Efficient Algorithms for Parallel Bi-core Decomposition

YIHAO HUANG, Phillips Academy, yhuang23@andover.edu
CLAIRE WANG, Phillips Academy, cwang23@andover.edu

Abstract. Graphs are used in the modeling of social networks, biological networks, user-product networks, and many other real-world relationships. Identifying dense regions within these graphs can often aid in applications including product-recommendation, spam identification, and protein-function discovery. A fundamental dense substructure discovery problem in graph theory is the k-core decomposition. However, the k-core decomposition does not directly apply to bipartite graphs, which are graphs that model the connections between two disjoint sets of entities, such as book-authorship, affiliation, and gene-disease association. Given the prevalence of bipartite graphs, solving the dense subgraph discovery problem on bipartite graphs has wide-reaching real-world impacts.

In this paper, we solve the bipartite analogue of the k-core decomposition problem, which is the bi-core decomposition problem. Existing sequential bi-core decomposition algorithms are not scalable to large-scale bipartite graphs with hundreds of millions of edges. Therefore, we develop a theoretically efficient parallel bi-core decomposition algorithm. Our algorithm improves the theoretical bounds of existing algorithms, reducing the length of the computation graph's longest dependency path, which asymptotically bounds the runtime of a parallel algorithm when there are sufficiently many processors. We prove the problem of bi-core decomposition to be P-complete. We also devise a parallel bi-core index structure to allow for fast queries of the computed cores. Finally, we provide optimized parallel implementations of our algorithms that are scalable and fast. Using 30 threads, our parallel bi-core decomposition algorithm achieves up to a 44x speedup over the best existing sequential algorithm and up to a 2.9x speedup over the best existing parallel algorithm. Our parallel query implementation is up to 22.3x faster than the existing sequential query implementation.

Additional Key Words and Phrases: parallel computing, bi-core, bipartite graphs, k-core, graph theory

CONTENTS

Abstract 1
Contents 2
1 Introduction 3
2 Related Work 5
3 Preliminaries 6
4 Sequential Bi-core Decomposition 8
4.1 Sequential Peeling 8
4.2 Computation Sharing 8
4.3 Memory-Efficient Bi-core Index 9
5 Parallel Bi-core Decomposition 10
5.1 Parallel Bucketing and Exponential Search 11
5.2 Parallel Bi-core Decomposition 12
5.3 P-completeness 14
5.4 Peeling Space Pruning Optimization 15
6 Parallel Bi-core Index Structure 17
6.1 Parallel Index Construction 17
7 Experiments 18
7.1 Experiment Setup 18
7.2 Implementation and Other Optimizations 19
7.3 Bi-core Decomposition 19
7.4 Bi-core Index Construction 21
7.5 Bi-core Index Query 21
8 Conclusion 22
References 22

1 INTRODUCTION

Motivation. The problem of discovering dense clusters in networks is fundamental in large-scale graph analysis. It has applications in community search of social networks, clustering word-documents, improving advertising, detecting frauds, and analyzing protein-gene-disease relations in bioinformatics and medicine [27, 36]. Classic algorithms for dense subgraph discovery include k-core [25] decomposition, k-truss [8], and nucleus decomposition [31]. However, these algorithms apply to general graphs, and do not take into account the bipartite structures that exist in many real-world graphs.

A bipartite graph G consists of two mutually exclusive sets of vertices U, V and edges that connect between them. They model the affiliation between two distinct types of entities. Notably, bipartite graphs have been used to model authorship networks, group membership networks, peer-to-peer exchange networks, gene-disease associations, protein-protein interactions, and enzyme-reaction links [15–17, 27, 49].

Traditional dense substructure analysis on bipartite graphs projects bipartite graphs to unipartite co-occurrence networks by connecting two vertices if they share a neighbor, creating an edge. Then, k-core decomposition or a similar analysis is performed on the unipartite co-occurrence network. Co-occurrence network analysis can lose important connectivity information and cause an explosion of number of edges, making it practically inefficient [36]. Therefore, bipartite analogues for classic unipartite dense subgraph discovery algorithms are crucial for efficient and accurate dense substructure analysis on bipartite graphs.

Given the practical values of bipartite graphs, generalizing problems and algorithms for unipartite graphs to bipartite graphs has become a recent popular direction of research [1, 32, 47, 53]. The bipartite equivalent of k-core (bi-core) was introduced by Ahmed *et al.* [1]. A (α, β)-core (or a bi-core) is the maximal subgraph where the induced degrees of all vertices in the first partition is $\geq \alpha$ and the induced degrees of all vertices in the second partition is $\geq \beta$.

Applications. Bi-core decomposition has been applied to recommendation systems, fraud detection, and community search [4, 14, 48]. Below, we list 3 specific applications of bi-core decomposition.

(1) *Fraudster Detection* Bi-core decomposition can be applied to a social network graph for fraudster/spammer detection by considering the bipartite graph connecting user accounts to posts they like/dislike/upvote/downvote. A common strategy of fraudulent online influence campaigns is to create a large number of fake social media accounts to like/dislike specific posts or online products in order to manipulate public opinions. These fake accounts are generally created to like/dislike a small number of posts; those posts generally receives lots of likes/dislikes. Therefore, a fraudulent influence campaign can be identified by identifying (α, β)-cores with a low α value (corresponding to each user's degree) and a high β value (corresponding to each post/product's degree).

(2) *Graph Visualization* Algarra *et al.* introduced k-core decomposition for visualizing bipartite biological networks modeling gene-protein, host-pathogen, and predator-prey interactions. k-core decomposition identifies dense communities within the bipartite graph, which represents communities of generalists (species that interact with many other species; for example, predators that prey on many species). These dense substructures help researchers identify critical species in an ecosystem. Similarly, bi-core decomposition can also be applied to the problem and can potentially generate more accurate representations since it addresses the imbalance between the two entities. For example, there are generally more predator species than prey species, so intuitively, the number of degrees a prey species needs to be considered a generalist should be higher than that of a predator species [29].

(3) *Community Search* Wang *et al.* applied bi-core decomposition to find significant bipartite communities, which are densely-connected bipartite subgraphs with high edge weights containing a specific query vertex [48]. Their

approach uses bi-core decomposition as a subroutine to narrow down the search range of significant bipartite communities.

Parallel Bi-core Decomposition. Liu *et al.* [24] proposed the state-of-the-art sequential index-based approach for bi-core decomposition. Their algorithm runs in $O(m^{\frac{3}{2}})$ time and $O(m)$ space, where m is the number of edges. They leveraged computation-sharing across different rounds of peeling to improve upon prior works [7, 24]. Liu *et al.* also introduces a parallel version of their algorithm. However, their parallel algorithm only parallelizes between rounds of peeling and does not parallelize the peeling process itself. As a result, it has long sequential dependencies, which limits its scalability to different large real-world graphs.

As the sizes of graphs such as social media networks and biology graphs increase, leveraging parallelism to speed up bi-core decomposition becomes increasingly crucial. The importance of parallelism also grows as it becomes increasingly difficult to increase CPU clock speeds, as a result of which, chip manufacturers have turned to increasing the number of cores in a CPU [40]. As the number of cores in a CPU grows, shared-memory parallelism in particular becomes necessary in order to take advantage of the growing number of cores [43].

Motivated by the need of an efficient parallel bi-core decomposition algorithm, we develop in this paper a shared-memory parallel bi-core decomposition algorithm. Our algorithm uses a peeling-based approach, where each round of peeling removes all vertices with the lowest induced degree concurrently from the graph until the graph is empty. We use the classic *work-span* model to analyze the theoretical complexity of our parallel algorithm. In short, the *work* is the total number of operations performed, and the *span* (or the *depth*) is the length of the longest chain of sequential dependencies. We prove that our algorithm achieves $O(m^{\frac{3}{2}})$ work, and because the work complexity matches the time complexity of the best sequential algorithm, our algorithm is *work-efficient*. Our algorithm achieves $O(\rho\log(n))$ span *w.h.p.*[1]. n is the number of vertices. We define ρ to be the bi-core peeling complexity, or the maximum number of rounds of peeling required until the graph is empty. Additionally, our algorithm uses $O(m)$ space.

Note that ρ is upperbounded by n, so our span is $O(\rho\log(n)) = O(n\log(n))$ *w.h.p.*. In comparison, the parallel algorithm introduced by Liu *et al.* has a span of $O(m)$. Thus, for sufficiently large graphs, our parallel algorithm has better span than Liu *et al.*'s algorithm. Moreover, on real world graphs, we find that $\rho\log(n)$ is generally 2–3 orders of magnitude smaller than m in practice.

Moreover, we prove the problem of bi-core decomposition to be P-complete. It is as such unlikely that there exists a parallel bi-core decomposition algorithm with polylogarithmic span.

In addition, to allow finding all vertices $v \in (\alpha, \beta)$-core in work linear to the size of the core, we develop a parallel index structure as an extension of Liu *et al.*'s sequential index structure. While Liu *et al.* provided a parallel bi-core decomposition algorithm, they did not provide a parallel index construction algorithm. We provide a work-efficient, $O(\log(n))$ span algorithm to construct our index structure in parallel and a work-efficient, $O(1)$ span algorithm to perform the query based on the index structure.

Finally, we implement all of our parallel algorithms and introduce practical optimizations to improve their performance on real-world graphs. We present a comprehensive experimental evaluation of our algorithms on real-world graphs that contain up to hundreds of millions of edges. We compare our experimental results against Liu *et al.*'s parallel and sequential algorithms, which we use as our baselines. Our bi-core decomposition algorithm achieves up to 44x speedup over Liu *et al.*'s sequential algorithm on a machine with 30 cores and two-way hyperthreading. Furthermore, it achieves a 2.9x speedup over their parallel algorithm. Our parallel index query achieves 22.3x speedup over Liu *et al.*'s sequential

[1] w.h.p. stands for *with high probability* (meaning a probability of $1 - \frac{C}{n^a}$ for some C and any $a \geq 1$)

index query. Overall, we show that our implementations demonstrate good scalability over different numbers of threads and over graphs of different sizes.

In summary, the contributions of our work are as follows.

(1) We introduce the first theoretically efficient shared-memory parallel bi-core decomposition algorithm with nontrivial parallelism. We provide an accompanying parallel index construction and query algorithm, and prove that the problem of bi-core decomposition to be P-complete.
(2) We introduce practical optimizations and provide fast implementations of our parallel algorithms, that outperform the existing state-of-the-art algorithms. Our code is publicly available at https://github.com/clairebookworm/gbbs.
(3) We perform an extensive empirical evaluation on our algorithms.

2 RELATED WORK

K-core Decomposition. The bi-core decomposition problem is an extension of the well-studied k-core decomposition problem, which on general graphs, asks for the largest k for each vertex v such that there exists an induced subgraph containing v where all vertices have induced degree at least k. The first efficient sequential algorithm for k-core decomposition was given by Matula and Beck [25], and there has been numerous work on parallelizations in both the distributed memory and shared memory settings [10, 11, 20, 26, 41].

Other Dense Subgraph Decompositions. k-clique decomposition, k-truss, and (r, s)-nucleus decomposition are all extensions of the k-core decomposition that focus on higher order substructures in order to discover dense substructures in a graph. k-clique decomposition [37, 42] involves computing the k-clique core number of each vertex v, or the largest c such that there exists an induced subgraph containing v where all vertices are incident upon at least c induced k-cliques. k-truss is a classic extension [2, 8, 21, 34, 44, 51, 52] that asks for the largest k for each edge e such that there exists an induced subgraph containing e where all edges are contained within at least k triangles. Notably, the k-core and k-truss decompositions are part of the MIT GraphChallenge [18], demonstrating their practical importance and popularity. The (r, s)-nucleus decomposition [33, 35] further generalizes the k-clique and k-truss decompositions, by asking for the largest k for each r-clique such that there exists an induced subgraph containing the r-clique in which all r-cliques are contained within at least k induced s-cliques. Notably, k-clique decomposition is $(1, k)$-nucleus decomposition, and k-truss is $(2, 3)$-nucleus decomposition.

Generalization of Decomposition Algorithms to Bipartite Graphs. Another direction of current work focuses on generalizing these unipartite decomposition algorithms to bipartite graphs by focusing on other higher order structures available in bipartite graphs. Zou [53] and Sarıyüce and Pinar [32] defined k-tip and k-wing decomposition on bipartite graphs. k-tip decomposition asks for the largest k for each vertex v such that there exists an induced subgraph in which every vertex is incident to at least k induced $(2, 2)$-bicliques. Similarly, k-wing decomposition asks for the largest k for each edge e such that there exists an induced subgraph in which every edge is incident to at least k induced $(2, 2)$-bicliques. Multiple sequential [30, 32, 45–47, 53] and parallel [23, 38] algorithms have been developed for k-tip and k-wing decomposition.

Ahmed *et al.* proposed the (α, β)-core decomposition problem, or the bi-core decomposition problem and gave the first sequential bi-core algorithm [1]. Ding *et al.* applied bi-core to recommender systems and provided a sequential bi-core algorithm based on the classic k-core peeling algorithm [14]. Liu *et al.* developed an efficient computation sharing sequential bi-core peeling algorithm and a memory-efficient indexing structure to store the bi-cores for efficient

Table 1. Graph Notation Summary

G	An undirected, simple, bipartite graph
U	One bipartition of the vertices in G
V	Another bipartition of the vertices in G
$\deg(x)$	Degree or induced degree of a vertex x, depending on context
$N(x)$	x's neighbors. In other words, the set of vertices that are adjacent to x
dmax_v	The maximum vertex degree in V
dmax_u	The maximum vertex degree in U
$\max_\alpha(\beta)$	The maximum α value such that (α, β)-core is nonempty
$\max_\beta(\alpha)$	The maximum β value such that (α, β)-core is nonempty
δ	The maximum δ value such that (δ, δ)-core is nonempty. In other words, it is the maximum k-core number of the graph G.

membership queries from vertices [24]. Wang *et al.* extended the problem to weighted bipartite graphs to find the bi-core component with the highest minimum edge weight containing a given query vertex [48].

3 PRELIMINARIES

In this section, we provide the definitions and notations that we use throughout this paper.

Graph Definitions.

We take every graph to be simple, undirected, and bipartite. A *bipartite graph* is a graph G consisting of two mutually exclusive sets of vertices U and V, such that every edge connects a vertex in U with a vertex in V. In other words, every edge is of the form (u, v) where $u \in U$ and $v \in V$. Let $\deg(u)$ denote the degree of a vertex u.

DEFINITION 1. *A bi-core, or an (α, β)-core, is the maximal induced subgraph $G' = (U', V')$ of G such that for every $u \in U'$, the induced degree* $\deg(u) \geq \alpha$*, and for every $v \in V'$, the induced degree* $\deg(v) \geq \beta$*.*

Note that in this definition, $\deg(u)$ denotes the induced degree of vertex u in the induced subgraph G'. For the remainder of this paper, we take $\deg(u)$ to be the vertex u's induced degree in G' instead of in G unless specified.

See Table 1 for a table of graph notations we use.

Note the two following facts:

(1) if $u \in (\alpha_1, \beta_1)$-core, then $u \in (\alpha_2, \beta_2)$-core if $\alpha_2 \leq \alpha_1$ and $\beta_2 \leq \beta_1$.
(2) Every nonempty (α, β)-core must have $\alpha \leq \delta$ and/or $\beta \leq \delta$.

We give a quick proof of the second fact here. Assume for the sake of contradiction there exists a nonempty (α, β)-core with $\alpha > \delta$ and $\beta > \delta$. Then, we know that (α, β)-core $\subseteq (\delta + 1, \delta + 1)$-core. Thus, the $(\delta + 1, \delta + 1)$-core is nonempty and δ is not the max unipartite k-core number of the graph, which is a contradiction. Thus, any nonempty (α, β)-core must have $\alpha \leq \delta$ and/or $\beta \leq \delta$.

Problem Statement. We formally define the bi-core decomposition problem as follows.

DEFINITION 2. *Given a graph G and values α, β, return all vertices in the (α, β)-core of graph G [24].*

Similar to Liu *et al.*'s algorithm [24], our algorithm for this problem involves a process consisting of three stages: bi-core peeling, index building, and bi-core querying. In particular, in the bi-core peeling stage, we compute the $\alpha_{\max}, \beta_{\max}$ values for each vertex as defined below. Given a bipartite graph $G = (V, U)$, for every vertex $u \in U$, we define $\beta_{\max \alpha}(u)$ for a fixed α to be the maximum β value such that $u \in (\alpha, \beta)$-core [24]. Similarly, for $v \in V$, we let $\alpha_{\max \beta}(v)$ denote, for a fixed β, the maximum α value such that $v \in (\alpha, \beta)$-core [24]. Thus, in the bi-core peeling stage, we compute the values $\beta_{\max \alpha}(u)$ for every α and for every $u \in U$, and we compute the values $\alpha_{\max \beta}(v)$ for every β and for every $v \in V$.

Note that with these values, one can determine for every vertex $u \in U$ whether or not it is in (α, β)-core for any α, β values. If $\beta_{\max\ \alpha}(u) \geq \beta$, then $u \in (\alpha, \beta)$-core. Similarly, if $\alpha_{\max\ \beta}(v) \geq \alpha$, then $v \in (\alpha, \beta)$-core.

Then, the index construction stage uses $\alpha_{\max}, \beta_{\max}$ values to construct an index structure that allows queries to be processed in the final query stage.

Model of Computation. We use the shared-memory model of parallel computation, and in particular, we use the classic work-span model for our analysis, which allows us to derive theoretical bounds on the algorithm's running time on P processors. The work of an algorithm is the total number of operations executed, and the span is the length of the longest dependency path [9]. *Brent's Theorem* [6] states that given an algorithm's work T_1 and span T_∞, the algorithm's running time on P processors T_P can be bounded by

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

We assume arbitrary forking for simplicity. In other words, forking n processes has a span of $O(1)$. With the provided model, we show that our algorithm is *work-efficient*, meaning that it has the same work complexity as the best sequential algorithm.

Parallel Primitives. We now define the parallel primitives that we use throughout our algorithms.

ATOMIC-COMPARE-AND-SWAP(p, a, b) takes as input a pointer p and two values a, b. It atomically reads p; if its value equals a it then updates the value to b. If the update is performed successfully, the function returns true, else it returns false.

PREFIX-SUM(A) takes as input a sequence A and returns a sequence B of the same length such that $B[i] = A[0] \oplus A[1] \oplus A[2] \cdots A[i-2] \oplus A[i-1]$. Here $\oplus$ is a binary associative operator with an identity value denoted by ε. We assume for the rest of the paper that the operator is the addition operator. PREFIX-SUM has $O(n)$ work and $O(\log(n))$ span where n is the length of the sequence [9].

SUFFIX-MIN(A) is a special case of prefix sum using min as the operator and it is performed on the reverse of A. Specifically, it returns sequence B such that $B[i] = \min(A[i+1], A[i+2], \cdots, A[n])$ where n is the index of A's last element.

REDUCE-MIN(A) takes as input a sequence of length n. It returns the minimum element in the sequence. REDUCE-MIN has work $O(n)$ and span $O(\log(n))$ [9].

FILTER(A, COND) takes as input a sequence and a condition for filtering. It retains all items for which the condition is true and then outputs these elements in a sequence, maintaining the original ordering. The function returns a sequence of filtered elements in $O(n)$ work and $O(\log(n))$ span [9].

HISTOGRAM(A) takes as input a sequence of indices. It applies a parallel semisort to the indices, which it then uses to create a histogram of the frequencies of each index. It takes $O(n)$ expected work and $O(\log(n))$ span *w.h.p.* [19]. We use the parallel semisort by Gu *et al.* [19], and apply FILTER and PREFIX-SUM to obtain the occurrence count for each index.

RADIX-SORT(A) takes as input a sequence of elements with an natural ordering defined. It sorts them in parallel with work $O(n)$ and span $O(\log(n))$ where n is the length of the array [5, 28, 39, 50].

4 SEQUENTIAL BI-CORE DECOMPOSITION

In this section, we present the sequential bi-core peeling algorithm introduced by Liu *et al.* [24] to provide the context for our parallelization in Section 5. We reiterate that the entirety of this section is not our work.

4.1 Sequential Peeling

First, we note that the problem of computing the $\alpha_{\max\ \beta}(v)$ values for all $v \in V$ and all $1 \leq \beta \leq \mathrm{dmax}_v$ is symmetric to the problem of finding the $\beta_{\max\ \alpha}(u)$ values for all possible u and α. As such, we focus our discussion on the problem of finding the $\alpha_{\max\ \beta}(v)$ values. Note that $\alpha_{\max\ \beta}(v) = \alpha$ if $v \in (\alpha, \beta)$-core but $v \notin (\alpha + 1, \beta)$-core. Thus, a peeling-based algorithm is often used to solve this problem [1, 14]. In a baseline peeling-based algorithm [24], we apply a subroutine PEEL-FIX-β for every β' between 1 and dmax_v. PEEL-FIX-β takes as input a fixed β' value, and increases the α value of the (α, β')-core from 1 to $\max_\alpha(\beta')$. For each (α, β') pair, it iteratively deletes vertices no longer within the current (α, β')-core. In other words, for each α from 1 to $\max_\alpha(\beta')$, the algorithm iteratively peels vertices not in each successive core. When deleting a vertex v to discover the $(\alpha + 1, \beta')$-core, we update $\alpha_{\max\ \beta'} \leftarrow \alpha$, because it is the highest α value for which $v \in (\alpha, \beta')$-core. The subroutine PEEL-FIX-α is symmetric.

4.2 Computation Sharing

Liu *et al.* observed that it is unnecessary to repeat the entirety of the peeling process for each possible β' value, because we rediscover some of the same information in the symmetric subroutine. Instead, it is sufficient to perform the peeling process for $1 \leq \beta' \leq \delta$. Essentially, when a vertex u is deleted while discovering the $(\alpha + 1, \beta')$-core, we know that $u \in (\alpha, \beta')$-core. Thus, we know that the $\beta_{\max \alpha}(u)$ value is at least β'. More precisely, because (i, β')-core $\supseteq (\alpha, \beta)$-core for $i < \alpha$ and thus $u \in (i, \beta')$-core, we can also update $\beta_{\max i}(u)$ to at least β', for all $i < \alpha$. Provided that the peeling process is performed for all $1 \leq \beta' \leq \delta$, Liu *et al.* showed that all $\beta_{\max \alpha}(u)$ entries with $\alpha > \delta$ will be updated to their correct values [24].

We give a brief explanation as follows, although further details can be found in their paper. Given $u \in (\alpha, \beta_{\max\ \alpha}(u))$-core and $\alpha > \delta$, we know $\beta_{\max\ \alpha}(u) \leq \delta$. Therefore, we must have peeled off $(\alpha, \beta_{\max\ \alpha})$-core in the peeling process and would have recorded that correct β value for the entry.

The pseudocode for Liu *et al.*'s computation sharing algorithm is in Algorithm 1. We discuss this pseudocode in more detail. On Lines 5–6 of Algorithm 1, we loop over all $1 \leq \beta' \leq \delta$ and run PEEL-FIX-β on each β' (note that Lines 3–4 are symmetric for PEEL-FIX-α). Each iteration of PEEL-FIX-β iteratively removes vertices from U with degree $\leq \alpha$ for α from 1 to $\max_\alpha(\beta')$ for the given β'. In more detail, on Line 18, DEL-UPDATE iteratively deletes all vertices v with induced degree $\deg(v) < \beta'$, because these vertices are not in any (α, β')-core for the given β'. Then, until the graph is empty, we execute Lines 20–26. On Line 20, we find the set of vertices $u \in U$ with the current minimum degree and store them in delU. We let α denote the current minimum degree. At this point, all remaining vertices are in (α, β')-core. We now continue with the peeling process, and iteratively delete all vertices in U with induced degree $\leq \alpha$, which we maintain in the set delU. Lines 21–23 update the $\beta_{\max i}(u)$ values for all $u \in$ delU and $1 \leq i \leq \alpha$, because as discussed earlier, these $\beta_{\max i}(u)$ are at least β'. On Line 24, the DEL-UPDATE subroutine removes the vertices in delU, which affects the degrees of their neighbors in V. If the degrees of these neighbors $v \in V$ fall below β', this means that each of these vertices v is not in the $(\alpha + 1, \beta')$-core. Thus, to continue searching for the next core, we peel these vertices as well and record them

Algorithm 1 Sequential Baseline 1 [24]

```
1: procedure SEQ-BI-CORE(G)
2:     $\delta \leftarrow$ PAR-K-CORE($G$)
3:     for $\alpha' = 1$ to $\delta$ do
4:         PEEL-FIX-$\alpha$($G$, $\alpha'$)
5:     for $\beta' = 1$ to $\delta$ do
6:         PEEL-FIX-$\beta$($G$, $\beta'$)
7: procedure DEL-UPDATE($G$, delX, $k$)
8:     delY $\leftarrow \emptyset$
9:     for all $x$ in delX do
10:        for all $y$ in $N(x)$ do
11:            $\deg(y) \leftarrow \deg(y) - 1$
12:            if $\deg(y) < k$ then
13:                add $y$ to delY
14:                remove $y$ from $G$            ▷ Or mark $y$ as removed in an array
15:        remove $x$ from $G$                    ▷ Or mark $x$ as removed in an array
16:    return delY
17: procedure PEEL-FIX-$\beta$($G$, $\beta'$)
18:    DEL-UPDATE($G$, $\{v \in V \mid \deg(v) < \beta'\}$, 1)    ▷ Iteratively delete vertices in $V$ with indeced degree less than $\beta'$
19:    while $U \neq \emptyset$ do
20:        delU, $\alpha \leftarrow$ FIND-MIN($G$)    ▷ Find min induced degree in $U$, store that to $\alpha$, store the set of all $u \in U$ with $\deg(u) \le \alpha$ to delU
21:        for all $u$ in delU do
22:            for $i = 1$ to $\alpha$ do
23:                $\beta_{max\ i}(u) \leftarrow \max(\beta_{max\ i}(u), \beta')$
24:        delV $\leftarrow$DEL-UPDATE($G$, delU, $\beta$)    ▷ Peel $U$ up to $\alpha$
25:        for all $v$ in delV do
26:            $\alpha_{max\ \beta'}(u) \leftarrow \alpha$    ▷ Update $\alpha_{max\ \beta'}$
27: procedure PEEL-FIX-$\alpha$(G,$\alpha'$)
28:    symmetric to PEEL-FIX-$\beta$
```

in delV. We repeat this peeling process until all vertices in V remaining have induced degree at least β'. We update the $\alpha_{\max\ \beta'}(u)$ values of delV on Line 26.

Note that when we remove a vertex from G, as on Lines 14–15, we do not have to remove the vertex physically. We can maintain an array that tracks whether each vertex is removed and simply mark the vertex as removed in the array. When traversing the graph, we can skip vertices marked as removed in the array.

The PEEL-FIX-α subroutine is symmetric to PEEL-FIX-β, swapping β' with α', α with β, V with U, and v with u.

Analysis. We first discuss the time complexity of PEEL-FIX-β. Note that the combined time of all of the DEL-UPDATE subroutines is bounded by $O(m)$. This is because these subroutines are called on each vertex at most once, where the vertex is additionally symbolically deleted from the graph. For each vertex x we delete, we traverse its neighbors to update their degrees. Thus, in total, it incurs $O(m)$ time. Lines 25–26 are bounded by $O(n)$ since there can be at most n updates to $\alpha_{\max \beta'}(v)$ over the entire PEEL-FIX-β routine. This is because this update is performed at most once for each vertex. Lines 21–23 are also bounded over all invocations by $O(m)$, because the maximal α value is $O(\deg(u))$, thus giving a total of $O(\sum_{u \in U} \deg(u)) = O(m)$ time. FIND-MIN can be achieved in $O(\text{dmax}_u) = O(n)$ with a linear search. Thus PEEL-FIX-β runs in $O(m)$ time [24] and PEEL-FIX-α is symmetric.

For the overall peeling procedure SEQ-BI-CORE, we make $O(\delta)$ calls to subroutines PEEL-FIX-β and PEEL-FIX-α. Here δ is the degeneracy of the graph, or the maximal value such that (δ, δ)-core is nonempty.

Since δ is bounded by $O(\sqrt{m})$ [24], Algorithm 1 runs in $O(\delta m)$, or $O(m^{\frac{3}{2}})$.

4.3 Memory-Efficient Bi-core Index

Liu *et al.* also introduced a memory-efficient bi-core indexing structure to allow for efficient queries of (α, β)-cores. Specifically, their data structure returns the set of all vertices in an (α, β)-core for a given (α, β) in time linear to the size of the core. The indexing structure consists of two data structures, $\mathbb{I}^U$ and $\mathbb{I}^V$, where $\mathbb{I}^U$ stores the vertices in U partition, while $\mathbb{I}^V$ stores vertices in V partition. Because these are symmetric, we discuss only $\mathbb{I}^U$ here.

Let $\mathbb{I}^U_{\alpha,\beta}$ be the set of vertices $u \in U$ such that $\beta_{\max\alpha}(u) = \beta$. In other words, $\mathbb{I}^U_{\alpha,\beta}$ is the set of all u that are in the (α, β)-core but not in the $(\alpha, \beta+1)$-core. Thus, by definition, $\mathbb{I}^U_{\alpha,\beta_1} \cap \mathbb{I}^U_{\alpha,\beta_2} = \emptyset$ for all $\beta_1 \neq \beta_2$. Further, note that if $u \in \mathbb{I}^U_{\alpha,\beta}$, then u is in the (α, β')-core for all $\beta' \leq \beta$. Therefore, the (α, β)-core is given by $\bigcup_{i=\beta}^{\max_\beta(\alpha)} \mathbb{I}^U_{\alpha,i}$. $\mathbb{I}^U$ is a lookup table containing all sets $\mathbb{I}^U_{\alpha,\beta}$ that is nonempty, allowing us to access a specific set in $O(1)$ time. Therefore, the operation of taking the union across all sets $\mathbb{I}^U_{\alpha,i}$ for $\beta \leq i \leq \max_\beta(\alpha)$ is in time linear to the number of vertices in these sets. Thus, we can find all vertices in (α, β)-core with time complexity linear to the number of vertices in the core.

Liu *et al.* implemented $\mathbb{I}^U$ as a jagged 2D pointer array with indices corresponding to α, β values. Each element of the array with index α, β points to the set $\mathbb{I}^U_{\alpha,\beta}$

Analysis. Liu *et al.* showed that the indexing structure takes $O(m)$ space, which we discuss here. First, we prove that the size of the 2D dynamic array is proportional to $O(m)$.

The size of the array is $\sum_{\alpha=1}^{\text{dmax}_u} \max_\beta(\alpha)$, by construction. To bound this value, we can consider the process of constructing the bipartite graph by adding vertices in the U partition while the V partition is already in place. When we add vertex u_i, it only affects cores with $\alpha \leq u_i$. Since the addition of each vertex can at most increase $\max_\beta(\alpha)$ by 1, the addition of vertex u_i increases $\sum_{\alpha=1}^{\text{dmax}_u} \max_\beta(\alpha)$ by at most $O(\deg(u_i))$. Therefore, after the construction process finishes, the space of the array is $O(\sum_{u\in U} \deg(u)) = O(m)$.

Now, we show that the space occupied by all of the vertex sets in $\mathbb{I}^U$ is bounded by $O(m)$.

For each $u \in U$, u exists in $\mathbb{I}^U_\alpha$ exactly once for each $\alpha \leq \deg(u)$. Thus each $u \in U$ exists in $\mathbb{I}^U$ exactly $\deg(u)$ times. Therefore, the total number of vertices in $\mathbb{I}^U$ is $O(\sum_{u\in U} \deg(u)) = O(m)$.

Thus, in total, since this argument is symmetric for $\mathbb{I}^V$, Liu *et al.*'s indexing data structure takes $O(m)$ total space.

5 PARALLEL BI-CORE DECOMPOSITION

The sequential nature of Liu *et al*'s [24] bi-core decomposition (Algorithm 1) limits its practical applicability to large graphs. While Liu *et al.* [24] provides a parallel version of their algorithm, their parallel algorithm only parallelizes between rounds of peeling (subroutines PEEL-FIX-α and PEEL-FIX-β), and does not parallelize the peeling process itself. As a result, it has a high span of $O(m)$. We present in this section a parallel bi-core decomposition algorithm using the same computation-sharing technique developed by Liu *et al* [24]. We prove that our algorithm is work-efficient and has span $O(\rho \log(n))$ *w.h.p.*, where ρ is the peeling complexity, which we define as follows.

DEFINITION 3. *The bi-core peeling complexity ρ is the maximum number of rounds needed to empty the graph by any call of* PAR-PEEL-FIX-α *or* PAR-PEEL-FIX-β*, where in each round of peeling, the set of vertices with the minimum induced degree is removed from the graph*

It is worth remarking that the bi-core peeling complexity ρ cannot be bounded by $O(\rho_k)$ where ρ_k is the k-core peeling complexity introduced by Dhulipala *et al.* [12].

Our algorithm is based on a peeling paradigm. For PAR-PEEL-FIX-β given a fixed β', in each round, we remove all vertices u with the lowest induced degree concurrently. In other words, we peel all vertices u with $\deg(u) \leq \alpha$ for the current α. In order to efficiently peel these vertices concurrently, we must update the degrees of the neighbors of the peeled vertices in parallel, using an approach that we describe in Section 5.2. Additionally, to reduce the span of the

Algorithm 2 Parallel Exponential Search

```
1: procedure HAS-MIN-DEG(buckets)    ▷ check if the given slice of array of buckets contain a nonempty bucket
2:     hasMinDeg ← false    ▷ hasMinDeg records whether the interval contain the next nonempty bucket
3:     parfor $i = 0$ to |buckets| do
4:         if buckets[$i$] exists then
5:             ATOMIC-COMPARE-AND-SWAP(hasMinDeg, false, true)
6:     return hasMinDeg
7: procedure NEXT-BUCKET(buckets, $k$)    ▷ find and return the next nonempty bucket and its corresponding degree
8:     $i \leftarrow 1$    ▷ $i$ is doubled in each iteration of the while loop. In each iteration, we search interval $(\frac{i}{2}, i]$ for the next nonempty bucket
9:     while HAS-MIN-DEG(buckets[$k + \frac{i}{2} + 1$ to $k + i$])= false do
10:        $i \leftarrow 2i$
11:    minDeg ← REDUCE-MIN(buckets[$k + \frac{i}{2} + 1$ to $k + i$])
12:    return buckets[minDeg], minDeg    ▷ Return next min deg
```

sequential search used in Algorithm 1, we use a parallel exponential search technique to find the next set of vertices with minimum degree; we discuss this in Section 5.1.

5.1 Parallel Bucketing and Exponential Search

To achieve polylogarithmic span while maintaining work-efficiency, we use an efficient parallel bucketing structure to store the subset of vertices to be peeled in each round. Dhulipala *et al.* introduced this parallel bucketing structure and applied it to their parallel k-core decomposition algorithm [11]. The data structure consists of an array of buckets, where the indices represent vertex degrees. Each bucket stores all vertices with current degree corresponding to its index. It supports two types of operations. UPDATE-VERTICES allows batch update of vertices' degrees. It moves those vertices to new buckets corresponding to their new degrees in parallel [11]. On the other hand, NEXT-BUCKET, given a degree value k, searches for the next subset of vertices with lowest induced degree $\geq k$. We introduce a parallel exponential search to complete this in logarithmic span. We provide its pseudocode in Algorithm 2.

NEXT-BUCKET finds and returns the next nonempty bucket with degree $\geq k$. First, we initialize i to 1. Then, in each iteration of the while loop on Lines 9–10 of Algorithm 2, we determine if the interval $(k + \frac{i}{2}, k + i]$ contains the next nonempty bucket. Then, we double i and repeat until the next nonempty bucket is found. For instance, we start the search from the interval $(k, k + 1]$. The HAS-MIN-DEG subroutine called on Line 9 checks in parallel whether the next minimum degree vertex exist in the given interval. If it does not exist in this interval, we proceed to interval $(k + 1, k + 2]$, and then to $(k + 2, k + 4]$, $(k + 2^i, k + 2^{i+1})$ for each i until a nonempty bucket is found.

On Line 11, we perform a parallel REDUCE-MIN on the sequence of the indices of nonempty buckets to obtain the next minimum degree with nonempty bucket. On Line 12, we return the next nonempty bucket.

Analysis.

First, let n be the number of vertices stored in the bucketing structure.

Given p calls to UPDATE-VERTICES updating a total of q vertices (a repeating vertex is counted repeatedly), UPDATE-VERTICES achieves an overall work complexity of $O(q)$ and span complexity of $O(p \log(n))$ [11].

We perform p calls to NEXT-BUCKET with the condition that the degree value k passed in at the i^{th} call equals the return value of the $(i - 1)^{\text{th}}$ call. In other words, we find the next nonempty bucket starting from the previous nonempty bucket. Given the condition, over all p calls to subroutine NEXT-BUCKET, we now show that it achieves an overall work complexity of $O(n)$ and span complexity of $O(p \log(n))$.

Assume NEXT-BUCKET is called with current degree value k and that the next minimum degree is $k + h$. Then, notice that NEXT-BUCKET searches at most $2h$ elements before terminating and returning $k + h$ as the next minimum degree. If it

Algorithm 3 Parallel Bi-core Decomposition

```
1: procedure PAR-BI-CORE($G$)
2:     parfor $\alpha' = 1$ to $\delta$ do
3:         PAR-PEEL-FIX-$\alpha$($G$, $\alpha'$)
4:     parfor $\beta' = 1$ to $\delta$ do
5:         PAR-PEEL-FIX-$\beta$($G$, $\beta'$)
6: procedure PAR-DEL-UPDATE($G$, $X_{\text{del}}$)
7:     $Y_{\text{update}} \leftarrow \emptyset$
8:     degs $\leftarrow$ array storing the corresponding degree of each vertex $x \in X_{\text{del}}$
9:     offsets $\leftarrow$ PREFIX-SUM(degs)
10:    parfor all $i, x$ in $X_{\text{del}}$ do
11:        remove $x$ from $G$
12:        parfor all $y$ in $N(x)$ do
13:            offset $\leftarrow$ offsets[$i$]+$j$    ▷ Obtain the location in $Y_{\text{update}}$ to store $y$ at. $j$ is the index of $y$ in $N(x)$
14:            $Y_{\text{update}}$[offset] $\leftarrow y$    ▷ Record $y$ for degree update
15:    $Y_{\text{hist}} \leftarrow$ HISTOGRAM($Y_{\text{update}}$)    ▷ Count occurrences of vertices
16:    parfor all $y$, count in $Y_{\text{hist}}$ do
17:        deg($y$) $\leftarrow$ deg($y$) $-$ count
18:    return $Y_{\text{update}}$
19: procedure PAR-PEEL-FIX-$\beta$($G$, $\beta'$)
20:    PAR-DEL-UPDATE($G$, $\{v | \deg(v) < \beta'\}$)    ▷ Remove all vertices in $V$ with degree $< \beta'$
21:    Store vertices in $U$ into buckets    ▷ Construct bucketing structure from vertices in $U$ based on their degrees
22:    while buckets $\neq \emptyset$ do
23:        $U_{\text{del}}, \alpha \leftarrow$ NEXT-BUCKET(buckets, $\alpha$)    ▷ Extract the next set of vertices with minimum degree
24:        parfor all $u$ in $U_{\text{del}}$ do
25:            parfor $i = 1$ to $\alpha$ do
26:                $\beta_{\max i}(u) \leftarrow \max(\beta_{\max i}(u), \beta')$    ▷ Update $\beta_{\max i}(u)$
27:        $V_{\text{update}} \leftarrow$ PAR-DEL-UPDATE($G$, $U_{\text{del}}$)    ▷ Peel $U$ up to $\alpha$
28:        $V_{\text{del}} \leftarrow$ FILTER($V_{\text{update}}$, $\deg(v) < \beta'$)
29:        parfor all $v$ in $V_{\text{del}}$ do
30:            $\alpha_{\max \beta'}(v) \leftarrow \max(\alpha_{\max \beta'}(v), \alpha)$    ▷ Update $\alpha_{\max \beta'}(v)$
31:        $U_{\text{update}} \leftarrow$ PAR-DEL-UPDATE($G$, $V_{\text{del}}$)    ▷ Remove peeled $v$
32:        buckets.UPDATE-VERTICES($U_{\text{update}}$)    ▷ Update vertices with changed degrees in the bucketing structure
33: procedure PAR-PEEL-FIX-$\alpha$(G, $\alpha'$)
34:    symmetric to PAR-PEEL-FIX-$\beta$
```

searches only h elements ahead, the algorithm has an overall work upperbounded by $O(\text{dmax}_v) = O(n)$. Since it searches only $2p$ ahead, its work is bounded by $O(2n) = O(n)$ as well.

Next, we show that the subroutine NEXT-BUCKET has a span of $O(p\log(n))$. Assume, as previously, that the current degree is k and the next degree is $k + h$. Note that NEXT-BUCKET takes at most $\log(h)$ iterations of its while loop on Line 9 of Algorithm 2 to find the next minimum degree $k + h$. To loosen the bound, $\log(h) = O(\log(n))$. Therefore, the overall span of p calls to NEXT-BUCKET is $O(p\log(n))$. An assumption we make in this derivation is that subroutine HAS-MIN-DEG as used on Line 9 has span $O(1)$. This is true because at most one ATOMIC-COMPARE-AND-SWAP operation can be successfully executed for a given interval. Additionally, note that REDUCE-MIN on Line 11 of Algorithm 2 has span $O(\log(n))$; it is executed only once for each call of NEXT-BUCKET, thus totaling a span of $O(p\log(n))$ as well.

5.2 Parallel Bi-core Decomposition

The parallel bi-core decomposition algorithm shares a similar structure to Algorithm 1. Instead of removing vertices with minimum induced degree sequentially, we delete all vertices with the same minimum induced degree in parallel. Further, to achieve optimal span, we replace the linear search used in Algorithm 1 to find the next lowest degree vertex with Algorithm 2 to find the next bucket of vertices with lowest induced degree.

The pseudocode is given in Algorithm 3 and we now discuss it in more detail.

First we discuss the subroutine PAR-DEL-UPDATE. Subroutine PAR-DEL-UPDATE takes as input a generic subset of vertices X_{del} and then peels all these vertices in parallel. On Lines 7–14, we construct an array Y_{update} that stores all neighbors y of X_{del}. Note that if y is incident to multiple vertices in X_{del}, it appears the same number of times in Y_{update}. This array is constructed in parallel by first building the array offsets as the PREFIX-SUM of degs on Line 9. offsets$[i]$ records the total number of neighbors of vertices $[1, 2, \cdots, i-1]$ in X_{del}. Therefore, offsets$[i] + j$ gives the index location to store the j^{th} neighbor of the i^{th} vertex in X_{del}, which is used on Lines 13-14 to store neighbor y into its place in Y_{update}. On Line 15, HISTOGRAM returns a sequence of pairs (y, count). For every y, count is the number of its occurrences in Y_{update}. On Lines 12–13, we iterate through each y and decrease its degree by its corresponding count. Note that on Line 11, it is unnecessary to remove x from G. We can simply maintain an array that tracks whether each vertex is removed and ignore vertices marked as removed in our traversals.

Since many threads may be updating the degree of the same vertex, our parallel aggregation approach is necessary to maintain low theoretical span.

Now, we discuss the main algorithm. On Line 20 in PAR-PEEL-FIX-β, we peel off all $v \in V$ with degree less than β' using the subroutine PAR-DEL-UPDATE. On Line 21, we construct a bucket representation of U, buckets, as discussed by Dhulipala *et al.* [11] and as described in Section 5.1. We call NEXT-BUCKET on buckets on Line 23 to obtain the next nonempty bucket, which we store into U_{del}. We also update the α value appropriately; importantly, U_{del} records all u with induced degree $\deg(u) \leq \beta$. Note that for all $u \in U_{\text{del}}$, $u \in (\alpha', \beta)$-core but $u \notin (\alpha', \beta+1)$-core. On Lines 24–26, we in parallel update the $\beta_{\max \alpha'}$ values. Note that Line 26 does not incur race conditions if we keep a copy of $\beta_{\max \alpha}$ and similarly $\alpha_{\max \beta}$ for each thread. On Line 27, we peel off all vertices in the current bucket, U_{del}. Lines 29–30 updates $\alpha_{\max \beta'}$ values in the same way as in Algorithm 1. Then, Line 31 calls PAR-DEL-UPDATE to peel off all vertices stored in V_{del}. Finally, on Line 32, we update the degrees of vertices in U_{update}, which consist of all vertices $u \in U$ whose degree is affected by peeling off V_{del}; UPDATE-VERTEX moves $y \in U_{\text{update}}$ to new buckets corresponding to their new degrees. Note that a minor detail we exclude here is that some vertices in U_{update} could take on a degree $< \alpha$. We simply set their degree to α, so they are peeled together in the next round of peeling.

PAR-PEEL-FIX-α is symmetric to PAR-PEEL-FIX-β, with all u, v and α, β flipped.

Analysis. PAR-PEEL-FIX-β has work complexity $O(m)$. First note that NEXT-BUCKET, and UPDATE-VERTICES [11] all have overall work across all iterations of the while loop bounded by $O(m)$ as discussed in Section 5.1.

Over all invocations of subroutine PAR-DEL-UPDATE, each vertex is peeled exactly once. Since we traverse the neighbor of each vertex in PAR-DEL-UPDATE once, the total work given by PAR-DEL-UPDATE in one call of PAR-PEEL-FIX-β is $O(\sum_{x \in V \text{ or } U} \deg(x)) = O(m)$.

The work of updating the $\beta_{\max \alpha}$ and $\alpha_{\max \beta}$ values in Algorithm 3 is the same as the sequential updates performed in Algorithm 1, totaling $O(m)$. FILTER, over all invocations, also total $O(m)$ work. Therefore, PAR-PEEL-FIX-β has work complexity $O(m)$. Thus, procedure PAR-BI-CORE has overall work complexity $O(\delta m)$ or more loosely $O(m^{\frac{3}{2}})$.

Now we analyze the span complexity. First note that PAR-DEL-UPDATE has span $O(\log(n))$ *w.h.p.*; this is because PREFIX-SUM and HISTOGRAM both have span upperbounded by $O(\log(n))$ *w.h.p.*. Each iteration of the while loop on Line 22 has span $O(\log(n))$ *w.h.p.* because FILTER, PAR-DEL-UPDATE, UPDATE-VERTICES [11], and NEXT-BUCKET all have span bounded by $O(\log(n))$ *w.h.p.*. The rounds of iterations of the while loop is bounded by $O(\rho)$. The span is therefore $O(\rho \log(n))$ *w.h.p.*. Overall, PAR-BI-CORE has span $O(\rho \log(n))$ *w.h.p.*.

5.3 P-completeness

The span of our algorithm is not polylogarithmic. However, this is to be expected as the problem of bi-core decomposition is P-complete, which we prove here. We prove the P-completeness of the decision version of the bi-core decomposition problem. The decision problem is: given values α, β and a simple bipartite graph, decide if (α, β)-core is nonempty in the graph. This is a generalization of the k-core decision problem on a bipartite graph: given value k, decide if (k, k)-core exists. We show the P-completeness of bi-core decision problem using a reduction from the k-core decision problem.

THEOREM 1. *The (α, β)-core decomposition problem is P-complete if and only if $\alpha \geq 3$ or $\beta \geq 3$. Otherwise, if $\alpha \leq 2$ and $\beta \leq 2$, it is in NC.*

When $\alpha \leq 2$ and $\beta \leq 2$. For $\alpha = 2$ or $\beta = 2$, the $(2, 2)$-core decomposition problem is equivalent to the k-core decomposition problem on the bipartite graph with $k = 2$, which has an NC solution [3].

If $\alpha = 1$, then the $(1, \beta)$-core decomposition problem is equivalent to finding all vertices $x \in V$ such that $\deg(x) \geq \beta$ as well as its neighbors in partition U. Similarly, we can solve $(\alpha, 1)$-core decomposition for some arbitrary α with $O(1)$ span.

When $\alpha \geq 3$ and $\beta \geq 3$. We perform the reduction from the k-core decision problem in a general graph G by constructing G' such that G' is bipartite and the k-core decision problem on G is equivalent to the (k, k)-core decision problem on G'.

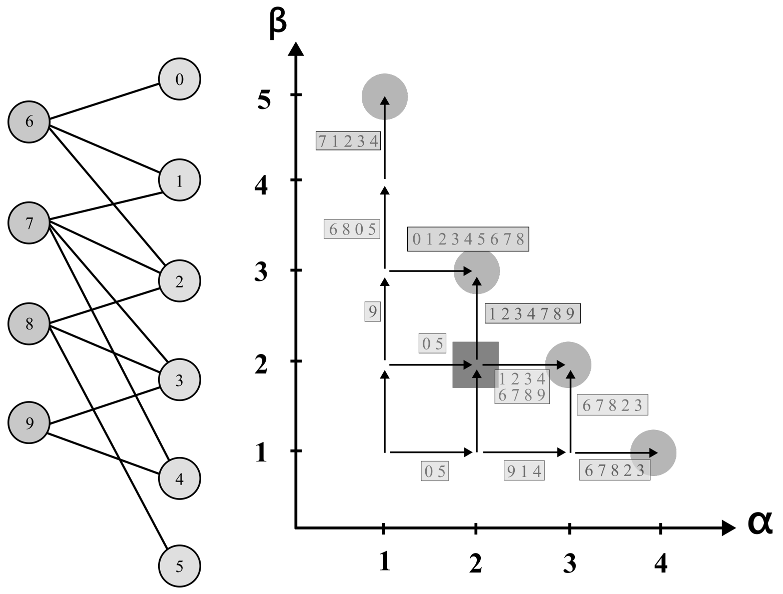
Let G' consist of two partitions U, V where each partition is a copy of all vertices of G. In other words, a vertex $x \in G$ is copied to x_u and x_v in G'. Now, we connect an edge (x_u, y_v) in G' if (x, y) is an edge in G.

Now, we show that for any value k, k-core is nonempty in G if and only if (k, k)-core is nonempty in G'. If the k-core of G is nonempty and comprises a vertex subset W, then for $w \in W$, $\deg(w) \geq k$, or there exists $\geq k$ edges of the form (w, p), where $p \in W$. Now consider $W' = W_V \bigcup W_U$ in G. Given that W_U, W_V are copies of W, and each $w \in W$ has $\geq k$ edges of the form (w, p), we know each $w_U \in W_U$ is incident to $\geq k$ edges of the form (w_U, p_V). Similarly for each $w_V \in W_V$. Therefore, W' forms a (k, k)-core on the bipartite graph and so the (k, k)-core is nonempty in G'.

Reversely, if the (k, k)-core in G' is nonempty, we show that the k-core in G is nonempty. Due to the symmetry of U, V partitions, if $w_U \in (k, k)$-core, then $w_V \in (k, k)$-core. Therefore, if the (k, k)-core of G' is W', then $W' = W_U \bigcup W_V$ and W_U, W_V are mirror images of each other. Let W be the vertex subset in G that corresponds to W_U, W_V. We show that it is a k-core in G. For each vertex w_U incident to edges of the form (w_U, p_V) where $p_V \in W_V$, its corresponding vertex w in W is incident to the corresponding edges of the form (w, p) and $p \in W$ because $p_V \in W_V$. Since $\deg(w_U) \geq k$ for each $w_U \in W_U$, $\deg(w) \geq k$ for each $w \in W$. Therefore, W forms a k-core of graph G and so the k-core of graph G is nonempty.

Given the correspondence between the k-core decision problem on graph G with the (k, k)-core decision problem on graph G', we have obtained an NC reduction from the k-core problem to the bi-core problem since constructing G' takes work $O(m)$ and span $O(1)$. Therefore, given that k-core decomposition is P-complete for $k \geq 3$, we know that bi-core decomposition is P-complete for $\alpha \geq 3, \beta \geq 3$.

When one of $\alpha, \beta = 2$. For the case where $\alpha = 2$ and β is some arbitrary value ≥ 3 (the reverse of this is symmetric and thus not shown). We show that deciding whether $(2, \beta)$-core is nonempty has a reduction from the k-core equivalent with $k = \beta$. Consider an arbitrary simple graph and the k-core problem on this graph. We create a middle-vertex for each edge in the graph. Putting these middle-vertices into the U partition and the original vertices into the V partition, we can create in $O(1)$ span a bipartite graph where all vertices in U has a degree of 2. Now, deciding if $(2, \beta)$-core nonempty is equivalent to deciding if the k-core of the original graph is nonemtpy, where $k = \beta$, because each wedge in the bipartite



(a) Example Graph (b) Example Peeling Space

Fig. 1. On the left is an example bipartite graph, and on the right is an illustration of the peeling space of the graph. This is discussed in more detail in Section 5.4

graph corresponds to an edge in the original graph. Given this reduction, we know the problem of bi-core decomposition is P-complete if the core (α, β) is of the form $(2, \beta)$ with $\beta \geq 3$ or $(\alpha, 2)$ with $\alpha \geq 3$.

Thus, we have proven that the bi-core decomposition problem is in NC if and only if $\alpha = 1$ or $\beta = 1$ or $\alpha, \beta = 2$. Otherwise, it is P-complete, meaning that it is most likely impossible to obtain an algorithm with polylogarithmic span.

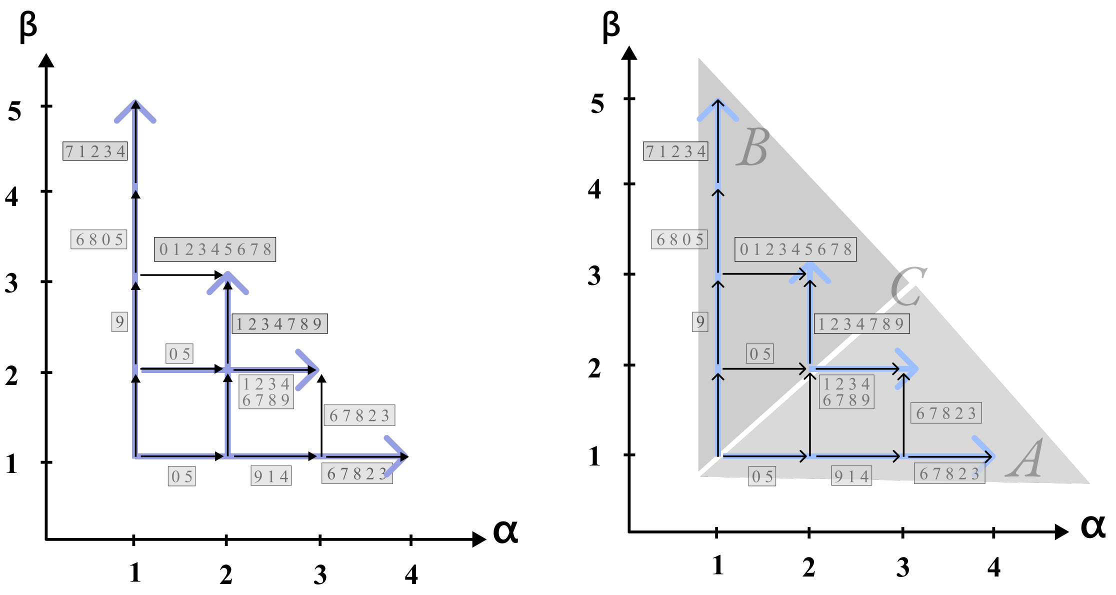
5.4 Peeling Space Pruning Optimization

In this section, we introduce a peeling space pruning optimization to our algorithm. This optimization is also applicable to the sequential bi-core decomposition algorithm in Algorithm 1. The baseline algorithm introduced by Liu *et al.* [24] performs a complete peeling from $\alpha = 1$ to $\alpha = \text{dmax}_u$ for each $1 \leq \beta' \leq \delta$. Then, it performs a complete peeling from $\beta = 1$ to $\beta = \text{dmax}_v$ for each $1 \leq \alpha' \leq \delta$. We observe that, in the process of peeling, all (α, β)-cores with $1 \leq \alpha \leq \delta$ and $1 \leq \beta \leq \delta$ are peeled twice, once when we perform peeling along increasing α values for different β' and another time when we perform peeling along increasing β values for different α'.

To avoid repetition, we can modify Algorithm 3 such that each PAR-PEEL-FIX-$\alpha(G, \alpha')$ starts peeling along β values from the (α', α')-core instead of from $(\alpha', 1)$-core. In other words, the algorithm starts iteratively increasing β value from α' to dmax_v and removing vertices no longer within the current (α', β)-core at the same time. Notably, we confine β to $\alpha' \leq \beta \leq \text{dmax}_v$ as opposed to $1 \leq \beta \leq \text{dmax}_v$ as used in Algorithms 1 and 3.

We illustrate this optimization with an example. Consider a graph and its peeling space visualization as shown in Figure 1.

Each integral intersection in the grid of Figure 1 represents an (α, β)-core. Edges represent a single-step peeling operation from (α, β)-core to $(\alpha, \beta+1)$-core (upward) or to $(\alpha+1, \beta)$-core (rightward). The numerals on an edge represents



(a) Algorithm 3's Peeling Path (b) Optimized Peeling Path

Fig. 2. This figure compares unoptimized vs optimized peeling paths. The left hand side shows the unoptimized peeling paths while the right hand side shows the optimized ones.

the indices of vertices that would be deleted by that specific peeling operation. The circled nodes represent (α, β)-cores that are empty, and the boxed node represents the (δ, δ)-core. Every core corresponding to a grid position that is not drawn is empty. The circled nodes form the boundary of the peeling space.

The peeling operations performed by Algorithm 3 can be visualized by the blue highlighted peeling paths in Figure 2 (a). For $\alpha' = 1$, we perform β-core peeling from $\beta = 1$ to $\beta = 5$. For $\alpha' = 2$, we again increase β from 1 to 3 while iteratively removing vertices not within the current bi-core. With the proposed optimization, for $\alpha' = 2$, we only perform peeling from $\beta = 2$ to $\beta = 5$, starting from the (α', α')-core, or the $(2, 2)$-core in this case. This is represented by the blue highlighted peeling paths in Figure 2 (b).

To show the correctness of the optimized algorithm, we divide the peeling space into 3 parts: part C with the diagonal (x, x)-cores, part B where all the (α, β)-cores satisfy $\beta > \alpha$ and part A where the (α, β)-cores satisfy $\alpha > \beta$. Note that part A of the peeling space corresponds visually to the part of peeling space below the diagonal (x, x)-cores; part B corresponds instead to the section above the diagonal (x, x)-cores. Thus, for the optimized algorithm, when peeling along increasing α values, it operates in part A of the peeling space; when peeling along increasing β values it operates in part B of the peeling space.

First, we note that the correct $\alpha_{\max\beta}(v)$ values are computed for all vertices v with $(\alpha_{\max\beta}(v), \beta)$-cores in part A or C of the peeling space. For a specific β value, if vertex $v \in (\alpha, \beta)$-core but $v \notin (\alpha + 1, \beta)$-core, with $\alpha \geq \beta$, then $\alpha_{\max\beta}(v)$ is recorded correctly to be α as we perform peeling along α values from $\alpha = \beta$ to its maximum value.

Then, we show that the optimized algorithm computes the correct $\alpha_{\max\beta}(v)$ values for all vertices v with $(\alpha_{\max\beta}(v), \beta)$-cores in part B of the peeling space. When peeling along increasing β values with $\alpha' = \alpha_{\max\beta}(v)$, the algorithm would remove v at $(\alpha_{\max\beta}(v), \beta')$-core, where β' is some core value higher than β. Given that, the updates of $\alpha_{\max\beta}(v)$ in

PAR-PEEL-FIX-α in Algorithm 3 as described in Section 5.2 ensures the $\alpha_{\max\beta}(v)$ value recorded is α', the correct value. Because A, B, and C form the entire peeling space, we have shown that for all β and v values, $\alpha_{\max\beta}(v)$ is correctly recorded.

Symmetric correctness arguments can be established for $\beta_{\max\alpha}(u)$ values to show that the overall optimized algorithm is correct.

6 PARALLEL BI-CORE INDEX STRUCTURE

To allow for linear-time queries of (α, β)-cores, we parallelize the indexing structure introduced by Liu *et al.* [24]. In this section, we discuss our parallel index construction and query algorithms. Both algorithms are work-efficient. The index construction algorithm has $O(\log(n))$ span, and the query algorithm has $O(1)$ span.

We define $\mathbb{PI}^U, \mathbb{PI}^V$ to be the parallel index structures for the U, V vertex partitions respectively. $\mathbb{PI}^U$ is the parallel version of $\mathbb{I}^U$, and $\mathbb{PI}^U_{\alpha,\beta}$ is the parallel version of $\mathbb{I}^U_{\alpha,\beta}$. Again, due to symmetry, we only discuss $\mathbb{PI}^U_{\alpha,\beta}$. $\mathbb{PI}^U_{\alpha,\beta}$ is a set containing the same elements as $\mathbb{I}^U_{\alpha,\beta}$ for any combination of α, β. In Liu *et al.*'s work, each set $\mathbb{I}^U_{\alpha,\beta}$ is stored separately, pointed to within $\mathbb{I}^U$. In our parallelization, we store all the sets of vertices contiguously in an array, ordered first by their α value and then by their β value. We call this array P. This is similar to the compressed sparse row (CSR) format used in graph representations. Then, we define M, a 2D jagged array where each $\mathrm{M}[\alpha][\beta]$ corresponds to a set $\mathbb{PI}^U_{\alpha,\beta}$ and contains the starting index of that set in the array P. By definition, $\mathbb{PI}^U_{\alpha,\beta}$ = all vertices in P in range $[\mathrm{M}[\alpha][\beta], \mathrm{M}[\alpha][\beta+1])$.

This way, $[\mathrm{M}[\alpha][\beta], \mathrm{M}[\alpha+1][0])$ directly gives the range of vertices in P that corresponds to $\bigcup_{i=\beta}^{\max_\beta(\alpha)} \mathbb{PI}^U_{\alpha,i} = (\alpha,\beta)$-core.

Thus, to query the (α, β)-core, we return all vertices in P in the range $[\mathrm{M}[\alpha][\beta], \mathrm{M}[\alpha+1][0])$. This takes $O(|(\alpha,\beta)\text{-core}|)$ work and $O(1)$ span.

6.1 Parallel Index Construction

The objective of the index construction algorithm is to take as input $\beta_{\max\alpha}(u)$ for every $u \in U$ and construct M and P.

To construct P, we perform parallel RADIX-SORT on the vertices based on their α, β values. Then, we use a parallel filter to find the indices at which the α or β value differs, which we store in an array TPT (total pointer table). We also find indices where the α value differs, which we store in FPT (first pointer table). Then, using these two pointer tables, the 2D jagged array M is constructed.

The pseudocode for our parallel index construction algorithm is given in Algorithm 4. We now discuss our algorithm in more detail. First, on Line 2, we store a list of tuples $(\alpha, \beta_{\max\alpha}(u), u)$ to P. This is the list of all possible combinations of α and vertex $u \in U$, with the $\beta_{\max\alpha}(u)$ value attached. We perform parallel RADIX-SORT on P based on the ordering of first the α values and then the β values on Line 3. We initialize an empty TPT on Line 4 with the same size as P. The parallel for loop on Lines 5–8 finds all indices at which either the α value or the β value in P changes. This is accomplished by marking the indices where changes happen on Line 7 and filtering out all the unmarked index positions on Line 8. Constructing TPT essentially breaks up the array P into blocks where each block has constant α, β value and corresponds to set $\mathbb{PI}^U_{\alpha,\beta}$. $\mathrm{TPT}[i]$ records the starting index location of the i^{th} block. Lines 9–13 repeats the entire process, but for FPT to filter out index positions where the α value of P changes. Note that the indices stored in FPT are not indices of positions in P. Instead, they are indices of positions in TPT. $[\mathrm{FPT}[\alpha-1], \mathrm{FPT}[\alpha])$ gives the range of blocks defined by TPT that has this particular α value. It corresponds to set $\mathbb{PI}^U_\alpha$.

Algorithm 4 Parallel Index Construction

```
1:  procedure BUILD-U-INDEX($\beta_{\max}$)
2:      P ← list of $(\alpha, \beta_{\max\alpha}(u), u)$ for every $u \in U$
3:      RADIX-SORT(P)                                         ▷ Sorts the list of tuples by first $\alpha$, then $\beta$ values
4:      Initialize TPT                                        ▷ Creates an empty TPT with the same size as P
5:      parfor $i = 0$ to P.size − 1 do
6:          if $i = 0$ or P[$i-1$].$\alpha \neq$ P[$i$].$\alpha$ or P[$i-1$].$\beta \neq$ P[$i$].$\beta$ then
7:              TPT[$i$]← $i$                                 ▷ Records index location where $\alpha$ or $\beta$ changes value
8:      FILTER(TPT, element is not empty)                     ▷ Filter out empty indices in TPT
9:      Initialize FPT                                        ▷ Creates an empty FPT with the same size as TPT
10:     parfor $i = 0$ to TPT.size − 1 do
11:         if $i = 0$ or P[TPT[$i-1$]].$\alpha \neq$ P[TPT[$i$]].$\alpha$ then
12:             FPT[$i$]← $i$                                 ▷ Records index location on TPT array of where $\alpha$ value changes
13:     FILTER(FPT, element is not empty)                     ▷ Filter out empty indices in FPT
14:     Initialize M        ▷ Creates empty M array with 1st dimension = FPT.size and 2nd dimension = $\max_\beta(\alpha)$
15:     parfor $\alpha = 1$ to FPT.size do
16:         parfor $j$ = FPT[$\alpha - 1$] to FPT[$\alpha$]−1 do
17:             start ← TPT[$j$]                              ▷ Gets start index location of $j$th block
18:             M[$\alpha$][P[start].$\beta$] ← start         ▷ Stores start location; P[start].$\beta$ gives $j$th block's corresponding $\beta$ value
19:         M[$\alpha$] ← SUFFIX-MIN(M[$\alpha$])
20:     return M
21: procedure BUILD-V-INDEX($\alpha_{\max}$)
22:     symmetric to BUILD-U-INDEX
```

Finally, based on FPT and TPT, we create M in the following manner. We obtain $M[\alpha]$ for each α value independently. Line 16 iterates in parallel over the blocks that have the particular α value. For each block j, we store its starting position $TPT[j]$ to $M[\alpha][\beta_j]$ where β_j is the β value corresponding with the jth block. This is done on lines 17–18. Now, we have constructed $M[\alpha][\beta']$ correctly for all (α, β') pairs such that β' appears in $\beta_{\max\alpha}(u)$ for some $u \in U$.

For pairs (α, β) such that β does not appear in $\beta_{\max\alpha}(u)$ for some $u \in U$, we point $M[\alpha][\beta]$ to the same destination as $M[\alpha][\beta']$ where β' is the smallest value larger than β and appears in $\beta_{\max\alpha}(u)$ for some $u \in U$. We accomplish this by performing SUFFIX-MIN on $M[\alpha]$ on Line 19.

Analysis. Since P.size = $O(m)$, the work is bounded by $O(m)$ since all operations on lines 3–14 have linear work with respect to the length of the array input. Lines 15–19 have work complexity $O(m)$, because we loop through the M table exactly once and from Section 4.3, we know the table takes $O(m)$ space.

Algorithm 4 has span $O(\log(n))$ *w.h.p.*, since RADIX-SORT, FILTER, and all other operations are bounded by $O(\log(m)) = O(\log(n))$ span *w.h.p.*.

7 EXPERIMENTS

In this section, we provide a comprehensive evaluation of our implementations of parallel bi-core algorithms.

7.1 Experiment Setup

We use real-world graphs from the KONECT graph database [22], the details of which are given in Table 2. Specifically, we used the graphs, in descending number of edges, Orkut, Web Trackers, LiveJournal, Delicious, TREC, Reuters, Epinions, Flickr.

We use Google Cloud Platform `c2-standard-60` instances for all our experiments, which are 30-core machines with two-way hyper-threading, with Intel 3.1 GHz Cascade Lake processors and 240 GB of memory; the processors have a max turbo clock-speed of 3.8 GHz.

Graph Name	Type	$\lvert U\rvert$	$\lvert V\rvert$	n	m	dmax	δ	$\rho_{\max}$
Orkut	Membership	2.78M	8.73M	11.51M	327M	318K	466	12100
Web Trackers	Inclusion	27.7M	12.7M	40.43M	140.6M	11.57M	437	4542
LiveJournal	Membership	3.20M	7.49M	10.69M	112M	1.05M	108	6831
Delicious	Purchase	833K	33.7M	34.6M	101.8M	143K	183	4771
TREC	Inclusion	556K	1.17M	1.73M	83.6M	457K	508	6029
Reuters	Inclusion	781K	284K	1.06M	60.6M	345K	192	4767
Epinions	Rating	120K	755K	880k	13.67M	162K	151	3049
Flickr	Membership	396K	104K	500k	8.55M	35K	147	2300

Table 2. Data and information on tested graphs.

7.2 Implementation and Other Optimizations

While our parallel bi-core decomposition algorithm, given in Algorithm 3, is theoretically efficient, it is practically slow due to the overhead incurred by the histogram-based PAR-DEL-UPDATE subroutine. We find that its high parallelism fails to compensate for this overhead.

To implement a practically fast bi-core decomposition algorithm, we do not implement the fully parallelized version of our algorithm, and only parallelize between different calls of PAR-PEEL-FIX-α and PAR-PEEL-FIX-β. This practical parallel algorithm, which corresponds to PAR-BASELINE, is similar to the parallel algorithm introduced by Liu *et al.* [24]; however, our implementation differs from theirs in that we employ a lazily instantiated bucketing structure, where we only instantiate a constant number of buckets at any given time in our bucketing structure. This technique was introduced by Dhulipala *et al.* [11] for implementing their k-core decomposition algorithm. We further apply the peeling space pruning optimization to PAR-BASELINE to obtain PAR-OPTIMIZED.

To reiterate, the 6 algorithms we implement are

(1) SEQ-BASELINE: Algorithm 1, the state of the art sequential algorithm proposed by Liu *et al.*
(2) SEQ-OPTIMIZED: Algorithm 1, but with peeling space pruning optimization introduced in section 5.4
(3) PAR-BASELINE: Liu *et al.*'s parallel algorithm, but with an optimized bucketing structure
(4) PAR-OPTIMIZED: PAR-BASELINE, but with the peeling space pruning optimization
(5) PAR-INDEX: Algorithm 4
(6) PAR-QUERY: Parallel index query algorithm described in Section 6.

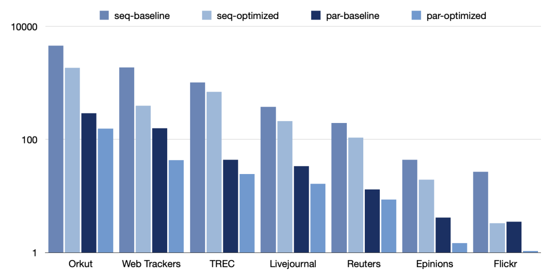
We use the Graph Based Benchmark Suite [13] to implement our algorithms. All code is written in C++ with the -O3 optimization level enabled. We perform each experiment 3 times and report the average running time.

Despite the difference in machines, with Liu *et al.* reporting their runtimes on a 3.4 GHz CPU, our sequential baseline closely reproduces the result reported by Liu *et al.* [24]. The paper reported a runtime of 4103 seconds for their sequential algorithm, while our reproduction attains a runtime of 4539 seconds. We are not able to obtain their source code.

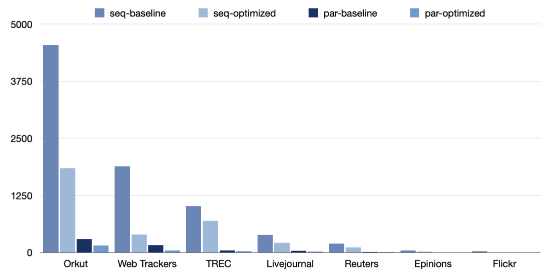
7.3 Bi-core Decomposition

In this section, we report the runtime of the 4 bi-core decomposition algorithms SEQ-BASELINE, SEQ-OPTIMIZED, PAR-BASELINE, and PAR-OPTIMIZED.

Performances Comparison.

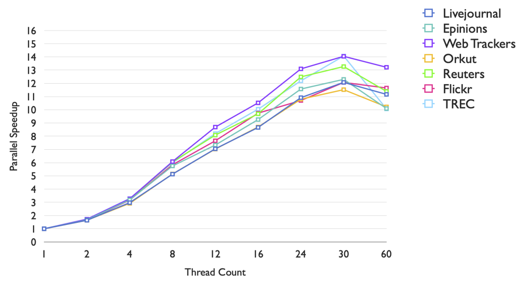


(a) runtimes (log-scale)

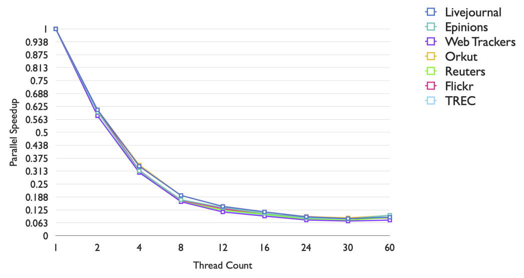


(b) runtimes

Fig. 3. This figure comparatively shows the runtime (in seconds) of the 4 bi-core decomposition algorithms, SEQ-BASELINE, SEQ-OPTIMIZED, PAR-BASELINE, PAR-OPTIMIZED. Graph (a) is in log-scale while graph (b) is in linear-scale. PAR-OPTIMIZED consistently performs all 3 other algorithms.



(a) Speedup Ratios for Tested Graphs



(b) Ratio of Runtimes Relative to Serial Time

Fig. 4. Graph (a) shows plots of the runtime of PAR-OPTIMIZED across different number of threads for various graphs. The plots show consistent speedups. Graph (b) shows the runtime ratios, which are the multiplicative inverses of the speedup numbers

Figure 3 shows the runtimes of the 4 algorithms for graphs in Table 2. The parallel algorithms are run with 30 threads. We do not run on 60 threads because the extra threads are hyperthreads and are not actual cores. Therefore, we observed that using 60 hyperthreads did not improve our running times.

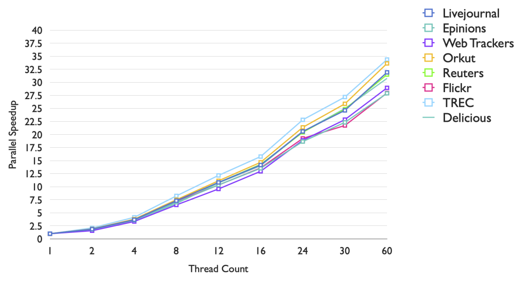
PAR-BASELINE and PAR-OPTIMIZED significantly outperforms SEQ-BASELINE and SEQ-OPTIMIZED. While the parallel algorithms can process graphs with hundreds of millions of edges within several minutes, the sequential algorithms can take more than an hour to finish.

Running on 30 threads, PAR-OPTIMIZED attains 23–44x speedup over SEQ-BASELINE, the sequential state of the art.

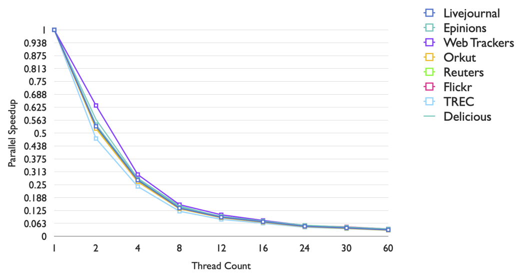
To compare against the parallel state of the art, also introduced by Liu *et al.*, we note that they reported a runtime for 12 threads of 732 seconds for the Orkut graph. On the other hand, still running on 12 threads, PAR-OPTIMIZED attains a runtime of 253 seconds for the Orkut graph. Thus, our algorithm achieves a 2.9x speedup over Liu *et al.*'s parallel algorithm, demonstrating the effectiveness of out introduced optimizations.

Figure 3 also demonstrates the effectiveness of the peeling space pruning optimization we introduce. SEQ-OPTIMIZED consistently outperforms SEQ-BASELINE by 2.1-2.8x. Similarly, PAR-OPTIMIZED is about 1.6-4.2x faster than PAR-BASELINE. This demonstrates the effectiveness of the optimization technique across sequential and parallel setting.

Analysis of Scalability.



(a) Speedup Ratios for Tested Graphs



(b) Ratio of runtimes relative to serial time

Fig. 5. Graph (a) shows plots of PAR-INDEX's speedup over different number of threads used. Graph (b) shows the runtime ratios.

As shown in Figure 4, PAR-OPTIMIZED achieves a 11.5–14.1x self-relative speedup when running on 30 threads compared to 1 thread. The speedup plateaus at 60 threads. Increasing from 30 threads to 60 threads introduces virtual hyper-threads and not physical cores. So it is expected that there is no significant runtime improvement.

Further, Figure 4 demonstrates that the speedup achieved by PAR-OPTIMIZED is consistent across different graphs, showing that PAR-OPTIMIZED is scalable to graphs of sizes up to hundreds of millions of edges without noticeable deterioration in its parallel speedup.

7.4 Bi-core Index Construction

Now, we report the runtimes of the bi-core index construction algorithm, or PAR-INDEX. Using 30 threads, PAR-INDEX terminates for most graphs within several seconds. For example, when run on Orkut, a graph with 327 millions edges, with 30 threads, PAR-INDEX takes only 3.76 seconds to finish. This is 2.4% of the runtime of PAR-OPTIMIZED. PAR-INDEX takes up similar percentage of time for other large graphs while taking up larger portion of time for smaller graphs. For example, on Flickr, PAR-INDEX is 10% of PAR-OPTIMIZED in terms of runtime. This observation matches the theoretical prediction based on PAR-INDEX's lower work complexity as compared to PAR-OPTIMIZED, which predicts that the runtime of PAR-OPTIMIZED grows faster than that of PAR-INDEX as the number of edges increases.

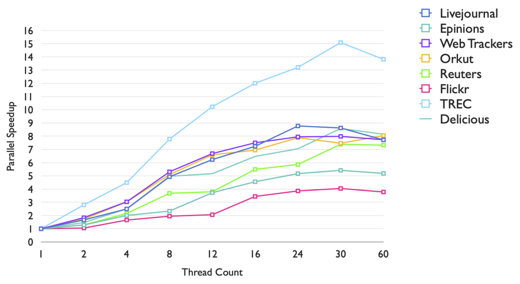
Analysis of Scalability.

Given its span of $O(\log(n))$, PAR-INDEX predictably has significant parallelism. Figure 5 demonstrates the parallel speedup of PAR-INDEX across different number of threads and different graphs used. Running on Delicious with 30 threads, PAR-INDEX achieves a near-linear self-relative speedup of 25.0x. A similar speedup number is achieved for all other graphs, as shown in Figure 5. This demonstrates PAR-INDEX to be scalable to graphs of large sizes.

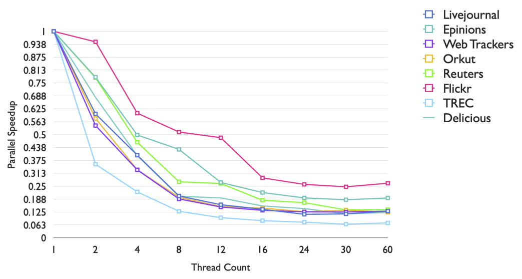
7.5 Bi-core Index Query

In this section, we give the runtimes of our bi-core index query algorithm. Following the convention adopted by Liu *et al.*, we report the runtime of completing 10 query calls with each query having $\alpha = 10, \beta = 10$. On the graph Delicious, Liu *et al.* reported a runtime of 0.04 second. PAR-QUERY, running on a single thread, outperforms it, attaining a runtime of 0.0154 second. Our implementation achieves a 2.6x speedup. This proves that our optimization of storing the index structure in CSR format is effective (it potentially reduces the number of cache misses). Using 30 threads, PAR-QUERY achieves a runtime 0.0018 second, which is a 22.3x speedup over the Liu *et al.*'s sequential query algorithm.

Analysis of Scalability.



(a) Speedup ratios for tested graphs



(b) Ratio of runtimes relative to serial time

Fig. 6. Graph (a) shows plots of PAR-QUERY's speedup over different number of threads used. Graph (b) shows the runtime ratios.

Figure 6 demonstrates the tangible self-relative speedups achieved by PAR-QUERY. It achieves up to 15.1x self-relative speedup using 30 threads as compared to 1 thread. The speedup achieved is mostly consistent across different graphs, but has higher volatility when compared to the PAR-OPTIMIZED or PAR-INDEX. This is expected due to the low runtime of the query algorithm as well as its strong dependence on cache efficiency, with parallel copy of contagious array being its only time-consuming operation.

8 CONCLUSION

In this paper, we study parallel algorithms for bi-core decomposition, which is an important theoretical problem with many real-world applications. We develop the first shared-memory work-efficient parallel bi-core decomposition algorithm with nontrivial span bounds. Our parallel algorithm improves the span complexity from the state-of-the-art $O(m)$ to $O(\rho \log(n))$ *w.h.p.*. Practically, the span we achieve is 2–3 orders of magnitude lower than the span of existing parallel algorithm. Further, we prove the problem of bi-core decomposition to be P-complete. We also introduce a parallel indexing structure to store the bi-cores, and provide a work-efficient parallel index construction algorithm and query algorithm. Finally, we introduce optimizations such as peeling space pruning and provide optimized implementations of our algorithms. We perform experimental evaluation of our algorithms on real-world bipartite graphs. Our parallel algorithms outperform sequential baselines by up to 44x for peeling and 22.3x for query. Experimental results also prove our bi-core decomposition algorithms are capable of scaling to real-world graphs with hundreds of millions of edges, processing them in minutes and performing bi-core queries in milliseconds.

ACKNOWLEDGEMENTS

We would like to thank Jessica Shi and Professor Julian Shun (MIT CSAIL) for their amazing support and guidance throughout the year. In addition, we thank the MIT PRIMES organization for this opportunity.

REFERENCES

[1] Adel Ahmed, Vladimir Batagelj, Xiaoyan Fu, Seok-hee Hong, Damian Merrick, and Andrej Mrvar. 2007. Visualisation and analysis of the internet movie database. In *2007 6th International Asia-Pacific Symposium on Visualization*. 17–24. https://doi.org/10.1109/APVIS.2007.329304

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

[3] Richard J. Anderson and Ernst W. Mayr. 1984. A P-complete problem and approximations to it.

[4] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. 2013. CopyCatch: Stopping Group Attacks by Spotting Lockstep Behavior in Social Networks. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/2488388.2488400

[5] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. 1991. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '91)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/113379.113380

[6] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206.

[7] Monika Cerinšek and Vladimir Batagelj. 2015. Generalized two-mode cores. *Social Networks* 42 (2015), 80–87. https://doi.org/10.1016/j.socnet.2015.04.001

[8] Jonathan Cohen. 2008. Trusses: Cohesive Subgraphs for Social Network Analysis. (2008).

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

[10] N. S. Dasari, R. Desh, and M. Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *IEEE International Conference on Big Data*. 9–16.

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

[12] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 393–404.

[13] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS) *(GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/3398682.3399168

[14] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. 2017. Efficient Fault-Tolerant Group Recommendation Using Alpha-Beta-Core. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. Association for Computing Machinery, New York, NY, USA, 2047–2050. https://doi.org/10.1145/3132847.3133130

[15] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2019. A Survey of Community Search Over Big Graphs. arXiv:cs.DB/1904.12539

[16] Valeria Fionda, Luigi Palopoli, Simona Panni, and Simona E. Rombo. 2007. Bi-grappin: bipartite graph based protein-protein interaction network similarity search. In *2007 IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2007)*. 355–361. https://doi.org/10.1109/BIBM.2007.13

[17] J. Garcia-Algarra, J. M. Pastor, M. L. Mouronte, and J. Galeano. 2017. A structural approach to disentangle the visualization of bipartite biological networks. *bioRxiv* (2017). https://doi.org/10.1101/192013 arXiv:https://www.biorxiv.org/content/early/2017/11/21/192013.full.pdf

[18] GraphChallenge [n.d.]. GraphChallenge. http://graphchallenge.mit.edu/.

[19] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. 2015. A Top-Down Parallel Semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 24–34.

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

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

[22] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. *International Conference on World Wide Web*, 1343–1350.

[23] Kartik Lakhotia, Rajgopal Kannan, Viktor Prasanna, and Cesar A. F. De Rose. 2020. Receipt: Refine Coarse-Grained Independent Tasks for Parallel Tip Decomposition of Bipartite Graphs. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 404–417.

[24] Boge Liu, L. Yuan, Xuemin Lin, Lu Qin, W. Zhang, and Jingren Zhou. 2020. Efficient (α, β)-core computation in bipartite graphs. *VLDB J.* 29 (2020), 1075–1099.

[25] David W. Matula and Leland L. Beck. 1983. Smallest-last Ordering and Clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (July 1983), 417–427.

[26] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2013. Distributed k-Core Decomposition. *IEEE Transactions on Parallel and Distributed Systems* 24, 2 (2013), 288–300. https://doi.org/10.1109/TPDS.2012.124

[27] Georgios A Pavlopoulos, Panagiota I Kontou, Athanasia Pavlopoulou, Costas Bouyioukos, Evripides Markou, and Pantelis G Bagos. 2018. Bipartite graphs in systems biology and medicine: a survey of methods and applications. *GigaScience* 7, 4 (02 2018). https://doi.org/10.1093/gigascience/giy014 giy014.

[28] Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms. *SIAM J. Comput.* 18, 3 (1989), 594–607. https://doi.org/10.1137/0218041 arXiv:https://doi.org/10.1137/0218041

[29] Jane B. Reece, Noel Meyers, Lisa A. Urry, Michael L. Cain, Steven A. Wasserman, Peter V. Minorsky, Robert B. Jackson, Bernard J. Cooke, and Neil A. Campbell. 2015 2015. *Campbell biology / Jane B. Reece, Noel Meyers, Lisa A. Urry, Michael L. Cain, Steven A. Wasserman, Peter V. Minorsky, Robert B. Jackson, Bernard Cooke* (tenth edition. australian and new zealand version. ed.). Pearson Frenchs Forest, NSW. xliv, 1315, A–49, B–1, C–1, D–1, E–2, F–3, CR–10, G–37, I–54 pages : pages.

[30] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2018. Butterfly Counting in Bipartite Networks. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*. 2150–2159.

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

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

[33] Ahmet Erdem Sariyuce, C. Seshadhri, and Ali Pinar. 2017. Parallel Local Algorithms for Core, Truss, and Nucleus Decompositions. (04 2017).

[34] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. 2018. Local Algorithms for Hierarchical Dense Subgraph Discovery. 12, 1 (2018). https://doi.org/10.14778/3275536.3275540

[35] Ahmet Erdem Sariyuce, C. Seshadhri, Ali Pinar, and Umit V. Catalyurek. 2015. Finding the Hierarchy of Dense Subgraphs Using Nucleus Decompositions. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE. https://doi.org/10.1145/2736277.2741640

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

[37] Jessica Shi, Laxman Dhulipala, and Julian Shun. 2021. Parallel Clique Counting and Peeling Algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*. SIAM, 135–146. https://doi.org/10.1137/1.9781611976830.13

[38] Jessica Shi and Julian Shun. 2020. Parallel Algorithms for Butterfly Computations. *ArXiv* abs/1907.08607 (2020).

[39] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. 68–70.

[40] Yifan Sun, Nicolas Agostini, Shi Dong, and David Kaeli. 2019. Summarizing CPU and GPU Design Trends with Product Data.

[41] Alok Tripathy, Fred Hohman, Duen Horng Chau, and Oded Green. 2018. Scalable K-Core Decomposition for Static Graphs Using a Dynamic Graph Data Structure. In *2018 IEEE International Conference on Big Data (Big Data)*. 1134–1141. https://doi.org/10.1109/BigData.2018.8622056

[42] Charalampos Tsourakakis. 2015. The K-Clique Densest Subgraph Problem. In *Proceedings of the 24th International Conference on World Wide Web (WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1122–1132. https://doi.org/10.1145/2736277.2741098

[43] Balaji Venu. 2011. Multi-core processors - An overview. (10 2011).

[44] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proceedings of the VLDB Endowment* 5 (05 2012). https://doi.org/10.14778/2311906.2311909

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

[46] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient Bitruss Decomposition for Large-scale Bipartite Graphs. 661–672. https://doi.org/10.1109/ICDE48307.2020.00063

[47] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2021. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal* (03 2021), 1–24. https://doi.org/10.1007/s00778-021-00658-5

[48] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. 2020. Efficient and Effective Community Search on Large-scale Bipartite Graphs. (11 2020).

[49] D. J. Watts and S. H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 409–10.

[50] Marco Zagha and Guy E. Blelloch. 1991. Radix sort for vector multiprocessors. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. 712–721. https://doi.org/10.1145/125826.126164

[51] Yang Zhang and Srinivasan Parthasarathy. 2012. Extracting Analyzing and Visualizing Triangle K-Core Motifs within Networks. (04 2012), 1049–1060. https://doi.org/10.1109/ICDE.2012.35

[52] Feng Zhao and Anthony Tung. 2012. Large scale cohesive subgraphs discovery for social network visual analysis. *Proceedings of the VLDB Endowment* 6, 85–96. https://doi.org/10.14778/2535568.2448942

[53] Zhaonian Zou. 2016. Bitruss Decomposition of Bipartite Graphs. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-32049-6_14