Parallel Algorithms for Butterfly Computations

Jessica Shi  
MIT CSAIL  
jeshi@mit.edu  

Julian Shun  
MIT CSAIL  
jshun@mit.edu  

Abstract

Butterflies are the smallest non-trivial subgraph in bipartite graphs, and therefore having efficient computations for analyzing them is crucial to improving the quality of certain applications on bipartite graphs. In this paper, we design a framework called PAR\textsc{Butterfly} that contains new parallel algorithms for the following problems on processing butterflies: global counting, per-vertex counting, per-edge counting, tip decomposition (vertex peeling), and wing decomposition (edge peeling). The main component of these algorithms is aggregating wedges incident on subsets of vertices, and our framework supports different methods for wedge aggregation, including sorting, hashing, histogramming, and batching. In addition, PAR\textsc{Butterfly} supports different ways of ranking the vertices to speed up counting, including side ordering, approximate and exact degree ordering, and approximate and exact complement coreness ordering. For counting, PAR\textsc{Butterfly} also supports both exact computation as well as approximate computation via graph sparsification. We prove strong theoretical guarantees on the work and span of the algorithms in PAR\textsc{Butterfly}.

We perform a comprehensive evaluation of all of the algorithms in PAR\textsc{Butterfly} on a collection of real-world bipartite graphs using a 48-core machine. Our counting algorithms obtain significant parallel speedup, outperforming the fastest sequential algorithms by up to 13.6x with a self-relative speedup of up to 38.5x. Compared to general subgraph counting solutions, we are orders of magnitude faster. Our peeling algorithms achieve self-relative speedups of up to 10.7x and outperform the fastest sequential baseline by up to several orders of magnitude.

1 Introduction

A fundamental problem in large-scale network analysis is finding and enumerating basic graph motifs. Graph motifs that represent the building blocks of certain networks can reveal the underlying structures of these networks. Importantly, triangles are core substructures in unipartite graphs, and indeed, triangle counting is a key metric that is widely applicable in areas including social network analysis [41], spam and fraud detection [8], and link classification and recommendation [57]. However, many real-world graphs are bipartite and model the affiliations between two groups. For example, bipartite graphs are used to represent peer-to-peer exchange networks (linking peers to the data they request), group membership networks (e.g., linking actors to movies they acted in), recommendation systems (linking users to items they rated), factor graphs for error-correcting codes, and hypergraphs [11, 35]. Bipartite graphs contain no triangles; the smallest non-trivial subgraph is a butterfly (also known as rectangles), which is a \((2, 2)\)-biclique (containing two vertices on each side and all four possible edges among them), and thus having efficient algorithms for counting butterflies is crucial for applications on bipartite graphs [5, 48, 59]. Notably, butterfly counting has applications in link spam detection [22] and document clustering [17]. Moreover, butterfly counting naturally lends itself to finding dense subgraph structures in bipartite networks. Zou [65] and Sariyüce and Pinar [49] developed peeling algorithms to hierarchically discover dense subgraphs, similar to the \(k\)-core decomposition for unipartite graphs [39, 50]. An example bipartite graph and its butterflies is shown in Figure 1.

There has been recent work on designing efficient sequential algorithms for butterfly counting and peeling [13, 48, 49, 59, 64, 65]. However, given the high computational requirements of butterfly computations, it is natural to study whether we can obtain performance improvements using parallel machines. This paper presents a framework for butterfly computations, called PAR\textsc{Butterfly}, that enables us to obtain new parallel algorithms for butterfly counting and peeling. PAR\textsc{Butterfly} is a modular framework that enables us to easily experiment with many variations of our algorithms. We not only show that our algorithms are efficient in practice, but also prove strong theoretical bounds on their work and span. Given that all real-world bipartite graphs fit on a multicore machine, we design parallel algorithms for this setting.

For butterfly counting, the main procedure involves finding wedges (2-paths) and combining them to count butterflies. See Figure 1 for an example of wedges. In particular, we want to find all wedges originating from each vertex, and then aggregate the counts of wedges incident to every distinct pair of vertices forming the endpoints of the wedge. With these counts, we can obtain global, per-vertex, and per-edge butterfly counts. The PAR\textsc{Butterfly} framework provides different ways to aggregate wedges in...
Parallel algorithms for butterfly counting and peeling algorithms are highly parallel and match the work of the best sequential algorithms for dense subgraph discovery algorithms developed by Zou [65] and Sariyüce and Pinar [49]. Our peeling algorithms iteratively remove the vertices (tip decomposition) or edges (wing decomposition) with the lowest butterfly count until the graph is empty. Each iteration removes vertices (edges) from the graph in parallel and updates the butterfly counts of neighboring vertices (edges) using the parallel wedge aggregation techniques that we developed for counting. We use a parallel bucketing data structure by Dhillipala et al. [18] and a new parallel Fibonacci heap to efficiently maintain the butterfly counts.

We prove theoretical bounds showing that some variants of our counting and peeling algorithms are highly parallel and match the work of the best sequential algorithm. For a graph \( G(V, E) \) with \( m \) edges and arboricity \( \alpha \), PAR\textsc{Butterfly} gives a counting algorithm that takes \( O(n \alpha m) \) expected work and \( O(\log m) \) span with high probability (w.h.p.). Using a parallel Fibonacci heap that we design, PAR\textsc{Butterfly} gives a vertex-peeling algorithm that takes \( O(\min(\max-b_v, \rho_v \log n) + \sum_{v \in V} \deg(v)^2) \) expected work and \( O(\rho_v \log^2 n) \) span w.h.p., and an edge-peeling algorithm that takes \( O(\min(\max-b_e, \rho_e \log n) + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')) \) expected work and \( O(\rho_e \log^2 m) \) span w.h.p., where \( \max-b_v \) and \( \max-b_e \) are the maximum number of per-vertex and per-edge butterflies and \( \rho_v \) and \( \rho_e \) are the number of vertex and edge peeling iterations required to remove the entire graph. Our work bounds for vertex-peeling and edge-peeling significantly improve upon Sariyüce and Pinar’s sequential algorithms, which take work proportional to the maximum number of per-vertex and per-edge butterflies.

We present a comprehensive experimental evaluation of all of the different variants of counting and peeling algorithms in the PAR\textsc{Butterfly} framework. On a 48-core machine, our counting algorithms achieve self-relative speedups of up to 38.5x and outperform the fastest sequential baseline by up to 13.6x. Our peeling algorithms achieve self-relative speedups of up to 10.7x and due to their improved work complexities, outperform the fastest sequential baseline by up to several orders of magnitude. Compared to PGD [2], a state-of-the-art parallel subgraph counting solution that can be used for butterfly counting as a special case, we are 349.6–5169x faster. We find that although the sorting, hashing, and histogramming aggregation approaches achieve better theoretical complexity, batching usually performs the best in practice due to lower overheads.

In summary, the contributions of this paper are as follows.
1. New parallel algorithms for butterfly counting and peeling.
2. A framework PAR\textsc{Butterfly} with different ranking and wedge aggregation schemes that can be used for parallel butterfly counting and peeling.
3. Strong theoretical bounds on algorithms obtained using PAR\textsc{Butterfly}.
4. A comprehensive experimental evaluation on a 48-core machine demonstrating high parallel scalability and fast running times compared to the best sequential baselines, as well as significant speedups over the state-of-the-art parallel subgraph counting solution.

The PAR\textsc{Butterfly} code can be found at https://github.com/jeshi96/parbutterfly. Due to space constraints, we have omitted some details from this version of the paper, and the full version can be found on arXiv [51].

2 Preliminaries

Graph Notation. We take every bipartite graph \( G = (U, V, E) \) to be simple and undirected. For any vertex \( v \in U \cup V \), let \( N(v) \) denote the neighborhood of \( v \), let \( N_2(v) \) denote the 2-hop neighborhood of \( v \) (the set of all vertices reachable from \( v \) by a path of length 2), and let \( \deg(v) \) denote the degree of \( v \). For added clarity when working with multiple graphs, we let \( N^G(v) \) denote the neighborhood of \( v \) in \( G \) and let \( N^G_2(v) \) denote the 2-hop neighborhood of \( v \) in \( G \). We use \( n = |U| + |V| \) to denote the number of vertices in \( G \), and \( m = |E| \) to denote the number of edges in \( G \).

A butterfly is a set of four vertices \( u_1, u_2 \in U \) and \( v_1, v_2 \in V \) with edges \((u_1, v_1), (u_1, v_2), (u_2, v_1), (u_2, v_2) \in
A wedge is a set of three vertices $u_1, u_2 \in U$ and $v \in V$, with edges $(u_1, v), (u_2, v) \in E$. We call the vertices $u_1, u_2$ endpoints and the vertex $v$ the center. Symmetrically, a wedge can also consist of vertices $v_1, v_2 \in V$ and $u \in U$, with edges $(v_1, u), (v_2, u) \in E$. We call the vertices $v_1, v_2$ endpoints and the vertex $u$ the center. We can decompose a butterfly into two wedges that share the same endpoints but have distinct centers.

The arboricity $\alpha$ of a graph is the minimum number of spanning forests needed to cover the graph. In general, $\alpha$ is upper bounded by $O(\sqrt{m})$ and lower bounded by $\Omega(1)$ [13]. Importantly, $\sum_{(u,v) \in E} \min(\deg(u), \deg(v)) = O(\alpha m)$.

We store our graphs in compressed sparse row (CSR) format, which requires $O(m+n)$ space. We initially maintain separate offset and edge arrays for each vertex partition $U$ and $V$, and assume that all arrays are stored consecutively in memory.

**Model of Computation.** We use the work-span model of parallel computation, with arbitrary forking, to analyze our algorithms. The work of an algorithm is defined to be the total number of operations, and the span is defined to be the longest dependency path [15, 31]. We aim for algorithms to be work-efficient, that is, a work complexity that matches the best-known sequential time complexity. We assume concurrent reads and writes and atomic adds are supported in $O(1)$ work and span.

**Parallel primitives.** We use the following primitives in this paper. **Prefix sum** takes as input a sequence $A$ of length $n$, an identity $\varepsilon$, and an associative binary operator $\oplus$, and returns the sequence $B$ of length $n$ where $B[i] = \bigoplus_{j<i} A[j] \oplus \varepsilon$. **Filter** takes as input a sequence $A$ of length $n$ and a predicate function $f$, and returns the sequence $B$ containing $a \in A$ such that $f(a)$ is true, in the same order that these elements appeared in $A$. Both algorithms take $O(n)$ work and $O(\log n)$ span [31].

We also use several parallel primitives in our algorithms for aggregating equal keys. **Semisort** groups together equal keys but makes no guarantee on total order. For a sequence of length $n$, parallel semisort takes $O(n)$ expected work and $O(\log n)$ span with high probability [26]. Additionally, we use parallel hash tables and histograms for aggregation, which have the same bounds as semisort [18, 19, 52].

**3 PARButterfly Framework**

In this section, we describe the PARButterfly framework and its components. Section 3.1 describes the procedures for counting butterflies and Section 3.2 describes the butterfly peeling procedures. Section 4 goes into more detail on the parallel algorithms that can be plugged into the framework, as well as their theoretical bounds.

**3.1 Counting Framework** Figure 2 shows the high-level structure of the PARButterfly framework. Step 1 assigns a global ordering to the vertices, which helps reduce the overall work of the algorithm. Step 2 retrieves all the wedges in the graph, but only where the second and third vertices of the wedge have higher rank than the first. Step 3 counts for every pair of vertices the number of wedges that share those vertices as endpoints. Step 4 uses the wedge counts to obtain global, per-vertex, or per-edge butterfly counts. For each step, there are several options with respect to implementation, each of which can be independently chosen and used together.

Figure 3 shows an example of executing each of the steps. The options within each step of PARButterfly are described in the rest of this section.

**3.1.1 Ranking** The ordering of vertices when we retrieve wedges is significant since it affects the number of wedges that we process. As we discuss in Section 4.1, Sanei-Mehri et al. [48] order all vertices from one bipartition of the graph first, depending on which bipartition produces the least number of wedges, giving them practical speedups in their serial implementation. We refer to this ordering as **side order**. Chiba and Nishizeki [13] achieve a lower work complexity for counting by ordering vertices in decreasing order of degree, which we refer to as **degree order**.

For practical speedups, we also introduce approximate degree order, which orders vertices in decreasing order of the logarithm of their degree (log-degree). Since the ordering of vertices in many real-world graphs have good locality, approximate degree order preserves the locality among vertices with equal log-degree. We show in the full paper that butterfly counting using approximate degree order is work-efficient.

**Degeneracy order**, also known as the ordering given by vertex coreness, is a well-studied ordering of vertices given by repeatedly finding and removing vertices of smallest degree [39, 50]. This ordering can be obtained serially in linear time using a $k$-core decomposition algorithm [39], and in parallel in linear work by repeatedly removing (peeling) all vertices with the smallest degree from the graph in parallel [18]. The span of peeling is proportional to the number of peeling rounds needed to reduce the graph to an empty graph. We define complement degeneracy order to be the ordering given by repeatedly removing vertices of largest degree. This mirrors the idea of decreasing order of degree, but encapsulates more structural information about the graph.
Figure 3: We execute butterfly counting per vertex on the graph in Figure 1. In Step 1, we rank vertices in decreasing order of degree. In Step 2, for each vertex \( v \) in order, we retrieve all wedges where \( v \) is an endpoint and where the other two vertices have higher rank (the wedges are represented as \( ((x, z), y) \) where \( x \) and \( z \) are endpoints and \( y \) is the center). In Step 3, we aggregate wedges by their endpoints, and this produces the butterfly counts for Step 4. Note that if we have \( w \) wedges that share the same endpoint, this produces \( \binom{w}{2} \) butterflies for each of the two endpoints and \( w - 1 \) butterflies for each of the centers of the \( w \) wedges.

However, using complement degeneracy order is not efficient. The span of finding complement degeneracy order is limited by the number of rounds needed to reduce a graph to an empty graph, where each round deletes all maximum degree vertices of the graph. As such, we define approximate complement degeneracy order, which repeatedly removes vertices of largest log-degree. This reduces the number of rounds needed and closely approximates the number of wedges that must be processed using complement degeneracy order. We show in the full paper that using complement degeneracy order and approximate complement degeneracy order give the same work-efficient bounds as using degree order. We implement both of these using the parallel bucketing structure of Dhulipala et al. [18].

In total, the options for ranking are side order, degree order, approximate degree order, complement degeneracy order, and approximate complement degeneracy order.

3.1.2 Wedge aggregation We obtain wedge counts by aggregating wedges by endpoints. PARButterfly implements fully-parallel methods for aggregation including sorting, hashing, and histogramming, as well as a partially-parallel batching method.

We can aggregate the wedges by semisorting key-value pairs where the key is the two endpoints and the value is the center. Then, all elements with the same key are grouped together, and the size of each group is the number of wedges shared by the two endpoints. We implemented this approach using parallel sample sort from the Problem Based Benchmark Suite (PBBS) [9, 53] due to its better cache-efficiency over parallel semisort.

We can also use a parallel hash table to store key-value pairs where the key is two endpoints and the value is a count. We insert the endpoints of all wedges into the table with value 1, and sum the values on duplicate keys. The value associated with each key then represents the number of wedges that the two endpoints share. We use a parallel hash table based on linear probing with an atomic addition combining function [52].

Another option is to insert the key-value pairs into a parallel histogramming structure which counts the number of occurrences of each distinct key. The parallel histogramming structure that we use is implemented using a combination of semisorting and hashing [18].

Finally, in our partially-parallel batching method we process a batch of vertices in parallel and find the wedges incident on these vertices. Each vertex aggregates its wedges serially, using an array large enough to contain all possible second endpoints. The simple setting in our framework fixes the number of vertices in a batch as a constant based on the space available, while the wedge-aware setting determines the number of vertices dynamically based on the number of wedges that each vertex processes.

In total, the options for combining wedges are sorting, hashing, histogramming, simple batching, and wedge-aware batching.

3.1.3 Butterfly aggregation There are two main methods to translate wedge counts into butterfly counts, per-vertex or per-edge. One method is to make use of atomic adds, and add the obtained butterfly count for the given vertex/edge directly into an array, allowing us to obtain butterfly counts without explicit re-aggregation.

The second method is to reuse the aggregation method chosen for the wedge counting step and use sorting, hashing, or histogramming to combine the butterfly counts per-vertex or per-edge.

3.1.4 Other options There are a few other options for butterfly counting in PARButterfly. First, butterfly counts can be computed per vertex, per edge, or in total. For wedge aggregation methods apart from batching, since the number of wedges can be quadratic in the size of the original graph, it may not be possible to fit all wedges in memory at once; a parameter in our framework takes into account the number

\[ \binom{w}{2} \] butterflies on \( v_1, v_3 \)
\[ + (2 - 1) \] butterflies on \( u_1, u_2 \)
\[ + \binom{2}{2} \] butterflies on \( v_2, v_3 \)
\[ + (2 - 1) \] butterflies on \( u_1, u_2 \)
\[ + \binom{2}{2} \] butterflies on \( u_1, u_2 \)
\[ + (2 - 1) \] butterflies on \( v_1, v_2 \)
of wedges that can be handled in memory and processes subsets of wedges using the chosen aggregation method until they are all processed. Similarly, for wedge aggregation by batching, a parameter takes into account the available space and appropriately determines the number of vertices per batch.

**PARBUTTERFLY** also implements both edge and colorful sparsification as described by Sanei-Mehri et al. [48] to obtain approximate total butterfly counts. For approximate counting, the sub-sampled graph is simply passed to the framework shown in Figure 2 using any of the aggregation and ranking choices, and the final result is scaled appropriately. Note that this can only be used for total counts. Due to space constraints, we describe and analyze the sparsification algorithms in the full version of the paper.

Finally, Wang et al. [60] independently describe an algorithm for butterfly counting using degree ordering, as done in Chiba and Nishizeki [13], and also propose a cache optimization for wedge retrieval. Their cache optimization involves retrieving precisely the wedges given by Chiba and Nishizeki’s algorithm, but instead of retrieving wedges by iterating through the lower ranked endpoint (for every \( v \), retrieve wedges \((v, w, u)\) where \( w, u \) have higher rank than \( v \)), they retrieve wedges by iterating through the higher ranked endpoint (for every \( u \), retrieve wedges \((v, w, u)\) where \( w, u \) have higher rank than \( v \)). Inspired by their work, we have augmented PARBUTTERFLY to include this cache optimization for all of our orderings.

### 3.2 Peeling Framework

Butterfly peeling classifies induced subgraphs by the number of butterflies that they contain. Formally, a vertex induced subgraph is a \( k\)-tip if it is a maximal induced subgraph such that for a bipartition, every vertex in that bipartition is contained in at least \( k \) butterflies and every pair of vertices in that bipartition is connected by a sequence of butterflies. Similarly, an edge induced subgraph is a \( k\)-wing if it is a maximal induced subgraph such that every edge is contained within at least \( k \) butterflies and every pair of edges is connected by a sequence of butterflies.

The tip number of a vertex \( v \) is the maximum \( k \) such that there exists a \( k\)-tip containing \( v \), and the wing number of an edge \( (u, v) \) is the maximum \( k \) such that there exists a \( k\)-wing containing \( (u, v) \). **Vertex peeling**, or **tip decomposition**, involves finding all tip numbers of vertices in a bipartition \( U \), and **edge peeling**, or **wing decomposition**, involves finding all wing numbers of edges.

The sequential algorithms for vertex peeling and edge peeling involve finding butterfly counts and in every round, removing the vertex or edge contained within the fewest number of butterflies, respectively. In parallel, instead of removing a single vertex or edge per round, we remove all vertices or edges that have the minimum number of butterflies.

The peeling framework is shown in Figure 4, and supports vertex peeling (tip decomposition) and edge peeling (wing decomposition). Because it also involves iterating over wedges and aggregating wedges by endpoint, it contains similar parameters to those in the counting framework. However, there are a few key differences.

First, ranking is irrelevant, because all wedges containing a peeled vertex must be accounted for regardless of order. Also, using atomic add operations to update butterfly counts is not work-efficient with respect to our peeling data structure (see Section 4.3), so we do not have this as an option in our implementation. Finally, vertex or edge peeling can only be performed if the counting framework produces per-vertex or per-edge butterfly counts, respectively.

Thus, the main parameter for the peeling framework is the choice of method for wedge aggregation: sorting, hashing, histogramming, simple batching, or wedge-aware batching. These are precisely the same options described in Section 3.1.2.

### 4 PARBUTTERFLY Algorithms

We describe in detail here our parallel algorithms for butterfly counting and peeling, and state their theoretical bounds. Due to space constraints, we defer the proofs of the bounds to the full version of this paper. Our theoretically-efficient parallel algorithms are based on the work-efficient sequential butterfly listing algorithm, introduced by Chiba and Nishizeki [13].

Wang et al. [59] proposed the first algorithm for butterfly counting per vertex, which is not work-efficient. They also give a simple parallelization of their counting algorithm that is not work-efficient. Moreover, Sanei-Mehri et al. [48] and Sarıyüce and Pinar [49] give sequential butterfly counting and peeling algorithms respectively, but neither are work-efficient.

#### 4.1 Preprocessing

The main subroutine in butterfly counting involves processing a subset of wedges of the graph; previous work differ in the way in which they choose wedges to process. As mentioned in Section 3.1.1, Chiba and Nishizeki [13] choose wedges by first ordering vertices by decreasing order of degree and then for each vertex in order, obtaining all wedges with said vertex as an endpoint and deleting the vertex. The ordering of vertices does not affect the correctness of the algorithm – in fact, Sanei-Mehri et al. [48] use this precise algorithm but with all vertices from one bipartition of the graph ordered before all vertices from the other bipartition. Importantly, Chiba and Nishizeki’s [13] decreasing degree ordering gives the work-efficient bounds \( O(\alpha m) \) on butterfly counting.

---

**PARBUTTERFLY Framework for Peeling**

1. Obtain butterfly counts: Obtain per-vertex or per-edge butterfly counts from the counting framework.
2. Peel: Iteratively remove vertices or edges with the lowest butterfly count from the graph until an empty graph is reached.

---

Figure 4: PARBUTTERFLY Framework for Peeling

Copyright © 2020 by SIAM
Unauthorized reproduction of this article is prohibited
Algorithm 1 Preprocessing

1: procedure PREPROCESS(G = (U, V, E), f)
2: V′ ← sort(U ∪ V; f) ⊢ Sort vertices in increasing order of rank
3: Let u′’s rank R[u] be its index in V′
4: E′ ← \{(R[u], R[v]) | (u, v) ∈ E\} ⊢ Rename vertices to their rank
5: G′ = (V′, E′)
6: parfor u ∈ V′ do
7: N′ G(u) ← sort({v | (u, v) ∈ E′}) ⊢ Sort neighbors by decreasing order of rank
8: Store deg u(u) and deg v(u) for all (u, v) ∈ E′
9: return G′

Throughout this section, we use decreasing degree ordering to obtain the same work-efficient bounds in our parallel algorithms. However, using approximate degree ordering, complement degeneracy ordering, and approximate complement degeneracy ordering also gives us these work-efficient bounds; we defer a proof of the work-efficiency of these orderings to the full paper. Furthermore, our exact and approximate counting algorithms work for any ordering; only the theoretical analysis depends on the ordering.

We use rank to denote the index of a vertex in some ordering, in cases where the ordering that we are using is clear or need not be specified. We define a modified degree, deg u(u), to be the number of neighbors u′ ∈ N(u) such that rank(u′) > rank(v). We also define a modified neighborhood, N r(u), to be the set of neighbors u′ ∈ N(u) such that rank(u′) > rank(v).

We give a preprocessing algorithm, PREPROCESS (Algorithm 1), which takes as input a bipartite graph and a ranking function f, and renames vertices by their rank in the ordering. The output is a general graph (we discard bipartite information). Note that when we mention vertices u and v on this general graph in the rest of this section, they have not necessarily originated from bipartitions U and V respectively. PREPROCESS also sorts neighbors by decreasing rank.

The following lemma summarizes the complexity of preprocessing.

**Lemma 4.1.** Preprocessing can be implemented in O(m) expected work and O(\log m) span w.h.p.

### 4.2 Counting algorithms

In this section, we present our parallel algorithms for butterfly counting.

The following equations describe the number of butterflies per vertex and per edge. Sanei-Mehri et al. [48] derived and proved the per-vertex equation, based on Wang et al.’s [59] equation for the total number of butterflies. We give a short proof of the per-edge equation.

**Lemma 4.2.** For a bipartite graph G = (U, V, E), the number of butterflies containing a vertex u is given by

\[
\sum_{u′ ∈ N G(u)} \frac{|N(u) \cap N(u′)|}{2}.
\]

The number of butterflies containing an edge (u, v) ∈ E is given by

\[
\sum_{u′ ∈ N(v) \setminus \{u\}} (|N(u) \cap N(u′)| - 1).
\]

**Proof.** The proof for the number of butterflies per vertex is given by Sanei-Mehri et al. [48]. For the number of butterflies per edge, we note that given an edge (u, v) ∈ E, each butterfly that (u, v) is contained within has additional vertices u′ ∈ U, v′ ∈ V and additional edges (u′, v), (u, v′), (u′, v′) ∈ E. Thus, iterating over all u′ ∈ N(v) (where u′ ≠ u), it suffices to count the number of vertices v′ ≠ v such that v′ is adjacent to u and to u′. In other words, it suffices to count v′ ∈ N(u) ∩ N(u′) \ \{v\}. This gives us precisely \sum_{u′ ∈ N(v) \setminus \{u\}} (|N(u) \cap N(u′)| - 1) as the number of butterflies containing (u, v).

Note that in both equations given by Lemma 4.2, we iterate over wedges with endpoints u and u′ to obtain our desired counts (Step 4 of Figure 2). We now describe how to retrieve the list of wedges (Step 2 of Figure 2).

#### 4.2.1 Wedge retrieval

There is a subtle point to make in retrieving all wedges. Once we have retrieved all wedges with endpoint u, Equation (4.1) gives the number of butterflies that u contributes to the second endpoints of these wedges, and Equation (4.2) gives the number of butterflies that u contributes to the centers of these wedges. As such, given the wedges with endpoint u, we can count not only the number of butterflies on u, but also the number of butterflies that u contributes to other vertices of our graph. Thus, after processing these wedges, there is no need to reconsider u.

From Chiba and Nishizeki’s [13] work, we must retrieve all wedges containing endpoints u in decreasing order of degree, and then delete u from the graph (i.e., do not consider any other wedge containing u).

We introduce here a parallel wedge retrieval algorithm, GET-WEDGES (Algorithm 2) that takes as input a preprocessed (ranked) graph. It iterates through all vertices u and retrieves all wedges with endpoint u such that the center and second endpoint both have rank greater than u (Lines 4–9).

**Algorithm 2** Parallel wedge retrieval

1: procedure GET-WEDGES(G = (V, E))
2: Use PREFIX-SUM to compute a function I that maps wedges to indices in order
3: Initialize W to be an array of wedges
4: parfor u1 ∈ V do
5: parfor i ← 0 to deg u1(u1) do
6: v ← N(u1)[i] ⊢ v is the neighbor of u1
7: parfor j ← 0 to deg u1(v) do
8: u2 ← N(v)[j] ⊢ u2 is the neighbor of v
9: W[I(i, j)] ← ((u1, u2), 1, v) ⊢ (u1, u2) are the endpoints, v is the center of the wedge
10: return W

Copyright © 2020 by SIAM
Unauthorized reproduction of this article is prohibited
This is equivalent to Chiba and Nishizeki’s algorithm which deletes vertices from the graph, but we do not modify the graph to allow all wedges to be processed in parallel. We process exactly the wedges that Chiba and Nishizeki process, and they prove that they process $O(\alpha m)$ wedges. GET-WEDGES (Algorithm 2) takes $O(\alpha m)$ work and $O(\log m)$ span.

After retrieving our wedges, we group wedges that share the same endpoints. We define a subroutine GET-FREQ that takes a sequence $S$, rearranges $S$ to group entries with the same key, and returns two arrays: a list of keys and their frequencies, and the indices of $S$ where entries of the same key are grouped. This can be implemented using semisorting, hashing, or histrogramming, as discussed in Section 3.1. For an input of length $n$, GET-FREQ takes $O(n)$ expected work and $O(\log n)$ span w.h.p. using any of the three aggregation methods [18, 23, 26]. We have $O(\alpha m)$ wedges, and so GET-FREQ takes $O(\alpha m)$ expected work and $O(\log m)$ span w.h.p.

The following lemma summarizes the complexity of wedge retrieval and counting.

**Lemma 4.3.** Retrieving a list of all wedges and counting the number of wedges that share the same endpoints can be implemented in $O(\alpha m)$ expected work and $O(\log m)$ span w.h.p.

Note that this is a better worst-case work bound than the work bound of $O(\sum_{v \in V} \deg(v)^2)$ using side order. In the worst-case $O(\alpha m) = O(m^{1.5})$ while $O(\sum_{v \in V} \deg(v)^2) = O(mn)$. We have that $mn = \Omega(m^{1.5})$, since $n = \Omega(m^{0.5})$.

**4.2.2 Per vertex** We now describe the butterfly counting per vertex algorithm, which is given as COUNT-V in Algorithm 3. We implement preprocessing and wedge retrieval in Lines 13 and 14, respectively.

We note that following Line 2, by counting the frequency of wedges by endpoints, for each fixed vertex $u_1$ we have obtained in $R$ all possible endpoints $(u_1, u_2) \in V' \times V'$ with the size $|N(u_1) \cap N(u_2)|$. By Lemma 4.2, for each endpoint $u_2$, $u_1$ contributes $(|N(u_1)|^2 - 1)$ butterflies, and for each center $v$, $u_1$ contributes $|N(u_1) \cap N(u_2)| - 1$ butterflies. Thus, we compute the per-vertex counts by iterating through $R$ to add the count per endpoint (Line 6) and iterating through $W$ to add the count per center (Line 9). The total complexity of butterfly counting per vertex is given as follows.

**Theorem 4.1.** Butterfly counting per vertex can be performed in $O(\alpha m)$ expected work and $O(\log m)$ span w.h.p.

**4.2.3 Per edge** We now describe the butterfly counting per edge algorithm, which is given as COUNT-E in Algorithm 4. We implement preprocessing and wedge retrieval as described previously.

As we discussed in Section 4.2.2, following Step 3 for each fixed vertex $u_1$ we have in $R$ all possible wedge endpoints $(u_1, u_2) \in V' \times V'$ with the size $|N(u_1) \cap N(u_2)|$. By Lemma 4.2, we compute per-edge counts by iterating through all of our wedge counts and adding $|N(u_1) \cap N(u_2)| - 1$ to our butterfly counts for the edges contained in the wedges with endpoints $u_1$ and $u_2$. We note that $W$ has already been aggregated, and $F$ gives us the sections of $W$ that hold wedges corresponding with the endpoints in $R$. As such, we iterate through $R$ to obtain our count $|N(u_1) \cap N(u_2)| - 1$, and use $F$ to iterate through $W$ to obtain the edges contained in the corresponding wedges. As in Section 4.2.2, we use GET-FREQ to obtain the total sums. The total complexity of butterfly counting per edge is given as follows.

**Theorem 4.2.** Butterfly counting per edge can be performed in $O(\alpha m)$ expected work and $O(\log m)$ span w.h.p.

**4.3 Peeling algorithms** In this section, we present our parallel algorithms for butterfly peeling. The sequential algorithm for butterfly peeling [49, 65] is precisely the sequential algorithm for $k$-core [39, 50], except instead of updating the number of neighbors per vertex per round, we...
update the number of butterflies per vertex or edge per round. Thus, we base our parallel butterfly peeling algorithm on the parallel bucketing algorithm for \( k \)-core in Julienne [18]. In parallel, our butterfly peeling algorithm removes (peels) all vertices or edges with the minimum butterfly count in each round, and repeats until the entire graph has been peeled.

Zou [65] give a sequential butterfly peeling per edge algorithm that they claim takes \( O(m^2) \) work. However, their algorithm repeatedly scans the edge list up to the maximum number of butterflies per edge iterations, so their algorithm actually takes \( O(m^2 + m \cdot \max-b_v) \) work, where \( \max-b_v \) is the maximum number of butterflies per edge. This is improved by Sariyüce and Pinar’s [49] work; Sariyüce and Pinar state that their sequential butterfly peeling algorithms per vertex and per edge take \( O(\sum_{v \in V} \deg(v)^2) \) work and \( O(\sum_{u \in U} \sum_{v_1, v_2 \in N(u)} \max(\deg(v_1), \deg(v_2))) \) work, respectively. They account for the work to update butterfly counts, but do not discuss how to extract the vertex or edge with the minimum butterfly count per round. In their implementation, their bucketing structure is an array of size equal to the number of butterflies, and they sequentially scan this array to find vertices to peel. They scan through empty buckets, and so the time complexity for their butterfly peeling implementation is on the order of the maximum number of butterflies per vertex or per edge.

We design a more efficient bucketing structure, which stores non-empty buckets in a Fibonacci heap [21], keyed by the number of butterflies. We have an added \( O(\log n) \) factor to extract the bucket containing vertices with the minimum butterfly count. Note that insertion and updating keys in Fibonacci heaps take \( O(1) \) amortized time per key, which does not increase our work. We need to ensure that batch insertions, decrease-keys, and deletions in the Fibonacci are work-efficient and have low span. We present a parallel Fibonacci heap and prove its bounds in the full version of this paper. We show that a batch of \( k \) insertions takes \( O(k) \) expected work and \( O(\log n) \) span w.h.p., a batch of \( k \) decrease-key operations takes \( O(k) \) amortized expected work and \( O(\log^2 n) \) span w.h.p., and a parallel delete-min operation takes \( O(\log n) \) amortized work and \( O(\log n) \) span.

A standard sequential Fibonacci heap gives work-efficient bounds for sequential butterfly peeling, and our parallel Fibonacci heap gives work-efficient bounds for parallel butterfly peeling. The work of our parallel algorithms improve over the sequential algorithms of Sariyüce and Pinar [49].

Our actual implementation uses the bucketing structure from Julienne [18], which is not work-efficient in the context of butterfly peeling, but is fast in practice. Julienne materializes only 128 buckets at a time, and when all of the materialized buckets become empty, Julienne will materialize the next 128 buckets. To avoid processing many empty buckets, we use an optimization to skip ahead to the next range of 128 non-empty buckets during materialization.

### 4.3.1 Per vertex

The parallel vertex peeling (tip decomposition) algorithm is in PEEL-V (Algorithm 5). We peel vertices considering only the bipartition of the graph that produces the fewest number of wedges (considering the vertices in that bipartition as endpoints), which mirrors Sariyüce and Pinar’s [49] sequential algorithm and gives us work-efficient bounds for peeling; more concretely, we consider the bipartition \( X \) such that \( \sum_{u \in X} (\deg_v(u)/2) \) is minimized. Without loss of generality, let \( \overline{U} \) be this bipartition.

Vertex peeling takes as input the per-vertex butterfly counts from the \textsc{ParButterfly} counting framework. We create a bucketing structure mapping vertices in \( \overline{U} \) to buckets based on their butterfly count (Line 11). While not all vertices have been peeled, we retrieve the bucket containing vertices with the lowest butterfly count (Line 16), peel them from the graph, and compute the wedges removed due to peeling (Line 16). Finally, we update the buckets of the remaining vertices with affected butterfly counts (Line 17).

The main subroutine in PEEL-V is \textsc{Update-V} (Lines 1–9), which returns a set of vertices whose butterfly counts have changed after peeling a set of vertices. To compute updated butterfly counts, we use the equations in Lemma 4.2 and precisely the same overall steps as in our counting algorithms: wedge retrieval, wedge counting, and butterfly counting. Importantly, in wedge retrieval, for every peeled vertex \( u_1 \), we must gather all wedges with an endpoint \( u_1 \), to account for all butterflies containing \( u_1 \) (from Equation (4.1)). We process all peeled vertices \( u_1 \) in parallel (Line 3), and for each one we find all vertices \( u_2 \) in its 2-hop neighborhood, each of which contributes a wedge (Lines 4–6). Finally, we aggregate the number of deleted butterflies per vertex (Line 7), and update

---

**Algorithm 5 Parallel vertex peeling (tip decomposition)**

1: procedure UPDATE-V\((G = (U, V, E), B, A)\)
2: parfor \( u_1 \in A \) do
3: parfor \( v \in N(u_1) \) do
4: parfor \( u_2 \in N(v) \) where \( u_2 \neq u_1 \) do
5: Store \( ((u_1, u_2), 1, v) \) in \( W \) if \( (u_1, u_2) \) is the key, 1 is the frequency
6: \( B' \leftarrow \text{COUNT-V-WEDGES}(G, W) \)
7: Subtract corresponding counts \( B' \) from \( B \)
8: return \( B \)

10: procedure PEEL-V\((G = (U, V, E), B)\) \( \triangleright \) \( B \) is an array of butterfly counts per vertex
11: Let \( K \) be a bucketing structure mapping \( \overline{U} \) to buckets based on \# of butterflies
12: \( f \leftarrow 0 \)
13: while \( f < \vert U \vert \) do
14: \( A \leftarrow \text{all vertices in next bucket (to be peeled)} \)
15: \( f \leftarrow f + \vert A \vert \)
16: \( B \leftarrow \text{UPDATE-V}(G, B, A) \) \( \triangleright \) Update # butterflies
17: Update the buckets of changed vertices in \( B \)
18: return \( K \)
the butterfly counts (Lines 8). The wedge aggregation and butterfly counting steps are precisely as given in our vertex counting algorithm (Algorithm 3). Like in Algorithm 2, we also need to compute a mapping from wedges to indices in \( W \) using prefix sums, but we omit this from the pseudocode for simplicity.

The work of PEEL-V is dominated by the work spent in the UPDATE-V subroutine, which is precisely the number of wedges with endpoints in \( U \), or \( O\left(\sum_{v \in V} \deg(v)^2\right) \). The work analysis for COUNT-V-WEDGES then follows from a similar analysis as in Section 4.2.2. Using our parallel Fibonacci heap, extracting the next bucket on Line 14 takes \( O(\log n) \) amortized work and updating the buckets on Line 17 is upper bounded by the number of wedges.

To analyze the span of PEEL-V, let \( \rho_v \) be the vertex peeling complexity of the graph, or the number of rounds needed to completely peel the graph where in each round, all vertices with the minimum butterfly count are peeled. Then, the overall span of PEEL-V is \( O(\rho_v \log^2 n) \) w.h.p.

If the maximum number of per-vertex butterflies is \( \Omega(\rho_v \log n) \), which is likely true in practice, then the work of the algorithm described above is faster than Sariyüz and Pinar’s [49] sequential algorithm, which takes \( O\left(\max-b_v + \sum_{v \in V} \deg(v)^2\right) \) work, where \( \max-b_v \) is the maximum number of butterflies per-vertex.

We must now handle the case where \( \max-b_v \) is \( O(\rho_v \log n) \). While we do not know \( \rho_v \) at the beginning of the algorithm, we can start running the algorithm as stated (with the Fibonacci heap), until the number of peeling rounds \( q \) is equal to \( \max-b_v / \log n \). If this occurs, then since \( q \leq \rho_v \), we have that \( \max-b_v \) is at most \( \rho_v \log n \) (if this does not occur, we know that \( \max-b_v \) is greater than \( \rho_v \log n \), and we finish the algorithm as described above). Then, we terminate and restart the algorithm using the original bucketing structure of Dhulipala et al. [19], which will give an algorithm with \( O\left(\max-b_v + \sum_{v \in V} \deg(v)^2\right) \) expected work and \( O(\rho_v \log^2 n) \) span w.h.p. The work bound matches the work bound of Sariyüz and Pinar and therefore, our algorithm is work-efficient.

The overall complexity of butterfly vertex peeling is as follows.

**Theorem 4.3.** Butterfly vertex peeling can be performed in \( O\left(\min(\max-b_v, \rho_v \log n) + \sum_{v \in V} \deg(v)^2\right) \) expected work and \( O(\rho_v \log^2 n) \) span w.h.p., where \( \max-b_v \) is the maximum number of per-vertex butterflies \( \rho_v \) is the vertex peeling complexity.

### 4.3.2 Per edge

While the bucketing structure for butterfly peeling by edge follows that for butterfly peeling by vertex, the algorithm to update butterfly counts within each round is different. Based on Lemma 4.2, in order to obtain all butterflies containing some edge \( (u_1, v_1) \), we must consider all neighbors \( u_2 \in N(v_1) \setminus \{v_1\} \) and then find the intersection \( N(u_1) \cap N(u_2) \). Each vertex \( v_2 \) in this intersection where \( v_2 \neq v_1 \) produces a butterfly \( (u_1, v_1, u_2, v_2) \). Thus, we must find each butterfly individually in order to count contributions per edge. This is precisely the serial update algorithm that Sariyüz and Pinar [49] use for edge peeling.

The algorithm for parallel edge peeling is given in PEEL-E (Algorithm 6). Edge peeling takes as input the per-edge butterfly counts from our counting framework. Line 14 initializes a bucketing structure mapping each edge to a bucket by butterfly count. While not all edges have been peeled, we retrieve the bucket containing vertices with the lowest butterfly count (Line 17), peel them from the graph and compute the wedges that were removed due to peeling (Line 19). Finally, we update the buckets of the remaining vertices whose butterfly count was affected due to peeling (Line 20).

The main subroutine is UPDATE-E (Lines 1–12), which returns a set of edges whose butterfly counts have changed after peeling a set of edges. For each peeled edge \( (u_1, v_1) \) in parallel (Line 3), we find all neighbors \( u_2 \) of \( v_1 \) where \( u_2 \neq u_1 \) and compute the intersection of \( N(u_1) \) and \( N(u_2) \) (Lines 4–5). All vertices \( v_2 \neq v_1 \) in the intersection contribute a deleted wedge, and we save the number of deleted wedges on the remaining edges in \( B' \) (Lines 6–9). Finally, we aggregate the number of deleted butterflies per edge (Line 10), and update the butterfly counts (Line 11). We need a mapping from edges to indices in \( B' \) (computed using prefix sums), but we omit this step for simplicity.

The work of PEEL-E is dominated by the total work spent in the UPDATE-E subroutine. For intersection (Line 5), we can use hash tables to store the adjacency lists of the vertices, and so we perform \( O\left(\min(\deg(u), \deg(u'))\right) \) work (by scanning

---

**Algorithm 6 Parallel edge peeling (wing decomposition)**

1. **procedure UPDATE-E** \((G = (U, V, E), B, A)\)
2. Initialize \( B' \) to store updated butterfly counts
3. **parfor** \((u_1, v_1) \in A\) do
4. **parfor** \(u_2 \in N(v_1)\) where \( u_2 \neq u_1 \) do
5. \( N \leftarrow \text{INTERSECT}(N(u_1), N(u_2))\)
6. Store \((u_2, v_1), |N| - 1, \) in \( B'\)
7. **parfor** \(v_2 \in N\) where \( v_2 \neq v_1\) do
8. Store \((u_1, v_1, v_2), 1, \) in \( B'\)
9. Store \((u_2, v_2), 1, \) in \( B'\)
10. \((B', \_\) \leftarrow \text{GET-FREQ}(B')\)
11. Subtract corresponding buckets in \( B''\) from \( B\)
12. **return** \( B\)
13. **procedure PEEL-E** \((G = (U, V, E), B)\) \(\triangleright B\) is an array of butterfly counts per edge
14. Let \( K \) be a bucketing structure mapping \( E \) to buckets based on \# of butterflies
15. \( f \leftarrow 0\)
16. while \( f < m \) do
17. \( A \leftarrow \) all edges in next bucket (to be peeled)
18. \( f \leftarrow f + |A|\)
19. \( B \leftarrow \text{UPDATE-E}(G, B, A)\) \(\triangleright \) Update # butterflies
20. Update the buckets of changed edges in \( B\)
21. **return** \( K\)
We use Cilk Plus’s work-stealing scheduler \cite{10,36} and we
\[
\sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u'))
\]
expected work. As in vertex peeling, to analyze the span of PEEL-E, we define \( \rho_c \) to be the edge peeling complexity of the graph, or the number of rounds needed to completely peel the graph where in each round, all edges with the minimum butterfly count are peeled. The overall span of PEEL-E is \( \Omega(\rho_c \log^2 m) \) w.h.p.

Similar to vertex peeling, if the maximum number of per-edge butterflies is \( \Omega(\rho_c \log m) \), which is likely true in practice, then the work of our algorithm is faster than the sequential algorithm by Sariyuce and Pinar \cite{49}. The work of their algorithm is \( \Omega(\max-b_e + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')) \) expected work and \( \Omega(\rho_c \log^2 m) \) span w.h.p. Our work bound matches the work bound of Sariyuce and Pinar and therefore, our algorithm is work-efficient.

To deal with the case where the maximum number of butterflies per-edge is small, we can start running the algorithm as stated (with the Fibonacci heap), until the number of peeling rounds \( q \) is equal to \( \max-b_e/\log m \). If this occurs, then since \( q \leq \rho_c \), we have that \( \max-b_e \) is at most \( \rho_c \log m \) (if this does not occur, we know that \( \max-b_e \) is greater than \( \rho_c \log m \), and we finish the algorithm as described above). Then, we terminate and restart the algorithm using the original bucketing structure of Dhulipala et al. \cite{19}, which will give an algorithm with work \( \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')) \) expected work and \( \Omega(\rho_c \log^2 m) \) span w.h.p. Our work bound matches the work bound of Sariyuce and Pinar and therefore, our algorithm is work-efficient.

The overall complexity of butterfly edge peeling is as follows.

**Theorem 4.4.** Butterfly edge peeling can be performed in \( \Omega(\min(\max-b_e, \rho_c \log m) + \sum_{(u,v) \in E} \sum_{u' \in N(v)} \min(\deg(u), \deg(u')) \) expected work and \( \Omega(\rho_c \log^2 m) \) span w.h.p., where \( \max-b_e \) is the maximum number of per-edge butterflies and \( \rho_c \) is the edge peeling complexity.

### 5 Experiments

#### 5.1 Environment

We run our experiments on an m5d.24xlarge AWS EC2 instance, which consists of 48 cores (with two-way hyper-threading), with 3.1 GHz Intel Xeon Platinum 8175 processors and 384 GiB of main memory. We use Cilk Plus’s work-stealing scheduler \cite{10,36} and we compile our programs with g++ (version 7.3.1) using the \(-O3\) flag. We test our programs on real-world bipartite graphs from the Koblenz Network Collection (KONECT) \cite{34}. We remove self-loops and duplicate edges. Figure 5 describes the properties of these graphs.

We compare our algorithms against Sanei-Mehri et al.’s \cite{48} and Sariyuce and Pinar’s \cite{49} work, which are the state-of-the-art sequential butterfly counting and peeling implementations, respectively.

When discussing aggregation methods, we use the prefix “A” to refer to using atomic adds for butterfly aggregation, and we take a lack of prefix to mean that the wedge aggregation method was used for butterfly aggregation. “BatchS” is simple batching and “BatchWA” is wedge-aware batching.

#### 5.2 Results

##### 5.2.1 Butterfly counting

Figure 6 shows runtimes over different aggregation methods for counting per vertex (per-edge count and total count runtimes are in the full version of this paper), for the seven datasets in Figure 5 with sequential counting times exceeding 1 second. The times are normalized to the fastest combination of aggregation and ranking methods per graph. We find that simple batching and wedge-aware batching give the best runtimes for butterfly counting in general. Among the work-efficient aggregation methods, hashing and histogramming with atomic adds are often faster than sorting, particularly for larger graphs due to increased parallelism and locality, respectively. Our fastest parallel runtimes for each dataset for total, per-vertex, and per-edge counts are shown in Figure 7.

We also implemented sequential algorithms for butterfly counting in PARBUTTERFLY that do not incur any parallelism overheads. Figure 7 includes the runtimes for our sequential counting implementations, as well as runtimes for implementations from previous works, all of which we tested on the same machine. The code from Sanei-Mehri et al. and Sariyuce and Pinar \cite{49} are serial implementations for global and local butterfly counting, respectively. PGD \cite{2} is a parallel framework for counting subgraphs of up to size 4 and ESCAPE is a serial framework for counting subgraphs of up to size 5. We timed only the portion of the codes that counted butterflies. Our configurations achieve parallel speedups between 6.3–13.6x over the best sequential implementations for large enough graphs.\footnote{By “large enough,” we mean graphs for which the sequential counting algorithms take more than 2 seconds to complete.} We also improve upon the previous best parallel implementations by 349.6–5169x due to having a work-efficient algorithm.

We examined self-relative speedups on livejournal for per-vertex and per-edge counting, respectively, and across all rankings, we achieve self-relative speedups between 10.4–30.9x for per vertex counting, between 9.2–38.5x for per edge counting, and between 7.1–38.4x for total counting.

##### 5.2.2 Ranking

We defer a full discussion of the effect of different rankings to the full version of the paper. In brief, different rankings change the number of wedges that we must process, and complement degeneracy and approximate complement degeneracy minimizes the number of wedges that we process across all of the real-world graphs considered. However, complement degeneracy is not feasible in practice, since the time for ranking often exceeds the time for the actual
5.2.3 Approximate counting Figure 8 shows runtimes for both colorful sparsification and edge sparsification on orkut, as well as the corresponding single-threaded times. We see that over a variety of probabilities $p$ we achieve self-relative speedups between 4.9–21.4x.

5.2.4 Cache optimization Using Wang et al.’s [60] cache optimization for total, per-vertex, and per-edge parallel butterfly counting gives speedups between 1.0–1.7x of our parallel butterfly counting algorithms without the cache optimization, considering the best aggregation and ranking methods for each case. We did not see speedups using the cache optimization on some small graphs, with runtimes

Figure 5: These are relevant statistics for the KONECT [34] graphs that we experimented on. Note that we only tested peeling algorithms for which Sariyuce and Pinar’s [49] serial peeling algorithms completed in less than 5.5 hours. As such, there are certain graphs for which we have no available $\rho_v$ and $\rho_e$ data, and these entries are represented by a dash.

![Figure 5](image)

**Dataset** | **Abbreviation** | **$|U|$** | **$|V|$** | **$|E|$** | **# butterflies** | $\rho_v$ | $\rho_e$
--- | --- | --- | --- | --- | --- | --- | ---
DBLP | dblp | 4,000,150 | 1,425,813 | 8,649,016 | 21,040,464 | 4,806 | 1,853
Github | github | 120,867 | 56,519 | 440,237 | 50,894,505 | 3,541 | 14,061
Wikipedia edits (it) | itwiki | 2,225,180 | 137,693 | 12,644,802 | 307,384,076 | 374 | 602,142
Discogs label-style | discogs | 383 | 1,617,943 | 5,740,842 | 77,383,418,076 | 10,676 | 123,859
Discogs artist-style | discogs_style | 383 | 1,617,943 | 5,740,842 | 77,383,418,076 | 374 | 602,142
LiveJournal | livejournal | 7,489,073 | 3,201,203 | 112,307,385 | 3,297,158,439,527 | 423 | 0
Wikipedia edits (en) | enwiki | 21,416,395 | 3,819,691 | 122,075,170 | 2,036,443,879,822 | 96.09 | 131,996
Delicious user-item | delicious | 33,778,223 | 833,081 | 101,798,957 | 56,892,252,403 | 165,850 | 123,859
Orkut | orkut | 8,730,857 | 2,783,196 | 327,037,487 | 22,131,701,295 | 206,756,209,850 | 1089.04
Web trackers | web | 27,665,730 | 12,756,244 | 140,613,762 | 20,067,567,209,850 | 19314.87 | 19314.87

Figure 6: These are the parallel runtimes for butterfly counting per vertex, considering different wedge aggregation and butterfly aggregation methods. We consider the ranking that produces the fastest runtime for each graph; * refers to side ranking, # refers to approximate complement degeneracy ranking, and ◦ refers to approximate degree ranking. All times are scaled by the fastest parallel time, as indicated in parentheses.

![Figure 6](image)

**Dataset** | **Abbreviation** | **Total Counts** | **Per-Vertex Counts** | **Per-Edge Counts** | **Total Counts** | **Per-Vertex Counts** | **Per-Edge Counts**
--- | --- | --- | --- | --- | --- | --- | ---
itwiki | itwiki | 0.10* | 1.38* | 1.63 | 1.03* | 1.43* | 6.06 | 0.37* | 3.24* | 19314.87
discogs | discogs | 0.90# | 1.30° | 4.12 | 234.48 | 2.08 | 0.93# | 1.53° | 96.09 | 0.90# | 5.01° | 1089.04
livejournal | livejournal | 3.83* | 35.41° | 37.80 | ≥ 5.5 hrs | 39.06 | 5.05* | 36.22° | 158.79 | 10.26° | 105.65° | ≥ 5.5 hrs
enwiki | enwiki | 8.29° | 39.47° | 69.10 | ≥ 5.5 hrs | 151.63 | 11.75° | 75.10° | 608.53 | 16.73° | 167.69° | ≥ 5.5 hrs
delicious | delicious | 13.52° | 165.03° | 162.00 | ≥ 5.5 hrs | 286.86 | 18.36° | 182.00° | 1027.12 | 23.58° | 321.02° | ≥ 5.5 hrs
orkut | orkut | 35.07° | 423.02° | 403.46 | ≥ 5.5 hrs | 1321.20 | 66.19° | 439.02° | 2841.27 | 131.07° | 1256.83° | ≥ 5.5 hrs
web | web | 12.18° | 115.53° | 4340 | ≥ 5.5 hrs | 172.77 | 15.89° | 195.43° | ≥ 5.5 hrs | 17.40° | 218.15° | ≥ 5.5 hrs

Figure 7: These are best runtimes in seconds for parallel and sequential butterfly counting from PARBUTTERFLY (PB), as well as runtimes from previous work. Note that PGD [2] is parallel, while the rest of the implementations are serial. Also, for the runtimes from our framework, we have noted the ranking used; * refers to side ranking, # refers to approximate complement degeneracy ranking, and ◦ refers to approximate degree ranking. The wedge aggregation method used for the parallel runtimes was simple batching, except the cases labeled with ◦ , which used wedge-aware batching.

Side ordering often outperforms the other rankings due to better locality, especially if the number of wedges processed by the other rankings does not greatly exceed the number of wedges given by side ordering.
under 4 seconds. More detailed experimental results are provided in the full version of the paper.

5.2.5 Butterfly peeling Figure 9 shows the runtimes over different wedge aggregation methods for vertex peeling (the runtimes do not include the time for counting butterflies). Edge peeling runtimes are in the full version of the paper. We only report times for the datasets for which finished within 5.5 hours. We find that for vertex peeling, aggregation by histogramming largely gives the best runtimes, while for edge peeling, all of our aggregation methods give similar results.

We compare our parallel peeling times to our single-threaded peeling times and serial peeling times from Sariyüce and Pinar’s [49] implementation, which we ran in our environment and which are in Figure 10. Compared to Sariyüce and Pinar [49], we achieve speedups between 1.3–30696x for vertex peeling and between 3.4–7.0x for edge peeling. Our speedups are highly variable because they depend heavily on the peeling complexities and the number of empty buckets processed. Our largest speedup of 30696x occurs for vertex peeling on discogs_style where we efficiently skip over many empty buckets, while the implementation of Sariyüce and Pinar sequentially iterates over the empty buckets.

Moreover, comparing our parallel peeling times to their corresponding single-threaded times, we achieve speedups between 1.0–10.7x for vertex peeling and between 2.3–10.4x for edge peeling. We did not see self-relative parallel speedups for vertex peeling on discogs_style, because the total number of vertices peeled (383) was too small.

6 Related Work
There have been several sequential algorithms designed for butterfly counting and peeling. Wang et al. [59] propose the first algorithm for butterfly counting over vertices in $O(\sum_{v \in V} \deg(v)^2)$ work, and Sanei-Mehri et al. [48] introduce a practical speedup by choosing the vertex partition with fewer wedges to iterate over. Sanei-Mehri et al. [48] also introduce approximate counting algorithms based on sampling and graph sparsification. Later, Zhu et al. [64] present a sequential algorithm for counting over vertices based on ordering vertices (although they do not specify which order) in $O(\sum_{v \in V} \deg(v)^2)$ work. They extend their algorithm to the external-memory setting and also design sampling algorithms.

Chiba and Nishizeki’s [13] original work on counting 4-cycles in general graphs applies directly to butterfly counting in bipartite graphs and has a better work complexity. Chiba and Nishizeki [13] use a ranking algorithm that counts the total number of 4-cycles in a graph in $O(\alpha m)$ work, where $\alpha$ is the arboricity of the graph. While they only give a total count in their work, their algorithm can easily be extended to obtain counts per-vertex and per-edge in the same time complexity. Butterfly counting using degree ordering was also described by Xia [62]. Sariyüce and Pinar [49] introduce algorithms for butterfly counting over edges, which similarly takes $O(\sum_{e \in E} \deg(e)^2)$ work. Zou [65] develop the first algorithm for butterfly peeling per edge, with $O(m^2 + m \cdot \max(b_e))$ work. Sariyüce and Pinar [49] give algorithms for butterfly peeling over vertices and over edges, which take $O(\max(b_v + \sum_{v \in V} \deg(v)^2))$ work and $O(\max(b_e + \sum_{u \in U} \sum_{v_1, v_2 \in N(u)} \max(\deg(v_1), \deg(v_2))))$ work, respectively.

In terms of prior work on parallelizing these algorithms, Wang et al. [59] implement a distributed algorithm using MPI that partitions the vertices across processors, and each processor sequentially counts the number of butterflies for vertices in its partition. They also implement a MapReduce algorithm, but show that it is less efficient than their MPI-based algorithm. The largest graph they report parallel times for is the deli graph with 140 million edges and $1.8 \times 10^{10}$ butterflies (the delicious tag-item graph in KONECT [34]). On this graph, they take 110 seconds on 16 nodes, whereas on the same graph we take 5.17 seconds on 16 cores.

Very recently, and independently of our work, Wang et al. [60] describe an algorithm for butterfly counting using degree ordering, as done in Chiba and Nishizeki [13], and also propose a cache optimization for wedge retrieval. Their parallel algorithm is our parallel algorithm with simple batching for wedge aggregation, except they manually schedule the threads, while we use the Cilk scheduler. They use their algorithm to speed up approximate butterfly counting, and also propose an external-memory variant.

There has been recent work on algorithms for finding subgraphs of size 4 or 5 [2, 16, 20, 28, 46], which can be used for butterfly counting as a special case. Marcus and Shavitt [38] design a sequential algorithm for finding subgraphs of up to size 4. Hocevar and Demsar [28] present a sequential algorithm for counting subgraphs of up to size 5. Pinar et al. [46] also present an algorithm for counting subgraphs of up to size 5 based on degree ordering as done in Chiba and Nishizeki [13]. Elenberg et al. [20] present a distributed algorithm for counting subgraphs of size 4. Ahmed et al. [2] present the PGD shared-memory framework for counting subgraphs of up to size 4. The work of their algorithm for counting 4-cycles is $O(\sum_{(u, v) \in E} (\deg(u) + \sum_{u' \in N(u)} \deg(u')))$, which is higher than that of our algorithms. Aberger et al. [1] design the EmptyHeaded framework for parallel subgraph finding based on worst-case optimal join algorithms [42]. For butterfly counting, their approach takes quadratic work. We were unable to obtain runtimes for EmptyHeaded because it ran out of memory in our environment. Dave et al. [16] present a parallel method for counting subgraphs of up to size 5 local to each edge. For counting 4-cycles, their algorithm is the same as PGD, which we compare with. There have also been various methods for approximating subgraph counts via sampling [3, 4, 12, 30, 32, 40, 47, 61]. Finally, there has also been significant work in the past decade on parallel triangle

Copyright © 2020 by SIAM
Unauthorized reproduction of this article is prohibited
batch aggregation and side ranking. We considered both the runtimes on 48 cores hyperthreaded and on a single thread. We ran these on orkut, using simple batch aggregation and side ranking.

### 7 Conclusion

We have designed a framework \textsc{ParButterfly} that provides efficient parallel algorithms for butterfly counting (global, per-vertex, and per-edge) and peeling (by vertex and by edge). We have also shown strong theoretical bounds in terms of work and span for these algorithms. The \textsc{ParButterfly} framework is built with modular components that can be combined for practical efficiency. \textsc{ParButterfly} outperforms the best existing parallel butterfly counting implementations, and we outperform the fastest sequential baseline by up to 13.6x for butterfly counting and by up to several orders of magnitude for butterfly peeling.

**Acknowledgements.** We thank Laxman Dhulipala for helpful discussions about bucketing. This research was supported by NSF Graduate Research Fellowship #1122374, DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

### References


