Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Efficient Algorithms for Parallel Bi-core Decomposition

Yihao Huang *† Claire Wang *† Jessica Shi ‡ Julian Shun ‡

Abstract

We present new shared-memory parallel algorithms for the bi-core decomposition problem, which discovers dense subgraphs in bipartite graphs and is the bipartite analogue of the classic k-core decomposition problem. We develop a theoretically-efficient parallel bi-core decomposition algorithm that discovers a hierarchy by peeling vertices from the graph in parallel. Our algorithm improves the span (parallel running time) over the state-of-the-art parallel bi-core decomposition algorithm, while matching the state-of-the-art sequential algorithm in work. We additionally prove the bi-core decomposition problem to be P-complete, meaning that a polylogarithmic span solution is unlikely under standard assumptions. We also devise a theoretically-efficient parallel bi-core index structure to allow for fast parallel queries of vertices in given cores.

Finally, we propose a novel practical optimization that prunes unnecessary computations, and we provide optimized parallel implementations of our bi-core decomposition algorithms that are scalable and fast. Using 30 cores with two-way hyper-threading, our implementation achieves up to a 4.9x speedup over the state-of-the-art parallel algorithm. Our parallel index structure can be constructed up to 27.7x faster than the state-of-the-art sequential counterpart. Due to the improved storage format of our index structure, our parallel queries are up to 116.3x faster than the state-of-the-art sequential queries.

1 Introduction

The problem of discovering densely connected subgraphs in networks is fundamental for large-scale graph analysis. It has applications in community search [24], clustering word-documents [10], improving advertising [14], detecting fraudsters or spammers [47], and analyzing protein-gene-disease relations in bioinformatics [31]. Classic algorithms for dense subgraph discovery include k-core decomposition [27, 29], k-truss [7], and nucleus decomposition [36]. However, these algorithms are designed towards general graphs, and do not take into account the specific structure of bipartite graphs.

A bipartite graph G consists of two bipartitions of vertices, U and V, where every edge connects a vertex in U to a vertex in V. These graphs model the affiliation between two distinct types of entities, such as in authorship networks [20], group membership networks [37], user-product networks [44], and protein-protein interactions [15]. Directly applying traditional dense substructure analysis techniques designed for general graphs to bipartite graphs does not allow for the bipartitions to be distinguished from one another, which can be important especially if they exhibit different structures. Another approach to discovering dense substructures in bipartite graphs uses graph projection to represent each bipartition as its own graph, and then applies traditional techniques to analyze each of the two resulting graphs; however, such methods still fail to capture important connectivity information and can cause an explosion in the number of edges, making them practically inefficient [34]. Thus, bipartite analogues for classic dense subgraph discovery algorithms are crucial for efficient and accurate dense substructure analysis on bipartite graphs.

Indeed, developing algorithms to apply specifically to bipartite graphs has become a recent popular direction of research [45, 1, 50, 34]. The focus of this work is on the bipartite equivalent of a k-core, known as a bi-core, which was introduced by Ahmed *et al.* [1]. Formally, an (α, β)-core (or a bi-core) is the maximal subgraph where the induced degree of each vertex in the first partition is at least α, and the induced degree of each vertex in the second partition is at least β. The bi-core decomposition has applications in a variety of fields, including spam detection on social networks [4], community search on bipartite networks [46], movie viewership analysis [1], and group recommendation [13]. For example, Beutel *et al.* [4] leveraged the bi-core decomposition to detect spammers on user-post social networks, where spammers and fake accounts often form dense subgraphs by interacting with each others' posts, and Wang *et al.* [46] used the bi-core decomposition as a subroutine for optimizing community search on bipartite networks, by reducing the search space for dense communities.

Parallel Bi-core Decomposition. Liu *et al.* [28] propose the current state-of-the-art sequential bi-core decomposition algorithm, which computes the (α, β)-core via multiple graph peeling processes. Their sequential algorithm takes $O(\delta m)$ time and $O(m)$ space, where m is the number of edges and δ denotes the degeneracy of the graph. Liu *et al.* also introduce

*Phillips Academy, Andover, MA

†This work represents an equal contribution between these authors.

‡MIT CSAIL, Cambridge, MA

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

a parallel algorithm; however, their parallel algorithm only parallelizes across different peeling processes and does not parallelize the peeling process itself. As a result, it has long sequential dependencies, which limits its parallel scalability. As the sizes of graphs increase, a bi-core decomposition algorithm with high parallelism and scalability becomes crucial.

We develop an efficient shared-memory parallel bi-core decomposition algorithm that parallelizes the peeling process. In each round of peeling, it removes all vertices with the lowest induced degree from the graph in parallel until the graph is empty. We use the classic *work-span* model to analyze the theoretical complexity of our parallel algorithm, where the *work* is the total number of operations, and the *span* (or the *depth*) is the length of the longest chain of sequential dependencies. We prove that for a graph with m edges our algorithm achieves $O(\delta m)$ work, which matches the best sequential time complexity. Our algorithm achieves $O(\rho \log n)$ span *w.h.p.*,[1] where n is the number of vertices, and ρ denotes the bi-core peeling complexity, which we define as the maximum number of rounds of peeling required to remove all vertices from the graph. Our algorithm uses $O(m)$ space.

Note that ρ is upper bounded by n, so our span is $O(\rho \log n) = O(n \log n)$ *w.h.p.* Also, the parallel algorithm introduced by Liu *et al.* has a span of $O(m)$, so our parallel algorithm improves upon Liu *et al.*'s algorithm when the number of edges m is $\Omega(\rho \log n)$. Moreover, on real world graphs, we find that $\rho \log n$ is generally 2–3 orders of magnitude smaller than m. We also prove the problem of bi-core decomposition to be P-complete by showing a reduction from the k-core problem to the bi-core decomposition problem. It is as therefore unlikely that there exists a parallel bi-core decomposition algorithm with polylogarithmic span under standard assumptions.

In addition, we develop a parallel index structure, which is an extension of Liu *et al.*'s sequential index structure. The index structure allows for efficient queries of all vertices $x \in (\alpha, \beta)$-core, for a given α and β, in work linear to the size of the core. Our parallel index structure is able to achieve this with $O(1)$ span. We also introduce an algorithm to construct our index structure in parallel given the bi-core numbers of each vertex in $O(m)$ work and $O(\log n)$ span *w.h.p.*

Finally, we introduce a practical heuristic that optimizes bi-core peeling by pruning the peeling space of the algorithm. We implement our algorithms and present a comprehensive evaluation on real-world graphs with up to hundreds of millions of edges. We compare our algorithms against Liu *et al.*'s parallel and sequential algorithms, which we use as our baselines. Our parallel bi-core decomposition algorithm achieves up to a 51.4x speedup (average 27.9x) over Liu *et al.*'s sequential algorithm on a machine with 30 cores and two-way hyperthreading. Furthermore, it achieves up to a 4.9x speedup (average 2.3x) over their parallel algorithm. Our parallel index construction algorithm attains up to a 27.7x speedup (average 18.4x) over Liu *et al.*'s sequential index construction algorithm, and our parallel index query achieves up to a 116.3x speedup (average 43.8x) over Liu *et al.*'s sequential 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.
2. We prove that the problem of bi-core decomposition is P-complete.
3. 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`.

The remainder of the paper is organized as follows. In Section 2, we present related work. In Section 3, we introduce notation and definitions. In Section 4, we introduce our shared-memory parallel bi-core decomposition algorithm and discuss the corresponding practical optimizations. In Section 5, we prove the P-completeness of the bi-core decomposition problem, and in Section 6, we introduce our parallel index construction and query algorithm. In Section 7, we present additional practical optimizations and our experimental results.

2 Related Work

k-core Decomposition. The bi-core decomposition problem is an extension of the well-studied k-core decomposition problem; for each vertex v, the k-core decomposition problem asks for the largest integer k such that v is contained within a subgraph where all vertices have induced degree at least k. The first efficient sequential algorithm for k-core decomposition was given by Matula and Beck [29], and there has been much work on parallelizations in both the distributed-memory and shared-memory settings [9, 21, 30, 41, 11].

Other Dense Subgraph Decompositions. k-clique decomposition, k-truss, and (r, s)-nucleus decomposition are all extensions of k-core decomposition that use higher order substructures to discover dense substructures in a graph. k-

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

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Notation	Definition
G	An undirected, simple, bipartite graph,
U	One bipartition of the vertices in G,
V	Another bipartition of the vertices in G,
x	A vertex in $U \cup V$,
u	A vertex in U,
v	A vertex in V,
$\deg(x)$	Degree of vertex x,
$\deg_{\text{in}}(x)$	Induced degree of vertex x considering only unpeeled vertices,
$N(x)$	A list of vertex x's neighbors,
$N_{\text{in}}(x)$	A list of vertex x's induced neighbors considering only unpeeled vertices,
dmax_v	The maximum vertex degree in V,
dmax_u	The maximum vertex degree in U,
$\alpha_{\max\ \beta}(v)$	The maximum α value for a given $v \in V$ and β such that $v \in (\alpha, \beta)$-core,
$\beta_{\max\ \alpha}(u)$	The maximum β value for a given $u \in U$ and α such that $u \in (\alpha, \beta)$-core,
$\max_\alpha(\beta)$	The maximum α value such that (α, β)-core is nonempty for fixed β,
$\max_\beta(\alpha)$	The maximum β value such that (α, β)-core is nonempty for fixed α, and
δ	The maximum δ value such that (δ, δ)-core is nonempty, or the degeneracy.

Table 1: Summary of graph notation.

clique decomposition [42, 38] 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 [7, 49, 48, 43, 22, 35, 2] 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 decomposition problems are part of the MIT GraphChallenge [17], demonstrating their practical importance and popularity. The (r, s)-nucleus decomposition [36, 35, 39] further generalizes the k-clique and k-truss decompositions, by asking for the largest c 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 c induced s-cliques. Notably, k-core decomposition is $(1, 2)$-nucleus decomposition, 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 work has focused on generalizing these decomposition algorithms to bipartite graphs by focusing on other higher-order structures in bipartite graphs. Zou [50] and Sarıyüce and Pinar [34] 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 [33, 50, 34, 43, 45] and parallel [40, 26] 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 [13]. More recently, 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 membership queries from vertices [28]. 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 [46].

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 $N(x)$ denote the set of neighbors of vertex x, and let $\deg(x)$ denote the degree of vertex x. In the context of the peeling process, we often discuss a subgraph of G consisting of all unpeeled vertices. In that case, we use $N_{\text{in}}(x)$ to refer to the induced neighbors of x and $\deg_{\text{in}}(x)$ to refer to the induced degree of x in the subgraph. We let dmax_v be the maximum degree of all vertices in V, and dmax_u is symmetrically defined

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

for U. We define a bi-core as follows:

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 $\deg_{in}(u) \geq \alpha$, and for every $v \in V'$, the induced degree $\deg_{in}(v) \geq \beta$.*

We define $\max_\alpha(\beta)$ to be the maximum α value, given a value β, such that the (α, β)-core is nonempty. Symmetrically, $\max_\beta(\alpha)$ is defined to be the maximum β value, given a value α, such that the (α, β)-core is nonempty. The degeneracy of the graph, δ, can be equivalently defined as the maximum δ such that the (δ, δ)-core is nonempty. Note that δ is upper bounded by $O(\sqrt{m})$ [27].

See Table 1 for a summary of these notations.

We note two additional facts:

1. if $x \in (\alpha_1, \beta_1)$-core, then $x \in (\alpha_2, \beta_2)$-core if $\alpha_2 \leq \alpha_1$ and $\beta_2 \leq \beta_1$ [28].
2. Every nonempty (α, β)-core must have $\alpha \leq \delta$ and/or $\beta \leq \delta$ [28].

Problem Statement. Now, we formally define the bi-core decomposition problem.

DEFINITION 2. *For a vertex $v \in V$ and fixed β, we define $\alpha_{\max\ \beta}(v)$ to be the maximum α such that $v \in (\alpha, \beta)$-core. Symmetrically, for a vertex $u \in U$ and fixed α, we define $\beta_{\max\ \alpha}(u)$ to be the maximum β such that $u \in (\alpha, \beta)$-core.*

We call the set of all $\alpha_{\max\ \beta}(v)$ and $\beta_{\max\ \alpha}(u)$ values the *bi-core numbers*. The *bi-core decomposition* problem is to compute $\beta_{\max\ \alpha}(u)$ for every $u \in U$ and every $\alpha \in [1, \text{dmax}_u]$, and symmetrically, $\alpha_{\max\ \beta}(v)$ for every $v \in V$ and every $\beta \in [1, \text{dmax}_v]$.

Note that with the bi-core numbers, we can easily determine whether any $u \in U$ is in the (α, β)-core for any α and β. If $\beta_{\max\ \alpha}(u) \geq \beta$, then $u \in (\alpha, \beta)$-core. Similarly, for any $v \in V$, 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, and in particular, we use the classic work-span model for our analysis, where the work of an algorithm is the total number of operations executed, and the span is the length of the longest dependency path [8]. Given work T_1 and span T_∞, the algorithm's running time on P processors T_P can be bounded by $T_P \leq T_1/P + O(T_\infty)$ using a work-stealing scheduler [6]. We assume arbitrary forking, where forking n processes has a span of $O(1)$. We show that our algorithms are *work-efficient*, meaning that they have the same work complexity as the best sequential algorithm for the same problem.

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

PREFIX-SUM(A) takes as input a sequence of length n and returns a sequence B of the same length such that $B[i] = A[0] \oplus \cdots \oplus A[i-1]$, where $\oplus$ is a binary associative operator. We assume that the operator is addition unless stated otherwise. The PREFIX-SUM operation also returns the total sum. It takes $O(n)$ work and $O(\log n)$ span [8].

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

FILTER(A, COND) takes as input a sequence of length n and a condition. It retains all elements such that the condition is true and outputs these elements in a sequence (in the original order). It takes $O(n)$ work and $O(\log n)$ span [8].

WRITE-MAX(a, b) takes as input a variable a and a value b. It atomically reads a, and if a's value is less than b, it then updates a's value to b. If the update is performed successfully, the function returns true, and otherwise, it returns false. We assume that this takes $O(1)$ work and span.

HISTOGRAM(A) takes as input a sequence of n 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.* [18].

RADIX-SORT(A) sorts a sequence of n integers. It takes $O(n)$ work and $O(\log n)$ span *w.h.p.* given that the range of the integers is bounded by $O(n \log^{O(1)} n)$ [32].

4 Parallel Bi-core Decomposition

In this section, we introduce our parallel bi-core decomposition algorithm, which is inspired by Liu *et al.*'s sequential algorithm [28]. The goal of the algorithm is to compute the $\alpha_{\max\ \beta}(v)$ values for every β and vertex $v \in V$, and to compute the $\beta_{\max\ \alpha}(u)$ values for every α and vertex $u \in U$. Note that $\alpha_{\max\ \beta}(v) = \alpha$ if $v \in (\alpha, \beta)$-core but $v \notin (\alpha + 1, \beta)$-core. Symmetrically, $\beta_{\max\ \alpha}(u) = \beta$ if $u \in (\alpha, \beta)$-core but $u \notin (\alpha, \beta + 1)$-core. Thus, a peeling-based subroutine is often used to solve this problem, to discover successive cores [1, 13].

4.1 Background Liu *et al.*'s [28] algorithm computes $\alpha_{\max\ \beta}(v)$ and $\beta_{\max\ \alpha}(u)$ values by calling a peeling subroutine PEEL-FIX-β for every $\beta' \in [1, \delta]$, where δ is the maximum k-core number of the graph. They also perform a symmetric subroutine PEEL-FIX-α for every $\alpha' \in [1, \delta]$. Due to the symmetry of these two subroutines, we only discuss PEEL-FIX-β.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

PEEL-FIX-β takes as input the fixed β' value, and increases the α value of the (α, β')-core from 1 to $\max_\alpha(\beta')$ while iteratively peeling 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 peeling a vertex v to discover the $(\alpha + 1, \beta')$-core, they update $\alpha_{\max\ \beta'}(v) \leftarrow \alpha$, because it is the highest α value for which $v \in (\alpha, \beta')$-core. Similarly, if they discover that $u \in (\alpha, \beta')$ but $u \notin (\alpha, \beta' + 1)$, they record β' as the value of $\beta_{\max\ \alpha}(u)$. A symmetric peeling subroutine PEEL-FIX-α is called for every $\alpha' \in [1, \delta]$. Liu *et al.* prove that these peeling subroutines correctly compute all $\alpha_{\max\ \beta}$ values and $\beta_{\max\ \alpha}$ values.

4.2 Parallel Bi-core Decomposition Algorithm The sequential nature of Liu *et al.*'s [28] algorithm limits its practical applicability to large graphs. While Liu *et al.* [28] also provide a parallel version of their algorithm, their parallel algorithm only parallelizes across rounds of peeling (calls to subroutines PEEL-FIX-α and PEEL-FIX-β), and does not parallelize the peeling process itself. As a result, their parallel algorithm has a high span of $O(m)$. We present in this section a parallel bi-core decomposition algorithm, and we prove that our algorithm is work-efficient and has span $O(\rho \log n)$ *w.h.p.*, where ρ is the peeling complexity, or the maximum number of rounds needed to empty the graph in any of the peeling subroutines, where in each round, we peel all vertices with the minimum induced degree.

Our parallel algorithm is also based on a peeling paradigm, and in particular, we parallelize the subroutines PEEL-FIX-α and PEEL-FIX-β, which we call PAR-PEEL-FIX-α and PAR-PEEL-FIX-β respectively. Our parallel peeling subroutines compute the exact same $\alpha_{\max \beta}$ and $\beta_{\max \alpha}$ values as Liu *et al.*'s sequential peeling subroutine, so as a result, the correctness of our algorithm follows from the correctness of Liu *et al.*'s algorithm. Because the two subroutines are symmetric, we only discuss PAR-PEEL-FIX-β here. PAR-PEEL-FIX-β takes as input a fixed β'. Then, it increases a variable α from 1 to $\max_\alpha(\beta')$ while peeling all vertices $u \in U$ where $u \in (\alpha, \beta')$-core but $u \notin (\alpha + 1, \beta')$-core in parallel in each iteration when α is increased. The order in which these vertices are peeled does not matter. Thus, this parallel peeling step computes the same result as Liu *et al.*'s sequential peeling step, which peels the vertices sequentially. Note that all such vertices $u \in U$ have induced degree satisfying $\deg_{\text{in}}(u) \leq \alpha$ for the current α value. Then, we peel all vertices $v \in V$ where $v \in (\alpha, \beta')$-core but $v \notin (\alpha + 1, \beta')$-core. Every peeled $v \in V$ must have induced degree satisfying $\deg_{\text{in}}(v) < \beta'$, so for each $v \in V$ that is peeled, we update the $\alpha_{\max\ \beta'}(v)$ value to be the current α value. We then update the α value to be the minimum value α' such that $\alpha' \geq \alpha$ and the (α', β')-core is nonempty.

Two challenges are involved in this process:

1. Finding the minimum α' such that there exists vertices $u \in U$ with induced degree $\deg_{\text{in}}(u) \leq \alpha'$, and returning this set of vertices in parallel.
2. Peeling a set of vertices from one partition and updating their neighbors' degrees in parallel.

To search for the minimum α' such that there exists u with $\deg_{\text{in}}(u) \leq \alpha'$ and to query for this set of vertices in parallel, we store all vertices U in a parallel bucketing structure Julienne by Dhulipala *et al.* [11]. Julienne organizes each $u \in U$ into buckets based on its induced degree, where vertices with $\deg_{\text{in}}(u) > \alpha$ are stored in the bucket indexed by $\deg_{\text{in}}(u)$, and vertices with $\deg_{\text{in}}(u) \leq \alpha$ are stored in a single bucket indexed by α. Finding the minimum α' and its corresponding vertices is then equivalent to finding the lowest-indexed nonempty bucket. Julienne supports this operation NEXT-BUCKET(α) $\rightarrow$ (α', $U_{\alpha'}$), where α is the bucket index to begin the search from, α' is the next lowest index corresponding to a nonempty bucket, and $U_{\alpha'}$ is the set of vertices inside the bucket with index α'. The operation has $O(\log n)$ span per query and $O(n)$ work over all queries in a subroutine of PAR-PEEL-FIX-β [11]. Julienne also supports updating the degrees of k vertices in $O(k)$ work and $O(\log n)$ span *w.h.p.* [11].

We now discuss how we resolve the second challenge in more detail. The pseudocode of our parallel bi-core decomposition algorithm is given in Algorithm 1.

First, we discuss the subroutine PAR-DEL-UPDATE, which updates the degrees of vertices after peeling a given set of vertices. PAR-DEL-UPDATE takes as input an array of vertices X_{del} and then peels all of these vertices in parallel. On Lines 7–12, 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 can be constructed in parallel using a parallel PREFIX-SUM on the degrees of the vertices in X_{del}. On Line 12, HISTOGRAM aggregates Y_{update} and returns a sequence of pairs $(y, \textit{count})$. For every y, *count* is the number of its occurrences in Y_{update}. On Lines 13–14, we iterate through each such y and decrease its degree by its corresponding *count*. Note that our pseudocode removes x from G on Line 9 for simplicity, but for our theoretical bounds, it is sufficient to mark removed vertices in a separate array and ignore them when traversing the graph in later iterations.

Now, we discuss the main subroutine, PAR-PEEL-FIX-β. Note that PAR-PEEL-FIX-α is symmetric. We refer to Figure 1 for an example of the computations in PAR-PEEL-FIX-β, with $\beta' = 2$ and starting with the graph in Step 0.

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Algorithm 1 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-DEL-UPDATE(G, X_del)
7:     Y_update ← []                                   ▷ Create an empty array
8:     parfor all x in X_del do
9:         remove x from G
10:        parfor all y in N_in(x) do
11:            add y to Y_update                       ▷ Record y in parallel for degree update
12:    Y_hist ← HISTOGRAM(Y_update)                    ▷ Count number of occurrence of each vertex
13:    parfor all (y, count) in Y_hist do
14:        deg_in(y) ← deg_in(y) − count
15:    return Y_update
16: procedure PAR-PEEL-FIX-β(G, β')
17:    PAR-DEL-UPDATE(G, {v| deg_in(v) < β'})          ▷ Remove all vertices in V with degree < β'
18:    Store vertices in U into buckets                ▷ Construct bucketing structure from vertices in U based on their degrees
19:    while buckets is not empty do
20:        (α, U_del) ← buckets.NEXT-BUCKET(α)         ▷ Extract the next set of vertices with minimum degree
21:        parfor all u in U_del do
22:            parfor i = 1 to α do
23:                WRITE-MAX(β_max i(u), β')           ▷ Update β_max i(u)
24:        V_update ← PAR-DEL-UPDATE(G, U_del)         ▷ Peel vertices u ∈ U with induced degree ≤ α
25:        V_del ← FILTER(V_update, deg_in(v) < β')    ▷ Determine vertices v ∈ V no longer in (α, β')-core
26:        parfor all v in V_del do
27:            WRITE-MAX(α_max β'(v), α)               ▷ Update α_max β'(v)
28:        U_update ← PAR-DEL-UPDATE(G, V_del)         ▷ Remove vertices v ∈ V no longer in (α, β')-core
29:        buckets.UPDATE-VERTICES(U_update)           ▷ Update vertices u ∈ U with changed degrees in the bucketing structure
30: procedure PAR-PEEL-FIX-α(G, α')
31:    symmetric to PAR-PEEL-FIX-β
```

On Line 17 in PAR-PEEL-FIX-β, we peel all $v \in V$ with degree less than β' using the subroutine PAR-DEL-UPDATE. In Figure 1, this removes vertices 0 and 5, resulting in the graph at Step 1. At this point, all remaining vertices are in the $(0, \beta')$-core. On Line 18, we initialize the parallel bucketing structure Julienne over the vertices in U, which we call *buckets*. We call NEXT-BUCKET on *buckets* on Line 20 to obtain the next nonempty bucket of vertices, which we store into U_{del}. We also update the α value. In our example graph in Figure 1, $U_{\text{del}} = \{6, 8, 9\}$ and the new α value is 2. Importantly, U_{del} records all u with induced degree $\deg_{\text{in}}(u) \leq \alpha$. Note that for all $u \in U_{\text{del}}$, $u \in (\alpha, \beta')$-core. Thus, on Lines 21–23, we can update the $\beta_{\max\ \alpha}(u)$ values to β' for all $u \in U_{\text{del}}$, which is done in parallel using a WRITE-MAX.

On Line 24, we peel all vertices in the current bucket, U_{del}. In Figure 1, this results in the graph in Step 2, where $V_{\text{update}} = \{1, 2, 3, 4\}$. On Line 25, we store in V_{del} all $v \in V_{\text{update}}$, where $\deg_{\text{in}}(v) < \beta'$. These vertices are no longer in (α, β')-core and must be removed. In the example, $V_{\text{del}} = \{1, 2, 3, 4\}$. Because these vertices $v \in V_{\text{del}}$ satisfy $v \in (\alpha, \beta')$-core but $v \notin (\alpha + 1, \beta')$-core, we update the $\alpha_{\max\ \beta'}(v)$ values to α for each vertex $v \in V_{\text{del}}$ on Lines 26–27. Then, Line 28 calls PAR-DEL-UPDATE to peel all vertices in V_{del}. Finally, on Line 29, we use the bucketing structure to update the degrees of vertices in U_{update}, which consists of all $u \in U$ whose degree is affected by peeling V_{del}; UPDATE-VERTEX moves $u \in U_{\text{update}}$ to new buckets corresponding to their new degrees. For vertices in U_{update} with an updated degree $< \alpha$, we set their degrees to α, so they are peeled in the next round of peeling. In our example, $U_{\text{update}} = \{7\}$, and the degree of vertex 7 is updated to 2. In the second iteration of the while loop, the remaining vertex 7 in the U partition is removed as $\deg_{\text{in}}(7) = 0 \leq 2 = \alpha$, thus completing the peeling procedure.

Analysis. We now show that PAR-PEEL-FIX-β takes $O(m)$ work. First, across all iterations of the while loop, NEXT-BUCKET takes $O(m)$ work [11]. UPDATE-VERTICES takes $O(1)$ work to update a single vertex, and since each vertex x is updated at most $\deg_{\text{in}}(x)$ times, the overall work of all updates is bounded by $O(\sum_{x \in U \cup V} \deg_{\text{in}}(x)) = O(m)$.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

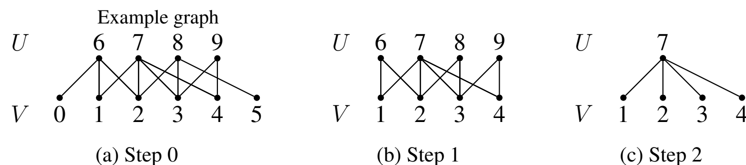


Figure 1: An example peeling process of PAR-PEEL-FIX-β where the input β' is 2. From Step 0 to Step 1, all vertices in V with induced degree < 2 are removed. From Step 1 to Step 2, all vertices in U with degree ≤ 2 are removed.

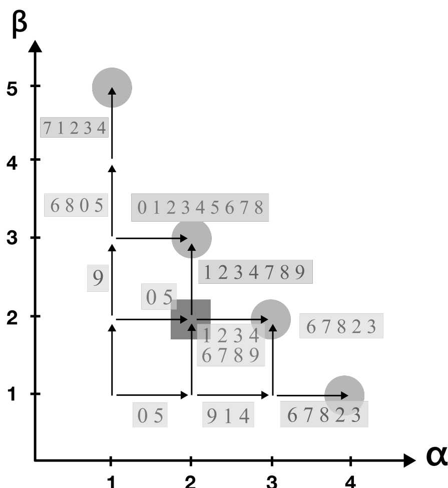


Figure 2: This shows the peeling space of the example graph in Figure 1, and is discussed in more detail in Section 4.3.

Across all calls of 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 of PAR-DEL-UPDATE in one call of PAR-PEEL-FIX-β is $O(m)$.

The work of updating the $\alpha_{\max\ \beta}$ values totals $O(n)$, since each vertex can only appear in V_{del} exactly once. The work of updating the $\beta_{\max\ \alpha}$ values is bounded by $O(m)$ because for each vertex u, the maximal α value for which $\beta_{\max\ \alpha}(u) > 0$ is bounded by $\deg_{\text{in}}(u)$. Thus, for each u, Line 23 is only executed $O(\deg_{\text{in}}(u))$ times, which totals to $O(\sum_{u \in U} \deg_{\text{in}}(u)) = O(m)$. FILTER, over all calls, also takes $O(m)$ work. Thus, PAR-PEEL-FIX-β has work $O(m)$, and overall, PAR-BI-CORE takes $O(\delta m) = O(m^{3/2})$ work.

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 $O(\log n)$ span *w.h.p.* Each iteration of the while loop on Line 19 has span $O(\log n)$ *w.h.p.* because FILTER, PAR-DEL-UPDATE, UPDATE-VERTICES, and NEXT-BUCKET all take $O(\log n)$ span *w.h.p.* The number of iterations of the while loop is ρ by definition. The span of PAR-PEEL-FIX-β is therefore $O(\rho \log n)$ *w.h.p.*, and overall, PAR-BI-CORE has span $O(\rho \log n)$ *w.h.p.*

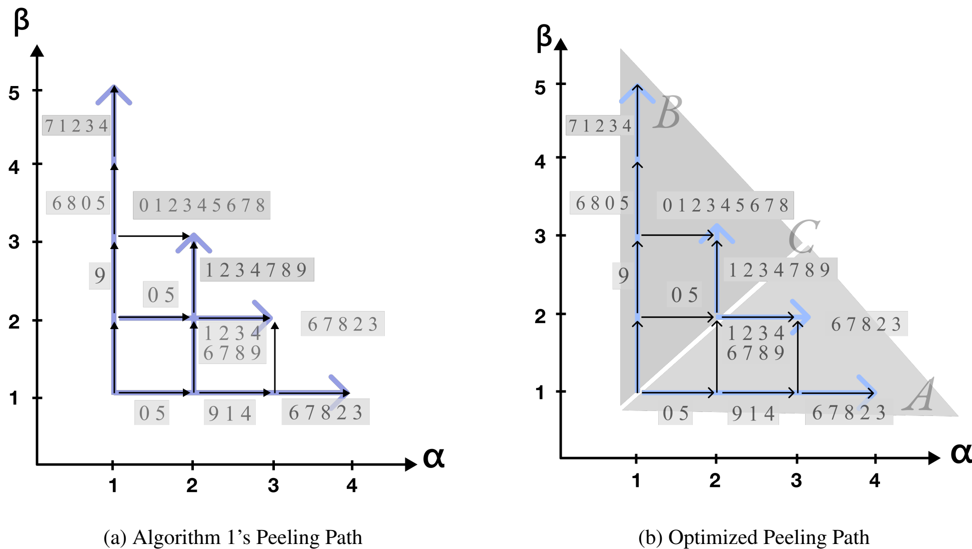
4.3 Peeling Space Pruning Optimization In this section, we introduce a peeling space pruning optimization to our algorithm, which is also applicable to the sequential bi-core decomposition algorithm by Liu *et al.* [28]. Liu *et al.*'s algorithm performs the peeling subroutine for every α, from $\alpha = 1$ to $\alpha = \max_\alpha(\beta')$, for each $1 \leq \beta' \leq \delta$. Then, it also performs the same subroutine for every β, from $\beta = 1$ to $\beta = \max_\beta(\alpha')$, 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 again when we perform peeling along increasing β values for different α'.

To avoid this repetition, we modify Algorithm 1. We will discuss the modification considering increasing α values, but the same can be applied to increasing β values. Instead of peeling from the $(1, \beta')$-core, we modify PAR-PEEL-FIX-$\beta(G, \beta')$ to peel from the (β', β')-core. In other words, the algorithm starts iteratively, increasing the α value from β' to $\max_\alpha(\beta')$ and removing vertices no longer in the current (α, β')-core at the same time. Notably, we confine α to $\beta' \leq \alpha \leq \max_\alpha(\beta')$ as opposed to $1 \leq \beta \leq \max_\alpha(\beta')$ as used in Liu *et al.*'s algorithm and in Algorithm 1.

We illustrate this optimization with an example in Figure 2, using the graph from Figure 1. Each grid intersection in Figure 2 represents an (α, β)-core. Edges represent a single-step peeling operation from the (α, β)-core to the $(\alpha, \beta + 1)$-

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy



(a) Algorithm 1's Peeling Path

(b) Optimized Peeling Path

Figure 3: This figure compares the unoptimized and optimized peeling paths of the example graph in Figure 1. The top figure shows the unoptimized peeling paths, while the bottom figure shows the optimized peeling paths.

core (upward), or to the $(\alpha+1,\beta)$-core (rightward). The labeled numbers on an edge represent the indices of vertices that would be deleted by that specific peeling operation. The circled nodes represent (α,β)-cores that are nonempty, 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 1 can be visualized by the blue highlighted paths in Figure 3a. For $\beta'=1$, we perform α-side peeling from $\alpha=1$ to $\alpha=4$. For $\beta'=2$, we again increase α from 1 to 3 while iteratively removing vertices not within the current bi-core. With the proposed optimization, for $\beta'=2$, we only perform peeling from $\alpha=2$ to $\alpha=3$, starting from the (β',β')-core, or the $(2,2)$-core in this case. This is represented by the blue highlighted peeling paths in Figure 3b.

To show the correctness of the optimized algorithm, we divide the peeling space into three parts: part A where all of the (α,β)-cores satisfy $\alpha>\beta$, part B where all of the (α,β)-cores satisfy $\beta>\alpha$, and part C with the diagonal (x,x)-cores. Note that part A of the peeling space corresponds to the section of the peeling space below the diagonal (x,x)-cores, and part B corresponds 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.

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 parts A and C of the peeling space. For a fixed β, if $v\in(\alpha,\beta)$-core but $v\notin(\alpha+1,\beta)$-core where $\alpha\geq\beta$, then $\alpha_{\max\ \beta}(v)$ is recorded correctly to be α as we perform the peeling process along α values from $\alpha=\beta$ to $\alpha=\max_\alpha(\beta)$. This update is performed on Line 27 of Algorithm 1.

Now, 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 a fixed α' such that $\alpha'=\alpha_{\max\ \beta}(v)$, the algorithm removes v at the (α',β')-core, where $\beta'>\alpha'$ and $\beta'\geq\beta$. Consider the update given by Line 23 in subroutine PAR-PEEL-FIX-β of Algorithm 1. Using the symmetric update in the subroutine PAR-PEEL-FIX-α, we update $\alpha_{\max\ \beta}(v)\leftarrow\max(\alpha_{\max\ \beta}(v),\alpha')$ for all $\beta\in[1,\beta']$. Thus, $\alpha_{\max\ \beta}(v)$ is set to its correct value, α'.

Because parts A, B, and C form the entire peeling space, we have shown that for all β and v, $\alpha_{\max\ \beta}(v)$ is correctly computed. Symmetric arguments apply for all $\beta_{\max\ \alpha}(u)$, to show that the overall optimized algorithm is correct.

5 P-completeness of Bi-core Decomposition

The span of our Algorithm 1 from Section 4.2 is not polylogarithmic. However, we show here that the bi-core decomposition problem in general is P-complete, which means that the problem is inherently sequential and cannot be solved with polylogarithmic span if we accept the standard assumption that P $\neq$ NC [19]. Note that NC contains problems that can be solved in polylogarithmic span, or more specifically, problems that can be solved in polylogarithmic time on a parallel machine with a polynomial number of processors [23]. We show that for small enough α and β, the bi-core decomposition

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

problem is in NC. More formally, we prove these results for the decision version of the problem, which is, given α and β, and a bipartite graph, decide if the (α, β)-core in the graph is nonempty. This is a generalization of the k-core decision problem on a bipartite graph: given k, decide if the (k, k)-core is nonempty. We show the P-completeness of bi-core decision problem using a reduction from the k-core decision problem. Note that the k-core decision problem is P-complete for $k > 2$ and in NC for $k = 2$ [3].

THEOREM 5.1. *The (α, β)-core decomposition problem is P-complete if and only if $\alpha \geq 3$ and $\beta \geq 2$, or $\beta \geq 3$ and $\alpha \geq 2$. Otherwise, it is in NC.*

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

When $\alpha = 1$ or $\beta = 1$. If $\alpha = 1$, then the $(1, \beta)$-core decomposition problem is equivalent to finding all $v \in V$ such that $\deg(v) \geq \beta$. These vertices and their neighbors in U form the $(1, \beta)$-core, and can be found in $O(1)$ span. Similarly, we can find the $(\alpha, 1)$-core for any α in $O(1)$ span.

When $\alpha \geq 3$ and $\beta \geq 3$. We perform the reduction from the k-core decision problem in a graph G (that is not necessarily bipartite) by constructing a bipartite graph G' such that 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 and V, where each partition is a copy of all vertices of G. In other words, each $x \in G$ is copied to $x_u \in U$ and $x_v \in V$ in G'. We form an edge (x_u, y_v) in G' if (x, y) is an edge in G.

Now, we show that for any k, the k-core is nonempty in G if and only if the (k, k)-core is nonempty in G'. If the k-core of G is nonempty and comprises a vertex subset W, then for $w \in W$, there exists at least k edges of the form (w, p), where $p \in W$. Let W_U be the set of all vertices $x_u \in U$ that represent copies of vertices $x \in W$, and symmetrically, let W_V be the set of all $x_v \in V$ that represent copies of $x \in W$. Consider $W' = W_U \bigcup W_V$ in G. Since W_U and W_V are copies of W, and each $w \in W$ has at least k edges of the form (w, p), we know each $w_U \in W_U$ is incident to at least k edges of the form (w_U, p_V) where $p_V \in W_V$ by construction. A similar argument applies for each $w_V \in W_V$. Therefore, W' forms a (k, k)-core on the bipartite graph, and 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 the U and 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 and W_V are mirror images of each other. Let W be the vertex subset in G that corresponds to W_U and W_V. We show that it is a k-core in G. For each vertex $w_U \in W_U$ 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 the induced degree $\deg(w_U) \geq k$ for each $w_U \in W_U$ on the subset $W_U \cup W_V$, we must have that the induced degree $\deg(w) \geq k$ for each $w \in W$ on the subset W. Therefore, W forms a k-core of graph G, and since W is nonempty, the k-core of graph G is nonempty.

Thus, we have shown an NC reduction from the k-core problem to the bi-core problem, since constructing G' takes work $O(m)$ and span $O(1)$. Since the k-core decomposition is P-complete for $k \geq 3$ [3], we have shown that the bi-core decomposition is P-complete for $\alpha \geq 3$ and $\beta \geq 3$.

When one of α, $\beta = 2$. We now consider the case where $\alpha = 2$ and $\beta \geq 3$; the reverse where $\beta = 2$ and $\alpha \geq 3$ is symmetric. We show that deciding whether the $(2, \beta)$-core is nonempty has a reduction from the k-core equivalent with $k = \beta$. Consider a graph G that is not necessarily bipartite, and the k-core problem on this graph. We construct a bipartite graph G' by replacing every edge in G with a path of length 2. In other words, for each edge (x, y), we add a *middle vertex* z and replace the edge with edges (x, z) and (z, y). We let U be the set of middle vertices, and V be the set of original vertices in G, and we note that U and V form the bipartitions of G'. Also, note that we can construct G' in $O(1)$ span. Now, deciding if the $(2, \beta)$-core nonempty in G' is equivalent to deciding if the k-core of G is nonempty where $k = \beta$, because every vertex in V has the same degree as in the original graph, and every vertex in U has degree exactly 2, so they will always be included in a β-core for $\beta \geq 3$. Given this reduction, the problem of bi-core decomposition is P-complete for $\alpha = 2$ and $\beta \geq 3$, and symmetrically for $\beta = 2$ and $\alpha \geq 3$.

Thus, we have shown 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.

6 Parallel Bi-core Index Structure

The algorithm in Section 4 computes the $\alpha_{\max\ \beta}(v)$ values for every β and vertex $v \in V$, and the $\beta_{\max\ \alpha}(u)$ values for every α and vertex $u \in U$. To enable fast queries for the set of vertices inside a particular (α, β)-core, we parallelize the sequential index structure by Liu *et al.* [28]. Their index structure is constructed sequentially using the computed $\alpha_{\max\ \beta}(v)$ and $\beta_{\max\ \alpha}(u)$ values, and allows for queries of (α, β)-cores in time proportional to the number of vertices in the core. Our

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

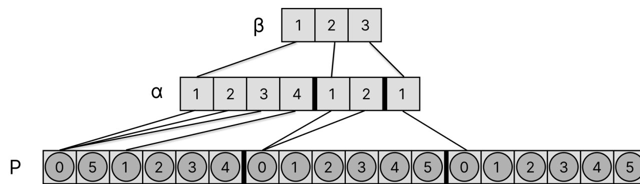


Figure 4: This figure shows the index structure $\mathbb{PI}^V$ for the example graph in Figure 1. The first level is indexed by β values, and the second level is indexed by α values, which then point to the corresponding set of vertices in the core. The lines between levels represent pointers in the structure.

parallel index construction algorithm takes $O(m)$ work and $O(\log n)$ span *w.h.p.*, and our parallel query algorithm takes linear work in the size of the core and $O(1)$ span.

We define $\mathbb{PI}^U$ and $\mathbb{PI}^V$ to be the parallel index structures for U and V respectively. Note that $\mathbb{PI}^V$ is our parallel version of $\mathbb{I}^V$ from Liu *et al.*'s work [28], and symmetrically, $\mathbb{PI}^U$ is the parallel version of $\mathbb{I}^U$, where $\mathbb{I}^V$ and $\mathbb{I}^U$ are Liu *et al.*'s sequential index structures. Because they are symmetric, we only discuss $\mathbb{PI}^V$ here. Liu *et al.* define $\mathbb{I}^V_{\alpha,\beta}$ to be the set containing all vertices $v \in V$ such that $v \in (\alpha, \beta)$-core but $v \notin (\alpha+1, \beta)$-core; notably, each $\mathbb{I}^V_{\alpha,\beta}$ points to a location in memory in their index structure $\mathbb{I}^V$. In our parallelization, we define $\mathbb{PI}^V_{\alpha,\beta}$ in the same way, and each $\mathbb{PI}^V_{\alpha,\beta}$ points to a location in memory in our index structure $\mathbb{PI}^V$.

A key difference is that in Liu *et al.*'s work, each set $\mathbb{I}^V_{\alpha,\beta}$ is stored separately. In our parallelization, we store all sets $\mathbb{PI}^V_{\alpha,\beta}$ contiguously in memory, ordered first by β and then by α. We call this array P. In order to access each set $\mathbb{PI}^V_{\alpha,\beta}$ efficiently, we define a 2D array M, where each $M[\beta][\alpha]$ corresponds to a set $\mathbb{PI}^V_{\alpha,\beta}$ and contains the starting index of that set in P. Note that unlike our other data structures, we define M to be 1-indexed, for the sake of clarity in querying for the (α, β)-core. By definition, $\mathbb{PI}^V_{\alpha,\beta}$ consists of all vertices in P in the range $[M[\beta][\alpha], M[\beta][\alpha+1]-1]$. Thus, the range $[M[\beta][\alpha], M[\beta+1][0]-1]$ gives precisely the vertices that correspond to $\bigcup_{i=\alpha}^{\max_\alpha(\beta)} \mathbb{PI}^V_{i,\beta}$, which is the set of all vertices in the (α, β)-core. Figure 4 shows an example of the $\mathbb{PI}^V$ index structure for the example graph from Figure 1.

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

6.1 Parallel Index Construction In this section, we detail our index construction algorithm for index structure $\mathbb{PI}^V$. Note that the algorithm for constructing $\mathbb{PI}^U$ is symmetric. The inputs to the construction algorithm are the $\alpha_{\max\ \beta}(v)$ values for every β and every vertex $v \in V$. The objective is to construct $\mathbb{PI}^V$, which consists of M and P.

First, we construct P. For every $v \in V$ and a β value such that $\alpha_{\max\ \beta}(v) > 0$, we store in an array a tuple $(\beta, \alpha_{\max\ \beta}(v), v)$. We then perform parallel RADIX-SORT on these tuples to obtain P. To construct M, we apply a parallel filter to find the indices of P at which the β or $\alpha_{\max\ \beta}(v)$ values change. We store these indices in an array, ***TPT*** (total pointer table). We also find indices where only the β values change, which we store in ***FPT*** (first pointer table). These two tables contain the required information to form M.

The pseudocode for our parallel index construction algorithm is in Algorithm 2. We now discuss our algorithm in more detail. First, on Line 2, we store a list of tuples $(\beta, \alpha_{\max\ \beta}(v), v)$ in P. This is the list of all possible combinations of β and $v \in V$, with the corresponding $\alpha_{\max\ \beta}(v)$ value, where $\alpha_{\max\ \beta}(v) > 0$. We perform parallel RADIX-SORT on P based on the ordering of first the β values and then the α values, on Line 3; note that the third value v in the tuple need not be sorted. For our example graph in Figure 1, the sorted $P = [(1,3,0),(1,3,5),(1,4,1),(1,4,2),(1,4,3),(1,4,4),$ $(2,2,0),(2,2,1),(2,2,2),(2,2,3),(2,2,4),(2,2,5),(3,1,0),(3,1,1),(3,1,2),(3,1,3),(3,1,4),(3,1,5)]$, as illustrated by the example index structure in Figure 4. Then, 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 β value or the α value in P changes. This is accomplished by marking the indices where consecutive values differ on Line 7 and filtering out all of the unmarked index positions on Line 8. Constructing ***TPT*** essentially breaks up the array P into blocks where each block has the same β and α values, and corresponds to set $\mathbb{PI}^V_{\alpha,\beta}$. ***TPT***$[i]$ records the starting index location of the ith block. Lines 9–13 repeat the entire process, but for ***FPT***, to filter out index positions where only the β 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***; this is to form the second level of the 2D array M. $[$***FPT***$[\beta-1],$ ***FPT***$[\beta]-1]$ gives the range of blocks defined by ***TPT*** with this particular β value, and it corresponds to set $\bigcup_{0\le\alpha\le\max_\alpha(\beta)} \mathbb{PI}^V_{\alpha,\beta}$.

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Algorithm 2 Parallel Index Construction

```
 1: procedure BUILD-V-INDEX(α_max β(v) values)
 2:     P ← array of (β, α_max β(v), v) for every v ∈ V
 3:     RADIX-SORT(P)                                         ▷ Sorts P by first β, and then α
 4:     Initialize TPT                                        ▷ Creates an empty TPT of length |P|
 5:     parfor i = 0 to P.size − 1 do
 6:         if i = 0 or P[i − 1].β ≠ P[i].β or P[i − 1].α ≠ P[i].α then
 7:             TPT[i] ← i                                    ▷ Records index where β or α changes value
 8:     FILTER(TPT, element is not empty)                     ▷ Filter out empty indices
 9:     Initialize FPT                                        ▷ Creates an empty FPT of length |P|
10:     parfor i = 0 to TPT.size − 1 do
11:         if i = 0 or P[TPT[i − 1]].β ≠ P[TPT[i]].β then
12:             FPT[i] ← i                                    ▷ Records index of TPT array where β changes
13:     FILTER(FPT, element is not empty)                     ▷ Filter out empty indices
14:     Initialize M    ▷ Creates empty M array with 1st dimension = FPT.size and 2nd dimension = max_α(β)
15:     parfor β = 1 to FPT.size do
16:         parfor j = FPT[β − 1] to FPT[β]−1 do
17:             start ← TPT[j]                                ▷ Obtains starting index of j-th block
18:             M[β][P[start].α] ← start    ▷ Stores starting index; P[start].α gives j-th block's corresponding α value
19:         M[β] ← SUFFIX-MIN(M[β])
20:     return M
21: procedure BUILD-U-INDEX(β_max α(u) values)
22:     symmetric to BUILD-V-INDEX
```

Finally, based on *FPT* and *TPT*, we create M in the following manner. We obtain $M[\beta]$ for each β value independently. Line 16 iterates in parallel over the blocks that have the particular β value. For each block j, on Lines 17–18, we store its starting position *TPT*$[j]$ to $M[\beta][\alpha_j]$, where α_j is the α value corresponding with the jth block. Thus, we have constructed $M[\beta][\alpha']$ for all (β, α') pairs such that α' equals $\alpha_{\max\ \beta}(v)$ for some $v \in V$. The 2D array M created from our example graph is visually represented by the first two levels of the index structure in Figure 4.

For pairs (β, α) where α does not appear in $\alpha_{\max\ \beta}(v)$ for some $v \in V$, we point $M[\beta][\alpha]$ to the same location as $M[\beta][\alpha']$ where α' is the smallest value $\geq \alpha$ that appears in $\alpha_{\max\ \beta}(v)$ for some $v \in V$. We accomplish this by performing SUFFIX-MIN on $M[\beta]$ on Line 19.

Analysis. Since P.size $= O(m)$, and since we only need to sort each tuple $(\beta, \alpha_{\max\ \beta}(v), v)$ by the first element β and second element $\alpha_{\max\ \beta}(v)$, if we compress the pairs $(\beta, \alpha_{\max\ \beta})$ into integer keys, the range of these keys is bounded by $O(\sum_{1\leq\beta\leq \text{dmax}_V} \max_\alpha(\beta)) = O(m)$ [28]. The compression can be done since we know the $\max_\alpha(\beta)$ value for each β. Thus, we can construct a mapping from each pair (β, α) to an index by running a prefix sum over the $\max_\alpha(\beta)$ values for each $\beta \in [1, \text{dmax}_V]$. The mapping can then be stored in a parallel hash table to be used during the RADIX-SORT, which as such takes $O(m)$ work *w.h.p.* Lines 4–13 also take $O(m)$ work, proportional to the sizes of *TPT* and *FPT*. Lines 14–19 take $O(m)$ work, because we loop through M exactly once and the table takes $O(m)$ space, as in its sequential counterpart in [28].

Algorithm 2 has span $O(\log n)$ *w.h.p.*, since RADIX-SORT, FILTER, SUFFIX-MIN, and hash table operations [16] are bounded by $O(\log n)$ span *w.h.p.*

7 Experiments

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

7.1 Experimental Setup We experiment using real-world bipartite graphs from the KONECT graph database [25], the details of which are given in Table 2. As seen in Table 2, the bi-core peeling complexities, or ρ, of these real-world graphs are in general 3–4 orders of magnitude smaller than their numbers of edges m. This indicates that the $O(\rho \log n)$ span *w.h.p.* achieved by our parallel bi-core decomposition algorithm is significantly lower than the $O(m)$ span achieved by the previous state-of-the-art parallel algorithm [28].

We use Google Cloud Platform `c2-standard-60` instances, 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 maximum turbo

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Name	$\|U\|$	$\|V\|$	m	dmax	δ	ρ
Orkut	2.78M	8.73M	327M	318K	466	12100
Web Trackers	27.7M	12.7M	140.6M	11.57M	437	4542
TREC	556K	1.17M	83.6M	457K	508	6029
LiveJournal	3.20M	7.49M	112M	1.05M	108	6831
Reuters	781K	284K	60.6M	345K	192	4767
Epinions	120K	755K	13.67M	162K	151	3049
Flickr	396K	104K	8.55M	35K	147	2300

Table 2: The graphs used in our experiments, along with the sizes, maximum degree (dmax), degeneracy (δ), and number of rounds required in peeling, or the bi-core peeling complexity, (ρ) are shown.

clock-speed of 3.8 GHz.

7.2 Implementation and Other Optimizations While our parallel bi-core decomposition algorithm (Algorithm 1) is theoretically efficient, it is practically slow due to the overhead incurred by the histogram-based PAR-DEL-UPDATE subroutine. We implemented the fully parallel algorithm and found it to be orders of magnitude slower than Liu *et al.*'s algorithm on certain graphs, and it is overall slower on all of the datasets that we experiment with. Thus, for a practically fast implementation, we do not parallelize the bi-core peeling process as described in Section 4. Instead, we only parallelize across different peeling processes, similar to Liu *et al.*'s parallel algorithm [28]. Our parallel implementation differs from Liu *et al.*'s work in that we utilize the Julienne bucketing structure [11] to search for the next set of vertices with minimum induced degree in each peeling iteration. Julienne was originally designed as a parallel bucketing structure, but we use a sequential version, since the additional parallelism does not improve our algorithm's performance. Liu *et al.*'s algorithm, on the other hand, uses a simple sequential search to find the next set of vertices with minimum induced degree. In addition, our parallel implementation includes the peeling space pruning optimization described in Section 4.3. We demonstrate in this section that our optimization techniques are effective.

We use Liu *et al.*'s sequential and parallel bi-core decomposition algorithms as baselines. In total, we perform our experimental analysis on the following bi-core decomposition algorithms:

1. LIU-SEQ: Liu *et al.*'s sequential algorithm [28];
2. LIU-PAR: Liu *et al.*'s parallel algorithm [28]; and
3. PAR: Our parallel algorithm, which differs from LIU-PAR in that we use Julienne [11] and the peeling space pruning optimization described in Section 4.3.

We also perform experiments on the following bi-core index construction and query algorithms. Note that Liu *et al.* do not provide parallel implementations for their bi-core index construction and query algorithms, so their sequential implementations are the state of the art.

1. LIU-CONS: Liu *et al.*'s sequential bi-core index construction algorithm [28];
2. LIU-QUERY: Liu *et al.*'s sequential bi-core query algorithm [28];
3. IND-CONS: Our parallel bi-core index construction algorithm as detailed in Algorithm 2; and
4. QUERY: Our parallel bi-core query algorithm, as described in Section 6.

We use the Graph Based Benchmark Suite (GBBS) [12] to implement our algorithms. All of our code is written in C++ and compiled with the `-O3` flag, and we use the work-stealing scheduler from PARLAYLIB by Blelloch *et al.* [5]. We perform each experiment three times and report the average running time.

Because QUERY and LIU-QUERY are extremely fast, we perform query experiments in batches of 10,000 bi-core queries per batch, and report the total time for the batch. For each (α, β)-core query in the batch, the (α, β)-core to be queried is uniformly at random selected from all non-empty bi-cores in the input graph. We test both QUERY and LIU-QUERY on the same set of randomly sampled (α, β)-cores.

We run our parallel algorithms, including PAR, IND-CONS, and QUERY, on 30 threads; using 60 hyper-threads, in general, does not improve their performances. However, we find that LIU-PAR benefits from hyper-threading, and we compare to LIU-PAR using 60 hyper-threads.

7.3 Bi-core Decomposition In this section, we discuss the performance of the bi-core decomposition algorithms LIU-SEQ, LIU-PAR, and PAR.

Comparison to Prior Work. Figure 5 shows the comparison of our parallel bi-core decomposition implementation to the

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

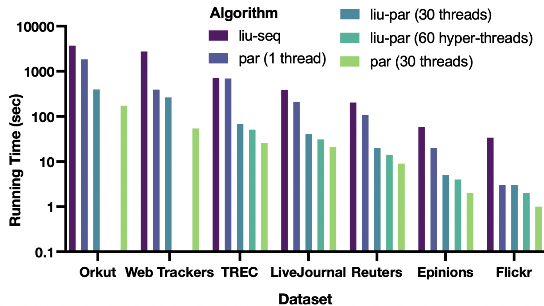


Figure 5: This figure compares the running time (in seconds) of various bi-core decomposition algorithms, namely LIU-PAR, LIU-SEQ, and PAR. LIU-PAR on 60 hyper-threads runs out of memory for the Orkut and Web Trackers graphs, hence the missing bars. However, LIU-PAR is able to finish running on all graphs on 30 threads.

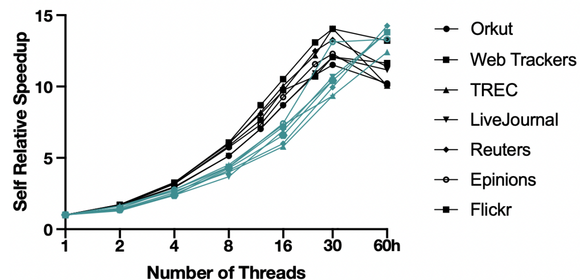


Figure 6: This figure compares the parallel self-relative speedups of PAR (in black) and LIU-PAR (in green) over different numbers of threads. LIU-PAR on 60 hyper-threads runs out of memory for the Orkut and Web Trackers graphs. Also, "60h" stands for 60 hyper-threads.

state-of-the-art sequential and parallel implementations [28]. Our parallel implementation PAR attains between 18.2–51.4x speedups over LIU-SEQ, with an average speedup of 27.9x. Compared to LIU-PAR running on 60 hyper-threads, PAR also achieves between 1.5–2.0x speedups over LIU-PAR, with an average speedup of 1.8x. However, we note that LIU-PAR runs out of memory when running on 60 hyper-threads for large graphs, specifically Orkut and Web Trackers. This is because LIU-PAR keeps a separate copy of the table storing the bi-core numbers (*i.e.* the $\alpha_{\max\ \beta}(v)$ and $\beta_{\max\ \alpha}(u)$ values) for each thread, so it consumes a significant amount of memory when running on a large number of threads. In comparison, our algorithm PAR keeps a single global copy of this table, which also reduces the overall memory consumption of PAR compared to LIU-PAR. For Orkut and Web Trackers, the running times reported in Figure 5 represent LIU-PAR's performance on 30 threads, which does not run out of memory on these graphs. Compared to running LIU-PAR on 30 threads, PAR is able to achieve 1.9–4.9x speedups, with an average of 2.7x. Considering the best running times of LIU-PAR for each graph, PAR is between 1.5–4.9x faster than LIU-PAR, with an average of 2.3x.

We outperform LIU-PAR due to our peeling space pruning optimization and our use of Julienne [11]. In particular, LIU-PAR performs more repeated work for cores that are processed by both peeling along increasing α values and peeling along increasing β value, as explained in detail in Section 4.3. Additionally, we note that our single-threaded running times achieve between 1.8–10.3x speedups over LIU-SEQ, with an average speedup of 3.9x. This demonstrates the effectiveness of our optimizations, particularly our peeling space pruning optimization, even in the sequential setting.

Analysis of Scalability. Figure 6 demonstrates the parallel scalability of our algorithm over different numbers of threads. PAR achieves up to a 14.6x self-relative speedup, with an average of 12.2x across our different input graphs, comparing our running time on 30 threads to our single-threaded running time. In comparison, LIU-PAR, when running on 60 hyper-threads, achieves a similar average self-relative speedup of 13.5x and a maximum of 14.3x. When LIU-PAR is run on 30 threads, it attains an average self-relative speedup of 10.5x, with a maximum of 13.1x.

We note that there is a plateau of the speedup achieved by PAR from 30 threads to 60 hyper-threads due to the strong inherent parallelism of our algorithm; the additional parallelism provided by hyper-threading does not improve its running time.

7.4 Bi-core Index

Now, we discuss the performance of the parallel bi-core index construction and query algorithms.

Bi-core Index Construction. As shown in Figure 7, our parallel index construction algorithm IND-CONS consistently outperforms the sequential algorithm LIU-CONS across different graphs. Using 30 threads, IND-CONS achieves 10.6–27.7x speedups over LIU-CONS, with an average speedup of 18.4x. We additionally note that constructing the index is not computationally intensive compared to computing the bi-core decomposition, and the running times of IND-CONS constitute 2.2%–8.3% of the running times of PAR.

Figure 8 shows the self-relative speedups of IND-CONS across different number of threads and over graphs of different sizes. We observe good scalability, with 20.7–28.5x self-relative speedups of IND-CONS on 30 threads; the average self-relative speedup is 24.7x.

Bi-core Index Query. Our parallel QUERY operation attains between 17.1–116.3x speedups over LIU-QUERY, as demonstrated by Figure 9, with an average of 43.8x. Our significant speedups are due to our parallelization and due to

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

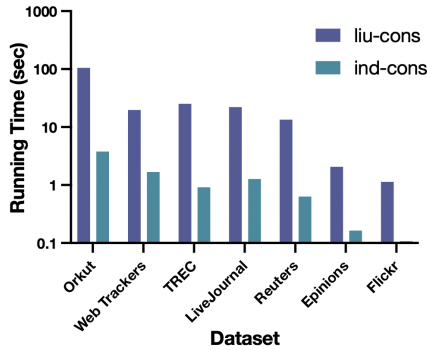


Figure 7: This figure shows the running times of Liu et al.'s sequential index construction algorithm, LIU-CONS and our parallel index construction algorithm, IND-CONS.

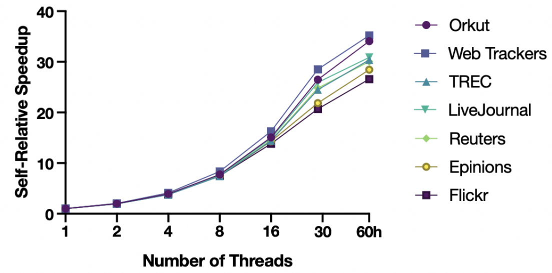


Figure 8: This figure shows the parallel self-relative speedup of IND-CONS over different numbers of threads. Note that "60h" stands for 60 hyper-threads.

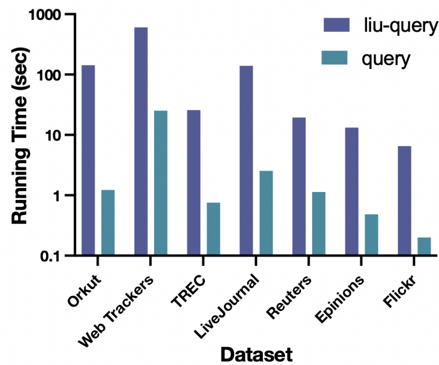


Figure 9: This graph shows the running times of Liu et al.'s sequential query algorithm, LIU-QUERY, and our parallel query algorithm, QUERY, on a batch of 10,000 queries.

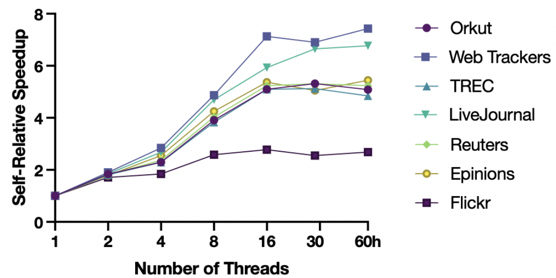


Figure 10: This graph shows the parallel self-relative speedups of PAR-QUERY over different numbers of threads, running on a batch of 10,000 queries. Note that "60h" stands for 60 hyper-threads.

our usage of a compact storage format for our index structure, which gives us better cache locality for batches of queries.

Figure 10 shows the parallel scalability of QUERY over different numbers of threads. QUERY achieves between 2.6–6.9x self-relative speedup on 30 threads, with an average of 5.3x. Overall, it shows good scalability over most of our input graphs, especially graphs with larger sizes.

8 Conclusion

In this paper, we develop a work-efficient shared-memory parallel bi-core decomposition algorithm with improved 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.* Furthermore, 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 work-efficient parallel index construction and query algorithms. Finally, we introduce a practical optimization reducing the amount of computation in the peeling process, and we provide fast implementations of all of our algorithms. We perform a thorough experimental evaluation of our algorithms on real-world bipartite graphs, and demonstrate that our parallel algorithms outperform the previous best parallel implementation by up to 4.9x for computing the bi-core numbers, and outperform the previous best sequential implementation by up to 27.7x for constructing the index structure and up to 116.3x for querying the bi-cores. We also show that our bi-core decomposition algorithms are scalable to various real-world graphs with up to hundreds of millions of edges.

Acknowledgements

This research is supported by MIT PRIMES, NSF GRFP #1122374, DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, cloud computing

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

credits from Google-MIT, FinTech@CSAIL Initiative, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] Adel Ahmed, Vladimir Batagelj, Xiaoyan Fu, Seok-hee Hong, Damian Merrick, and Andrej Mrvar. Visualisation and analysis of the internet movie database. In *International Asia-Pacific Symposium on Visualization*, pages 17–24, 2007.

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

[3] Richard J. Anderson and Ernst W. Mayr. A p-complete problem and approximations to it. In *Stanford Technical Report*, 1984.

[4] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. Copycatch: Stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the International Conference on World Wide Web*, page 119–130, 2013.

[5] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Brief announcement: ParlayLib – a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 507–509, 2020.

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[7] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis. *Technical Report, National Security Agency*, 2008.

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

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

[10] Inderjit S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 269–274, 2001.

[11] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[12] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. The graph based benchmark suite (GBBS). In *GRADES-NDA*, pages 1–8, 2020.

[13] Danhao Ding, Hui Li, Zhipeng Huang, and Nikos Mamoulis. Efficient fault-tolerant group recommendation using alpha-beta-core. In *ACM on Conference on Information and Knowledge Management*, page 2047–2050, 2017.

[14] Daniel C Fain and Jan O Pedersen. Sponsored search: A brief history. *Bulletin-American Society For Information Science And Technology*, 32(2):12, 2006.

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

[16] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

[17] GraphChallenge. `http://graphchallenge.mit.edu/`.

[18] Yan Gu, Julian Shun, Yihan Sun, and Guy E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

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

[20] Rut Jesus, Martin Schwartz, and Sune Lehmann. Bipartite networks of wikipedia's articles and authors: a meso-level approach. In *Proceedings of the International Symposium on Wikis and Open Collaboration*, pages 1–10, 2009.

[21] H. Kabir and K. Madduri. Parallel k-core decomposition on multicore platforms. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1482–1491, 2017.

[22] Humayun Kabir and Kamesh Madduri. Parallel k-truss decomposition on multicore systems. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[23] Dexter C Kozen. *Theory of Computation*, volume 121. Springer, 2006.

[24] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. Trawling the web for emerging cyber-communities. *Computer Networks*, 31(11-16):1481–1493, 1999.

[25] Jérôme Kunegis. KONECT: the Koblenz network collection. In *International Conference on World Wide Web (WWW)*, pages 1343–1350, 05 2013.

[26] Kartik Lakhotia, Rajgopal Kannan, Viktor Prasanna, and Cesar A. F. De Rose. Receipt: Refine coarse-grained independent tasks for parallel tip decomposition of bipartite graphs. *Proc. VLDB Endow.*, 14(3):404–417, November 2020.

[27] Don R Lick and Arthur T White. k-degenerate graphs. *Canadian Journal of Mathematics*, 22(5):1082–1096, 1970.

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

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited

[29] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, July 1983.
[30] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 24(2):288–300, 2013.
[31] Georgios A Pavlopoulos, Panagiota I Kontou, Athanasia Pavlopoulou, Costas Bouyioukos, Evripides Markou, and Pantelis G Bagos. Bipartite graphs in systems biology and medicine: a survey of methods and applications. *GigaScience*, 7(4), 02 2018.
[32] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, 1989.
[33] Seyed-Vahid Sanei-Mehri, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. Butterfly counting in bipartite networks. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 2150–2159, 2018.
[34] Ahmet Erdem Sariyüce and Ali Pinar. Peeling bipartite networks for dense subgraph discovery. In *ACM International Conference on Web Search and Data Mining (WSDM)*, pages 504–512, 2018.
[35] Ahmet Erdem Sariyüce, C. Seshadhri, and Ali Pinar. Local algorithms for hierarchical dense subgraph discovery. *VLDB Endowment*, 12(1):43–56, 2018.
[36] Ahmet Erdem Sariyüce, C. Seshadhri, Ali Pinar, and Ümit V. Çatalyürek. Nucleus decompositions for identifying hierarchy of dense subgraphs. *ACM Trans. Web*, 11(3):16:1–16:27, July 2017.
[37] EN Sawardecker, CA Amundsen, M Sales-Pardo, and Luis AN Amaral. Comparison of methods for the detection of node group membership in bipartite networks. *The European Physical Journal B*, 72(4):671–677, 2009.
[38] Jessica Shi, Laxman Dhulipala, and Julian Shun. Parallel clique counting and peeling algorithms. In *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 135–146, 2021.
[39] Jessica Shi, Laxman Dhulipala, and Julian Shun. Theoretically and practically efficient parallel nucleus decomposition. *Proceedings of the VLDB Endowment*, 15(3):583–596, November 2021.
[40] Jessica Shi and Julian Shun. Parallel algorithms for butterfly computations. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 16–30, 2020.
[41] Alok Tripathy, Fred Hohman, Duen Horng Chau, and Oded Green. Scalable k-core decomposition for static graphs using a dynamic graph data structure. In *IEEE International Conference on Big Data (Big Data)*, pages 1134–1141, 2018.
[42] Charalampos Tsourakakis. The k-clique densest subgraph problem. In *Proceedings of the International Conference on World Wide Web*, page 1122–1132, 2015.
[43] Jia Wang and James Cheng. Truss decomposition in massive networks. *Proc. VLDB Endow.*, 5(9):812–823, May 2012.
[44] Jun Wang, Arjen P De Vries, and Marcel JT Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 501–508, 2006.
[45] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal*, pages 1–24, 03 2021.
[46] Kai Wang, Wenjie Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. Efficient and effective community search on large-scale bipartite graphs. In *IEEE International Conference on Data Engineering (ICDE)*, pages 85–96, 2021.
[47] Si Zhang, Dawei Zhou, Mehmet Yigit Yildirim, Scott Alcorn, Jingrui He, Hasan Davulcu, and Hanghang Tong. Hidden: hierarchical dense subgraph detection with application to financial fraud detection. In *SIAM International Conference on Data Mining (SDM)*, pages 570–578, 2017.
[48] Yang Zhang and Srinivasan Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1049–1060, 2012.
[49] Feng Zhao and Anthony Tung. Large scale cohesive subgraphs discovery for social network visual analysis. *Proceedings of the VLDB Endowment*, 6:85–96, 12 2012.
[50] Zhaonian Zou. Bitruss decomposition of bipartite graphs. In *Database Systems for Advanced Applications*, pages 218–233, 2016.

Downloaded 01/22/23 to 12.138.45.132 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © 2023 by SIAM
Unauthorized reproduction of this article is prohibited