \), 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 8.1 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 arbor-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 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 concurrently.

- $R = 3a, Q = 4c$: 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 $(L[R], uf.parent(R)) = (1a, 3b)$ (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 \textsc{link-efficient} call to \((R, L[Q]) = (2a, 3a) \text{ (Line 26)}\).

\(- R = 2a, Q = 3a \): 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 \textsc{link-efficient} on \((R, L[Q]) = (2a, 1a) \text{ (Line 23)}\).

\(* R = 1a, Q = 2a \): 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 \textsc{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, \textsc{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 \textsc{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 is a significant overhead.

Our main innovation in \textsc{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, \textsc{nh} uses \( \binom{k}{3} \cdot n_s + n_r \) additional space, where \( n_s \) is the number of \( s \)-cliques. In addition, \textsc{nh} is sequential, whereas \textsc{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 \textsc{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, \textsc{construct-tree-efficient}, is fully parallel.

### 7.4 Practical Version of \textsc{arb-nucleus-hierarchy}

Finally, we make certain optimizations to our theoretically-efficient \((r, s)\) nucleus decomposition hierarchy algorithm, \textsc{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 lists (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 8.1. For each such pair of \( r \)-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

**Environment and Inputs.** We run our experiments on a Google Cloud Platform instance with 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 \textsc{arb-nucleus-hierarchy} (Algorithm 1, with the optimiza-
tions described in Section 7.4), which we call \textsc{anh-te}, and our nucleus decomposition hierarchy
framework \textsc{arb-nucleus-decomp-hierarchy-framework} (Algorithm 3) using both \textsc{link-basic}
(Algorithm 4) and \textsc{link-efficient} (Algorithm 5), which we call \textsc{anh-bl} and \textsc{anh-el}, respectively.

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

We compare our hierarchy algorithms to \textsc{nh}, the state-of-the-art sequential \((r, s)\) nucleus decom-
position hierarchy implementation by Sariyüce and Pinar [49] for \((1, 2)\), \((2, 3)\), and \((3, 4)\) nucleus
decomposition. Note that \textsc{nh} does not generalize to other \( r \) and \( s \) values. \textsc{nh}, like \textsc{anh-bl} and
\textsc{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 \textsc{phcd}, the state-of-the-art parallel \( k \)-core hierarchy
implementation by Chu et al. [11].

#### 8.1 Comparison of \textsc{anh-te}, \textsc{anh-el}, and \textsc{anh-bl}

Figure 6 compares our exact \((r, s)\) nucleus decomposition hierarchy algorithms, \textsc{anh-te}, \textsc{anh-el},
and \textsc{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 8.2). However, our running times do include the time
required to compute the \((r, s)\)-clique-core numbers of each \( r \)-clique, which notably for \textsc{anh-te} is
given by \textsc{arb-nucleus} from [55].

<table>
<thead>
<tr>
<th>Graph</th>
<th>( n )</th>
<th>( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon</td>
<td>334,863</td>
<td>925,872</td>
</tr>
<tr>
<td>dblp</td>
<td>317,080</td>
<td>1,049,866</td>
</tr>
<tr>
<td>youtube</td>
<td>1,134,890</td>
<td>2,987,624</td>
</tr>
<tr>
<td>skitter</td>
<td>1,696,415</td>
<td>11,095,298</td>
</tr>
<tr>
<td>livejournal</td>
<td>3,997,962</td>
<td>34,681,189</td>
</tr>
<tr>
<td>orkut</td>
<td>3,072,441</td>
<td>117,185,083</td>
</tr>
<tr>
<td>friendster</td>
<td>65,608,366</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Sizes of our input graphs, which are from SNAP [37].
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-TE 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 $\binom{s}{2}$ 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.

To provide some context into how long the hierarchy construction takes compared to the coreness computation, we measured the time for just computing coreness values and compared it to the total time for each of our algorithms. The coreness computation time represents 46.53%, 35.26%, and 36.08% of the total time on average in ANH-EL, ANH-BL, and ANH-TE, respectively. Note that this ignores the locality effects of interleaving the computations in ANH-EL and ANH-BL. Nevertheless, we see that on average, ANH-EL spends the least amount of time on hierarchy construction, indicating the effectiveness of our optimizations.

### 8.2 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. 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 (1,2), (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. We get slowdowns due to the additional overheads of parallel subroutines and the fact that our code has a general structure for supporting arbitrary (r,s) values, whereas the code of [49] is specialized for the specific values of (1,2), (2,3), and (3,4). These slowdowns are somewhat counteracted by our faster clique enumeration algorithm, leading to speedups in some cases.

**Usefulness of the Hierarchy.** In Figure 10, we show the edge densities of various (2,s)-nuclei in the youtube graph. The edge density of a subgraph S is defined to be the number of edges in S divided by $\binom{|S|}{2}$ (the total possible number of edges). Edge density is the metric used to evaluate the quality of subgraphs from nucleus decompositions [52], and finding dense subgraphs subject to a size constraint has important applications.

For a fixed r and s, having the nucleus hierarchy enables us to efficiently find all of the c-(r,s) nuclei for any value of c by simply removing all internal nodes corresponding to c’-(r,s) nuclei for $c’ < c$, and having each resulting subtree be a c-(r,s) nucleus that contains all of its leaves. On the other hand, given only the core numbers and no hierarchy, to find the c-(r,s) nuclei for a given
value of $c$, one would need to run connected components on all $r$-cliques with core number $\geq c$, with connectivity defined by $s$-clique adjacency, which is much more expensive than simply cutting the hierarchy. Indeed, Figure 10 shows the time needed to generate all $(2, s)$ nuclei for various values of $s$ with and without the hierarchy for the Youtube graph. We see that using the hierarchy is 5.84–834.09x faster than not using the hierarchy.

8.3 Performance of Approximate Hierarchy

We considered $\delta = 0.1, 0.5$, and 1 for our experiments for approximate $(r, s)$ nucleus decomposition ($\delta$ 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 $\delta$ 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 $\delta$, and these errors are still much lower than the theoretical guarantee of $(\binom{r}{s} + \delta) \cdot (1 + \delta)$ given by Theorem 6.1. Note that the theoretical bound is a worst case bound. In practice, $(r, s)$-cliques whose core numbers are close to the end of the range of each bucket will have a much better approximation. Furthermore, graphs with a low peeling complexity will likely have more $(r, s)$-cliques peeled before moving to the next round (Line 17 of Algorithm 2), which will make the approximation better.

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.

Acknowledgements. This research was supported by NSF Graduate Research Fellowship #1122374, DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, cloud computing credits from Google-MIT, FinTech@CSAIL Initiative, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by the Semiconductor Research Corporation (SRC) and DARPA.

References


Parallel Algorithms for Hierarchical Nucleus Decomposition


[64] 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.


Received July 2023; revised October 2023; accepted November 2023