S.T. Yau High School Science Award

Research Report

The Team

Registration Number:

Name of team member: Claire Wang
School: Phillips Academy Andover
City, Country: Andover, MA

Name of team member: Yihao Huang
School: Phillips Academy Andover
City, Country: Andover, MA

Title of Research Report

Efficient Algorithm for Parallel Bi-core Decomposition

Date

September 15, 2021

Efficient Algorithm for Parallel Bi-core Decomposition

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

Abstract

Many real-world statistics and problems can be modeled by graphs, such as user-product networks, social networks, and biological networks. Identifying dense regions within these graphs is useful for product-recommendation, spam identification, and protein-function discovery. k-core decomposition is a fundamental graph theory problem that discovers dense substructures of a graph. However, k-core decomposition does not directly apply to bipartite graphs, which are graphs that model the connections between two disjoint sets of entities. Bipartite graphs are widely used to model authorship, affiliations, and gene-disease associations, to name a few. In this paper, we solve the analog 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, in this paper, we develop a theoretically efficient parallel bi-core decomposition algorithm. Compared to existing parallel algorithms, our algorithm reduces the length of the longest dependency path of the computational graph which measures the asymptotic bound of a parallel algorithm given sufficiently many threads. We provide an optimized parallel implementation that is scalable and fast. Using 30threads, our parallel algorithm achieves up to 34.8x self-relative speedup. Our code achieves up to 4.1x speedup compared with the best existing parallel algorithm.

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

Acknowledgements

Jessica Shi, CSAIL, jeshi@mit.edu
Julian Shun, CSAIL, jshun@mit.edu

Commitments on Academic Honesty and Integrity

We hereby declare that we

1. are fully committed to the principle of honesty, integrity and fair play throughout the competition.
2. actually perform the research work ourselves and thus truly understand the content of the work.
3. observe the common standard of academic integrity adopted by most journals and degree theses.
4. have declared all the assistance and contribution we have received from any personnel, agency, institution, etc. for the research work.
5. undertake to avoid getting in touch with assessment panel members in a way that may lead to direct or indirect conflict of interest.
6. undertake to avoid any interaction with assessment panel members that would undermine the neutrality of the panel member and fairness of the assessment process.
7. observe the safety regulations of the laboratory(ies) where we conduct the experiment(s), if applicable.
8. observe all rules and regulations of the competition.
9. agree that the decision of YHSA is final in all matters related to the competition.

We understand and agree that failure to honour the above commitments may lead to disqualification from the competition and/or removal of reward, if applicable; that any unethical deeds, if found, will be disclosed to the school principal of team member(s) and relevant parties if deemed necessary; and that the decision of YHSA is final and no appeal will be accepted.

(Signatures of full team below)

X. Claire Wang
Name of team member:

X. Yihao Huang
Name of team member:

X
Name of team member:

X. Jessica Shi
Name of supervising teacher:

Efficient Algorithm for Parallel Bi-core Decomposition

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

CONTENTS

1 INTRODUCTION

The problem of discovering dense clusters and subgraphs in networks is fundamental in large-scale graph analysis. It is used for community search in social networks, cluster word-documents, improving advertising and marketing, detect frauds, and analyze protein-gene-disease relations in bioinformatics and medicine [32, 40]. Classical problems for dense subgraph discovery include k-core [30] decomposition, k-truss [14], and nucleus decomposition [35]. However, these algorithms apply to general graphs, and do not take advantage of 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 [51], peer-to-peer exchange networks, gene-disease associations, protein-protein interactions, and enzyme-reaction links [20–22, 32].

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 is often inaccurate and memory-inefficient [40]. Therefore, bipartite analogs 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 [49]. The bipartite equivalent of k-core (bi-core) was introduced by Ahmed *et al.* [8]. 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 [10, 19, 50].

(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 creating a large number of fake social media accounts to like/dislike specific posts or online products in order to manipulate public opinions. Since 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 peeling discovers 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 important species in an ecosystem. Alternative to k-core peeling, bi-core peeling can also be applied to the problem and can potentially generate more accurate representations since bi-core decomposition addresses the imbalance between the two entities. For example, there are possibly more predator species than prey species, so intuitively, the number of degrees a prey specie needs to be considered a generalist should be lower than that of a predator specie [33].
(3) Community Search Wang *et al.* applied bi-core decomposition to compute (α, β)-community search. Specifically, their algorithm prunes the searching space by first computing the (α, β)-core and only searching the (α, β)-community within it.

Liu *et al.* [29] proposed an efficient sequential index-based approach for bi-core decomposition. Their algorithm runs in $O(m^{\frac{3}{2}})$, meaning that asymptotically, its runtime is bounded by the expression $m^{\frac{3}{2}}$. The algorithm uses memory linear to m. Liu *et al.* leveraged computation-sharing across different rounds of peeling to reduce the time complexity from $O(n^2)$ of earlier works to $O(m^{\frac{3}{2}})$ [12, 29]. However, the sequential nature of the algorithm limits its application to large real-world graphs.

Parallelism is increasingly vital for faster processing in as it becomes increasingly difficult to increase CPU clock speeds [42]. As the number of cores in a processor grows, shared-memory parallelism, in particular, becomes increasingly important [45].

We develop in this paper a shared-memory parallel bi-core decomposition algorithm. Our algorithm uses a peeling-based approach, where in each round of peeling, we remove all vertices with the lowest induced degree concurrently until the graph is empty. We prove that our algorithm achieves $O(m^{\frac{3}{2}})$ work, meaning that the total number of operations performed is asymptotically bounded by the expression $m^{\frac{3}{2}}$. Our algorithm is work-efficient, meaning that it has the same

work complexity as the best sequential algorithm. Our algorithm achieves $O(\rho \log(n))$ *w.h.p.* span, meaning that the length of the longest dependency path in the computational graph is bounded asymptotically by the expression $\rho \log(n)$. ρ is the bi-core peeling complexity; it represents the maximum number of rounds of peeling. Our algorithm uses $O(m)$ space. Note that the parallel algorithm introduced by Liu *et al* has a span of $O(m)$. Since $\rho \leq n$ theoretically and is generally much smaller than n in practice as shown in Table 2, $O(\rho \log(n))$ is a significantly better span than $O(m)$.

We furthermore implement our parallel algorithm and introduce optimizations to it. Finally, we present a comprehensive experimental evaluation of our algorithm on real-world graphs that contain up to hundreds of millions of edges. We compare our experimental results against Liu *et al.*'s algorithm, which we use as our baseline. Our algorithm achieves up to 4.1x speedup over the parallel baseline. Further, it achieves up to 16.6-35.4x speedup over the sequential baseline when using 24 threads. We demonstrate good parallel scalability over different numbers of threads, and we provide an evaluation of the bi-core peeling complexity as it relates to our running times.

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

(1) We introduce the first theoretically efficient shared-memory parallel bi-core decomposition algorithm.
(2) We introduce optimizations and provide an implementation of our parallel bi-core decomposition algorithm, which is publicly available at https://github.com/ClaireBookworm/gbbs.
(3) We perform extensive empirical evaluations of our algorithm.

2 RELATED WORK

k-core Decomposition. The bi-core decomposition problem is an extension of the k-core decomposition problem, which is well-studied, with the first efficient sequential algorithm given by Matula and Beck [30]. Parallel algorithms, both distributed memory [31] and shared memory [17], have also been developed (e.g., [16–18, 25, 31, 43]).

Other Dense Subgraph Decomposition. k-clique peeling, k-truss, and (r, s)-nucleus decomposition all discover dense substructures in a graph, similar to the k-core decomposition. k-clique peeling [13, 44] peels vertices based on the number of incident k-cliques and is a generalization of k-core peeling; k-truss peeling [9, 14, 26, 38, 46, 52, 53] peels edges based on their number of incident triangles; (r, s)-nucleus decomposition [37, 39] further generalizes both k-core peeling and k-truss, by peeling r-cliques based on their number of incident s-cliques. The MIT GraphChallenge [23]

Generalization of Decomposition Algorithms to Bipartite Graphs. Another direction of work focuses on generalizing unipartite peeling algorithms to bipartite graphs. Zou [54] and Sarıyüce and Pinar [36] generalized k-clique and k-truss peeling to k-tip decomposition and k-wing decomposition. K-tip peeling peels vertices based on the number of incident $(2, 2)$-bicliques; k-wing decomposition peels edges instead based on the number of incident $(2, 2)$-bicliques. Multiple sequential [34, 36, 47–49, 54] and parallel [28, 41] algorithms have been proposed for tip and wing decomposition problems. Ahmed *et al.* proposed the (α, β)-core decomposition problem, or the bi-core decomposition problem and gave the first sequential bi-core peeling algorithm [8]. Then, Cerinšek and Batagelj gave the first sequential bi-core peeling algorithm. Ding *et al.* applied bi-core to recommender systems and provided a sequential bi-core peeling algorithm based on the k-core peeling algorithm [19]. Liu *et al.* developed an efficient computation sharing sequential bi-core peeling algorithm and a memory-efficient indexing structure to store the bi-cores [29]. Wang *et al.* extended the problem to weighted bipartite graphs to find the bi-core with the highest edge weights containing a given query vertex [50].

3 PRELIMINARIES

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

Table 1. Graph Notation Summary

G	An undirected, simple, bipartite graph
U	One of the vertex subset of G, one of the bipartition
V	The other vertex subset of G, the other bipartition
$\deg(x)$	Degree or induced degree of generic vertex x, depending on context
dmax_v	The maximum vertex degree in V
dmax_u	The maximum vertex degree in U
$\max_\alpha(\beta)$	The max α value such that (α, β)-core is nonempty
$\max_\beta(\alpha)$	The max β value such that (α, β)-core is nonempty
δ	The max δ value such that (δ, δ)-core is nonempty. In other words, it is the maximum unipartite k-core number of graph G

Graph Definitions.

See Table 1 for a table of graph notations.

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 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* $u \in U'$*,* $\deg(u) \geq \alpha$*, and for every* $v \in V'$*,* $\deg(v) \geq \beta$*.*

Here $\deg(u)$ denotes the induced degree of vertex u in the induced subgraph G'. We make no distinction between the notation of a vertex's degree in G versus in G' and assume that it is clear from the context.

Note the following:

REMARK 1. *if* $u \in (\alpha_1, \beta_1)$*-core, then* $u \in (\alpha_2, \beta_2)$*-core if* $\alpha_2 \leq \alpha_1$ *and* $\beta_2 \leq \beta_1$*.*

REMARK 2. *Every nonempty* (α, β)*-core must have* $\alpha \leq \delta$ *and/or* $\beta \leq \delta$*.*

Assume for the sake of contradiction there exist a nonempty (α, β)-core with $\alpha > \delta$ and $\beta > \delta$, then by remark 1, 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, reaching a contradiction.

Problem Statement. For $u \in U$, we define $\beta_{max\ \alpha}(u)$ for a fixed α to be the maximum β value such that $u \in (\alpha, \beta)$-core. 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.

Given these definitions, we define the problem of bi-core decomposition as follows:

DEFINITION 2. *The* bi-core decomposition *problem finds the* $\beta_{max\ \alpha'}(u)$ *for every* $u \in U$ *and every fixed* α'*, and* $\alpha_{max\ \beta'}(v)$ *for every* $v \in V$ *and every fixed* β' *[29].*

With these values, we 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 due to remark 1. Similarly, if $\alpha_{\max\ \beta}(v) \geq \alpha$, then $v \in (\alpha, \beta)$-core.

Model of Computation. We use the shared memory model of parallel computation; we use the 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 [15]. *Brent's Theorem* [11] states that given an algorithm's work T_1 and span T_∞, the algorithm's runtime 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. Or in other words, forking n processes has a span of $O(1)$. With the provided conditions, we show that our algorithm is *work-efficient*, meaning that it has the same work complexity as the best sequential algorithm.

Evaluated with the work-span model, our algorithm has work $O(m)$ and span $O(\rho \log(n))$ *w.h.p.* [1] ρ is the bi-core peeling complexities, as defined in Section 5.1.

Parallel Primitives. Here, we define the parallel primitives that we use throughout our algorithms.

ATOMIC-WRITE-MAX(a, b) takes as input two values, a and b. If $b > a$, then it atomically updates a to b.

PREFIX-SUM(A) takes as input a sequence and returns as output a sequence of the same length such that each element equals the sum of all elements before it and itself in the original sequence. PREFIX-SUM has $O(n)$ work and $O(\log(n))$ *w.h.p.* span where n is the length of the sequence [15].

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 $O(\log(n))$*w.h.p.* [15].

FILTER(A) takes as input a sequence and a condition for filtering. It retains all items for which the condition is true and then outputs the resulting sequence. The function returns a sequence of filtered elements in $O(n)$ work and $O(\log(n))$ *w.h.p.* span [15].

HISTOGRAM(A) takes as input a sequence of indices. It applies semisort to the indices and create a histogram of the frequencies of each index. It takes $O(n)$ expected work and $O(\log(n))$ span *w.h.p.* [24].

4 SEQUENTIAL BI-CORE DECOMPOSITION

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 $1 \leq \beta \leq \text{dmax}_v$ is symmetric to the problem of finding the $\beta_{\max\ \alpha}(u)$ values for all possible u and α. Therefore, 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. Therefore, a peeling-based algorithm is often applied to solve the problem [8, 19]. In a baseline peeling-based algorithm, we apply PEEL-FIX-β to every β' between 1 and dmax_v. PEEL-FIX-β takes as input a specific fixed β' value. Then the algorithm increases the α value of the (α, β')-core from 1 to $\max_\alpha(\beta')$ while iteratively deleting vertices no longer within the current (α, β')-core. In other words, the algorithm peels by α from 1 to $\max_\alpha(\beta')$. While deleting a vertex v when increasing the α value to $\alpha + 1$, we update $\alpha_{\max\ \beta'} \leftarrow \alpha$, because it is the highest α value for which $v \in (\alpha, \beta')$-core. We perform the mirror operations for PEEL-FIX-α.

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

4.2 Computation Sharing

Liu *et al.* observed that it is unnecessary to perform the α-core peeling for all possible β' values. It is sufficient to perform α-core peeling for $1 \leq \beta' \leq \delta$. The only modification needed is to also update the $\beta_{\max\ \alpha}(u)$ value while deleting a vertex u in the peeling process. When we delete u while increasing the α value to $\alpha + 1$, we know that $u \in (\alpha, \beta')$-core. So we can update $\beta_{\max\ \alpha}(u)$ value to at least β'. Due to remark 1, we can also update $\beta_{\max\ i}(u)$ to at least β', for $i < \alpha$. Provided that α-core peeling 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. Given $u \in (\alpha, \beta_{\max\ \alpha}(u))$-core and $\alpha > \delta$, we know $\beta_{\max\ \alpha}(u) \leq \delta$ due to remark 2.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.

Algorithm 1 Sequential Baseline 1 [29]

```
1:  procedure SEQ-BI-CORE(G)
2:      for α′ = 1 to δ do
3:          PEEL-FIX-α(G,α′)
4:      for β′ = 1 to δ do
5:          PEEL-FIX-β(G,β′)
6:  procedure PEEL-FIX-β(G,β′)
7:      DEL-UPDATE(v if deg(v) < β′)                    ▷ Peel V from 1 to β′ − 1
8:      while U ≠ ∅ do
9:          delU, α ← FIND-MIN(deg(u))                  ▷ Get min unpeeled degree in U
10:         for all u in delU do
11:             for i = 1 to α do
12:                 β_max i(u) ← max(β_max i(u), β′)
13:         delV ←DEL-UPDATE(delU, β)                   ▷ Peel U up to α
14:         for all v in delV do
15:             α_max β′(u) ← α                         ▷ Update α_max β′
16:         DEL-UPDATE(delV, α)                         ▷ Remove peeled v
17: procedure PEEL-FIX-α(G,α′)
18:     mirror image of PEEL-FIX-β
19: procedure DEL-UPDATE(delX, k)
20:     delY ← ∅
21:     for all x in delX do
22:         for all y in neighbor(x) do
23:             deg(y) ← deg(y) − 1
24:             if deg(y) < k then
25:                 add y to delY
26:                 mark y as removed
27:         mark x as removed
28:     return delY
```

We now provide a more detailed description of the algorithm. On Lines 4–5 of Algorithm 1, we loop over all $1 \leq \beta' \leq \delta$ and run PEEL-FIX-β on each β'. Each iteration of PEEL-FIX-β peels α from 1 to $\max_\alpha(\beta')$ for the given β'. Specifically, it does the following. On Line 7, DEL-UPDATE deletes all vertices v with $\deg(v) < \beta'$ because these vertices are not in any (α, β')-core for the given β'. Then, until all $v \in V$ are peeled off, we execute Lines 9–16. On Line 9, we find the set of vertices u with the current minimum degree and store it to delU. We store the current minimum degree to α. At this point, all remaining vertices are in (α, β')-core. A round of peeling will peel off all vertices with induced degree $\leq \alpha$, which are stored in delU. Lines 10–12 record the $\beta_{\max\ \alpha'}(u)$ values for all $u \in$ delU as described in section 5.2. Line 13 deletes all vertices $u \in$ delU. The DEL-UPDATE function iteratively removes vertices in delU while updating the degrees of their neighbors. If the neighbors' degree fall below β', that means $v \notin (\alpha + 1, \beta')$-core and can be peeled, so they are recorded in delY. We update the $\alpha_{\max\ \beta'}(u)$ values as described in section 5.1 on line 15 and peel vertices in delV on line 16.

For PEEL-FIX-β, we simply mirror PEEL-FIX-α, swapping α' with β, β' with α, V with U, and v with u.

4.3 Analysis and Implementation Details

To analyze the runtime of the entire algorithm, we first focus on the runtime of PEEL-FIX-α. We separate it into several parts. First, notice the runtime of all DEL-UPDATE operations is bounded by $O(m)$. This is because for each generic vertex v we delete, we traverse its neighbors to update their degrees. Thus each vertex's neighbors will only be traversed once. That gives a runtime of $O(\sum_{x \in V \text{ or } U} \deg(i)) = O(m)$ for the part of DEL-UPDATE. In our implementation, we do not delete vertices or edges from the graph during peeling but simply mark vertices as deleted. This does not change the complexity because each edge (u, v) can only be traversed twice in each peeling, once when u is peeled and once when v is peeled. Line 14–15 is bounded by $O(n)$ since there can be at most n updates to αmax $\beta'(v)$, because there are only $O(n)$ vs. Line 10–12 is also bounded by $O(m)$ because the maximal α value is $O(\deg(u))$, thus giving a total runtime of $O(\sum_{u \in U} \deg(u)) = O(m)$. FIND-MIN can be achieved in $O(\text{dmax}_u) = O(n)$ with sequential search. In our implementation, we organize vertices into an array of buckets; each bucket stores all vertices with degree corresponding to the bucket's index; FIND-MIN sequentially searches for the next nonempty bucket. Thus PEEL-FIX-α runs in $O(m)$ time[29]. Since δ is bounded by $O(\sqrt{m})$, Algorithm 1 runs in $O(\delta m)$ or more loosely $O(m^{\frac{3}{2}})$.

5 PARALLEL BI-CORE DECOMPOSITION ALGORITHM

The sequential nature of Liu *et al*'s bi-core decomposition algorithm (1) limits its practical applicability to large graphs. We present in this section a parallel bi-core decomposition algorithm using the same computation-sharing technique developed by Liu *et al*. 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 defines as follows:

DEFINITION 3. *The bi-core peeling complexity is the maximum number of rounds of peeling executed by any call of* PEEL-FIX-α *or* PEEL-FIX-β*).*

Our algorithm is peeling-based, with the process of bi-core decomposition separated into rounds of peeling. For 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. Notably, these vertices are peeled concurrently as opposed to sequentially in Algorithm 1. Due to the concurrency of the peeling, we need to update peeled vertices' neighbors using a parallel aggregated-peeling approach described in section 6.2. Meanwhile, to reduce the span of the sequential search used in Algorithm 1, we introduce parallel exponential search on a parallel bucketing structure to find the next bucket of vertices with minimum degree; we discuss this in the following section.

5.1 Parallel Bucketing and Exponential Search

To achieve polylogarithmic span while maintaining work-efficiency, we use a bucketing structure to store the subset of vertices to be peeled at each round. Dhulipala *et al.* introduced this parallel bucketing structure and applied it to parallel k-core peeling [18]. The data structure consist of an array of buckets indexed by degrees. Each bucket stores all vertices with current degree corresponding to its index. When the degrees of vertices change due to other vertices being peeled, we call UPDATE-VERTICES to update those vertices to new buckets corresponding to the new degrees in parallel. In the beginning of each round of peeling, we call NEXT-BUCKET to search for the next subset of vertices with lowest induced degree $\geq$ the current degree value. We give the pseudo code for the parallel exponential search used in NEXT-BUCKET in Algorithm 3 and we implement our parallel bi-core peeling algorithm in Algorithm 4.

NEXT-BUCKET finds and returns the next nonempty bucket with degree $\geq k$. In each iteration of the while loop on Line 3–4 of Algorithm 3, we determine if the interval $(k + \frac{i}{2}, k + i]$ contains the next nonempty bucket. Then, we double

Algorithm 2 Parallel Exponential Search

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

i and repeats until the next nonempty bucket is found. For example, we start the search from the interval $(k, k+1]$. HAS-MIN-DEG on Line 3 Algorithm 3 checks 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 all i. Note that one minor detail we exclude from consideration here is that some vertices in V', after the latest round of peeling could take on a degree $\leq k$. We simply set their degree to k to delete them all together in the next round of peeling. Or, in other words, the next bucket can sometimes be the current bucket with new vertices added and the old ones peeled. On Line 5, we call REDUCE-MIN on the sequence of the indices of nonempty buckets to obtain the next minimum degree with nonempty bucket. On Line 6, we return the next nonempty bucket.

Analysis. First, we prove the work-efficiency of NEXT-BUCKET over all calls to the algorithm in each call to PAR-PEEL-FIX-α. Assume NEXT-BUCKET is called with current degree value k and that the next minimum degree is $k+p$. Then, notice that NEXT-BUCKET searches at most $2p$ elements before terminating and returning $k+p$ as the next minimum degree. If it searches only k elements ahead, the algorithm has an overall work (in each complete β-core peeling) of $O(\text{dmax}_v) = O(n)$. Since it searches only $2p$ ahead, its work is bounded by $O(2n) = O(n)$ as well.

Next, we show that NEXT-BUCKET has a span of $O(\rho \log(n))$ over all iterations in one call to PAR-PEEL-FIX-α. Assume, as previously, that the current degree is k and the next degree is $k+p$. Note that NEXT-BUCKET takes at most $\log(k)$ iterations of its while loop on Line 3 Algorithm 3 to find the next minimum degree $k+p$. To loosen the bound, $\log(k) = O(\log(n))$. Note that there are $O(\rho)$ rounds of peeling (or $O(\rho)$ calls to NEXT-BUCKET) in a complete β-core peeling (or a call to PEEL-FIX-α). Therefore, the total span of NEXT-BUCKET is $O(\rho \log(n))$. An assumption we make in this derivation is that HAS-MIN-DEG as called on Line 32 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 5 Algorithm 3 has span $O(\log(n))$ is executed only once for each round, totaling a span of $O(\rho \log(n))$ as well.

5.2 Parallel Aggregated-Peeling

At the crux of PAR-PEEL-FIX-α and PAR-PEEL-FIX-β of Algorithm 4 is the PAR-DEL-UPDATE operation which takes as input a generic subset of vertices X_{del} and then peels all its vertices in parallel. On Lines 3–6, we iterate through all neighbors y of X_{del} and store them in an array Y_{update}. On Line 11, HISTOGRAM returns a sequence of pairs (y, count). For every y, count stores the number of its occurrences in delY. On Lines 12–13, we iterate through each y and decrease its degree by its corresponding count.

Since many threads may be updating the degree of the same vertex, our aggregation-based approach is necessary to avoid the atomic update operations that would otherwise add sequential elements into the algorithm.

Algorithm 3 Parallel Aggregated-Peeling

```
1: procedure PAR-DEL-UPDATE(X_del)
2:     Y_update ← ∅
3:     degs ← degrees of vertices in X_del
4:     offsets ← PREFIX-SUM(degs)
5:     parfor all i, x in X_del do
6:         mark x as removed
7:         parfor all j, y in neighbor(x) do
8:             offset ← offsets[i]+j
9:             Y_update[offset] y to Y_update                ▷ Record y for degree update
10:    Y_update ← FILTER(Y_update, not marked as deleted)
11:    Y_update ← HISTOGRAM(Y_update)                        ▷ Count occurrences of vertices
12:    parfor all y, count in Y_update do
13:        deg(y) ← deg(y) − count
14:    return Y_update
```

Analysis. In each complete β-core peeling, all vertices are peeled off exactly once. Since we traverse the neighbor of each vertex in PAR-DEL-UPDATE once, the total work contributed by the parallel peeling procedure PAR-DEL-UPDATE in one complete β-core peeling is $O(\sum_{x \in V \text{ or } U} \deg(i)) = O(m)$. This is the same work required for the sequential Algorithm 1. Thus, PAR-DEL-UPDATE is work-efficient.

The span of PAR-DEL-UPDATE is bounded by $O(\log(n))$. Lines 5–9 have span $O(1)$ due to arbitrary forking. PREFIX-SUM, FILTER, HISTOGRAM all have span bounded by $O(\log(n))$. Therefore, the overall span is $O(\log(n))$.

5.3 Parallel Bi-core Decomposition

Algorithm 4 Parallel bi-core decomposition

```
1: procedure PAR-BI-CORE(G)
2:     parfor α′ = 1 to δ do
3:         PAR-PEEL-FIX-α(G, α′)
4:     parfor β′ = 1 to δ do
5:         PAR-PEEL-FIX-β(G, β′)
6: procedure PAR-PEEL-FIX-α(G, α′)
7:     PAR-DEL-UPDATE({u if deg(u) < α′})                    ▷ Peel U from 1 to α′ − 1
8:     buckets ← BUCKET(V, G)                                ▷ Store vertices in a bucketing structure by degree
9:     while buckets ≠ ∅ do
10:        V_del, β ← buckets.NEXT-BUCKET(β)                 ▷ Extract the next set of vertices with minimum degree
11:        parfor all v in V_del do
12:            parfor i = 1 to β do
13:                α_max i(v) ← max(α_max i(v), α′)          ▷ Update α_max i(v)
14:        U_updated ← PAR-DEL-UPDATE(V_del)                 ▷ Peel V up to β
15:        U_del ← FILTER(U_updated, deg(u) < α′)
16:        parfor all u in U_del do
17:            β_max α′(u) ← max(β_max α′(u), β)             ▷ Update β_max α′(u)
18:        V_updated ← PAR-DEL-UPDATE(U_del)                 ▷ Remove peeled u
19:        buckets.UPDATE-VERTICES(V_updated)                ▷ Update vertices with changed degrees in the bucketing structure
20: procedure PAR-PEEL-FIX-β(G, β′)
21:     symmetric to PAR-PEEL-FIX-α
```

The general structure of Algorithm 4 is similar to that of Algorithm 1, but the core components–DEL-UPDATE,FIND-MIN–are replaced with parallel algorithms.

On Line 7 of PAR-PEEL-FIX-α, we peel off all $u \in U$ with degree less than α'. On Line 8, BUCKET returns a bucket representation of V as implemented by Dhulipala *et al.* [18]. We call NEXT-BUCKET on buckets on Line 10 to obtain the next nonempty bucket, storing it into V_{del} while also updating the β value. V_{del} records all v with induced degree $\deg(v) \leq \beta$; note that for all $v \in$ delV, $v \in (\alpha', \beta)$-core but $v \notin (\alpha', \beta + 1)$-core. On Lines 11–13, we update the $\alpha_{\max\ \beta'}$

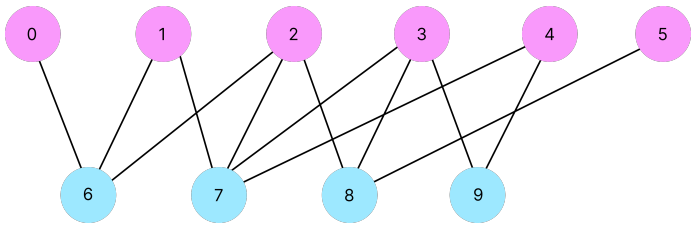


Fig. 1. Example Graph

values similar to Algorithm 1, but in parallel. Note that there is no determinacy race between different branches of the parallel for-loops in each β-core peeling; this is because each branch spawned on Lines 11–12 correspond to a different (β, v) pair. Therefore, if we keep a copy of $\alpha_{\max\ \beta}$ and $\beta_{\max\ \alpha}$ for each thread, Line 13 do not incur determinacy race issues. On Line 14, we peel off all vertices in the current bucket, V_{del}. Lines 16–17 updates $\beta_{\max\ \alpha'}$ values in the same way as 1; by similar arguments, this update does not incur determinacy issues either. Then, Line 18 calls DEL-UPDATE to peel off all vertices stored in U_{del}. Finally, on Line 19, we update the degrees of vertices in V_{update}, which consist of all vertices $v \in V$ whose degree is affected by peeling off U_{del}; UPDATE-VERTEX moves $v \in V_{\text{update}}$ to new buckets corresponding to their new degrees.

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

Analysis. PAR-PEEL-FIX-α has work complexity $O(m)$. This is because PAR-DEL-UPDATE, NEXT-BUCKET, and UPDATE-VERTICES [18] all have overall work across all iterations of the while loop bounded by $O(m)$.

Each iteration of the while loop on Line 9 has span $O(\log(n))$ because FILTER, PAR-DEL-UPDATE, and UPDATE-VERTICES [18] all have span bounded by $O(\log(n))$. Thus, excluding NEXT-BUCKET from consideration, PAR-PEEL-FIX-β attains a span of $O(\rho_\beta \log(n))$. ρ_β is the number of rounds of peeling executed by PAR-PEEL-FIX-β given the specific β value. NEXT-BUCKET does not change the span complexity because its span across all iterations of the while loop is also bounded by $O(\rho_\beta \log(n))$.

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 baseline sequential bi-core decomposition algorithm 1. The baseline algorithm introduced by Liu *et al.* performs a complete α-core peeling (from $\alpha = 1$ to $\alpha = \text{dmax}_u$) for each $1 \le \beta' \le \delta$. Then, it performs β-core peeling (from $\beta = 1$ to $\beta = \text{dmax}_v$) for each $1 \le \alpha' \le \delta$. We observe that, in the process of peeling, all (α, β)-cores with $1 \le \alpha \le \delta$ and $1 \le \beta \le \delta$ are peeled twice, once when we perform α-core peeling for different β values and another time when we perform β-core peeling for different α values.

To avoid repetition, we can modify Algorithm 4 such that each PAR-PEEL-FIX-$\alpha(G, \alpha')$ starts β-core peeling 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 $\alpha' \le \beta \le \text{dmax}_v$ as opposed to $1 \le \beta \le \text{dmax}_v$ as used in Algorithm 4 and 1.

We illustrate the optimization with an example. Consider a graph as shown in figure 1, we visualize its peeling space in figure 3.

Each position in the grid of figure 3 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 the indices

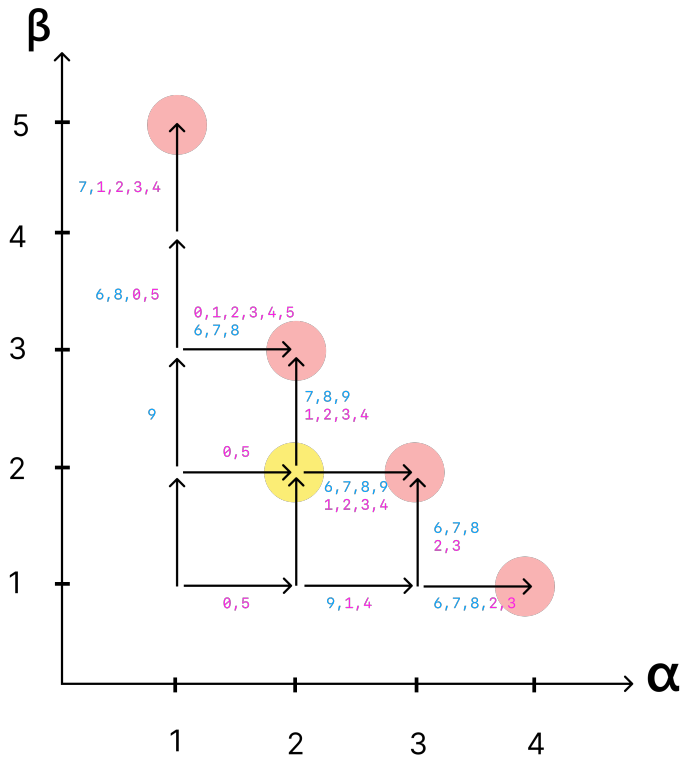


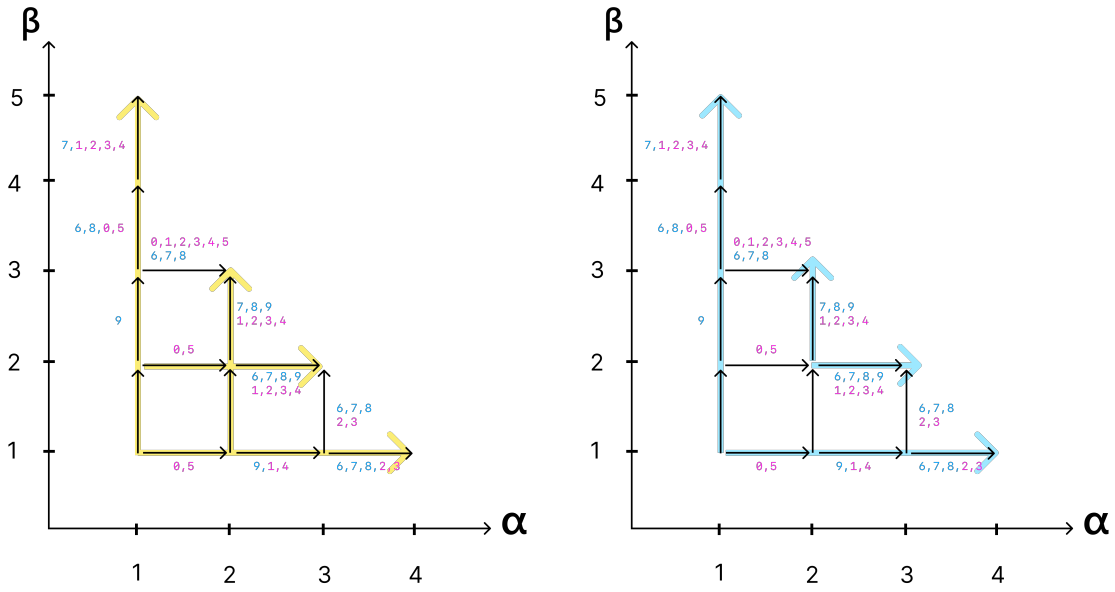
Fig. 2. Peeling Space

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

The peeling operations performed by algorithm 4 can be visualized by the yellow peeling paths in figure 3. 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 β-core 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 peeling paths in figure 3.

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 correspond visually to the part of peeling space to the right of the diagonal (x, x)-cores; part B correspond instead to the section above the diagonal (x, x)-cores. Thus, the optimized algorithm's α-core peeling peels in part A of the peeling space; the β-core peeling peels 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, then $\alpha_{\max\ \beta}(v)$ is recorded correctly to be α. This is true for all $1 \leq \beta \leq \delta$ and $v \in (\beta, \beta)$-core.



(a) Algorithm 4's Peeling Path (b) Optimized Peeling Path

Fig. 3. Unoptimized vs Optimized Peeling Paths

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 performing β-core peeling 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 $\alpha_{\max}{}_{\beta}(v)$ values performed on Lines 11–13 of Algorithm 4 ensures the $\alpha_{\max}{}_{\beta}(v)$ value recorded is α', the correct value. This is valid for all v and β such that $\alpha_{\max}{}_{\beta}(v) < \beta$ and so all $\alpha_{\max}{}_{\beta}(v)$ values satisfying the conditions is correct.

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

5.5 Implementation and Other Optimizations

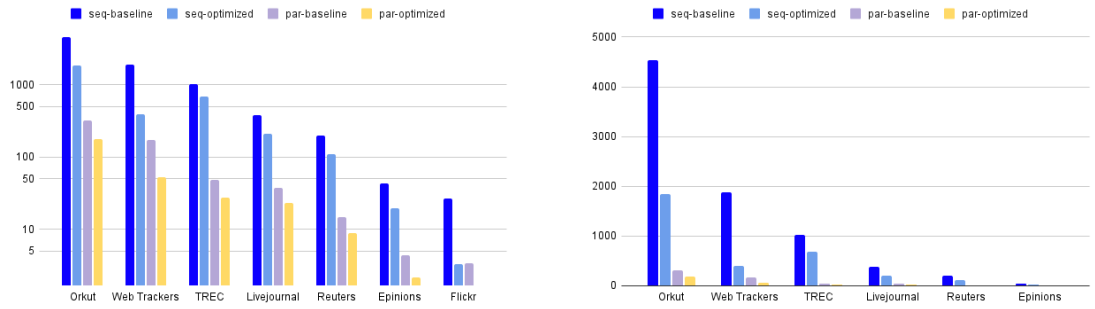
Directly implementation of our theoretically efficient Algorithm 4 with or without the optimization mentioned above is practically inefficient. On machines with only 30-cores and 60 vCPUs, the significant parallelism of Algorithm 4 is unnecessary and only incur additional overhead due to the scheduler and the aggregated peeling used. To implement a practically fast bi-core decomposition algorithm, we forgo the parallel aggregated-peeling and only parallelize between different α-core peelings and β-core peelings. We find that parallelism internal to each complete α-core or β-core peeling is superfluous. We also introduce several optimizations to speed up our parallel algorithm.

(1) Lazy Bucket Instantiation Instead of keeping track of all the buckets and vertices within, we only instantiate the next 16 buckets. This technique was used by Dhulipala *et al.* for implementing k-core decomposition [18].

(2) Load Balancing When distributing different α-core, β-core peelings to different threads, we employ a smart load-balancing based on the observations that α-core peelings with higher β values take less time to complete (and similar for β-core peelings).

Graph Name	Type	$\|U\|$	$\|V\|$	n	m	dmax	δ	$\rho_{\max}$
Orkut	Membership	2.78M	8.73M	11.51M	327M	318K	466	12100
Web Trackers	Inclusion	27.7M	284K	40.43M	140.6M	11.57M	437	4542
LiveJournal	Membership S	3.20M	7.49M	13.89M	112M	1.05M	108	6831
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. Graphs Statistics



(a) runtimes (log-scale) (b) runtimes

Fig. 4. Peeling runtime (in seconds) for different algorithms

6 EXPERIMENTS

We provide in this section a comprehensive evaluation of our implementation of our parallel bi-core decomposition algorithm.

We used the KONECT graph database [27], as shown in Table 2. We use Google Cloud Platform `c2-standard-60` instances for all our experiments, which run on 30 cores Intel 3.1 GHz Cascade Lake processors with two-way hyper-threading and 240 GB of memory; the processors have max turbo clock-speed of 3.8 GHz.

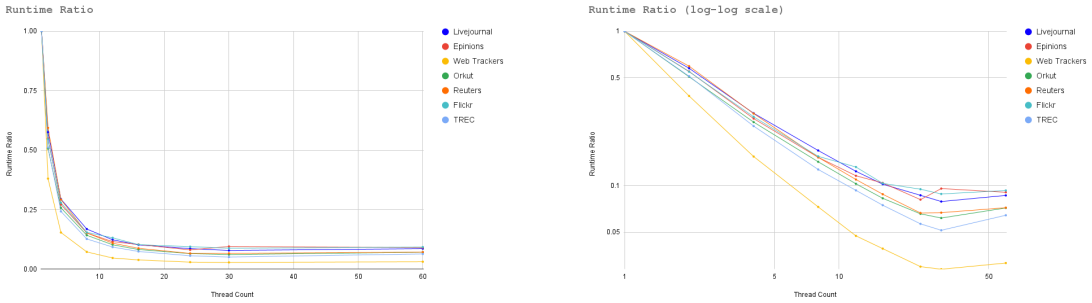
Graphs. The graphs we used (shown in Table 2) are Orkut [4], Web Trackers [7], LiveJournal [3], epinions [1], TREC [6], flickr [2], and reuters [5].

Algorithms. We benchmark the following 4 algorithms. We benchmark parallel algorithms using 1, 2, 4, 8, 12, 16, 24, 30, 60 threads. All code is written in C++ with -O3 optimization level.

(1) SEQ-BASELINE: Algorithm 1 as described by Liu *et al.*
(2) SEQ-OPTIMIZED: Algorithm 1 but with peeling space pruning optimization introduced in section 5.4
(3) PAR-BASELINE: Algorithm 4
(4) PAR-OPTIMIZED: Algorithm 4 with peeling space pruning optimization

Comparison of Performances of Algorithms.

Figure 4 shows the runtimes of our algorithms for these graphs. The parallel algorithms are run with 24 threads (each thread is a virtual hyperthread, so this correspond to a 12-core machine).



(a) Ratio of Runtimes Relative to Serial Time

(b) Logarithmic Scale

Fig. 5. Normalized Runtimes vs Number of Threads

First, note that SEQ-BASELINE is an accurate reproduction of Liu *et al.*'s sequential algorithm, the code of which we have not obtain at the time of writing this paper. On Orkut, our SEQ-BASELINE reports a runtime of 4538 seconds on a 3.1GHz CPU compared to Liu *et al.*'s 4103 seconds runtime on a 3.4GHz CPU. Orkut is the only graph for which Liu *et al.* reported a concrete numerical runtime.

We observe that PAR-BASELINE consistently outperforms the sequential baselines by large margins with 7.8-14.4x speedups over SEQ-BASELINE running on 12 cores and 24 vCPUs. This outperforms the speedup achieved by Liu *et al.*'s parallel algorithm by large margins. For example, Liu *et al.*'s parallel algorithm achieves 5.6x speedup over their sequential baseline on Orkut running on a 12-core machine with 3.4GHz CPU clock speed. In comparison, our parallel algorithm achieves 14.4x speedup over SEQ-BASELINE. On Orkut, PAR-BASELINE reports a runtime of 316 seconds with 3.1GHz CPUs compared to Liu *et al*'s reported 732 seconds with 3.4GHz CPUs. This proves that optimizations such as parallel load-balancing and lazily instantiated bucketing structure improves performance.

The peeling space pruning optimization strategy we introduced also consistently improves the runtimes of the sequential baseline by 2.1-2.8x. Comparing PAR-BASELINE to PAR-OPTIMIZED, the optimization improves runtimes by 1.6-3.2x, demonstrating the effectiveness of the optimization across sequential and parallel peeling.

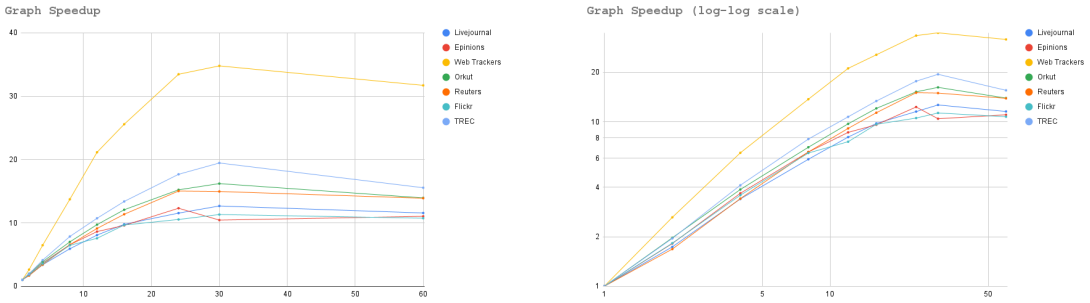
Overall, comparing PAR-OPTIMIZED to SEQ-BASELINE, our parallel algorithm attains 16.2-35.5x speedup over the sequential baseline. PAR-OPTIMIZED takes 178 seconds to complete bi-core decomposition for Orkut compared to the 731 seconds provided by Liu *et al.*, which we again emphasize is on different and potentially faster hardware. If we disregard the hardware difference, our parallel algorithm achieves 4.1x speedup over their parallel algorithm.

Analysis of Scalability.

Figure 5 and Figure 6 illustrates the parallel speedup achieved by our algorithm for different graphs. PAR-OPTIMIZED achieves consistent speedup across different graphs, showing that PAR-OPTIMIZED is robust across graphs of different sizes and structures.

Furthermore, PAR-OPTIMIZED demonstrates good scalability across different number of threads. Since the machine we use have only 30 cores and the 60 threads are just vCPUs not actual additional cores, the parallel speedup plateau when moving from 30-threads to 60-threads is, to a certain degree, expected. The scheduling overhead of PAR-OPTIMIZED also contributes to the speedup plateau.

Analysis of ρ vs runtime.



(a) Speedup Ratios for Tested Graphs (b) Logarithmic scale (x and y-axis)

Fig. 6. Speedup vs Number of Threads

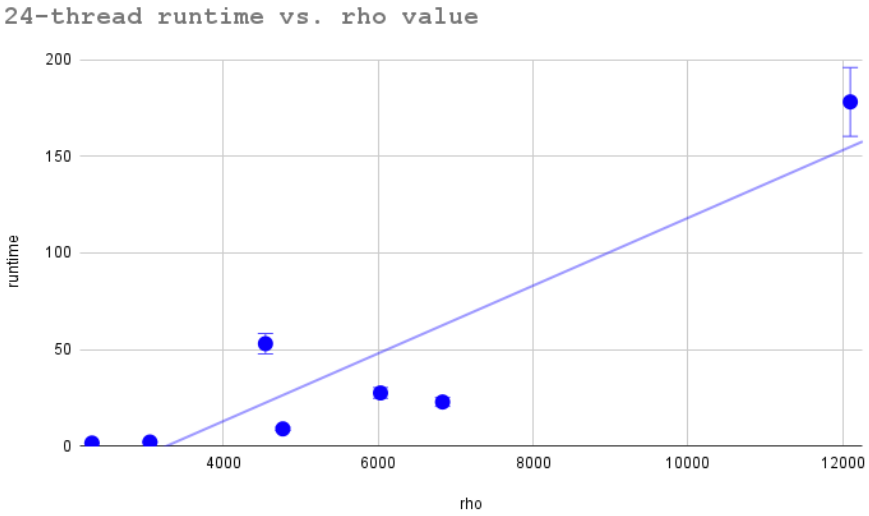


Fig. 7. Rho value vs. Runtimes

The span of our algorithm is $O(\rho \log(n))$ *w.h.p.* Theoretically, ρ is bounded by $O(\max(\text{dmax}_u, \text{dmax}_v)) = O(n)$. Therefore, it is theoretically reasonable to assume that ρ is the dominant term in the span. That is empirically true. For graphs given in Table 2, the ρ values are significantly larger than $\log_2(n)$ values. Thus, it is natural to assume there is an approximately linear correlation between the runtimes of PAR-OPTIMIZED and the ρ values. Indeed, figure 7 experimentally demonstrates a near-linear correlation between the ρ values and the runtimes. This demonstrates that the bi-core peeling complexity we introduced is an important value in both theoretical and empirical evaluation of parallel bi-core decomposition algorithms.

7 CONCLUSION

In this paper, we study various parallel algorithms for bi-core decomposition, which is an important problem to focus on. We also develop a shared-memory work-efficient parallel bi-core decomposition algorithm with strong span bounds. In addition, we provide the pseudo-code for the algorithm and derive its complexity bounds. Finally, our experiments on various bipartite networks prove the improved performance of our peeling algorithms, showing that it is both performant and scalable for larger graphs.

REFERENCES

[1] 2017. Epinions Network Dataset – KONECT. http://konect.cc/networks/epinions

[2] 2017. Flickr – KONECT. http://konect.cc/networks/flickr-groupmemberships/

[3] 2017. LiveJournal Network Dataset– KONECT. http://konect.cc/networks/livejournal

[4] 2017. Orkut Network Dataset – KONECT. http://konect.cc/networks/orkut-groupmemberships

[5] 2017. Reuters network dataset – KONECT. http://konect.cc/networks/reuters

[6] 2017. TREC Network Dataset – KONECT. http://konect.cc/networks/gottron-trec

[7] 2017. Web Trackers Dataset – KONECT. http://konect.cc/networks/webtrackers

[8] 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

[9] 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

[10] 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

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

[12] 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

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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

[20] 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

[21] 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

[22] 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

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

[24] 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.

[25] 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.

[26] 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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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

[39] 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

[40] 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.

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

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

[43] 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

[44] 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

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

[46] 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

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

[48] 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

[49] 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

[50] 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).

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

[52] 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

[53] 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

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